Conception d'algorithmes

Principes et 150 exercices non corrigés

Patrick Bosc, Marc Guyomard et Laurent Miclet

Seconde édition
entièrement revue et corrigée
Novembre 2018

Préfacé par Colin de la Higuera,
Président (2012 – 2015) de la Société des Informaticiens de France

Préface

Computational thinking is the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent.

La pensée algorithmique est l'ensemble des processus mentaux permettant de formuler les problèmes et leurs solutions dans une forme qui rend possible la résolution par un agent de traitement de l'information.

(Jeannette M. Wing, 2006)

À un moment où celles et ceux qui, depuis longtemps, militent pour que l'informatique soit enseignée au même titre que d'autres disciplines scientifiques se voient rejointes par un nombre croissant de scientifiques de sciences humaines, de personnalités politiques, de journalistes, il devient utile de rediscuter la façon d'aborder l'enseignement de la discipline informatique.

Cet enseignement doit permettre l'appropriation de sa culture, et à partir des bons points d'entrée, l'exploration des méandres de sa complexité depuis les sujets les plus classiques jusqu'aux applications les plus innovantes.

En effet, si l'on assiste à une convergence assez générale vers l'idée qu'il faut que l'enfant, l'adolescente et l'adolescent d'aujourd'hui, citoyenne et citoyen de demain, ait acquis les compétences et les connaissances suffisantes pour ne pas **subir** le monde numérique, la façon de le former fait beaucoup moins l'unanimité.

Parmi les idées reçues répandues, celle que la génération Y, celle des *enfants du numérique*, n'a pas grand-chose à apprendre, a fait long feu. Elle était sans doute liée au regard attendri de personnes admiratives devant un bambin qui se saisit d'une tablette, ou d'un adolescent capable d'utiliser ses deux pouces pour taper un message sur son téléphone portable. Maintenant, l'adulte doit comprendre que de pareilles performances sont purement motrices et ne doivent pas faire croire que la nouvelle génération est *nativement* dotée de capacités dont la sienne est dépourvue.

Une deuxième idée reçue tient au lieu commun *« a-t-on besoin de savoir comment fonctionne un moteur pour pouvoir conduire une voiture ? »*. Elle se justifiait par un modèle ancien, dépassé, celui où il y avait d'un côté les informaticiens, de l'autre le reste de l'humanité, et dans lequel un être humain n'avait qu'à faire appel à un informaticien quand il avait besoin de résoudre un problème informatique. Aujourd'hui, sans doute parce que dans la résolution de tout problème - ou presque - il y a une part d'informatique, les limites de cette séparation binaire de l'humanité volent en éclats.

Une troisième idée reçue consiste à penser qu'une *simple éducation aux usages* est suffisante pour la grande majorité des jeunes, quelques-uns pouvant cependant être formés à l'informatique parce qu'ils deviendront informaticiennes ou informaticiens. On peut cependant penser qu'avec la quantité croissante d'usages, leur enseignement direct n'est plus rentable : s'il pouvait être plus raisonnable il y a quelques années d'enseigner l'usage d'une suite bureautique que d'enseigner l'informatique, il faut aujourd'hui, si l'on en reste aux usages qui s'avèrent indispensables y inclure également le travail coopératif avec les outils du *cloud*, les questions de sécurité, d'intégrité des données, de réseau, l'interrogation de bases de données, l'organisation de ses archives, les bonnes pratiques sur l'Internet...

C'est tout simplement devenu plus compliqué d'enseigner les usages que d'enseigner l'informatique !

Vers 2014, avec l'arrivée de la notion de culture-code, on voit un accès universel à la société numérique : il suffirait de s'initier à la programmation. On en vante les aspects ludiques, le levier en termes de créativité et on lui donne aussi comme vertu de pouvoir pallier des manques éducatifs.

Or, la raison pour laquelle de nombreux informaticiens ont volontiers accompagné la fièvre du code n'est pas qu'il importait de savoir construire des pages web ou des applications pour téléphones portables ! Le code n'est pas une fin en soi mais une clé au monde numérique. La clé ouvre la route à un changement de paradigme. Ce n'est pas un outil supplémentaire qu'on nous offre, c'est une autre façon d'aborder les choses.

Avant d'attaquer ce point, et d'expliquer - enfin - en quoi ce livre est très utile, introduisons cette question de changement de paradigme avec un exemple emprunté à Seymour Papert. Il s'agit de résoudre la multiplication suivante :

$$\text{XLIV} \star \text{XVII}$$

La méthode de résolution que l'on peut imaginer consiste à transformer la notation romaine en notation arabe, de poser 44*17 et probablement d'utiliser un outil numérique pour finir.

Mais on peut également se demander comment faisaient les Romains de l'époque, puis les Européens qui jusqu'au Moyen Âge ont eu à utiliser les nombres romains...

Pour résoudre des questions faisant intervenir de l'arithmétique, il était nécessaire de recoder l'information autrement, d'une façon permettant l'utilisation d'algorithmes appropriés que chacun pouvait utiliser. Le développement du commerce a entraîné, au Moyen âge, le remplacement de la numération romaine par la numération arabe. Adieu les chiffres romains, bienvenue aux chiffres arabes.

C'est une révolution similaire qui est nécessaire aujourd'hui : celle-ci n'est pas technologique. Elle se situe au niveau de nos façons de raisonner, de résoudre des problèmes.

On constate maintenant qu'un nombre sans cesse croissant de problèmes se résolvent en passant par trois étapes : transformation des données du problème en information, traitement algorithmique de cette information, restitution de la solution sous une forme utile, acceptable, agréable.

Cette façon de résoudre un problème, en passant par la transformation en information et sa résolution informatique s'appelle le *computational thinking* en anglais. En français, plusieurs traductions existent : la pensée informatique, la pensée computationnelle, la pensée algorithmique [1].

Les exemples sont très nombreux... Les mécanismes de freinage ou de direction d'une voiture ne sont plus des liaisons physiques entre le conducteur et les roues, mais des systèmes captant le geste du conducteur et transformant celui-ci en données informatiques, puis optimisant en fonction de ces données de façon à minimiser les risques, et transformant enfin l'information résultante en force exercée sur les objets physiques finaux (les roues).

Le résultat est celui que l'on veut, mais en plus, cette information peut être exploitée : il est possible de vérifier la constance de certains paramètres, de calculer la date de remplacement, de transmettre des informations au programme qui gère de son côté la puissance du moteur...

Il suffit de regarder autour de nous... les articles de presse, les emplois du temps d'un lycée, les horaires de notre bus, le contrôle aérien, la gestion d'un match de rugby, de

1. https://interstices.info/jcms/c_43267/la-pensee-informatique

très nombreux problèmes du domaine médical... Tout y passe : on capte, on travaille l'information, on restitue.

Or, la seconde étape de cette construction n'obéit pas à la simple logique du codage : elle repose pour beaucoup sur l'algorithmique. La transformation de nos données en résultats, qu'elle se fasse en une passe ou de façon continue, nécessite l'emploi d'algorithmes, de savoir choisir parmi ceux-ci, de les prouver aussi.

La pensée algorithmique repose donc de manière forte sur une culture algorithmique. Et celle-ci s'acquiert en utilisant des livres comme ce *Conception d'algorithmes : Principes et 150 exercices corrigés* que j'accueille avec plaisir, et qui doit devenir une référence en matière d'enseignement de l'algorithmique dans le monde francophone.

L'algorithmique, pour beaucoup d'informaticiens, est un art : l'écriture de l'algorithme est le moment où l'on cherche à trouver l'idée géniale, la structure cachée, celle qui va permettre de résoudre la question. Les auteurs nous invitent à conserver cet aspect artistique, mais à développer aussi une approche systématique... La même idée géniale se retrouve-t-elle dans plusieurs algorithmes correspondant à des problèmes différents ? Qu'est-ce qui, au niveau de l'analyse, permet d'aborder ces problèmes, finalement, de façon homogène ?

Un enjeu pédagogique particulier est très bien défendu dans ce livre : celui d'écrire des algorithmes prouvables. Si trouver des idées algorithmiques permettant de résoudre un problème est tout à fait amusant, qu'il est difficile ensuite de prouver que l'algorithme qui vient d'être écrit résout bien le problème !

Patrick Bosc, Marc Guyomard et Laurent Miclet, enseignants chevronnés, ont bâti sur leur expérience devant des étudiants d'IUT, d'école d'ingénieur, d'université, et proposent ici une approche tout à fait intéressante : plutôt que de proposer un cours avec quelques exercices, les auteurs ont choisi ici de résumer le cours en retenant les éléments les plus pertinents, de proposer des exercices et surtout de concentrer la plus grande partie de leur analyse dans les corrections de ceux-ci : en fait, ce sont ces corrections qui constituent le fil conducteur de l'ouvrage.

Pour conclure, on peut se poser la question du public de ce livre ? Si en 2015 ce public peut être constitué d'enseignants et d'étudiants des filières informatiques des universités et des écoles d'ingénieurs, qui auront intérêt à l'étudier dans le cadre de leurs études, mais également comme livre de référence de leur vie professionnelle si dans celle-ci ils sont conduits à résoudre des problèmes avec des algorithmes, on peut aussi présager qu'un public bien plus large sera bientôt confronté à des questions similaires... La prise de conscience du phénomène informatique a été, en France, soudaine : en l'espace de trois ou quatre années, l'informatique s'est installée au lycée (ISN, puis ICN), dans les classes préparatoires, au collège et on parle même de l'école primaire ! Pour faire face à ces besoins, il est envisagé de former des éducateurs, animateurs enseignants, des professeurs de toutes les disciplines... Bien entendu, un texte comme celui-ci n'a pas immédiatement sa place : s'il est prévu de commencer à enseigner l'informatique à tous les niveaux, il n'en demeure pas moins que les enseignants vont s'adresser, un peu partout, à des débutants. Mais l'enfant de six ans qui va commencer à coder avec Scratch en 2016... aura besoin à un moment ou un autre de connaissances plus étendues, et d'une enseignante ou d'un enseignant qui les maîtrisera.

Je souhaite que cet enseignant francophone ait appris avec ce livre.

Bonne lecture

Colin de la Higuera
Professeur à l'Université de Nantes
Président (2012-2015) de la Société informatique de France

Remerciements

Ce livre doit tout, ou presque, à nos établissements d'enseignement et de recherche : l'IUT de Lannion, l'ENSSAT et l'Université de Rennes 1. Nous sommes redevables en particulier aux cours et travaux dirigés d'« algorithmique avancée » qui ont été enseignés notamment par André Couvert, Jean-Michel Hélary, René Pédrono, Sophie Pinchinat et Michel Raynal. Nous remercions particulièrement Nelly Barbot, Arnaud Delhay, Allel Hadjali, Amaury Habrard et Damien Lolive pour leur aide lors de la rédaction de cet ouvrage.

Ce livre a été composé en LaTeX par les auteurs avec les logiciels libres TeXstudio, Texmaker, TeXnicCenter, TeXShop et MiKTeX. Les figures ont été faites avec PGF/TikZ. Le code comporte environ 2.400.000 signes.

Your total ignorance of that which you profess to teach merits the death penalty. I doubt whether you would know that St Cassian of Imola was stabbed to death by his students with their styli.

(J. K. Toole)

There is a computer disease that anybody who works with computers knows about. It's a very serious disease and it interferes completely with the work. The trouble with computers is that you 'play' with them !

(R. Feynman)

The Richard Feynman Problem-Solving Algorithm :
1. write down the problem ;
2. think very hard ;
3. write down the answer.

(M. Gell-mann)

Please do not ask me for solutions to the exercises. If you're a student, seeing the solution will rob you of the experience of solving the problem yourself, which is the only way to learn the material. If you're an instructor, you shouldn't assign problems that you can't solve yourself !

(J. Erickson)

Computer Science is no more about computers than astronomy is about telescopes.

(E. W. Dijkstra)

However beautiful the strategy, you should occasionally look at the results.

(W. Churchill)

Solving problems is a practical skill like, let us say, swimming.

(G. Pólya)

Carotte et bâton sont les deux mamelles de la pédagogie.

(P. Struillou)

Students identify computer science with a computer driving license.

(J. Hromkovič)

An' here I sit so patiently
Waiting to find out what price
You have to pay to get out of
Going thru all these things twice

(B. Dylan)

L'art de la citation est l'art de ceux qui ne savent pas réfléchir par eux-mêmes.

(Voltaire)

Table des matières

Présentation

Ce qu'est cet ouvrage,

Le sujet de cet ouvrage est l'algorithmique et son but est de l'enseigner. Selon nous, cette discipline peut se définir comme « l'art et la science d'écrire un algorithme pour résoudre un problème donné, de préférence en un temps minimal ». Cet ouvrage vise par conséquent à enseigner des méthodologies de conception d'algorithmes efficaces. Il cherche à le faire essentiellement par l'exemple.

Il est construit selon une double règle.

— Les chapitres couvrent un ensemble de méthodes qui s'appliquent à des structures de données diverses. Par conséquent, à un chapitre correspond une méthodologie de construction d'algorithme, non pas une structure de données. Par exemple, le chapitre programmation dynamique comporte des problèmes résolus dans les tableaux, d'autres dans les arbres, les graphes, les séquences, etc.

— Un chapitre est le plus souvent constitué d'une présentation informelle par un exemple, puis de la technique permettant d'écrire un algorithme selon cette méthode. Ensuite, au moins un problème complet est traité en détail. La suite du chapitre est constituée de problèmes, énoncés et corrigés. Les problèmes un peu complexes sont abordés de façon progressive afin de mettre en évidence l'aspect constructif de la solution proposée. Les corrigés sont détaillés afin de rendre le plus explicite possible les points clés du raisonnement suivi.

ce qu'il n'est pas,

Cet ouvrage n'est ni un cours de *Structure de Données et Algorithmes* ni même un cours d'*Algorithmique*. En effet, premièrement, il n'aborde pas les problèmes de l'organisation efficace des données. Les structures de données ne sont précisées que lorsqu'elles jouent un rôle central dans l'efficacité de l'algorithme. Pour l'essentiel, on suppose que le lecteur connaît le sujet et sait employer les bons outils (ou les bons paquetages ou les bonnes classes). On parlera donc de tableaux, de séquences, de graphes et autres structures sans généralement préciser la manière dont elles sont implantées.

Deuxièmement, il vise à enseigner la conception et les stratégies algorithmiques, mais pas sous la forme d'un cours complet. Il essaye plutôt de proposer des exemples qui aident à comprendre pourquoi et comment telle méthode peut être appliquée à tel problème.

et ce qu'il vise à être.

L'algorithmique est, avons-nous dit, l'art et la science d'écrire un algorithme. Nous souhaitons augmenter ces deux qualités chez le lecteur. Nous voudrions que, face à un nouveau problème, il puisse développer, grâce aux exemples qu'il aura vus, une sorte d'*intuition* de la méthode à employer (c'est le côté artiste, ou artisan). Mais nous désirons aussi qu'il sache *prouver* que l'emploi de cette méthode est effectivement plus efficace que celui d'une technique naïve (c'est le côté scientifique). Enseigner par l'exemple ne veut pas dire sacrifier la rigueur. Beaucoup de constructions d'algorithmes se font à partir de notions

bien fondées, comme la récurrence ou le raisonnement par l'absurde. Cependant, trop souvent, les problèmes proposés dans certaines notes de cours ou certains ouvrages sont résolus sans que la preuve de la solution soit apparente. Cela donne parfois l'impression que cette solution sort du chapeau du magicien (c'est-à-dire de l'astuce du professeur) et encourage l'idée exagérée qu'une bonne intuition peut remplacer une démonstration. Nous pensons que la déduction et l'induction sont deux modes de raisonnement nécessaires dans la construction d'un algorithme.

À qui s'adresse cet ouvrage ?

Ce livre est destiné à tous ceux qui, pour une raison ou pour une autre, sont concernés par les sciences du numérique et veulent apprendre ou enseigner l'algorithmique. Il sera évidemment utile aux élèves et aux étudiants en sciences du numérique, non pas comme un ouvrage d'initiation, mais plutôt comme un manuel de référence destiné à accompagner son lecteur non seulement pendant l'apprentissage, mais aussi dans sa vie professionnelle. Nous pensons particulièrement aux élèves de terminale en spécialité ISN, ou dans certains BTS, aux étudiants en IUT Informatique, Réseaux et Télécom, GEII, aux étudiants en licence informatique ou à connotation informatique, aux élèves des classes préparatoires scientifiques et des écoles d'ingénieurs.

Ce livre est également destiné aux enseignants, qui trouveront au fil des pages quantité de matériel pour les cours et les séances d'exercices (sans parler des examens). Ils utiliseront les introductions, les exercices et les corrections pour montrer à leurs apprenants comment se modélise un problème et comment se construit rationnellement une solution. Enfin, il est aussi destiné, en dehors du système d'enseignement classique, à tous ceux qui veulent se munir de connaissances solides sur une des bases de la science informatique : la construction des algorithmes.

Plan

Comme nous l'avons dit, cet ouvrage a été organisé pour présenter successivement différentes méthodologies de construction d'algorithmes. Cependant, il commence par un chapitre intitulé « Mathématiques et informatique : notions utiles », dans lequel sont rappelées un certain nombre de bases mathématiques (essentiellement les principes de démonstration, en particulier par récurrence) et de structures de données (ensembles, graphes, arbres, files) et où une vingtaine d'exercices sont proposés. Le chapitre 2 « Complexité d'un algorithme », dans la même intention de consolider les connaissances de base, rappelle les notions essentielles de ce domaine et donne quelques exercices.

Dans le chapitre 3, « Spécification, invariants, itération » sont exposés les principes de la construction rationnelle de boucle, accompagnés d'une quinzaine d'exercices dont quelques-uns sont assez difficiles. Le chapitre suivant, intitulé « Diminuer pour résoudre, récursivité » traite de la construction d'algorithmes récursifs, illustrée par une petite dizaine d'exercices. Il montre en particulier comment les méthodes de démonstration par récurrence s'appliquent pour certifier l'exactitude de procédures récursives résolvant un problème de taille donnée n, en faisant appel à la résolution de problèmes identiques de taille $(n-1)$.

On aborde dans le chapitre 5 la méthodologie des « Essais successifs ». Les techniques d'énumération récursives des solutions à un problème combinatoire sont décrites et des « patrons » de programmes sont donnés. Les principes d'élagage de l'arbre des solutions

sont décrits, qui sont illustrés par une vingtaine d'exercices. Le chapitre suivant reste dans le même sujet, mais traite les méthodes PSEP (Séparation et évaluation progressive, ou *branch and bound*) qui sont, elles, itératives. Quatre exercices sont donnés pour concrétiser cette méthode.

Le chapitre 7 est consacré aux « Algorithmes gloutons », qui cherchent à résoudre des problèmes analogues à ceux des chapitres précédents, en n'effectuant aucun retour en arrière ; il est important de faire la preuve qu'ils résolvent le problème posé. On y présente donc les façons usuelles pour démontrer qu'un tel algorithme est exact. Une quinzaine d'exercices sont proposés pour illustrer cette technique.

Dans le chapitre 8, on s'intéresse à une approche particulièrement féconde de conception d'algorithmes : « Diviser pour Régner ». Une classification des algorithmes de ce type est proposée et leur construction est illustrée par une trentaine d'exercices, dont certains sont difficiles.

Enfin, le chapitre 9 décrit la méthodologie de « Programmation dynamique » qui est également très féconde pour construire une solution optimale à un problème combinatoire. La richesse de cette technique est telle que nous proposons plus d'une trentaine d'exercices, donnant lieu à une grande variété de constructions algorithmiques dont plusieurs ont un intérêt pratique avéré. Là aussi, certains problèmes proposés sont difficiles.

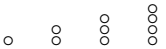
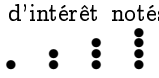
Nous avons essayé de choisir des exercices couvrant autant que possible la variété de chaque domaine abordé. Nous avons aussi cherché à ne pas proposer deux problèmes trop proches, tout en illustrant sur quelques problèmes l'applicabilité de plusieurs méthodologies. Au total, cet ouvrage traite près de cent cinquante exercices : pour chacun l'énoncé a été rédigé avec autant de précision que possible et les questions sont organisées pour aider le lecteur à construire la solution. Le corrigé, quant à lui se veut méthodique, rigoureux et complet.

Conventions de lecture

On trouvera dans la section « Notations » page 387 les conventions que nous utilisons pour les formules mathématiques et surtout pour les algorithmes. Il sera évidemment indispensable au lecteur de s'y référer en cas de doute sur la signification de tel ou tel symbole.

Tout au long de l'énoncé et du corrigé de chaque exercice, une note marginale rappelle les numéros de l'exercice et de la question en cours. La note ci-contre indique par exemple que l'on commence la question 3 de l'exercice 42. Sa réponse est signalée par la même note, avec R à la place de Q.

42 - Q 3

Chaque exercice est doublement coté, pour son intérêt intrinsèque et pour sa difficulté. Il y a quatre niveaux croissants d'intérêt notés  et quatre niveaux croissants de difficulté notés  . La notion d'intérêt d'un problème est assez subjective. Nous avons opté pour un croisement de divers critères, dont le principal est la qualité avec laquelle ce problème illustre la méthodologie à laquelle il appartient. Pour la difficulté, la cotation tient naturellement compte de la façon dont l'enoncé a été rédigé : un problème intrinsèquement difficile peut être adouci par une série de questions préparatoires graduées.

Commentaires sur les sources et sur la bibliographie

La bibliographie située en fin d'ouvrage est volontairement limitée à des livres, ou presque. Autrement dit, nous ne renvoyons jamais aux articles originaux où ont été publiés (s'il l'ont été) les algorithmes que nous présentons ici. C'est en effet le rôle d'un livre complet sur le sujet que de citer exhaustivement ses sources, plutôt que celui d'un livre d'exercices. Nous renvoyons donc aux livres de la bibliographie les lecteurs soucieux de connaître l'historique des algorithmes présentés. De ce point de vue, les ouvrages de D. Knuth [44], de G. Brassard et P. Bratley [15], de T. Cormen et *al.* [17] et de E. Horowitz et *al.* [39] sont particulièrement conseillés.

Aucun des algorithmes présentés ici ne se veut original. Pour l'essentiel, les sujets des exercices proviennent des livres de la bibliographie que nous présentons ci-dessous. Ils ont été soigneusement réécrits à des fins pédagogiques. Les autres ont pour origine des sources variées, en particulier le patrimoine commun des cours et travaux dirigés enseignés à l'Université de Rennes 1 et à l'ENSSAT de Lannion. Notre originalité n'est pas dans la création de nouveaux problèmes. En revanche, elle réside dans la construction de l'énoncé des exercices et dans la rigueur de l'élaboration et de la description des solutions.

Dans son excellent petit livre d'exercices, I. Parberry [55] donne son jugement sur une trentaine de livres d'enseignement d'algorithmique. Nous y renvoyons volontiers le lecteur pour qu'il se fasse un avis sur la bibliographie de langue anglaise datant d'avant 1995. Pour notre part, outre les inégalables ouvrages de T. Cormen et *al.* [17] et de D. Knuth [44], nous avons une préférence pour les livres écrits par U. Manber [48], par D. Gries [33], et plus récemment par J. Kleinberg et E. Tardos [43] et par J. Edmonds [27]. Les livres de R. Johnsonbaugh et M. Shaeffer [40], de E. Horowitz et *al.* [39], de S. Baase et A. Van Gelder [8], de R. Neapolitan et K. Naimipour [54], de A. Levitin [46] et de T. Goodrich and R. Tamassia [30] sont également des ouvrages récents à recommander.

La bibliographie en langue française sur l'algorithmique est plus mince. La référence (surtout pour les structures de données, moins pour l'algorithmique proprement dite) a longtemps été, à juste titre, l'ouvrage de C. Froidevaux et *al.* [28]. La traduction française du livre déjà cité de T. Cormen et *al.* [17] peut désormais lui être préférée. D'autres traduction ont été proposées, comme celles des ouvrages de R. Sedgewick [58]. Les livres de M. Quercia [56], de J.-M. Léry [47] et de J. Courtin et I. Kowarsky [19] (ce dernier est le plus complet) traitent plus de structures de données que d'algorithmique au sens strict. Le livre d'exercices d'algorithmique de L. Bougé et *al.* [14] et celui de A. Darte et S. Vaudenay [21] sont des recueils de problèmes du concours de l'ENS Lyon, toujours intéressants mais elliptiques en ce qui concerne la construction des solutions. L'excellent livre d'exercices et de problèmes d'algorithmique de B. Baynat et *al.* [9] est organisé par structures de données, non pas par types d'algorithmes. Il donne des solutions détaillées et des rappels de cours. Nous le conseillons bien volontiers, de même que le très bon ouvrage de J. Beauquier et *al.* [10], organisé selon le même principe et désormais disponible gratuitement sur le web. On trouve aussi en français de remarquables livres sur les algorithmes dans les graphes (par exemple celui de M. Gondran et M. Minoux, [29]), dans les séquences (M. Crochemore, [20]) et pour des problèmes d'algèbre et d'analyse (P. Naudin et C. Quitté, [53]).

Un ouvrage concis, mais de référence, sur la construction de programmes est celui de P. Berlioux et Ph. Bizard [13]. Celui de J. Arsac [5] mérite aussi d'être lu avec attention. Le livre récent de J. Julliand [41] est un excellent document d'introduction aux méthodes formelles de conception de programmes.

CHAPITRE 1

Mathématiques et informatique : quelques notions utiles

> Who can does ; who cannot do, teaches ; who cannot teach, teaches teachers.
>
> *(Paul Erdos)*

Tout au long de cet ouvrage, nous nous plaçons dans le cadre d'un développement *progressif* des algorithmes dans lequel chaque étape repose sur une construction saine. À cet effet, ce chapitre présente une variété d'outils permettant d'assurer la validité des constructions utilisées. Par ailleurs, au cours de cette démarche de développement, une première version d'un programme fait souvent appel, d'une part à des conditions exprimées dans le langage des prédicats, et d'autre part à des structures de données spécifiées dans le langage de la théorie des ensembles. Cette démarche est cependant insuffisante lorsque l'on cherche à obtenir une solution efficace (notamment sur le plan de la complexité temporelle – voir chapitre 2). Il faut alors procéder à un raffinement sur la base de structures de données taillées sur mesure. C'est pourquoi cette introduction est consacrée à divers objets et outils mathématiques, ainsi qu'aux principales structures de données utilisées ultérieurement.

Ce chapitre débute par des notions relatives aux raisonnements conduits fréquemment par la suite : i) le calcul propositionnel et le calcul des prédicats, ii) la démonstration par l'absurde, et enfin iii) la démonstration par récurrence, outil central des chapitres 4 et 8. Nous abordons ensuite les relations de récurrence et leur calcul, qui occupent une place privilégiée dans le chapitre 9. Nous en profitons pour donner notre acception des termes récurrence, induction et récursivité. La suite du chapitre s'articule principalement autour de la notion d'ensemble, et nous passons en revue les concepts de produit cartésien, de sac (multiensemble), de relation et de fonction ainsi que les principaux opérateurs s'y rattachant. Nous nous intéressons aux listes vues comme une illustration de la notion de structure inductive définie à partir de celle de produit cartésien. Nous nous arrêtons plus longuement sur les notions de graphe et d'arbre qui sont à l'origine de nombreux exercices, avant d'achever ce chapitre en présentant la notion de file de priorité dont l'intérêt se manifeste principalement à travers les chapitres 6 et 7, consacrés respectivement à la démarche PSEP et aux algorithmes gloutons.

Pour le lecteur intéressé, une construction rigoureuse de la logique classique et de la théorie des ensembles peut être trouvée dans [1]. Pour ce qui concerne la problématique du raffinement formel de structures de données, on peut consulter avec profit [36].

1.1 Aspects liés au raisonnement

1.1.1 Calcul des propositions et calcul des prédicats

L'ensemble prédéfini $\mathbb{B}$ est défini par $\mathbb{B} \mathrel{\widehat{=}} \{\textsf{vrai}, \textsf{faux}\}$. Ces deux valeurs seront parfois représentées par V/F ou v/f dans des tableaux pour des raisons de place. Si a et b sont des éléments de $\mathbb{B}$, alors :

- « a **et** b » vaut **vrai** si et seulement si a et b valent tous les deux **vrai**.
- « a **ou** b » vaut **faux** si et seulement si a et b valent tous les deux **faux**.
- « **non** a » vaut **vrai** si et seulement si a vaut **faux**.
- « $a \Rightarrow b$ » vaut **faux** si et seulement si **non** a et b valent tous les deux **faux**.
- « $a \Leftrightarrow b$ » vaut **vrai** si et seulement si a et b ont tous les deux la même valeur.

L'existence d'expressions indéfinies (par exemple pour cause d'accès en dehors du domaine d'une fonction, ou pour cause de division par 0) exige d'utiliser des opérateurs dits « courts-circuits » qui arrêtent l'évaluation dès que le résultat est acquis. Les deux opérateurs courts-circuits « **et alors** » et « **ou sinon** » se définissent de la manière suivante :

- Si a et b sont définis :

  a **et alors** $b \quad \Leftrightarrow \quad a$ **et** b.

  a **ou sinon** $b \quad \Leftrightarrow \quad a$ **ou** b.
- Si a est défini mais pas b :

  faux et alors $b \quad \Leftrightarrow$ **faux**.

  vrai et alors b n'est pas défini.

  faux ou sinon b n'est pas défini.

  vrai ou sinon $b \quad \Leftrightarrow$ **vrai**.
- Si a n'est pas défini, alors, quel que soit b :

  a **et alors** b n'est pas défini.

  a **ou sinon** b n'est pas défini.

Les quantificateurs $\forall$ et $\exists$ du calcul des prédicats se définissent par :

- « $\forall x \cdot (D \Rightarrow T(x))$ » vaut **vrai** si et seulement si, pour tout x appartenant au domaine D alors $T(x)$ vaut **vrai**. La formule vaut donc **vrai** si D est vide.
- « $\exists x \cdot (D$ **et** $T(x))$ » vaut **vrai** si et seulement s'il existe un x appartenant au domaine D tel que $T(x)$ vaut **vrai**. La formule vaut donc **faux** si D est vide.

La notation $\nexists$ est une abréviation pour **non** $\exists$.
La double égalité :

$$(a \Rightarrow b) = (\textbf{non } a \textbf{ ou } b) = (\textbf{non } b \Rightarrow \textbf{non } a)$$

est le fondement du raisonnement par contraposition, dans lequel on établit la validité de l'implication **non** $Q \Rightarrow$ **non** P pour prouver que P implique Q.

1.1.2 Démonstration par l'absurde

Le *raisonnement par l'absurde* ou *démonstration par l'absurde* (en anglais *proof by contradiction*) repose sur deux principes de la logique des propositions :

- le *tiers exclu*, qui affirme qu'une propriété qui n'est pas fausse est forcément vraie, donc que la conjonction d'une propriété et de sa négation prend la valeur logique **faux**,
- la définition de l'implication : $(P \Rightarrow Q) \mathrel{\widehat{=}} (\textbf{non}\ P\ \textbf{ou}\ Q)$.

Ces deux propriétés ont en particulier pour conséquence que (**non** P⇒**faux**) est équivalent à P, ce qui valide le raisonnement par l'absurde. En effet :

(**non** P ⇒ **faux**)
⇔ définition de l'implication
(**non**(**non** P) **ou faux**)
⇔ involutivité de la négation
(P **ou faux**)
⇔ propriété de la disjonction
P.

Démontrer une proposition P par l'absurde se fait en deux étapes :

1. on suppose que **non** P (c'est-à-dire que P est fausse),
2. on constate une contradiction entre cette supposition et une implication de cette supposition.

Il est à noter que si l'on ne parvient pas à une contradiction, on ne peut *rien conclure* quant à P.

Premier exemple Considérons la proposition P : « il n'y a pas de plus petit nombre rationnel strictement plus grand que 0 ». Pour prouver P par l'absurde, on commence par supposer la négation de P qui s'énonce : « il existe un plus petit nombre rationnel strictement positif r ». De la négation de P, on cherche à déduire une contradiction.

Soit $s = r/2$. Par construction, s est un nombre rationnel strictement plus grand que 0 et strictement plus petit que r. Mais cela est contradictoire avec P, qui affirme que r est le plus petit nombre rationnel.

Ainsi, on peut conclure que la proposition P est nécessairement vraie et qu'il n'existe donc pas de plus petit nombre rationnel strictement plus grand que 0.

Second exemple On veut montrer qu'il n'existe pas de rationnel positif dont le carré est 2, autrement dit prouver la proposition P s'énonçant « la racine carrée de 2 est irrationnelle ».

Démonstration. On va d'abord supposer qu'il existe un élément $x = p/q$ de $\mathbb{Q}_+$ (ensemble des rationnels positifs) tel que $x^2 = 2$, avec p et q premiers entre eux (c'est-à-dire que p/q est une fraction irréductible). On a :

$$\left(\frac{p}{q}\right)^2 = 2$$
$\Leftrightarrow$ arithmétique
$$p^2 = 2q^2$$
$\Leftrightarrow$ reformulation
$$2 \text{ divise } p^2$$
$\Leftrightarrow$ tout carré pair est celui d'un nombre pair
$$2 \text{ divise } p$$
$\Leftrightarrow$ $p = 2p'$
$$\exists p' \cdot (p^2 = 4p'^2 = 2q^2)$$
$\Leftrightarrow$ reformulation
$$2 \text{ divise } q^2$$
$\Leftrightarrow$ tout carré pair est celui d'un nombre pair
$$2 \text{ divise } q.$$

On en déduit que 2 divise à la fois p et q, donc que p et q ne sont pas premiers entre eux, ce qui contredit le fait que p/q est une fraction irréductible.

De façon analogue, on peut montrer qu'il n'existe pas de rationnel négatif dont le carré est 2, sinon l'opposé de ce nombre serait un rationnel positif dont le carré serait 2. Par conséquent, la racine carrée de 2 est irrationnelle [1].

1.1.3 Démonstration par récurrence à un indice

Nous abordons dans cette section un autre type de preuve, à savoir la démonstration *par récurrence* ou *par induction* à un indice. Nous en déclinons successivement les variantes les plus usuelles, en particulier celles qui sont utilisées dans les chapitres 4 et 8.

Démonstration par récurrence simple

Ce type de démonstration est aussi appelé démonstration par récurrence *faible*. On considère une propriété $P(n)$, dépendant de l'entier $n \in \mathbb{N}$ que l'on veut démontrer.

Si l'on peut démontrer les deux propriétés suivantes :

Base $P(n_0)$ est vraie pour un certain $n_0 \in \mathbb{N}$

Récurrence $\forall n \cdot ((n \geqslant n_0 \text{ et } P(n)) \Rightarrow P(n+1))$

alors :

Conclusion $\forall n \cdot (n \geqslant n_0 \Rightarrow P(n))$

On appelle *hypothèse de récurrence* le fait de supposer vraie $P(n)$, pour (essayer de) démontrer $P(n+1)$. La base s'appelle aussi l'initialisation, et l'étape de récurrence est aussi appelée *induction*.

Démonstration. La validité de la démonstration par récurrence s'établit grâce à un raisonnement par l'absurde. Soit $X = \{k \in \mathbb{N} \mid k \geqslant n_0 \text{ et } P(k) = \text{faux}\}$, l'ensemble des entiers supérieurs à k_0 tels que la propriété $P(k)$ est fausse. Si X est non vide, il admet un plus petit élément (puisque dans tout ensemble de nombres entiers il y a un élément inférieur à

1. Cette démonstration est due à Euclide, environ 250 av. J.-C.

tous les autres), que nous notons m. D'après la base, on sait que $m > n_0$. Donc, $m-1 \geqslant n_0$ et, comme m est le plus petit élément de X, $m-1 \notin X$ et $P(m-1)$ est vraie. En utilisant l'étape de récurrence, on déduit que $P(m)$ est vraie, ce qui est contradictoire avec $m \in X$. Par conséquent, X est vide.

Exemple Démontrons par récurrence la formule :

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Base Cette formule est vraie pour $n_0 = 1$. En effet, on a :

$$\sum_{i=1}^{1} i = 1$$

et pour $n_0 = 1$ la formule donne :

$$\frac{1(1+1)}{2} = 1$$

donc le même résultat.

Récurrence Supposons la formule vraie pour n, avec $n \geqslant 1$, et essayons de montrer qu'elle est vraie pour $(n+1)$. Autrement dit, essayons de prouver que l'hypothèse de récurrence

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

implique

$$\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}.$$

C'est en effet exact, puisque :

$$\begin{aligned}
& \sum_{i=1}^{n+1} i \\
= \quad & && \text{arithmétique} \\
& \left(\sum_{i=1}^{n} i\right) + (n+1) \\
= \quad & && \text{hypothèse de récurrence} \\
& \frac{n(n+1)}{2} + (n+1) \\
= \quad & && \text{arithmétique} \\
& \frac{(n+1)(n+2)}{2}.
\end{aligned}$$

Conclusion La formule est vraie pour $n_0 = 1$. De plus, si elle est vraie pour n (avec $n \geqslant 1$), alors elle est vraie pour $(n+1)$. Par conséquent, cette démonstration par récurrence a prouvé que la formule est vraie pour tout entier strictement positif.

Démonstration par récurrence partielle

$P(n)$ désigne une propriété dépendant de l'entier $n \in \mathbb{N}$.

Si l'on peut démontrer les deux propriétés suivantes :

Base $P(1)$ est vraie

Récurrence $\forall n \cdot ((k \in \mathbb{N} \text{ et } n = 2^k \text{ et } P(n)) \Rightarrow P(2n))$

alors :

Conclusion $\forall n \cdot ((k \in \mathbb{N} \text{ et } n = 2^k) \Rightarrow P(2n))$

Cette variante concerne une proposition sur les entiers qui sont des puissances de 2. On peut aussi démontrer une propriété sur les nombres pairs, ou d'une manière générale sur tout sous-ensemble infini de $\mathbb{N}$ constructible par induction (voir la section 1.3, page 15).

Démonstration par récurrence forte

La démonstration par récurrence *forte* (ou *totale* ou *généralisée*) paraît d'abord plus difficile à appliquer que la récurrence classique. Elle est cependant commode et (à juste titre) très employée. Ici encore, on considère une propriété $P(n)$, dépendant de l'entier $n \in \mathbb{N}$.

Si l'on peut démontrer la propriété suivante :

Récurrence $\forall n \cdot ((n \geqslant n_0 \text{ et } \forall m \cdot (m \in n_0 \mathinner{..} n-1 \Rightarrow P(m))) \Rightarrow P(n))$

alors :

Conclusion $\forall n \cdot (n \geqslant n_0 \Rightarrow P(n))$

La validité de ce schéma de démonstration dérive directement de celle de la récurrence simple.

Exemple Démontrons par récurrence forte la propriété :

> Tout nombre entier supérieur ou égal à 2 est le produit de plusieurs nombres premiers.

On admet que 1 (appelé aussi *l'unité*) est un nombre premier.

Récurrence Supposons que tout entier m inférieur à n et supérieur ou égal à $n_0 = 2$, soit le produit de plusieurs nombres premiers. Pour n, on distingue deux cas exclusifs :

- n est premier, et il est alors le produit de lui-même et de l'unité, donc de deux nombres premiers.
- n admet un diviseur d, autre que 1 et lui-même. Par hypothèse de récurrence, on a donc $n = d \cdot p$, où d et p sont deux entiers inférieurs ou égaux à n et supérieurs ou égaux à 2. Chacun d'eux est donc le produit de plusieurs nombres premiers, et n est donc lui-même le produit de plusieurs nombres premiers.

Conclusion Tout entier supérieur ou égal à 2 est donc le produit de plusieurs nombres premiers.

1.1.4 Induction de partition

Une forme importante de raisonnement par récurrence est l'induction de partition (aussi appelée induction DpR compte tenu de son importance dans la technique de conception d'algorithmes appelée « Diviser pour Régner » que nous étudierons au chapitre 8).

On considère un prédicat $P(i,s)$ dépendant des entiers naturels i et s tels que $i \leqslant s$. L'induction de partition consiste à prouver la validité de P sur tout intervalle $I = i\,..\,s$: i) en l'établissant en tout point de I, et ii) en montrant que la validité de P sur chacun des intervalles $I_1 = i\,..\,m$ et $I_2 = m+1\,..\,s$ (partition de I) entraîne celle de P sur I.

Si l'on peut démontrer les deux propriétés suivantes :

Base $\forall i \cdot (i \in \mathbb{N} \Rightarrow P(i,i))$

Induction $\forall (i,m,s) \cdot ((i \in \mathbb{N}$ **et** $s \in \mathbb{N}$ **et** $m \in \mathbb{N}$ **et** $m \in i..s-1$ **et** $P(i,m)$ **et** $P(m+1,s)) \Rightarrow P(i,s))$

alors :

Conclusion $\forall (i,s) \cdot ((i \in \mathbb{N}$ **et** $s \in \mathbb{N}$ **et** $i \leqslant s) \Rightarrow P(i,s))$

On prouve maintenant la validité de ce schéma de démonstration.

Démonstration. On procède à une preuve par l'absurde. Soit F l'ensemble des couples qui ne satisfont pas le prédicat P, soit :

$$F = \{(i,s) \mid (i,s) \in \mathbb{N} \times \mathbb{N} \text{ et } i \leqslant s \text{ et non } P(i,s)\}.$$

Dans l'hypothèse où $F \neq \varnothing$, soit (j,k) l'un des couples de F qui minimise la valeur $l = (k-j+1)$. D'après la base, $l \neq 1$ et il existe au moins une valeur m telle que $j \leqslant m < k$. Les couples (j,m) et $(m+1,k)$ n'appartiennent pas à F puisque $(m-j+1) < l$ et $(k-(m+1)+1) < l$). Il en résulte que les prédicats $P(j,m)$ et $P(m+1,k)$ sont tout deux satisfaits. D'après l'induction, on en déduit que $P(j,k)$ est satisfait, ce qui contredit le fait que $(j,k) \in F$. Donc $F = \varnothing$ et $\forall (i,s) \cdot (i \in \mathbb{N}$ **et** $s \in \mathbb{N}$ **et** $i \leqslant s \Rightarrow P(i,s))$.

Remarque 1 L'induction de partition doit impérativement se fonder sur une base $P(i,s)$ pour laquelle $s-i+1 = 1$. En particulier, elle ne peut s'appuyer sur une base où $s-i+1 = 0$. En effet, dans ce cas, il n'existe pas de valeur m telle que $i \leqslant m < s$. De même, l'induction ne peut se fonder sur une base $P(i,s)$ telle que $s-i+1 > 1$. Prenons l'exemple de $P(i,i+1)$ pour base, et tentons de démontrer $P(i,i+2)$. Les deux valeurs possibles pour m sont $m = i$ et $m = i+1$. Si $m = i$, la première hypothèse d'induction est $P(i,i)$ et si $m = i+1$, la seconde hypothèse d'induction est $P(i+2,i+2)$. Aucun de ces prédicats n'étant une instance de la base, le schéma associé est invalide.

Remarque 2 L'induction simple classique est un cas particulier de l'induction de partition obtenu en prenant $m = i$.

Exemple On va à nouveau montrer que :

$$\sum_{k=1}^{n} k = \frac{n \cdot (n+1)}{2}$$

mais cette fois en utilisant le principe de l'induction de partition. Posons $S(i,s) = \sum_{k=i}^{s} k$ et appelons $P(i,s)$ le prédicat $S(i,s) = (s^2 - i^2 + s + i)/2$. Si nous parvenons à démontrer $P(i,s)$, on aura en particulier $P(1,n)$ et on pourra en conclure que $S(1,n) = (n^2+n)/2 = \sum_{k=1}^{n} k$.

Base Pour tout i, on a : $S(i,i) = (i^2 - i^2 + i + i)/2 = i = \sum_{k=i}^{i} k$.

Hypothèses d'induction Prenons $m = \lfloor (i+s)/2 \rfloor$. On a :

— $S(i,m) = \dfrac{m^2 - i^2 + m + i}{2}$,

— $S(m+1,s) = \dfrac{s^2 - (m+1)^2 + m + 1 + i}{2}$.

Induction proprement dite Montrons que, moyennant les deux hypothèses ci-dessus, $S(i,s) = (s^2 - i^2 + s + i)/2$ pour $i < s$:

$$
\begin{aligned}
& S(i,s) \\
= & && \text{définition} \\
& \sum_{k=i}^{s} k \\
= & && \text{arithmétique } (i < s), \text{ avec } m = \lfloor (i+s)/2 \rfloor \\
& \sum_{k=i}^{m} k + \sum_{k=m+1}^{s} k \\
= & && \text{définition} \\
& S(i,m) + S(m+1,s) \\
= & && \text{hypothèses d'induction} \\
& \frac{m^2 - i^2 + m + i}{2} + \frac{s^2 - (m+1)^2 + s + m + 1}{2} \\
= & && \text{arithmétique} \\
& \frac{s^2 - i^2 + s + i}{2}
\end{aligned}
$$

Conclusion $$\sum_{k=i}^{s} k = \frac{s^2 - i^2 + s + i}{2}.$$

Remarque 1 L'un des artifices utilisés dans l'exemple ci-dessus consiste à remplacer une constante (1 dans la borne inférieure de la quantification) par une variable (i). Cette technique sera rencontrée dans la construction des itérations, dans le cadre du « renforcement par introduction de variables » (voir section 3.3.3, page 68) consistant à démontrer *plus que nécessaire* afin (en général) de faciliter la démonstration.

Remarque 2 Lors de l'utilisation de l'induction de partition dans la conception d'algorithmes « Diviser pour Régner », la principale difficulté consiste le plus souvent à découvrir les hypothèses d'induction et à montrer comment elles se composent pour former la solution définitive, ce qui sera alors appelé « rassemblement ».

1.1.5 Fondement de la démonstration par induction

Dans les paragraphes précédents, nous avons vu quelques cas de démonstration par récurrence, tous fondés sur l'ordre naturel des nombres entiers. Il est intéressant et très

utile de généraliser ce type de démonstration à des propriétés non plus caractérisées par un nombre entier, mais par un élément d'un ensemble totalement ordonné.

La démonstration par récurrence (plus souvent appelée dans ce cas *par induction*) provient de la propriété suivante.

Propriété 1 :
Soit E *un ensemble totalement ordonné par la relation* $\preceq$ *et* P *une propriété dépendant d'un élément* x *de* E. *Si la propriété suivante est vraie :*

$$\forall x \cdot (x \in E \text{ et } \forall y \cdot ((y \in E \text{ et } y \preceq x \text{ et } P(y)) \Rightarrow P(x))$$

alors $\forall x \cdot (x \in E \Rightarrow P(x))$.

La démonstration de cette propriété est une simple transposition de celle donnée à la section 1.1.3 pour la récurrence simple.

1.1.6 Démonstration par récurrence à plusieurs indices

La propriété précédente permet en particulier de justifier une démonstration par récurrence (induction) de propriétés dépendant de plusieurs indices entiers. Dans le cas de deux indices, on considère une propriété (formule) $P(m,n)$ dépendant de deux entiers m et n, que l'on veut démontrer par récurrence. On peut utiliser divers schémas, dont le suivant :

Si l'on peut démontrer les deux propriétés suivantes :

Base $\forall i \cdot (i \in \mathbb{N} \Rightarrow P(i,0))$ **et** $\forall j \cdot (j \in \mathbb{N} \Rightarrow P(0,j))$

Récurrence $\forall (i,j) \cdot ((i \in \mathbb{N}_1 \text{ et } j \in \mathbb{N}_1 \text{ et } P(i-1,j) \text{ et } P(i,j-1)) \Rightarrow P(i,j))$

alors :

Conclusion $\forall (m,n) \cdot ((m \in \mathbb{N} \text{ et } n \in \mathbb{N}) \Rightarrow P(m,n))$

La validité de ce schéma provient du fait que l'ensemble $\mathbb{N}^2$ des couples d'entiers est muni d'un ordre total (induit par l'ordre naturel sur $\mathbb{N}$), défini de la manière suivante :

$$(a,b) \preceq (c,d) \mathrel{\widehat{=}} a < c \text{ ou } (a = c \text{ et } b \leqslant d).$$

Considérons le point q de coordonnées (i,j). Le schéma proposé ci-dessus vise à établir que si P est satisfaite en tout point : i) du rectangle $0\,..\,i-1, 0\,..\,j-1$ (les cas $i = 0$ ou $j = 0$ sont couverts par la base), ii) de $(i,0)$ à $(i,j-1)$ de la ligne i, et iii) $(0,j)$ à $(i-1,j)$ de la colonne j, alors P est également satisfaite au point (i,j). En d'autres termes, si P est vérifiée en tout point « inférieur » (au sens de l'ordre $\preceq$) au point (i,j), $P(i,j)$ est également vraie, ce qui correspond à l'instance de la proposition 1 pour le couple $(\mathbb{N}^2, \preceq)$.

Ce schéma de démonstration est utilisé dans l'exercice 21, page 48. Il peut être transposé à $\mathbb{N}_1$ si besoin avec la base portant sur $\mathbb{N}_1$, la récurrence sur $\mathbb{N}_1 - \{1\}$ et la conclusion étant alors :

$$\forall (m,n) \cdot ((m \in \mathbb{N}_1 \text{ et } n \in \mathbb{N}_1) \Rightarrow P(m,n)).$$

D'autres schémas sont applicables pour démontrer une récurrence à deux indices (ou plus). Leur validité est assurée pour autant qu'ils se conforment à la propriété 1 pour un certain ordre total (voir exercice 8, page 40).

1.2 Relations de récurrence

Après avoir introduit la notion de relation de récurrence et en avoir donné plusieurs exemples, nous abordons la question de l'établissement d'une relation de récurrence et du calcul algorithmique associé.

1.2.1 Généralités, exemples et formes closes

Par définition, une suite $S(n)$, avec $n \in \mathbb{N}$, est *définie par une relation de récurrence* si l'on a, pour n assez grand, une relation du type : $S(n) = f(S(n-1), \ldots, S(n-p))$. De plus, il faut connaître des valeurs permettant d'initialiser le calcul de S. Nous ne considérons ici que les suites dont les valeurs sont des nombres entiers positifs ou nuls (notamment parce que ces suites sont associées à des calculs de complexité).

Dans la relation de récurrence définissant la suite de Fibonacci :

$$\begin{aligned} &\mathcal{F}(1) = 1 \\ &\mathcal{F}(2) = 1 \\ &\mathcal{F}(n) = \mathcal{F}(n-1) + \mathcal{F}(n-2) \qquad n > 2 \end{aligned}$$

ou la suite des nombres de Padovan :

$$\begin{aligned} &\mathcal{P}(1) = 1 \\ &\mathcal{P}(2) = 1 \\ &\mathcal{P}(3) = 1 \\ &\mathcal{P}(n) = \mathcal{P}(n-2) + \mathcal{P}(n-3) \qquad n > 3 \end{aligned}$$

la fonction f est linéaire. En revanche, celle qui apparaît dans la relation définissant les factorielles (nombre de permutations d'un ensemble à n éléments) ne l'est pas :

$$\begin{aligned} &\mathrm{Fact}(1) = 1 \\ &\mathrm{Fact}(n) = n \cdot \mathrm{Fact}(n-1) \qquad n > 1. \end{aligned}$$

Les mathématiques fournissent, grâce à la théorie des *fonctions génératrices* (voir par exemple [31]), de puissants moyens pour trouver des formes *closes* (c'est-à-dire non récurrentes) à certaines suites définies par des relations de récurrence, en particulier si f est linéaire, mais dans bien d'autres cas aussi. Ainsi, la suite de Fibonacci s'exprime sous la forme close :

$$\mathcal{F}(n) = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right] \qquad n \geqslant 1.$$

La suite des nombres de Catalan, dont nous reparlerons dans les exercices 5, page 37, et 13, page 42, est définie par la récurrence :

$$\begin{aligned} &\mathrm{Cat}(1) = 1 \\ &\mathrm{Cat}(n) = \sum_{i=1}^{n-1} \mathrm{Cat}(i) \cdot \mathrm{Cat}(n-i) \qquad n > 1 \end{aligned}$$

et aussi par :

$$\begin{aligned} &Cat(1) = 1 \\ &Cat(n) = \frac{4n-6}{n} \cdot Cat(n-1) && n > 1. \end{aligned}$$

Elle s'exprime sous la forme close[2] :

$$Cat(n) = \frac{(2n-2)!}{(n-1)! \cdot n!} = \frac{1}{n} \cdot C_{2n-2}^{n-1} \qquad n \geqslant 1.$$

D'un point de vue algorithmique, la forme close apparaît attractive puisque simple à calculer. Cependant, si ce choix est raisonnable pour le calcul du n^e nombre de Catalan, il peut ne pas être approprié, comme le montre l'exercice 12, page 41, à propos du calcul des nombres de Fibonacci.

Une relation de récurrence peut servir de définition à une suite définie sur plus d'un indice entier. Par exemple, les nombres d'Ackerman sont définis par la récurrence imbriquée à deux indices :

$$\begin{aligned} &A(0,n) = n+1 && n \geqslant 0 \\ &A(m,0) = A(m-1,1) && m > 0 \\ &A(m,n) = A(m-1,A(m,n-1)) && m > 0 \text{ et } n > 0. \end{aligned}$$

Un autre exemple de relation de récurrence à deux indices est celui des nombres de Stirling, qui se définissent comme le nombre de partitions à p blocs d'un ensemble ayant n éléments. Ils se construisent par une récurrence beaucoup plus simple que la précédente, et ils sont étudiés à l'exercice 16, page 44.

1.2.2 Établissement et calcul d'une relation de récurrence

Principes

Nous nous intéressons maintenant à l'établissement et au calcul de suites définies par une (relation de) récurrence. Il s'avère que de nombreux problèmes, liés au dénombrement par exemple, mais aussi au calcul de valeur optimale d'une fonction (voir chapitre 9), se résolvent en mettant en évidence une récurrence permettant le calcul de la grandeur voulue. Il importe que la récurrence soit complète, c'est-à-dire que toutes les situations pouvant survenir soient prises en compte.

D'un point de vue algorithmique, comme il est fréquent que l'on ne connaisse pas de forme close, on procède au calcul en construisant un programme itératif dans lequel un tableau à une ou deux dimensions (voire plus) permet de stocker la succession des valeurs de la suite calculées. Cette approche sera discutée et justifiée au chapitre 4.

Le principe retenu consiste à calculer la valeur d'une cellule à partir de celles déjà présentes dans le tableau. L'ordre de remplissage est donc primordial, et la progression du calcul dépend des relations de dépendance entre éléments. Si pour les récurrences à une seule dimension l'affaire est généralement simple (progression selon l'ordre croissant ou décroissant des valeurs l'indice), une récurrence à plusieurs indices mérite plus d'attention. Les exemples qui suivent illustrent la démarche proposée.

2. On admet que les formes closes peuvent contenir tout type de quantificateur, ici le produit sous-tendant le calcul des factorielles.

Quelques exemples

Récurrence à un indice : arbres binaires à n nœuds On veut calculer le nombre $nbab(n)$ d'arbres binaires (voir section 1.6, page 30) ayant n nœuds (feuilles comprises). On va d'abord établir la récurrence définissant ce nombre. On remarque qu'il y a un seul arbre binaire n'ayant aucun élément : l'arbre vide. Pour établir une récurrence sur $nbab(n)$, on considère la décomposition d'un arbre binaire à n nœuds en arbres plus petits. Pour n positif, un arbre à n nœuds est constitué d'une racine (un nœud), éventuellement d'un sous-arbre gauche ayant i nœuds, et éventuellement d'un sous-arbre droit ayant $(n-i-1)$ nœuds. Examinons le nombre de façons possibles de faire la décomposition :

- on peut placer zéro nœud à gauche (sous-arbre vide), ce qui fait $nbab(0)(=1)$ façon, et $(n-1)$ nœuds dans le sous-arbre droit, avec par définition $nbab(n-1)$ possibilités, ce qui fait au total $nbab(0)\cdot nbab(n-1)$ cas,
- on peut placer un nœud à gauche, et $(n-2)$ nœuds à droite, ce qui fait $nbab(1)\cdot nbab(n-2)$ cas,
- ...

Plus généralement, on place i nœuds à gauche, d'où $nbab(i)$ sous-arbres gauches, et $(n-i-1)$ nœuds à droite, d'où $nbab(n-i-1)$ sous-arbres droits, ce qui fait au total $nbab(i)\cdot nbab(n-i-1)$ arbres de cette sorte. Cela vaut pour toutes les valeurs de i de 0 à $(n-1)$. Finalement, le nombre d'arbres ayant n nœuds est :

$$\begin{aligned} nbab(n) &= nbab(0)\cdot nbab(n-1) + nbab(1)\cdot nbab(n-2) + \cdots \\ &\quad \cdots + nbab(i)\cdot nbab(n-i-1) + \cdots \\ &\quad \cdots + nbab(n-1)\cdot nbab(0) \end{aligned}$$

et on en déduit la récurrence :

$$\begin{aligned} &nbab(0) = 1 \\ &nbab(n) = \sum_{i=0}^{n-1} nbab(i)\cdot nbab(n-i-1) \qquad n \geqslant 1. \end{aligned}$$

Après avoir vérifié par énumération que $nbab(1) = 1$ et que $nbab(2) = 2$, la valeur suivante est :

$$\begin{aligned} nbab(3) &= \sum_{i=0}^{2} nbab(i)\cdot nbab(n-i-1) \\ &= nbab(0)\cdot nbab(2) + nbab(1)\cdot nbab(1) + nbab(2)\cdot nbab(0) \\ &= 2+1+2 = 5. \end{aligned}$$

La figure 1.1, page 13, met en évidence les arbres binaires pour $n \in 0\,..\,3$.

En l'absence de connaissances complémentaires, on effectue le calcul de $nbab(n)$ en remplissant un tableau $NBAB[0\,..\,n]$. Vu que $nbab(n)$ dépend de valeurs d'indice inférieur, le calcul progresse par valeurs croissantes de l'indice après initialisation de la cellule d'indice 0. Le programme qui en résulte est le suivant :

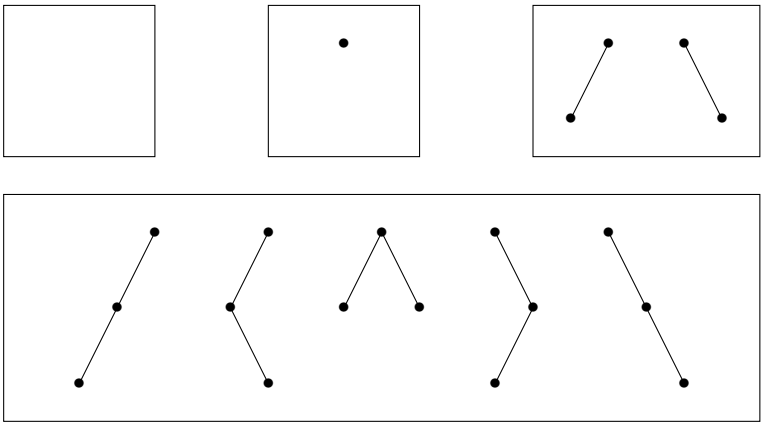


Fig. 1.1 – *Visualisation de tous les arbres binaires de taille* 0, 1, 2 *et* 3, *en nombres respectifs : 1, 1, 2, 5*

```
1.  constantes
2.     n ∈ ℕ₁ et n = ...
3.  variables
4.     NBAB ∈ 0..n → ℕ₁
5.  début
6.     NBAB[0] ← 1 ;
7.     pour i parcourant 1..n faire
8.        NBAB[i] ← 0 ;
9.        pour j parcourant 0..i − 1 faire
10.          NBAB[i] ← NBAB[i] + NBAB[j] · NBAB[i − j − 1]
11.       fin pour
12.    fin pour ;
13.    écrire(le nombre d'arbres binaires ayant , n, nœuds est , NBAB[n])
14. fin
```

On verra dans l'exercice 13, page 42, une alternative à ce calcul.

Récurrence à un indice : factorielle On considère maintenant un cas où la récurrence (à une dimension) est fournie d'emblée. On cherche à calculer la valeur de $n!$. On remarque que seule la valeur $(n-1)!$ est nécessaire au calcul de $n!$; il est ici inutile de mémoriser les valeurs successives de factorielle dans un tableau, ce qui « allège » d'autant le programme.

On va donc procéder de façon itérative, mais sans stocker les valeurs calculées, d'où le programme :

```
1.  constantes
2.     n ∈ ℕ₁ et n = ...
3.  variables
4.     Facto ∈ ℕ₁
5.  début
```

```
6.   Facto ← 1 ;
7.   pour i parcourant 2 .. n faire
8.     Facto ← Facto · i
9.   fin pour ;
10.  écrire(pour n = , n, la valeur de n ! est , Facto)
11. fin
```

Récurrence à deux indices : nombres de Delannoy On met maintenant l'accent sur la progression du calcul de la récurrence à deux indices définissant les nombres de Delannoy. Soit la récurrence :

$$\begin{array}{ll} nbdel(0,j) = 1 & 0 \leqslant j \leqslant m \\ nbdel(i,0) = 1 & 1 \leqslant i \leqslant n \\ nbdel(i,j) = \left(\begin{array}{l} nbdel(i,j-1) + \\ nbdel(i-1,j) + \\ nbdel(i-1,j-1) \end{array} \right) & \left\{ \begin{array}{c} 1 \leqslant i \leqslant n \\ \text{et} \\ 1 \leqslant j \leqslant m \end{array} \right. . \end{array}$$

Pour calculer $nbdel(n,m)$ avec m et n deux entiers donnés, on associe à $nbdel$ un tableau $NBD[0\,..\,n, 0\,..\,m]$. On observe que, dans le cas général, pour calculer l'élément $NBD[i,j]$, il faut connaître : i) la valeur de la cellule de même indice de ligne et d'indice de colonne immédiatement inférieur, ii) la valeur de la cellule de même indice de colonne et d'indice de ligne immédiatement inférieur, et iii) la valeur de la cellule d'indices ligne et colonne immédiatement inférieurs. On peut donc remplir NBD par ligne (mais aussi par colonne ou par diagonale). Après avoir initialisé la ligne 0 à 1 (premier terme de la récurrence), on remplit les lignes par valeurs croissantes de l'indice i (jusqu'à n) : la cellule $(i,0)$ est mise à 1 (second terme de la récurrence), puis on remplit les cellules $(i,1)$ à (i,m) en utilisant le dernier terme de la récurrence.

Construction d'un algorithme de calcul d'une récurrence

De façon générale, étant donnée une formule de récurrence (complète) dont on ne connaît pas de forme close, le programme qui permet le calcul s'élabore ainsi.

1. Définition de la structure tabulaire associée au calcul (dimensions et taille de chacune d'elles).
2. Détermination des relations de dépendance existant entre l'élément en partie gauche et ceux apparaissant en partie droite dans le terme général de la récurrence. On en déduit une progression du calcul qui garantit, qu'à l'aide de l'initialisation de la récurrence, le calcul de tout élément de la structure tabulaire ne fait appel qu'à des éléments déjà calculés.
3. Le cas échéant, améliorations telles que la limitation du calcul aux seuls éléments nécessaires.
4. Production du code associé.

Comme on l'a vu pour l'exemple du calcul de factorielle (voir aussi l'exercice 12, page 41), la structure tabulaire peut être remplacée par quelques variables scalaires. Il convient d'être vigilant sur la représentation des nombres, surtout en présence d'une récurrence dont les valeurs croissent extrêmement rapidement. Il peut être prudent de travailler en représentation flottante plutôt qu'avec des entiers. Par exemple, les nombres d'Ackerman

sont tels que $A(3,n) = 2^{n+3} - 3$ et $A(4,n) = 2^{2^{\cdot^{\cdot^{2}}}} - 3$ ($n+3$ occurrences de 2), et ce nombre devient très vite gigantesque ($A(4,2) = 2^{2^{2^2}} - 3 = 2^{16} - 3 = 2^{65536} - 3$). Enfin, la relation de dépendance liant partie gauche et partie droite peut être non triviale. Nous renvoyons ici aussi le lecteur aux nombres d'Ackerman avec le calcul de $A(6,1) = A(5,65533)$.

1.3 Récurrence, induction, récursivité, etc.

Nous avons vu aux sections précédentes ce que sont une démonstration par récurrence (aussi appelée démonstration par induction) et une suite définie par une relation de récurrence. Dans cette section, nous allons essayer d'éclaircir une terminologie souvent perçue de manière un peu confuse.

Démonstration par récurrence ou par induction

Nous l'avons dit, ces termes sont équivalents en ce qui concerne la démonstration. On parle le plus souvent de démonstration *par récurrence* quand l'ensemble ordonné est $\mathbb{N}$. Cependant, lors de la construction d'algorithmes fondés sur ce type de raisonnement, nous utiliserons fréquemment le terme générique *raisonnement inductif*, dont le cœur sera une *induction*. En cas de raisonnement sur les entiers, nous préciserons s'il s'agit d'un raisonnement par récurrences simple ou forte, qui constituent les deux variantes auxquelles il sera fait appel le plus souvent dans les chapitres 4 et 8.

Programme récursif

En informatique, un algorithme (comme un programme qui le code) est dit récursif s'il s'appelle lui-même directement ou par l'intermédiaire d'autres programmes. La fonction *Fact* de la page 16 illustre la notion d'algorithme récursif, de même que de nombreux programmes des chapitres 4 et 8.

La mise en œuvre « canonique » d'une récurrence est un algorithme récursif. Cependant, comme on le verra au chapitre 4, on préférera très souvent, pour des raisons d'efficacité, une version itérative remplissant une structure tabulaire (comme cela a été le cas dans les exemples précédents). Ceci ne doit pas être confondu avec le fait qu'un programme récursif peut être transformé (de façon automatique avec gestion explicite d'une pile) en un programme itératif, qui lui, aura le même comportement (en particulier les mêmes performances) que le programme récursif initial. Il existe malgré tout un type d'algorithme récursif pouvant être transformé avec intérêt en un algorithme itératif : lorsque la récursivité est dite *terminale*. Nous allons développer cette notion dans le cadre des fonctions, mais ce qui est dit se transpose aux procédures. Une fonction est (directement) récursive terminale si elle obéit au schéma suivant :

```
1. fonction F(...) résultat ... pré
2.    ...
3. début
4.    si C alors
5.       résultat V
6.    sinon
7.       résultat F(...)
8.    fin si
9. fin
```

Il est alors possible d'obtenir une programme itératif (dont l'exécution est plus efficace), ce qu'un compilateur évolué sait faire.

Exemple La fonction :

```
1. fonction Fact(p) résultat ℕ1 pré
2.    p ∈ ℕ1
3. début
4.    si p = 1 alors
5.       résultat 1
6.    sinon
7.       résultat p · Fact(p − 1)
8.    fin si
9. fin
```

n'est pas récursive terminale. Elle peut être mise sous cette forme de la façon suivante :

```
1. fonction Fact1(p) résultat ℕ1 pré
2.    p ∈ ℕ1
3. début
4.    résultat FactRecTerm(p, 1)
5. fin
6. fonction FactRecTerm(p, q) résultat ℕ1 pré
7.    p ∈ ℕ1 et q ∈ ℕ1
8. début
9.    si p = 1 alors
10.      résultat q
11.   sinon
12.      résultat FactRecTerm(p − 1, q · p)
13.   fin si
14. fin
```

On note qu'ici la fonction *Fact1* n'est pas récursive et que la fonction *FactRecTerm* est récursive terminale.

1.4 Ensembles

1.4.1 Notations de base

Un couple est constitué de deux expressions séparées par une virgule ou par le symbole $\mapsto$. La formule $x, y+3$ est un couple, qui peut également se noter $x \mapsto y+3$, ou encore $(x, y+3)$ (toute expression peut être parenthésée). Dans cette notation, le premier élément est appelé l'origine et le second l'extrémité.

$C \mathrel{\widehat{=}} (x, y+3)$ attribue le nom C à l'objet $(x, y+3)$.

Les opérateurs relationnels $=$ et $\neq$, avec leur sens habituel, sont supposés toujours disponibles pour comparer deux objets mathématiques.

1.4.2 Définition d'ensembles

Soit E un ensemble, $x \in E$ est une expression booléenne qui établit que x est un élément de l'ensemble E. Sa négation se note $\notin$.

L'ensemble qui ne contient aucun élément (l'ensemble vide) se note $\varnothing$.

Soit E et F deux ensembles. L'expression $E \times F$, produit cartésien de E et de F, se définit par l'équivalence $(a, b) \in E \times F \equiv (a \in E \textbf{ et } b \in F)$.

Un ensemble peut se définir par les propriétés de ses éléments. C'est la définition *en compréhension* ou en *intension*. Elle se note $\{x \mid x \in E \textbf{ et } P\}$, où E est un ensemble et P un prédicat de sélection, et définit l'ensemble des éléments de E possédant la propriété P. Ainsi

$$\{x \mid x \in \mathbb{N} \textbf{ et } \nexists \cdot (y \in \mathbb{N} \textbf{ et } x = 2 \cdot y)\}$$

définit l'ensemble des entiers impairs. La propriété suivante du produit cartésien :

$$E \times F = \{(x, y) \mid x \in E \textbf{ et } y \in F\}$$

découle de la définition en compréhension des ensembles. L'avantage de la seconde notation est qu'elle permet, *via* une *convention*, de désigner aisément un champ particulier d'un couple. En effet, supposons que l'on déclare le point p du plan par $p \in \{(x, y) \mid x \in \mathbb{R} \textbf{ et } y \in \mathbb{R}\}$. On pourra alors désigner l'abscisse de p par p.x (ou l'ordonnée par p.y). Cet avantage est également exploité dans les structures inductives, à la section 1.4.7, page 21. La notation « $\times$ » n'autorise pas cette facilité. Cette dualité sera exploitée par la suite.

Un ensemble peut également se définir par l'énumération de ses éléments, c'est la définition en *extension*. La notation utilisée est $\{1, 3, \textsf{vrai}, 1.4\}$. Ce dernier ensemble peut également s'écrire $\{1, \textsf{vrai}, 1.4, 3\}$ (l'ordre n'est pas pertinent), ou $\{1, 1, 3, \textsf{vrai}, 1.4, \textsf{vrai}, 3\}$ (un élément est présent ou ne l'est pas, le nombre d'occurrences n'est pas pertinent dès lors que l'élément est présent). Un ensemble peut également être défini de façon inductive. Un ensemble X défini de façon inductive (on dit aussi de manière plus ambiguë « défini récursivement ») est un cas particulier d'ensemble récursif. La fonction qui sert à le définir est de nature inductive (ou récurrente), c'est-à-dire que :

- certains éléments de X sont donnés explicitement,
- les autres éléments sont définis en fonction d'éléments appartenant déjà à X.

Par exemple, l'ensemble P des nombres pairs peut se définir comme *le plus petit* sous-ensemble de $\mathbb{N}$ satisfaisant l'équation :

$$P = \{0\} \cup \{n \mid n \in \mathbb{N} \textbf{ et } n-2 \in P \Rightarrow n \in P\}.$$

Nous verrons dans la suite de cet ouvrage plusieurs exemples d'ensembles définis de façon inductive, en particulier les arbres. Très souvent, les propriétés de tels ensembles sont prouvées par un raisonnement par induction.

Si E est un ensemble, $\mathbb{P}(E)$ représente l'ensemble des parties de E et se définit par

$$x \in \mathbb{P}(E) \Leftrightarrow \forall y \cdot (y \in x \Rightarrow y \in E).$$

Ainsi, si $E \mathrel{\widehat{=}} \{1,2,3\}$, $\mathbb{P}(E) = \{\varnothing, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$.

1.4.3 Opérations sur les ensembles

On suppose connues les définitions des opérateurs ensemblistes suivants : $\cap$ (intersection), $\cup$ (union), $\subseteq$ (inclusion), $\subset$ (inclusion stricte), $\not\subset$ (non inclusion), $\not\subseteq$ (non inclusion stricte), $-$ (différence).

Pour un ensemble fini E, $\mathrm{card}(E)$ désigne le nombre d'éléments de E.

1.4.4 Ensembles particuliers

Outre l'ensemble $\mathbb{B}$ déjà défini, les ensembles numériques suivants sont supposés connus : $\mathbb{N}$ (entiers naturels), $\mathbb{Z}$ (entiers relatifs), $\mathbb{R}$ (réels numériques, c'est-à-dire « réels discrétisés » en représentation informatique), $\mathbb{C}$ (nombres complexes).

Certains sous-ensembles des ensembles ci-dessus se notent de manière particulière. Il s'agit de $\mathbb{N}_1$ (qui représente $\mathbb{N} - \{0\}$), $\mathbb{R}_+$ (qui représente les éléments de $\mathbb{R}$ positifs ou nuls), $\mathbb{R}_+^*$ (qui représente les éléments de $\mathbb{R}$ strictement positifs).

Pour ce qui concerne les complexes, i est la constante telle que $i^2 = -1$. Si c est un complexe, sa partie réelle se note $\mathrm{re}(c)$ et sa partie imaginaire $\mathrm{im}(c)$. Ainsi, pour $c \mathrel{\widehat{=}} 1-2i$, $\mathrm{re}(c) = 1$ et $\mathrm{im}(c) = -2$.

Si E est un ensemble numérique totalement ordonné, l'opérateur $\min(E)$ (resp. $\max(E)$) désigne le plus petit (resp. le plus grand) élément de E. Si E est vide $\min(E) = \infty$ et $\max(E) = -\infty$.

Les intervalles de $\mathbb{Z}$ se notent $exp_1 \mathinner{..} exp_2$. Si $exp_2 < exp_1$ l'intervalle dénote l'ensemble vide. Dans la mesure où $\mathbb{Z}$ est un ensemble ordonné, on peut assimiler un intervalle à une liste (voir section 1.4.7, page 21). Il en est de même d'une expression telle que $exp_1 \mathinner{..} exp_2 - (F)$. Cette particularité est utilisée dans les boucles **pour** avec l'option **parcourant** où, par exemple, le domaine de variation de la variable de la boucle $1 \mathinner{..} 5 - \{3,4\}$ correspond à l'ensemble $\{1,2,5\}$ et est parcouru dans cet ordre lors de l'exécution de la boucle.

1.4.5 Relations, fonctions, tableaux

Relations

Soit E et F des ensembles et $R \subseteq E \times F$. R est appelé relation (binaire) entre la source E et la destination F. Si $F = E$, le couple (E, R) est appelé *graphe* (voir section 1.5, page 22). Les fonctions sont des relations binaires particulières, et les tableaux sont des fonctions particulières. Les notions définies pour les relations s'appliquent donc naturellement à ces deux dernières notions.

Soit R une relation binaire entre E et F. La relation inverse (ou réciproque) de R se note R^{-1} et se définit par :

$$R^{-1} \mathrel{\widehat{=}} \{(b, a) \mid (b, a) \in F \times E \textbf{ et } (a, b) \in R\}.$$

Soit R une relation entre E et F. Le domaine de R se note dom(R) et se définit comme le sous-ensemble de E, dont les éléments constituent l'origine d'au moins l'un des couples de R. Plus formellement :

$$dom(R) \mathrel{\widehat{=}} \{a \mid a \in E \textbf{ et } \exists b \cdot (b \in F \textbf{ et } (a, b) \in R)\}.$$

Soit R une relation entre E et F. Le co-domaine de R se note codom(R) et se définit comme le domaine de R^{-1}. C'est aussi le sous-ensemble de F dont les éléments constituent l'extrémité d'au moins l'un des couples de R.

Soit E un ensemble, la relation identité sur E se note id(E) et se définit par :

$$id(E) \mathrel{\widehat{=}} \{(a, b) \mid (a, b) \in E \times E \textbf{ et } a = b\}.$$

Soit S une relation entre F et G, et R une relation entre E et F. La relation $R \circ S$ est appelée composition de R et de S. Elle se définit par :

$$R \circ S \mathrel{\widehat{=}} \{(a, c) \mid (a, c) \in E \times G \textbf{ et } \exists b \cdot (b \in F \textbf{ et } (a, b) \in R \textbf{ et } (b, c) \in S)\}.$$

Fonctions

Une fonction f de E dans F est une relation dans laquelle il n'existe pas deux couples (a, b) et (a, c) tels que $b \neq c$. On note $f \in E \rightarrow F$. Dans la suite, il sera souvent nécessaire de distinguer différents types de fonctions. Soit $f \in E \rightarrow F$.

1. Si f est une fonction *partielle*, il peut exister des éléments de E qui ne sont l'origine d'aucun couple de f.
2. Si f est une fonction *totale*, tout élément de E est l'origine d'un (et d'un seul) couple de f.
3. Si f est une fonction *injective*, tout élément de F est l'extrémité d'au plus un couple de f.
4. Si f est une fonction *surjective*, tout élément de F est l'extrémité d'au moins un couple de f.
5. Si f est une fonction *bijective*, f est à la fois injective et surjective.

Notons qu'une fonction totale est aussi une fonction partielle. La figure 1.2, page 20, montre les relations d'inclusion qui existent entre les différents types de fonction.

Dans la suite, un prédicat tel que « $f \in E \rightarrow F$ **et** TI(f) » doit se comprendre comme « f est une fonction totale injective de E dans F ».

Tableaux

Un tableau t est une fonction totale d'un intervalle $i \mathinner{..} s$ dans un ensemble F. On note $t \in i \mathinner{..} s \rightarrow F$. Si i appartient au domaine de définition du tableau, t[i] désigne l'élément d'indice i de t. La notation ensembliste peut toujours être employée, mais lorsque le contexte permet de déterminer les bornes, la notation $[t_1, \ldots, t_n]$ peut être utilisée en

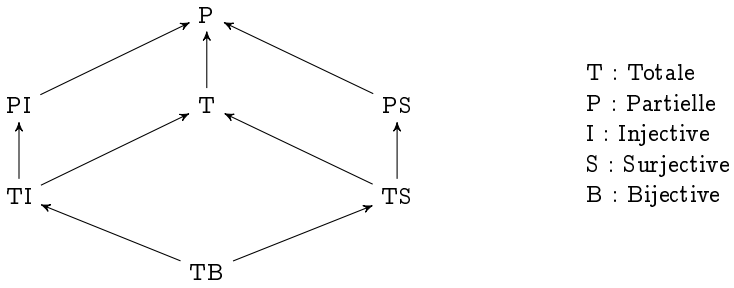


Fig. 1.2 – *Diagramme d'inclusion des différents types de fonction.* $A \to B$ *signifie que les fonctions du type* A *sont incluses au sens large dans les fonctions du type* B. *D'après [1].*

tant que constante de type tableau. Une tranche de tableau définie sur le domaine $i \,..\, s$ se note $t[i \,..\, s]$. Ainsi, si l'intervalle $3 \,..\, 5$ est inclus dans le domaine de définition de t, $t[3 \,..\, 5] \leftarrow [6, -1, 2]$ est une affectation correcte. Elle est équivalente à $t[3 \,..\, 5] \leftarrow \{3 \mapsto 6, 5 \mapsto 2, 4 \mapsto -1\}$. Par ailleurs, après l'affectation, $\text{codom}(t[3 \,..\, 5]) = \{-1, 2, 6\}$, tandis que $\text{dom}(t[3 \,..\, 5]) = \{3, 4, 5\}$.

Un tableau à deux dimensions se définit sur le produit cartésien de deux intervalles. Une constante de tableau à deux dimensions peut toujours être représentée comme un tableau de tableau (comme par exemple $[[0, 1], [1, 1]]$), mais la notation à deux dimensions, plus lisible, est en général utilisée (le même tableau : $\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$).

1.4.6 Sacs

Un sac (ou multiensemble) d'éléments de l'ensemble E est une structure mathématique dans laquelle le nombre d'occurrences d'un élément donné est significatif. Un sac S sur E peut être considéré comme une fonction s de E dans $\mathbb{N}$, telle que, pour $v \in E$, $s(v)$ est le nombre d'occurrences de v dans S. Pour faciliter l'écriture, nous abandonnons la notation ensembliste pour adopter une notation *ad hoc*. Le tableau ci-dessous répertorie les notations et leur équivalent ensembliste.

Sacs	Ensembles	Sacs	Ensembles
$\sqsubseteq$	$\in$	$\not\sqsubseteq$	$\notin$
$\not\sqsubset$	$\varnothing$	$\sqsubseteq$	$\subseteq$
$\dot{-}$	$-$	$\not\sqsubseteq$	$\not\subseteq$
$\sqcap$	$\cap$	$\lvert\ldots\rvert$	card
$\sqcup$	$\cup$	smin	min
$\sqsubset$	$\subset$	smax	max

$\mathbf{sac}(E)$ représente l'ensemble des sacs finis de l'ensemble fini E et $[\![\ldots]\!]$ est équivalent à la notation $\{\ldots\}$ pour les ensembles définis en extension. Enfin $\text{mult}(v, S)$ est la fonction qui délivre la multiplicité (le nombre d'occurrences) de v dans le sac S.

1.4.7 Produit cartésien et structures inductives

Les structures inductives, c'est-à-dire les structures définies par induction (voir page 17), jouent un rôle important comme structures de données dans l'ensemble de l'ouvrage. Les deux cas les plus typiques sont les listes finies et les arbres finis. Ces derniers sont traités à la section 1.6, page 30. Nous nous limitons ici à introduire les listes finies. Quel sens doit-on attribuer à une formule telle que :

$$\text{liste} = \{/\} \cup \{(val, svt) \mid val \in \mathbb{N} \text{ et } svt \in \text{liste}\}. \tag{1.1}$$

Informellement, l'ensemble ainsi défini est l'union entre la liste vide (notée /) et l'ensemble des couples constitués d'un entier et d'une liste d'entiers. $(3,(1,(8,/)))$ est un exemple d'une telle liste. En principe, il ne faut pas confondre une telle structure linguistique – qui est une *représentation externe* des listes – avec les listes proprement dites. Dans la pratique, nous nous autorisons cet abus de langage. Il est nécessaire d'ajouter que l'on ne s'intéresse qu'aux structures *finies* et qu'une liste ainsi définie est le *plus petit* ensemble satisfaisant l'équation 1.1. Compte tenu de la convention portant sur l'utilisation de la notation pointée (voir section 1.4.2, page 17), si l'on a l'affectation $l \leftarrow (3,(1,(8,/)))$, on aura $l.val = 3$ et $l.svt = (1,(8,/))$. Les listes ne permettent pas l'accès direct à leurs éléments, il faut toujours passer par l'un des champs (comme $l.svt.svt.val$ par exemple).

Dans la pratique, lorsque c'est possible, on s'autorise la notation plus concise $\langle \ldots \rangle$. Ainsi $(3,(1,(8,/)))$ peut également se noter $\langle 3,1,8 \rangle$. L'opération de concaténation de listes est notée $\cdot$.

1.4.8 Chaînes, séquences

Les chaînes (ou *séquences*, parfois *mots*) sont des tableaux flexibles à droite dont la borne gauche est implicitement 1. Leurs éléments sont souvent appelés *caractères*. Ce sont donc des fonctions totales pour lesquelles les opérateurs ensemblistes habituels sont disponibles (dom, codom, etc.). La longueur de la chaîne c est notée $|c|$. L'opération de concaténation (notée $\cdot$) permet d'allonger une chaîne en lui adjoignant une autre chaîne mais aussi un caractère (qui est alors implicitement converti en chaîne). Si $c = c[1] \ldots c[n]$ dénote une chaîne, $\overline{c} = c[n] \ldots c[1]$ est la chaîne miroir de c. D'un point de vue algorithmique, une chaîne c passée en paramètre d'entrée véhicule implicitement son domaine de définition ($\text{dom}(c)$). Si c est une chaîne définie sur le domaine $1..n$, la tranche $c[i..s]$ est une chaîne définie sur le domaine $1..s-i+1$ à condition que $i..s \subseteq 1..n$. La chaîne vide ε est définie sur le domaine $1..0$. La déclaration d'une constante symbolique, d'une variable ou d'un paramètre c de type **chaîne** se fait par $c \in$ **chaîne** avec comme vocabulaire implicite les lettres, chiffres et caractères spéciaux. Le recours à un vocabulaire spécifique Σ est réalisé par $c \in$ **chaîne**(Σ). L'exemple suivant illustre l'utilisation des chaînes, en particulier des constantes non délimitées par des guillemets pour lesquelles on fait usage d'une police spéciale.

1. **constantes**
2. $a = abcd$ /% constante symbolique de type **chaîne** %/
3. **variables**
4. $b \in$ **chaîne**$(\{a,b,c,d,1,2,3,4\})$ /% variable de type **chaîne** sur le vocabulaire $\Sigma = \{a,b,c,d,1,2,3,4\}$ %/

```
5.  début
6.    écrire(Impair(a)) ; /% la fonction Impair(x) (voir code) délivre une chaîne
      constituée des éléments d'indice impair de x %/
7.    lire(b) ; écrire(Impair(b)) ;
8.    b ← acbd · 1234 · a[3..4] · a[1] ; /% exemple de concaténation %/
9.    écrire(Impair(b))
10. fin
```

```
1.  fonction Impair(x) résultat chaîne pré
2.    x ∈ chaîne et y ∈ chaîne
3.  début
4.    y ← ε ; /% y devient une chaîne vide %/
5.    pour i parcourant dom(x) faire
6.      si 2 · ⌊i/2⌋ ≠ i alors
7.        y ← y · x[i] /% allongement de y par l'élément suivant d'indice
          impair de x %/
8.      fin si
9.    fin pour ;
10.   résultat y
11. fin
```

Les chaînes de caractères sont tout particulièrement utilisées dans l'exemple sur les séquences traité page 323, ainsi que dans les exercices 21 page 48, 108 page 289, 106 page 278, 136 page 365, 137 page 366, et 138 page 370.

1.5 Graphes

Intuitivement, un graphe est un ensemble de points dont certains peuvent être reliés deux à deux. C'est donc une notion plutôt simple qui permet de modéliser de nombreux problèmes présentant une grande utilité pratique, comme la recherche, par un appareil GPS, du meilleur trajet pour rejoindre son lieu de vacances.

La théorie des graphes est aussi la source de plusieurs défis mathématiques et/ou informatiques complexes comme celui du problème des quatre couleurs, dont la démonstration complète et automatique a été réalisée par l'assistant de preuve Coq.

Plus simplement, pour nous, la notion de graphe est à l'origine de nombreux exercices intéressants qui jalonnent cet ouvrage.

La théorie des graphes peut être vue comme une émanation de la théorie des ensembles (à travers la notion de relation binaire). Mais, s'étant développées de manière séparée, les deux théories utilisent des vocabulaires qui souvent divergent. Nous étudions tout d'abord les graphes orientés, avant de nous arrêter sur les graphes non orientés. Le tour d'horizon s'achève par le cas des graphes valués. Dans tous les cas, nous ne considérerons que des graphes finis.

1.5.1 Graphes orientés

Définition 1 (Graphe orienté) :
Un graphe orienté G *est un couple* (N, V)*, où* N *est un ensemble (fini) et* V *une relation binaire entre* N *et* N *(*V *est donc un sous-ensemble fini de* $N \times N$*).*

Les éléments de N sont appelés *sommets* ou *nœuds* ; ceux de V sont appelés *arcs*. Le cardinal de N est appelé *ordre* du graphe.

Exemple Le couple G défini par :

$$G = \begin{pmatrix} \{a,b,c,d,e,f\}, \\ \{(a,b),\ (a,c),\ (a,f),\ (b,a),\ (b,c),\ (b,d),\ (c,e),\ (d,d),\ (d,f),\ (e,a),\ (e,f),\ (f,c)\} \end{pmatrix}$$

est un graphe orienté.

La figure 1.3 offre quatre représentations possibles pour ce graphe. Le schéma (b), qui se présente sous la forme d'une matrice carrée, est souvent utilisé en tant que représentation informatique.

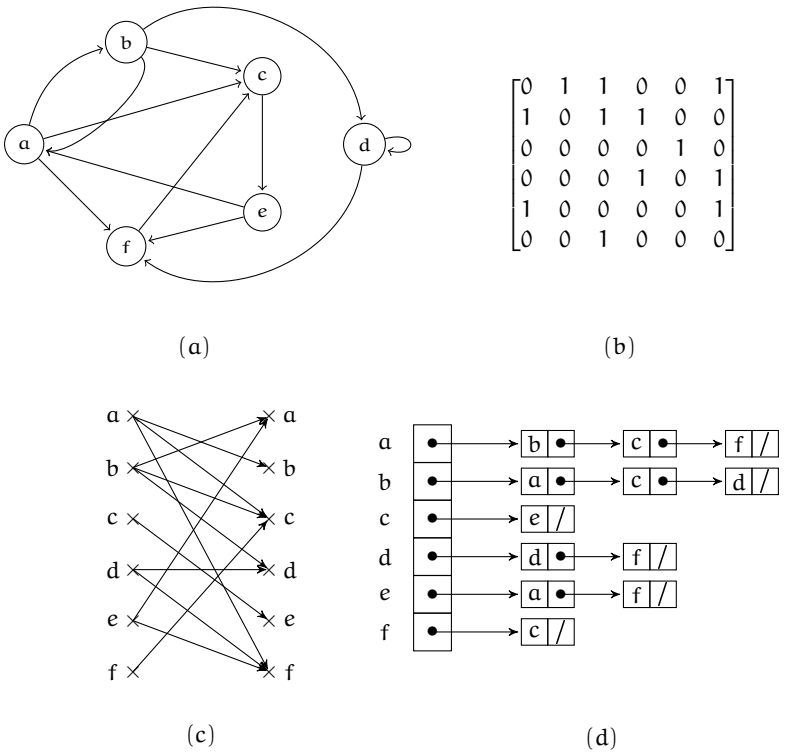


Fig. 1.3 – *Quatre représentations d'un graphe orienté. Le schéma* (a) *est la représentation sagittale classique,* (b) *est la représentation par la matrice d'adjacence,* (c) *est une représentation bipartite, enfin* (d) *est une représentation par listes des successeurs.*

Définition 2 (Boucle (sur un sommet)) :
On appelle boucle sur un sommet s *un arc qui relie ce sommet à lui-même, donc de la forme* (s, s) *avec* $s \in N$.

Définition 3 (Demi-degré) :
Soit G *un graphe et* s *un sommet de* G. *Le demi-degré extérieur (resp. intérieur) de* s *dans* G, *noté* $d_G^+(s)$ *(resp.* $d_G^-(s)$*), est le nombre d'arcs ayant pour origine (resp. pour extrémité) le sommet* s.

Définition 4 (Sous-graphe et graphe induit par un ensemble de sommets) :
Un graphe orienté $G' = (N', V')$ *est un sous-graphe du graphe orienté* $G = (N, V)$ *si* $N' \subset N$ *et* $V' = \{(u, v) \in V \mid u \in N', v \in N'\}$. *On dit aussi que* G' *est le graphe induit de* G *par l'ensemble de sommets* N'.

En d'autres termes, un sous-graphe G' d'un graphe G est obtenu en prenant un sous-ensemble strict N' de l'ensemble N des sommets de G et en ne gardant que les arcs dont origine et extrémité appartiennent à N'.

Définition 5 (Successeur d'un sommet dans un graphe orienté) :
Soit $G = (N, V)$ *un graphe orienté et* $s \in N$. *L'ensemble des successeurs de* s *dans* G, *noté* $\mathrm{Succ}_G(s)$ *est l'ensemble des sommets atteints depuis* s *à travers la relation* V.

Définition 6 (Prédécesseur d'un sommet dans un graphe orienté) :
Soit $G = (N, V)$ *un graphe orienté et* $s \in N$. *L'ensemble des prédécesseurs de* s *dans* G, *noté* $\mathrm{Préd}_G(s)$, *est l'ensemble* $\mathrm{Succ}_{(N,V^{-1})}(s)$.

Exemple Dans l'exemple du graphe G de la figure 1.3, page 23, nous avons $d_G^+(d) = 2$, $d_G^-(d) = 2$, $Succ_G(a) = \{b, c, f\}$ et $Préd_G(c) = \{a, b, f\}$. On notera que l'arc (d, d) constitue une boucle (sur le sommet d). Le graphe $G_1 = (\{a, b, d, f\}, \{(a, b), (b, a), (a, f), (b, d), (d, d), (d, f)\})$ est le graphe induit de G par l'ensemble de sommets $\{a, b, d, f\}$.

S'il n'y a pas d'ambiguïté, l'indice G est omis.

Définition 7 (Chemin dans un graphe) :
Soit $G = (\{s_1, \ldots, s_n\}, V)$ *un graphe. Un chemin* P *dans* G *est une liste non vide de sommets* $\langle s_{i_1}, \ldots, s_{i_q} \rangle$ *telle que tout couple de sommets adjacents dans la liste* P *est un arc de* G *(*$(s_{i_k}, s_{i_{k+1}}) \in V$ *pour* $k \in 1 \,..\, q-1$*).*

Définition 8 (Longueur d'un chemin) :
La longueur du chemin $P = \langle s_{i_1}, \ldots, s_{i_q} \rangle$ *est égale au nombre d'arcs* $(s_{i_k}, s_{i_{k+1}})$ *qui le composent. On la note* $|P|$.

Définition 9 (Chemin élémentaire dans un graphe) :
Un chemin élémentaire est un chemin dans lequel il n'existe pas de répétition de sommets.

Définition 10 (Chemin simple dans un graphe) :
Un chemin simple est un chemin dans lequel il n'existe pas d'arc emprunté plus d'une fois. Tout chemin élémentaire est simple.

Définition 11 (Chemin hamiltonien dans un graphe) :
Un chemin hamiltonien est un chemin qui passe une fois et une seule par chacun des sommets. La liste des sommets est une permutation de N.

Définition 12 (Chemin eulérien dans un graphe) :
Un chemin eulérien est un chemin qui passe une fois et une seule par chacun des arcs.

Exemples de chemins Dans le graphe de la figure 1.3, page 23 :

- $\langle a, b, a, c, e, a, b, d, d\rangle$ est un chemin. Il est constitué de la liste d'arcs suivante : $\langle (a, b), (b, a), (a, c), (c, e), (e, a), (a, b), (b, d), (d, d)\rangle$ et est donc de longueur 8. Il n'est ni simple ni élémentaire.
- $\langle b, c, e\rangle$ est un chemin élémentaire, c'est donc aussi un chemin simple (de longueur 2).
- $\langle b, a, b\rangle$ n'est pas un chemin élémentaire : le sommet b est rencontré deux fois.
- $\langle a, b, c, e, f\rangle$ est un chemin simple. Il est aussi élémentaire.
- $\langle a, b, a, c\rangle$ est également un chemin simple mais il n'est pas élémentaire.
- $\langle a, b, c, e, a, b\rangle$ n'est pas un chemin simple : l'arc (a, b) est parcouru deux fois. Il n'est donc pas élémentaire.
- $\langle a, b, d, f, c, e\rangle$ est un chemin hamiltonien.

Dans le schéma (a) de la figure 1.4, page 26, le chemin $\langle d, b, c, a, b, e, d, c, e\rangle$ est eulérien.

Définition 13 (Circuit dans un graphe) :
Soit $G = (\{s_1, \ldots, s_n\}, V)$ *un graphe et* $C = \langle s_{i_1}, \ldots, s_{i_q}\rangle$ *un chemin dans* G. C *est un circuit dans* G *si et seulement si* $s_{i_q} = s_{i_1}$.

Les qualificatifs *élémentaire*, *simple*, *hamiltonien* et *eulérien* se transposent aisément de la notion de chemin à celle de circuit.

Exemples de circuits Dans le graphe de la figure 1.3, page 23 :

- $\langle a, b, a, b, d, d, f, c, e, a\rangle$ est un circuit puisque c'est un chemin d'une part et parce qu'il débute et finit par le même sommet a de l'autre.
- $\langle c, e, f, c\rangle$ est un circuit élémentaire.
- $\langle a, b, a, c, e, a\rangle$ est un circuit simple, il n'est pas élémentaire.
- $\langle a, b, d, f, c, e, a\rangle$ est un circuit hamiltonien.

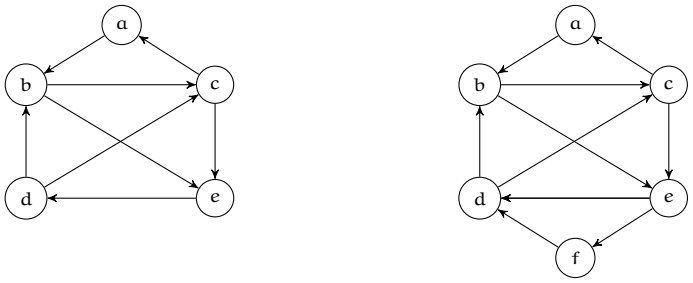


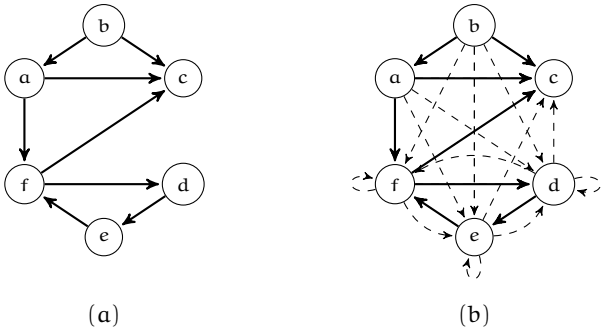
Fig. 1.4 – *Deux graphes orientés. Le graphe* (a) *possède un chemin eulérien* ($\langle d, b, c, a, b, e, d, c, e\rangle$) *mais pas de circuit eulérien. Le graphe* (b) *possède au moins un circuit eulérien* ($\langle d, b, c, a, b, e, d, c, e, f, d\rangle$).

Dans le graphe (b) de la figure 1.4, page 26, le chemin $\langle d, b, c, a, b, e, d, c, e, f, d\rangle$ est un circuit eulérien.

Définition 14 (Fermeture transitive d'un graphe) :
Soit $G = (N, V)$ *un graphe.* $G^+ = (N, V^+)$ *est la fermeture transitive de* G *si et seulement si* V^+ *est la plus petite relation transitive incluant* V.

Autrement dit, G^+ est la fermeture transitive de G si, lorsque dans G il existe un *chemin* entre x et y, il existe un *arc* entre x et y dans G^+.

Dans le schéma ci-dessous, le graphe (b) représente la fermeture transitive du graphe (a). Les arcs résultant de la fermeture apparaissent en pointillés.



Définition 15 (Descendant/ascendant d'un sommet) :
Soit x *un sommet d'un graphe* G. *Le sommet* y *est un descendant de* x *dans* G *s'il existe un arc* (x, y) *dans* G^+. *Le sommet* x *est alors un ascendant de* y.

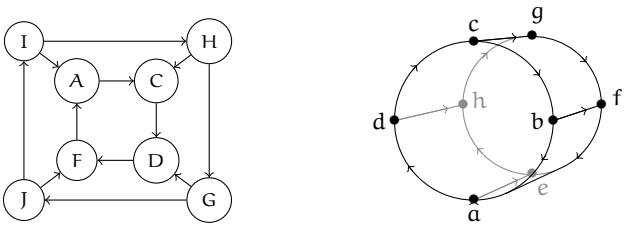
Définition 16 (Isomorphisme de graphes) :
Soit $G_1 = (N_1, V_1)$ *et* $G_2 = (N_2, V_2)$ *deux graphes orientés. On dit que ces graphes sont*

isomorphes, si et seulement s'il existe une bijection f *entre* N_1 *et* N_2 *telle que pour tout couple de sommets* (i, j) *de* N_1 *:*

$$(i, j) \in V_1 \Leftrightarrow (f(i), f(j)) \in V_2.$$

Dans le schéma ci-dessous, le graphe (a) est représenté dans le plan, tandis que (b) (le « cylindre » en fil de fer) est représenté en trois dimensions. Ce sont deux graphes isomorphes par la bijection p suivante :

$$p = \{(H, c), (G, b), (J, a), (I, d), (C, g), (D, f), (F, e), (A, h)\}.$$



(a) (b)

1.5.2 Graphes non orientés

Définition 17 (Graphe non orienté) :
Un graphe non orienté G *est un couple* (N, V)*, où* N *est un ensemble fini et* V *un ensemble de parties de* N *à un ou deux éléments.*

Comme dans les graphes orientés, les éléments de N sont appelés *sommets* ou *nœuds*. Les éléments de V sont appelés *arêtes*, ou *boucles* s'ils ne contiennent qu'un seul sommet.

Exemple Le couple G défini par :

$$G = \left(\begin{array}{l} \{a,b,c,d,e,f\}, \\ \{\{a, b\}, \{a, e\}, \{a, f\}, \{b, c\}, \{b, d\}, \{d\}, \{c, e\}, \{a, f\}, \{e, f\}, \{d, f\}\} \end{array} \right)$$

est un graphe non orienté. La figure 1.5 offre trois représentations possibles pour ce graphe.

Les notions de *chaîne* et de *cycle* sont les homologues, pour les graphes non orientés, des notions de chemin et de circuit dans les graphes orientés. Les qualificatifs *élémentaire*, *simple*, *hamiltonien* et *eulérien* se transposent aux graphes non orientés. Il en va de même pour la définition d'un sous-graphe, d'un graphe induit par un ensemble de sommets et de la fermeture transitive.

1.5.3 Graphes valués

Intuitivement, un graphe orienté *valué* est un graphe dans lequel chaque arc est doté d'un attribut (en général un réel, mais cela peut être toute autre valeur). Cet attribut peut représenter une distance, un temps, un coût, un potentiel, etc.

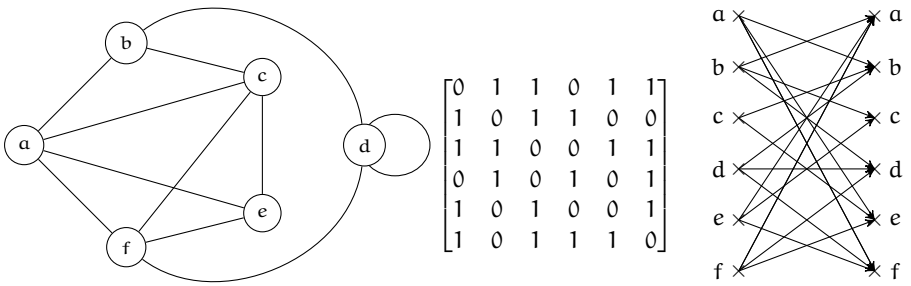


Fig. 1.5 – *Trois représentations d'un graphe non orienté. Le schéma* (a) *est la représentation traditionnelle ;* (b) *est la représentation par une matrice d'adjacence (cette matrice est toujours symétrique) ;* (c) *est une représentation bipartite.*

Définition 18 (Graphe orienté valué) :
Un graphe orienté valué G *est un triplet* (N, V, P)*, tel que* (N, V) *est un graphe orienté et* P *une fonction totale de* V *dans un ensemble* E *de valuations quelconques.*

La figure 1.6 reprend le graphe orienté de la figure 1.3 page 23, en valuant tous ses arcs dans $\mathbb{R}$ ($E = \mathbb{R}$).

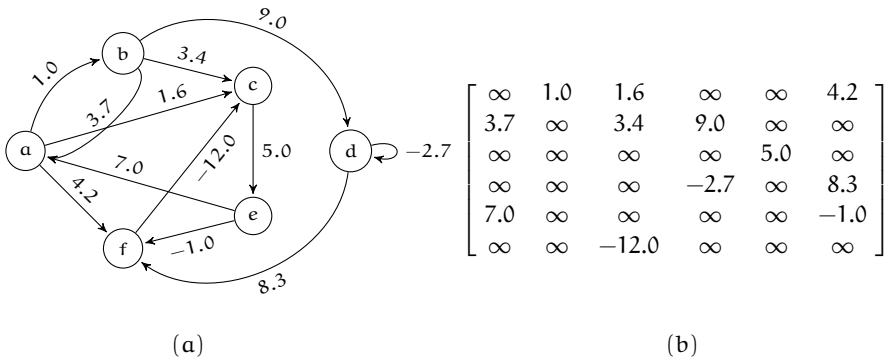


Fig. 1.6 – *Deux représentations d'un graphe orienté valué. Le schéma* (a) *est la représentation sagittale classique,* (b) *est la représentation matricielle dans laquelle les arcs absents sont dotés d'un symbole conventionnel (ici* ∞*).*

Un graphe *non* orienté *valué* est un graphe dans lequel chaque arête est dotée d'un attribut. Formellement, on peut en donner la définition suivante :

Définition 19 (Graphe non orienté valué) :
Un graphe non orienté valué G *est un triplet* (N, V, P) *tel que* (N, V) *est un graphe non orienté et* P *une fonction totale de* V *dans un ensemble* E *de valuations quelconques.*

Dans la pratique, la représentation d'un graphe valué peut se limiter à celle de N et de P, comme le montre le schéma (b) de la figure 1.6 page 28.

Références à la notion de graphe Parmi les innombrables problèmes portant sur les graphes (la recherche de plus courts chemins/chaînes, entre deux sommets, entre un sommet donné et tous les autres, entre tous les couples de sommets, la détermination de l'existence d'un isomorphisme entre graphes, le coloriage d'un graphe, etc.), certains sont abordés sous forme d'exercices dans cet ouvrage. On peut citer :

Graphes orientés non valués :

- Déplacements d'un cavalier sous contrainte (exercice 15, page 43).
- Parcours exhaustif d'un échiquier par un cavalier (exercice 49, page 113).
- Parcours d'un cavalier aux échecs (exercice 53, page 149).
- Circuits et chemins eulériens dans un graphe – tracés d'un seul trait (exercice 54, page 152).
- Chemins hamiltoniens : les dominos (exercice 55, page 154).
- Isomorphisme de graphes (exercice 57, page 157).
- Coloriage d'un graphe (exercice 58, page 159).
- Tournois et chemins hamiltoniens (exercice 84, page 236).

Graphes orientés valués :

- Diffusion d'information à moindre coût depuis une source : algorithme de Prim et de Dijkstra (exercice 77, page 210).
- Chemin de valeur minimale – cas particulier (exercice 129, page 350).
- Chemins de valeur minimale depuis une source – Algorithme de Bellman-Ford (exercice 130, page 352).
- Chemins de valeur minimale – Algorithme de Roy-Warshall et algorithme de Floyd – Algèbre de chemins (exercice 131, page 355).
- Chemin de coût minimal dans un tableau (exercice 132, page 358).
- Distance entre séquences : algorithme de Wagner et Fisher (exercice 137, page 366).

Graphes non orientés valués :

- Diffusion d'information à moindre coût depuis une source : algorithmes de Prim et de Dijkstra (exercice 77, page 210).
- Le voyageur de commerce (exercice 56, page 156, et exemple page 182).

1.6 Arbres

Outre leur intérêt intrinsèque (en compilation, en démonstration automatique, en traitement des langues naturelles, etc.), les arbres sont à la base de représentations efficaces pour une grande variété de structures de données (ensembles, files, files de priorité, tableaux flexibles, etc. – voir [36]). Bien que de nombreux types d'arbres soient utilisés tout au long de cet ouvrage (voir par exemple les arbres de récursion, chapitre 5, page 117), nous nous limitons ici aux seuls arbres binaires et ternaires. Il existe plusieurs façons de définir cette notion. Le passage par la théorie des graphes en est une. Un arbre binaire/ternaire y est défini comme un graphe particulier (un graphe non orienté, sans cycle, connexe, dont un sommet – la racine – est distinguée). Notre préférence est cependant en faveur d'une définition inductive, qui se prête mieux à l'usage que nous en faisons.

La figure 1.7, page 30, fournit un exemple d'arbre binaire et un exemple d'arbre ternaire. Il s'agit de graphes *orientés*. Sur ce point, la terminologie graphique et celle adoptée ici divergent, puisqu'en théorie des graphes, ces structures seraient dénommées « arborescences ». Le terme *chemin* est utilisé dans le sens de la théorie des graphes. Par abus de langage, nous utilisons le vocable « chemin » pour des trajets qui font abstraction du sens des flèches. Quant au terme *distance*, il est utilisé indifféremment pour les chemins et pour les « chemins ».

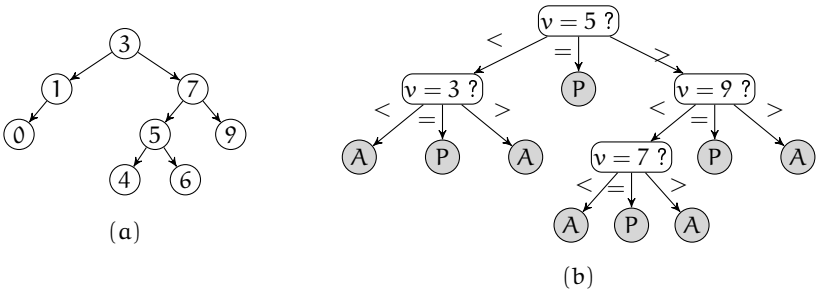


Fig. 1.7 – *Schéma* (a) *: un arbre binaire* – *Schéma* (b) *: un arbre (de décision) ternaire (*A *: absent* P *: présent)*

L'exemple ci-dessous montre comment se définit inductivement un arbre binaire *étiqueté*[3] (à chaque nœud est associée une valeur, ici un entier naturel) :

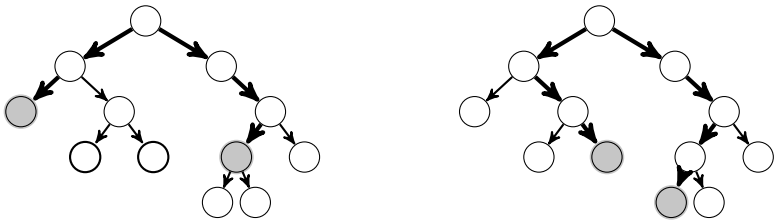
1. **constantes**
2. $\quad ab = \{/\} \cup \{(g, n, d) \mid g \in ab \text{ et } n \in \mathbb{N} \text{ et } d \in ab\}$
3. **variables**
4. $\quad t \in ab$
5. **début**
6. $\quad t \leftarrow /\,;$
7. $\quad t \leftarrow (((/, 0, /), 1, /), 3, (((/, 4, /), 5, (/, 6, /)), 7, (/, 9, /)))\,;$

3. Même si, par commodité, ils peuvent occuper la même place dans les schémas, il convient de ne pas confondre le *nom* d'un sommet ou d'un nœud, qui est un identifiant, avec l'étiquette, qui est une valeur quelconque. La définition inductive des arbres permet en général de faire l'impasse sur le *nom* du nœud.

8. écrire(t.g)
9. fin

ab est l'ensemble de tous les arbres binaires finis d'entiers. Il se définit (ligne 2 du code) comme l'union de deux ensembles : l'ensemble $\{/\}$ (l'arbre vide) et l'ensemble des triplets (g, n, d), où g et d sont des arbres binaires (respectivement le sous-arbre gauche et le sous-arbre droit) et n un entier naturel. À la ligne 4, la variable t est définie comme l'un des éléments de l'ensemble ab. L'instruction de la ligne 6 affecte l'arbre vide à t. Celle de la ligne suivante affecte à t l'arbre du schéma (a) de la figure 1.7. La ligne 8 affiche, sur un terminal standard, le sous-arbre gauche de t (noté t.g). Dans un arbre binaire, une *feuille* est un nœud sans aucun fils (sans aucun successeur) ; un *point simple* est un nœud n'ayant qu'un seul fils. La *hauteur* d'un arbre binaire est la distance (c'est-à-dire le nombre d'arcs) entre la racine et la feuille la plus éloignée. Le *poids* d'un arbre binaire est son nombre de nœuds. Parmi tous les « chemins » simples possibles dans un arbre non vide, il en existe (au moins) un dont la longueur est maximale. Cette longueur est appelée *diamètre*. Un diamètre ne passe pas nécessairement par la racine (voir exercice 92, page 254).

Dans les schémas ci-dessous, les « chemins » entre les nœuds grisés sont matérialisés par des arcs en gras. Dans le schéma (a) (resp. (b)) la longueur du « chemin » considéré est de cinq arcs (resp. sept arcs). La hauteur des deux arbres est de 4, leur diamètre de 7. Le « chemin » en gras du schéma (b) est un diamètre.



(a) (b)

La plupart des définitions ci-dessus se transposent aux arbres ternaires sans difficulté. Parmi les arbres binaires particuliers, citons les arbres *filiformes*, les arbres *complets*, les arbres *pleins* et les arbres *parfaits*. Un arbre filiforme est un arbre dont aucun nœud ne possède deux fils. Un arbre complet est un arbre qui ne possède pas de point simple. Les arbres (a) et (b) de la figure 1.8, page 32, sont des arbres complets. Ce n'est pas le cas des trois autres arbres de la figure. Un arbre plein est un arbre complet dont toutes les feuilles sont situées à la même hauteur. C'est le cas de l'arbre (b) de la figure 1.8, page 32. C'est le seul arbre de ce type dans cette figure. Un arbre parfait est tel que : i) toutes les feuilles sont situées sur les deux derniers niveaux, c'est-à-dire les niveaux plus bas, ii) quand l'arbre parfait n'est pas un arbre plein, c'est-à-dire quand les feuilles sont effectivement réparties sur deux niveaux, les feuilles du dernier niveau sont regroupées sur la gauche, iii) il existe au plus un point simple, et il est situé sur l'avant-dernier niveau. Le schéma (c) de la figure 1.8 fournit un exemple d'arbre parfait. L'arbre (b) est aussi un arbre parfait particulier (car sans point simple). En revanche, les arbres (d) et (e) de la figure 1.8, page 32, ne sont pas des arbres parfaits, le premier parce qu'il présente deux points simples, et le second parce que les feuilles du dernier niveau ne sont pas regroupées à gauche. L'arbre (a) n'est pas non plus un arbre parfait. Il est facile – et souvent utile – de fournir une définition inductive des notions étudiées dans cette section (voir exercice92, page 254).

Nous mettons en garde le lecteur contre le caractère non consensuel des définitions portant sur les arbres.

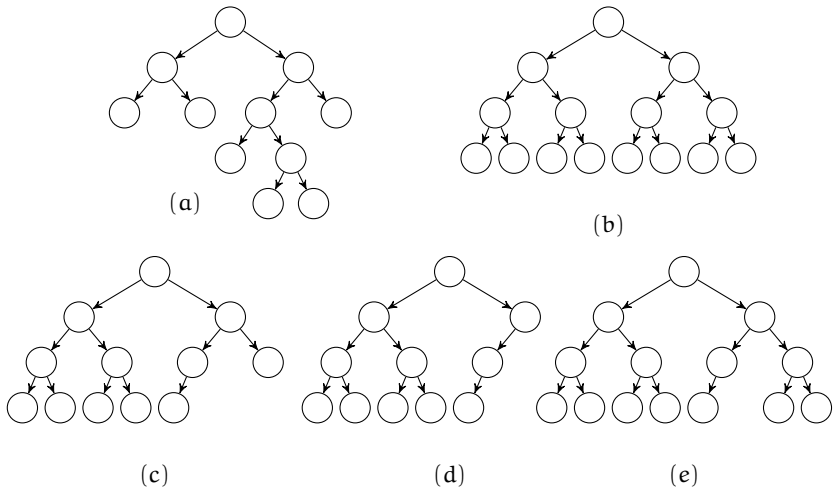


Fig. 1.8 – *Arbres complets, plein et parfaits – Exemples et contre-exemples*

Un type d'arbre binaire particulier mérite notre attention, il s'agit des arbres binaires *de recherche* (ou abr en abrégé). Un abr est soit un arbre vide, soit un arbre binaire dont les nœuds contiennent des valeurs numériques et qui est tel que, si v est la racine, son sous-arbre gauche (resp. droit) est un abr qui contient, s'il n'est pas vide, des valeurs inférieures (resp. supérieures) à v. L'arbre (a) de la figure 1.7, page 30, est un abr.

À plusieurs occasions, nous faisons usage d'arbres particuliers appelés « arbres de décision ». Un arbre de décision permet de structurer une classification (voir figure 8.1, page 245), de représenter l'ensemble des exécutions d'un algorithme (voir figure 87, page 249), parfois dans le but de calculer sa complexité (voir figure 94, page 256). Un arbre de décision se présente en général comme un arbre binaire dont chaque nœud est un prédicat, chaque branche une réponse possible (**vrai** ou **faux**). Le prédicat peut être remplacé par une question à choix multiples qui dote chaque nœud d'autant de branches. L'arbre (b) de la figure 1.7, page 30, est un arbre de décision ternaire qui permet de savoir si une certaine valeur v appartient à l'ensemble $\{3, 5, 7, 9\}$. Dans cet arbre, pour chaque nœud, on cherche à situer v par rapport à la valeur présente dans le nœud considéré. Les feuilles répondent à la question relative à l'arbre de décision (P : la valeur numérique présente dans la feuille appartient à l'ensemble $\{3, 5, 7, 9\}$, A : la valeur présente dans la feuille ne figure pas dans l'ensemble $\{3, 5, 7, 9\}$ – elle est absente).

Références à la notion d'arbre La notion d'arbre est au cœur de plusieurs exercices de cet ouvrage. Citons en particulier les titres suivants :

- Recherches dichotomique, trichotomique et par interpolation (exercice 87, page 249).
- Minimum local dans un arbre binaire (exercice 91, page 253).
- Diamètre d'un arbre binaire (exercice 92, page 254).

— Écrous et boulons (exercice 94, page 256).

— Le plus grand carré et plus grand rectangle sous un histogramme (exercice 114, page 306).

— Lâchers d'œufs par la fenêtre (le retour) (exercice 128, page 347).

— Arbres binaires de recherche pondérés (exercice 133, page 360).

— Triangulation optimale d'un polygone convexe (exercice 140, page 374).

Cette notion apparaît également à la figure 8.1, page 245, en tant qu'arbre de décision pour présenter une typologie des méthodes « Diviser pour Régner ».

1.7 Files de priorité

1.7.1 Définition

Une file de priorité est une structure de données dont chaque élément est muni d'une valeur numérique qui représente une priorité. Si la convention est que plus la valeur est petite plus elle est prioritaire, une suppression a pour effet d'enlever de la file une occurrence de la plus petite valeur. Dans sa version standard, une file de priorité se définit par les cinq opérations suivantes :

— *InitFdP*(f) : procédure qui vide la file de priorité f.

— *TêteFdP*(f) : fonction qui délivre l'élément prioritaire (la tête) de la file de priorité f. Cette fonction a comme précondition que f possède au moins un élément. Elle ne modifie pas la file.

— *SupprimerFdP*(f) : procédure qui enlève de la file f l'élément prioritaire (la tête de file). Cette procédure a également comme précondition que f possède au moins un élément.

— *AjouterFdP*(f, e) : procédure qui introduit dans la file f l'élément e (cet élément étant supposé véhiculer sa propre priorité).

— *EstVideFdP*(f) : fonction booléenne qui délivre vrai si et seulement si la file de priorité f est vide.

Il est parfois nécessaire d'enrichir ce jeu d'opérations par des opérations complémentaires, comme celle qui consiste à modifier la priorité d'un élément quelconque de la file (voir par exemple l'exercice 77, page 210). Ces modifications se font au coup par coup. La taille d'une file de priorité f est notée |f|.

1.7.2 Mises en œuvre

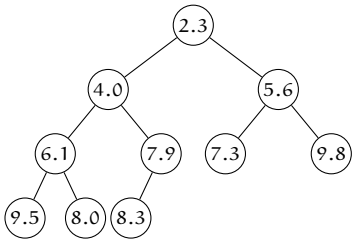
La littérature (voir par exemple [36]) présente plusieurs mises en œuvre plus ou moins sophistiquées de files de priorité. Rappelons deux d'entre elles, pour des files de priorité de n éléments :

— Les listes triées, pour lesquelles l'opération d'ajout est en $\mathcal{O}(n)$, l'opération de (ré)initialisation est, selon que l'on récupère ou non la place occupée, en $\Theta(n)$ ou $\Theta(1)$. Les autres opérations sont en $\Theta(1)$. Ce type de solution ne peut convenir que si n est petit.

— Les tas. Du point de vue structurel, un tas est un arbre binaire parfait (les feuilles sont situées sur – au plus – deux niveaux consécutifs) à gauche (les feuilles du dernier niveau sont regroupées sur la gauche). Pour ce qui concerne le contenu, un tas est un minimier (pour tout nœud, sa valeur est inférieure ou égale à celle de tous ses descendants).

Une caractéristique importante est qu'un tas peut se représenter avantageusement par un tableau T tel que, pour un i donné, $T[2i]$ et $T[2i+1]$, s'ils existent, sont les fils de $T[i]$.

Le schéma (a) suivant montre, sous sa forme arborescente, un tas défini sur $\mathbb{R}_+$. Le schéma (b) est la représentation en tableau du schéma (a).



(a)

	1	2	3	4	5	5	7	8	9	10
T	2.3	4.0	5.6	6.1	7.9	7.3	9.8	9.5	8.0	8.3

(b)

En termes de sommets visités (ou de conditions évaluées), le coût de l'opération *Supprimer* est en $\Theta(\log_2(n))$, celui de *Ajouter* est en $\mathcal{O}(\log_2(n))$. Les autres opérations sont en temps constant.

Références à la notion de file de priorité Cette notion est au cœur de deux chapitres de cet ouvrage : le chapitre 6 sur la démarche PSEP (Programmation par Séparation et Évaluation Progressive) et le chapitre 7 sur les algorithmes gloutons.

1.8 Files FIFO

Une file FIFO (First In, First Out : premier entré, premier sorti) peut être vue comme une file de priorité particulière dont les éléments ne portent pas de priorité explicite. La gestion des priorités est effectuée implicitement, puisqu'un élément est toujours inséré en fin de file et que lors d'un retrait, c'est l'élément en tête de file (le plus anciennement entré donc) qui est extrait. FIFO(E) représente toutes les files FIFO sur des éléments appartenant à E. Ainsi, si l'on écrit $P \in \text{FIFO}(\mathbb{R}_+ \times \mathbb{R}_+)$, on déclare que P est une file de couples de réels non négatifs. Les opérations nécessaires à la gestion des files FIFO sont les suivantes :

— *InitFifo*(P) : procédure qui vide la file P.

— *AjouterFifo*(P, e) : procédure qui ajoute l'élément e en queue de la file P.

— *EstVideFifo*(P) : cette fonction délivre vrai si et seulement si la file P est vide.

— *TêteFifo*(P) : si P est une file d'éléments de type E, cette fonction délivre l'élément en tête de file, sans le supprimer de la file. Cette fonction a comme précondition que P n'est pas vide.

— *SupprimerFifo*(P) : cette procédure supprime de la file P l'élément en tête de file. Cette procédure a comme précondition que P n'est pas vide.

Comme pour les files de priorité, il est parfois nécessaire d'enrichir ce jeu d'opérations ou de modifier légèrement certaines d'entre elles.

Références à la notion de file FIFO La notion de file FIFO est utilisée dans le chapitre 7 sur les algorithmes gloutons.

1.9 Exercices

1.9.1 Démonstrations

Exercice 1. Élément neutre unique ◦ •

Cet exercice vise essentiellement à pratiquer la démonstration par l'absurde de façon rigoureuse.

Soit un ensemble S muni d'un opérateur interne $\oplus$. Soit $u \in S$ un élément neutre à gauche pour $\oplus$, autrement dit :

$$\forall x \cdot (x \in S \Rightarrow u \oplus x = x).$$

Soit v un élément neutre à droite pour $\oplus$.

Question 1. Démontrer de façon directe que si l'opérateur $\oplus$ est commutatif, on a $u = v$. 1 - Q 1

Question 2. Démontrer par l'absurde cette propriété en relâchant la propriété de commutativité de l'opérateur $\oplus$. 1 - Q 2

Exercice 2. Élément minimum d'un ensemble muni d'un ordre partiel ◦ •

L'intérêt de cet exercice est de procéder à la démonstration d'une même propriété à la fois par l'absurde et par récurrence.

Soit E un ensemble fini muni d'une relation d'ordre partiel notée $\preceq$ et F un sous-ensemble strict non vide de E. On appelle antécédent strict de x un élément y différent de x tel que $y \preceq x$. Un élément m de F est dit *minimum* si m ne possède aucun antécédent strict dans F.

Question 1. Montrer par l'absurde que F possède au moins un élément minimum. 2 - Q 1

Question 2. Montrer par récurrence que F possède au moins un élément minimum. 2 - Q 2

2 - Q 3 **Question** 3. On considère l'ensemble E constitué des couples d'entiers naturels et la relation d'ordre partiel $\preceq$ définie par :

$$(a, b) \preceq (c, d) \mathrel{\widehat{=}} a \leqslant c \text{ et } b \leqslant d.$$

Soit $F(\subset E) = \{(a, b) \mid a \in 1\,..\,3 \text{ et } b \in 1\,..\,2 \text{ et } a \neq b\}$. Dénombrer les éléments minimaux de F.

Exercice 3. Factorielle et exponentielle ◦ •

Dans cet exercice, on établit deux résultats sur la comparaison de valeurs de factorielles et d'exponentielles. Le premier constitue un résultat utile sur l'ordre des deux classes de complexité associées (voir chapitre 2).

3 - Q 1 **Question** 1. Démontrer par récurrence simple que :

$$\forall n \cdot (n \in \mathbb{N}_1 \Rightarrow n! \geqslant 2^{n-1}).$$

3 - Q 2 **Question** 2. Pour quelle valeur minimale n_0 a-t-on :

$$\forall n \cdot ((n \in \mathbb{N}_1 \text{ et } n_0 \in \mathbb{N}_1 \text{ et } n \geqslant n_0) \Rightarrow n! > 2^{2n}) \ ?$$

Exercice 4. Par ici la monnaie ◦ •

Dans cet exercice, on commence par valider un nouveau schéma de démonstration par récurrence simple. Ensuite, on l'applique à une propriété sur la construction d'une somme monétaire. On demande également d'en faire la preuve au moyen du schéma habituel de démonstration par récurrence simple. On peut alors constater si une méthode est plus « commode » que l'autre, l'observation faite n'ayant pas vertu à être généralisée.

4 - Q 1 **Question** 1. Démontrer la validité du schéma alternatif de démonstration par récurrence simple ci-dessous :

Si l'on peut démontrer les deux propriétés suivantes :

Base $P(n_0)$ et $P(n_0 + 1)$ sont vraies pour un certain $n_0 \in \mathbb{N}$

Récurrence $\forall n \cdot ((n \geqslant n_0 \text{ et } P(n)) \Rightarrow P(n + 2))$

alors :

Conclusion $\forall n \cdot (n \geqslant n_0 \Rightarrow P(n))$

Question 2. Démontrer à l'aide de ce schéma que toute somme n de six centimes ou plus peut être obtenue avec des pièces de deux et de sept centimes. 4 - Q 2

Question 3. Démontrer ce même résultat en utilisant le schéma de démonstration par récurrence simple de la section 1.1.3, page 4. 4 - Q 3

Question 4. Quel est le nombre maximum de pièces de sept centimes utilisées en s'appuyant sur chacun de ces schémas ? 4 - Q 4

Question 5. Selon vous, laquelle de ces deux preuves est-elle la plus simple à établir ? 4 - Q 5

Exercice 5. Nombres de Catalan ○ •

Dans cet exercice, on démontre par récurrence simple que la forme close de la relation de récurrence proposée pour les nombres de Catalan est correcte. On établit également une majoration de la valeur du n^e nombre de Catalan.

On s'est intéressé aux nombres de Catalan définis notamment par la relation de récurrence (voir page 11) :

$$\begin{aligned} &Cat(1) = 1 \\ &Cat(n) = \frac{4n-6}{n} \cdot Cat(n-1) \qquad n > 1. \end{aligned}$$

Question 1. Montrer par récurrence simple que cette relation de récurrence admet comme forme close pour tout entier n positif (voir page 11) : 5 - Q 1

$$Cat(n) = \frac{(2n-2)!}{(n-1)!\, n!}.$$

Question 2. Montrer par récurrence simple que pour tout entier n positif, on a : 5 - Q 2

$$Cat(n) \leqslant \frac{4^{n-1}}{n}.$$

Exercice 6. Démonstrations par récurrence simple erronées ○ •

Cet exercice vise à attirer l'attention sur le respect des hypothèses pour effectuer une démonstration par récurrence correcte. Deux exemples sont successivement proposés, dans lesquels la proposition P à démontrer est à l'évidence *fausse. Le raisonnement avancé conduisant à la démontrer, il est intéressant de mettre en évidence l'erreur commise.*

Premier cas

On envisage de démontrer par récurrence la propriété P_1 suivante :

Tout couple d'entiers naturels est constitué de deux entiers égaux.

La récurrence se fait sur le maximum des deux nombres a et b, noté $\max(a, b)$.

Base Si $\max(a, b) = 0$, alors il est clair que $a = b = 0$.

Récurrence Supposons la propriété P_1 vraie quand le maximum de a et b est p. L'hypothèse de récurrence est donc :

si $\max(a, b) = p$ alors $a = b = p$.

On doit montrer qu'alors P_1 est vraie quand le maximum de a et b est $(p + 1)$. Soit (a, b) un couple tel que $\max(a, b) = p + 1$. Le maximum de $(a - 1)$ et de $(b - 1)$ est donc p. Par hypothèse de récurrence, on a : $a - 1 = b - 1 = p$, d'où : $a - 1 + 1 = b - 1 + 1 = p + 1$, soit finalement $a = b = p + 1$.

Conclusion La propriété P_1 est vraie pour tout couple d'entiers naturels.

6 - Q 1 **Question 1.** Où est l'erreur ?

Second cas

On se propose de démontrer par récurrence la propriété P_2 suivante :

n *points quelconques du plan sont toujours alignés.*

La récurrence se fait sur le nombre n de points.

Base Pour $n = 2$, la proposition P_2 est vraie, puisque deux points sont toujours alignés.

Récurrence Supposons la propriété P_2 vraie pour p points (hypothèse de récurrence). Montrons qu'alors les $(p + 1)$ points nommés $A_1, A_2, A_3, \ldots, A_{p+1}$ sont alignés. D'après l'hypothèse de récurrence, les p premiers points $A_1, A_2, A_3, \ldots, A_p$ sont alignés sur une droite (d_1) et les p derniers points $A_2, A_3, \ldots, A_{p+1}$ sont alignés sur une droite (d_2). Les deux droites (d_1) et (d_2) ont en commun les deux points A_2 et A_3 et sont donc forcément confondues. On a $(d_1) = (d_2) = (A_2A_3)$ et les points $A_1, A_2, A_3, \ldots, A_p, A_{p+1}$ sont donc alignés.

Conclusion la proposition P_2 est vraie pour tout nombre de points supérieur ou égal à 2.

6 - Q 2 **Question 2.** Trouver la faille.

Exercice 7. Démonstration par récurrence forte erronée d'une formule pourtant exacte ○ ⁝

Dans la lignée du précédent, cet exercice vise à mettre en évidence une erreur dans un raisonnement par récurrence. Cependant, ici, la formule à démontrer est juste et on en demande une démonstration directe convenable.

On se propose de démontrer par récurrence simple que, pour n entier supérieur ou égal à 1, on a :

$$n = \sqrt{1+(n-1)\sqrt{1+n\sqrt{1+(n+1)\sqrt{1+(n+2)\dots}}}}$$

Pour commencer, on admettra que cette expression a un sens, c'est-à-dire qu'elle converge quand n augmente indéfiniment (ce qui est vrai, comme on le constatera ultérieurement).

La démonstration par récurrence forte se fait alors ainsi :

Base Pour $n = 1$, la partie droite de la formule devient $\sqrt{1+0\sqrt{1+1(\dots)}} = 1$ et l'égalité est donc vérifiée.

Hypothèse de récurrence Pour tout $n > 1$:

$$(n-1) = \sqrt{1+(n-2)\sqrt{1+(n-1)\sqrt{1+n\sqrt{1+(n+1)\dots}}}}.$$

Récurrence On a :

$$\begin{aligned}
&(n-1) = \sqrt{1+(n-2)\sqrt{1+(n-1)\sqrt{1+n\sqrt{1+(n+1)\dots}}}} \\
\Rightarrow &\qquad \text{élévation au carré} \\
&(n-1)^2 = 1+(n-2)\sqrt{1+(n-1)\sqrt{1+n\sqrt{1+(n+1)\dots}}} \\
\Leftrightarrow &\qquad \text{identité remarquable} \\
&n^2-2n+1 = 1+(n-2)\sqrt{1+(n-1)\sqrt{1+n\sqrt{1+(n+1)\dots}}} \\
\Leftrightarrow &\qquad \text{arithmétique} \\
&n(n-2) = (n-2)\sqrt{1+(n-1)\sqrt{1+n\sqrt{1+(n+1)\dots}}} \\
\Leftrightarrow &\qquad \text{division des deux membres par } n-2 \\
&\frac{n(n-2)}{n-2} = \sqrt{1+(n-1)\sqrt{1+n\sqrt{1+(n+1)\dots}}} \\
\Leftrightarrow &\qquad \text{arithmétique} \\
&n = \sqrt{1+(n-1)\sqrt{1+n\sqrt{1+(n+1)\dots}}}.
\end{aligned}$$

On a démontré que l'hypothèse de récurrence implique la formule à prouver.

Question 1. Où est l'erreur de raisonnement ? 7 - Q 1

Question 2. Cette formule est pourtant exacte. En donner une preuve correcte. 7 - Q 2

Exercice 8. Schéma alternatif de démonstration de récurrence à deux indices ◦ •

On a vu à la section 1.1.6, page 9, un schéma de démonstration par récurrence à deux indices entiers. Le but de cet exercice est d'en valider un autre.

Montrer que le schéma de démonstration ci-après est correct :

Si l'on peut démontrer les deux propriétés suivantes :

Base $P(i,1)$ est vraie pour tout $i \geqslant 1$ **et**
$P(1,j)$ est vraie pour tout $j \geqslant 1$

Récurrence $\forall (i,j) \cdot \left(\left(\begin{array}{l} i \in \mathbb{N}_1 \text{ et} \\ j \in \mathbb{N}_1 \text{ et} \\ P(i,j) \end{array} \right) \Rightarrow \left(\begin{array}{l} P(i+1,j) \text{ et} \\ P(i,j+1) \text{ et} \\ P(i+1,j+1) \end{array} \right) \right)$

alors :

Conclusion $P(m,n)$ est vraie pour tous les couples d'entiers tels que $m \geqslant 1$ et $n \geqslant 1$

Exercice 9. De 7 à 77 et plus si ... ◦ •

Cet exercice est consacré à la démonstration par récurrence d'une propriété des éléments d'une suite d'entiers donnée d'emblée sous sa forme close.

Soit l'entier $A(n,p)$ défini par $A(n,p) = 3^{2n} - 2^{n-p}$ $\quad n \in \mathbb{N}_1$ **et** $p \in \mathbb{N}$ **et** $n \geqslant p$.

9 - Q 1 **Question 1.** Montrer par récurrence simple que si $A(n,p)$ est (resp. n'est pas) divisible par 7, alors $A(n+1,p)$ l'est aussi (resp. ne l'est pas non plus).

9 - Q 2 **Question 2.** Que dire de la divisibilité par 7 des nombres des suites $A(n,0)$, $A(n,1)$, $A(n,2)$ et $A(n,3)$?

Exercice 10. Une petite place svp ◦ •

On s'intéresse ici à une suite de nombres. On en démontre deux propriétés, une par récurrence simple, la seconde de façon directe, la démonstration par récurrence n'apparaissant pas dans ce cas la plus aisée.

On définit la suite récurrente de nombres $A(n)$ par :

$$\begin{aligned} A(1) &= 1 \\ A(n) &= A(n-1) + \frac{n^2 - 3n + 1}{(n-1)^2 \cdot n^2} \qquad n \geqslant 2. \end{aligned}$$

Question 1. Montrer par récurrence simple que la forme close des nombres $A(n)$ est $(n^2 - n + 1)/n^2$ pour tout $n \geqslant 1$. Qu'en déduire quant à la nature de ces nombres ? 10 - Q 1

Question 2. Montrer que pour tout $n > 2$, $A(n)$ est dans l'intervalle *ouvert* $A(2)\,..\,A(1)$. 10 - Q 2

Exercice 11. Suite du lézard ◦ ⁝

Dans cet exercice, on cherche à construire toute suite infinie de nombres binaires qui est égale à celle obtenue en ne prenant qu'un terme sur trois, mais aussi à celle obtenue en ne gardant que les deux termes sur trois restants. On écarte les deux suites triviales composées exclusivement de 0 ou de 1.

Soit une suite binaire $S = \langle s_1, s_2, \ldots, s_n, \ldots\rangle$ autre que celle (triviale) composée uniquement de 0 (resp. 1). On note $S/3$ la suite construite en prenant un élément sur trois dans S de la façon suivante : $S/3 = \langle s_3, s_6, \ldots, s_{3n}, \ldots\rangle$. On note $S - S/3$ la suite qui reste de S quand on a enlevé $S/3$, ce qui donne : $S - S/3 = \langle s_1, s_2, s_4, s_5, s_7, s_8, \ldots, s_{3n-2}, s_{3n-1}, s_{3n+1}, s_{3n+2}, \ldots\rangle$. Par exemple, pour $S = \langle 0,0,1,1,0,0,1,1,0,0,1,1,\ldots\rangle$, on a :

$$S/3 = \langle 1,0,0,1,\ldots\rangle \quad \text{et} \quad S - S/3 = \langle 0,0,1,0,1,1,0,1,\ldots\rangle.$$

Question 1. Donner un raisonnement par récurrence permettant de construire les deux suites S_1 et S_2 (autres que $\langle 0,0,0,\ldots\rangle$ et $\langle 1,1,1,\ldots\rangle$) telles que $S = S/3 = S - S/3$. On les appelle les suites « du lézard » (voir [22]). 11 - Q 1

Question 2. En donner les 20 premiers termes. 11 - Q 2

Exercice 12. À propos de la forme close de la suite de Fibonacci ∞ •

Cet exercice met en lumière le fait que même si l'on connait la forme close d'une récurrence, celle-ci peut ne pas être transposable dans un algorithme.

On rappelle que la suite de Fibonacci est définie (voir page 10) par la récurrence :

$$\begin{aligned} &\mathcal{F}(1) = 1 \\ &\mathcal{F}(2) = 1 \\ &\mathcal{F}(n) = \mathcal{F}(n-1) + \mathcal{F}(n-2) \qquad n > 2 \end{aligned}$$

et qu'elle admet la forme close :

$$\mathcal{F}(n) = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right] \qquad n \geqslant 1.$$

12 - Q 1 **Question** 1. Expliquer pourquoi en pratique on n'utilise pas la formule ci-dessus pour calculer $\mathcal{F}(n)$ avec n fixé.

12 - Q 2 **Question** 2. Proposer un algorithme itératif de calcul du n^e nombre de Fibonacci.

Remarque Dans l'exercice 103, page 268, plusieurs autres façons d'effectuer ce calcul sont examinées.

Exercice 13. Nombre d'arbres binaires à n nœuds ◦ •

Cet exercice complète l'exemple donné page 12. Son objectif est de construire une variante de l'algorithme de calcul du nombre d'arbres binaires ayant n nœuds fondée sur une forme close, donc a priori *plus efficace.*

On a établi que le nombre d'arbres binaires possédant n nœuds est donné par la récurrence :

$$\begin{aligned} &\mathrm{nbab}(0) = 1 \\ &\mathrm{nbab}(n) = \sum_{i=0}^{n-1} \mathrm{nbab}(i) \cdot \mathrm{nbab}(n-i-1) \qquad n \geqslant 1. \end{aligned}$$

13 - Q 1 **Question** 1. Montrer que cette récurrence s'écrit aussi sous la forme :

$$\mathrm{nbab}(n) = \mathrm{Cat}(n+1) \qquad n \geqslant 0$$

$\mathrm{Cat}(n)$ étant le n^e nombre de Catalan (voir définition page 10).

13 - Q 2 **Question** 2. En déduire un algorithme itératif de calcul de $\mathrm{nbab}(n)$.

Exercice 14. Identification d'une forme close ○ ⁝

L'intérêt de cet exercice est double. D'une part, on y étudie une façon pratique de calculer la valeur des éléments d'une récurrence à deux indices. D'autre part, on en établit une forme close qui, pour être prouvée, requiert de valider un nouveau schéma de démonstration par récurrence à deux indices.

On considère la suite récurrente à deux indices définie par :

$$\begin{aligned} a(0,j) &= j+1 && j \geqslant 0 \\ a(i,0) &= 2\cdot a(i-1,0) + a(i-1,1) && i > 0 \\ a(i,j) &= a(i-1,j-1) + 2\cdot a(i-1,j) + a(i-1,j+1) && i>0 \text{ et } j>0. \end{aligned}$$

Question 1. Calculer $a(3,2)$. Proposer une structure tabulaire et décrire la progression pour calculer plus généralement la valeur de l'élément $a(i,j)$ $(i,j \geqslant 0)$. 14 - Q 1

Question 2. Écrire le programme itératif de calcul de $a(n,m)$ pour n et m entiers donnés. 14 - Q 2

Question 3. Proposer une forme close pour $a(i,j)$ et la démontrer par récurrence. Quel intérêt cette expression présente-t-elle d'un point de vue algorithmique ? 14 - Q 3

1.9.2 Dénombrements

Exercice 15. Déplacements d'un cavalier sous contrainte ○ •

On considère les différents parcours d'un cavalier (dans l'esprit du jeu d'échecs) évoluant sous contrainte entre deux coins extrêmes d'une grille et on les dénombre grâce à une récurrence.

On s'intéresse aux parcours d'un cavalier (au sens du jeu d'échecs) sur une grille ayant $n > 4$ lignes et $m > 3$ colonnes. On veut connaître le nombre de façons différentes dont il dispose pour se rendre de la case $(1,1)$ à la case d'arrivée (n,m). On adopte la convention selon laquelle (i,j) désigne la case de la grille située en ligne i et colonne j. Contrairement au jeu d'échecs où un cavalier dispose de huit possibilités de déplacement (sauf s'il est amené à sortir de l'échiquier), on impose ici que le cavalier ne puisse se déplacer qu'en augmentant l'indice de colonne (le second), comme sur la figure 1.9.

Question 1. Soit $nparc(i,j)$ le nombre de parcours différents partant de la case $(1,1)$ et menant à la case (i,j). Établir la relation de récurrence complète du calcul de $nparc(i,j)$. 15 - Q 1

Question 2. Écrire l'algorithme itératif associé au calcul de $nparc(n,m)$ pour n et m fixés. Donner la valeur de $nparc(5,7)$. 15 - Q 2

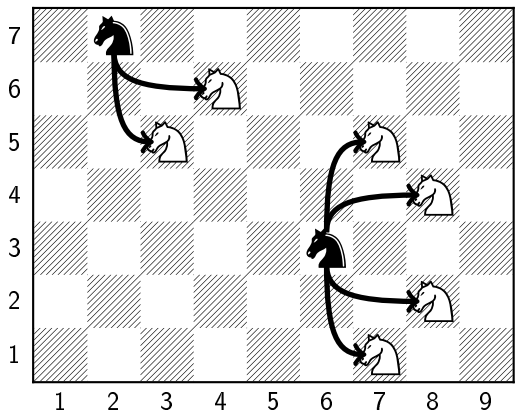


Fig. 1.9 – *Le cavalier situé en* $(7,2)$ *(resp.* $(3,6)$*), tracé en noir, peut aller en deux (resp. quatre) positions, tracées en blanc.*

Exercice 16. Nombre de partitions à p blocs d'un ensemble à n éléments ○ •

L'objectif essentiel de cet exercice est d'établir la récurrence des nombres de Stirling.

On note $\mathcal{S}(p,n)$ le nombre de partitions à p blocs d'un ensemble à n éléments. Les valeurs $\mathcal{S}(p,n)$ sont appelées *nombres de Stirling*. Par exemple, $\{\{a,c\},\{b,d\}\}$ et $\{\{b\},\{a,c,d\}\}$ sont deux des sept partitions à deux blocs de l'ensemble à quatre éléments $\{a,b,c,d\}$.

16 - Q 1 **Question** 1. Donner toutes les partitions à un, deux, trois et quatre blocs de cet ensemble à quatre éléments (un bloc ne peut pas être vide).

16 - Q 2 **Question** 2. Quelle est la relation de récurrence définissant $\mathcal{S}(p,n)$?

16 - Q 3 **Question** 3. Proposer une progression du calcul de $\mathcal{S}(p,n)$, puis l'algorithme itératif correspondant.

16 - Q 4 **Question** 4. Donner la valeur de $\mathcal{S}(p,n)$ pour $1 \leqslant p \leqslant 5$ et $1 \leqslant n \leqslant 7$.

Exercice 17. La montée de l'escalier ○ •

Cet exercice vise d'une part à établir une récurrence, d'autre part à en effectuer le calcul algorithmique en évitant l'évaluation de certaines cellules.

On considère un escalier de m marches que l'on veut gravir avec des sauts de a_1 ou a_2 ou ... ou a_n marches. Le problème est de trouver le nombre $nbf(s, m)$ de façons différentes de gravir *exactement* les m marches en s sauts.

Par exemple, si $m = 12$ marches et si les sauts possibles sont $a_1 = 2, a_2 = 3$ et $a_3 = 5$ marches, on peut gravir exactement les 12 marches de l'escalier par (entre autres) les suites de sauts :

$(2,2,2,2,2,2)$, $(2,3,2,3,2)$, $(2,2,2,3,3)$,
$(3,3,3,3)$, $(2,2,3,5)$, $(3,2,5,2)$, $(2,2,5,3)$.

Question 1. Donner deux façons de gravir exactement un escalier de 12 marches en *exactement trois* sauts de $a_1 = 2$ ou $a_2 = 3$ ou $a_3 = 5$ marches. 17 - Q 1

Question 2. Donner la formule de récurrence du calcul de $nbf(s, m)$. 17 - Q 2

Question 3. Proposer une progression permettant le calcul de $nbf(s, m)$. En déduire le programme itératif associé. 17 - Q 3

Question 4. Calculer les valeurs $nbf(s, m)$ pour $m \leqslant 12$, $s \leqslant 6$, $n = 2$, $a_1 = 2, a_2 = 5$. 17 - Q 4

Question 5. On remarque dans l'exemple précédent que certaines zones du tableau associé à nbf valent 0. Expliquer pourquoi et proposer une amélioration de l'algorithme. 17 - Q 5

Exercice 18. Le jeu patagon

Cet exercice introduit le jeu patagon, *auquel on joue seul pour amasser un montant maximal. On cherche à dénombrer les façons de jouer dites raisonnables, qui respectent la règle du jeu et sont susceptibles de conduire au gain maximal. L'exercice 144, page 380, s'intéresse à déterminer la (une) façon de jouer permettant d'atteindre le gain maximal.*

Dans le *jeu patagon*, le joueur est devant n objets, disposés en ligne. Chaque objet a une certaine valeur qui est visible. Le joueur peut prendre autant d'objets qu'il veut en cherchant naturellement à amasser une valeur totale aussi grande que possible. Il y a cependant une contrainte (et une seule) : le joueur n'a pas le droit de prendre deux objets placés l'un à côté de l'autre dans la configuration initiale.

Question 1. Avec la ligne d'objets suivante, où un objet est représenté par sa valeur, quel sera le meilleur choix ? 18 - Q 1

23	41	40	21	42	24

Question 2. On convient de représenter le choix du joueur par le vecteur caractéristique de taille n associé à ses choix. Dans l'exemple précédent, le vecteur $[0, 1, 0, 1, 0, 1]$ représente un jeu qui rapporte $0+41+0+21+0+24 = 86$ unités de monnaie patagone. Montrer que pour $n > 1$, il y a strictement moins de 2^n façons différentes de jouer. 18 - Q 2

18 - Q 3 **Question** 3. Certaines façons de jouer sont à coup sûr moins bonnes que d'autres. Par exemple, jouer $[0,1,0,1,0,0]$ (qui rapporte 62) est moins bon que $[0,1,0,1,0,1]$. On appelle *raisonnable* une façon de jouer qui, tout en respectant la règle, ne laisse pas d'objets qu'il aurait été possible de prendre. C'est ainsi que jouer $[0,1,0,1,0,1]$ est raisonnable, tandis que jouer $[0,1,0,0,0,1]$ ou $[0,1,0,1,0,0]$ ne l'est pas. Comment caractériser les façons raisonnables de jouer ?

18 - Q 4 **Question** 4. La suite de l'exercice a pour but de dénombrer les façons raisonnables de jouer, autrement dit le nombre de vecteurs binaires raisonnables de taille n. Pour un jeu de taille n, on définit $nfr_0(n)$ comme le nombre de façons raisonnables de jouer dont la valeur en position n du vecteur associé vaut 0. De même, on définit $nfr_1(n)$ comme le nombre de façons raisonnables de jouer dont la valeur en position n du vecteur associé vaut 1. Pour un jeu de taille n, le nombre total $nfr(n)$ de façons raisonnables de jouer vaut donc la somme de ces deux valeurs.

Montrer que :

$$nfr_0(1) = 0,\ nfr_1(1) = 1,\ nfr_0(2) = 1,\ nfr_1(2) = 1,\ nfr_0(3) = 1,\ nfr_1(3) = 1$$

et que pour $n > 3$:

$$nfr_0(n) = nfr_1(n-1)$$
$$nfr_1(n) = nfr_1(n-2) + nfr_1(n-3).$$

18 - Q 5 **Question** 5. Montrer que :

$$nfr(1) = 1,\ nfr(2) = 2,\ nfr(3) = 2$$

et que pour $n > 3$:

$$nfr(n) = nfr(n-2) + nfr(n-3).$$

18 - Q 6 **Question** 6. Donner les valeurs de $nfr_0(n), nfr_1(n)$ et $nfr(n)$ pour n allant de 1 à 15. Quelle relation indépendante de nfr_0 a-t-on entre nfr et nfr_1 ? Aurait-on pu l'établir avant ?

18 - Q 7 **Question** 7. Donner l'algorithme de calcul de $nfr(n)$ pour n fixé.

Exercice 19. Le jeu à deux tas de jetons ◦ ⁝

Dans cet exercice, on s'intéresse au nombre nfg *de façons de gagner dans un jeu où l'on ajoute et soustrait des jetons situés sur deux tas. Une propriété du nombre* nfg *est mise en évidence et prouvée par récurrence.*

On considère un jeu où l'on dispose de deux tas de jetons P et Q contenant respectivement p et q jetons ($p, q > 1$). Le but est d'atteindre une des deux situations, ($p = 0, q = 1$) ou ($p = 1, q = 0$), appelée *état gagnant* dans la suite. On passe d'un état des tas à un autre de la façon suivante : on enlève deux jetons à P (resp. Q), puis on jette un jeton et on ajoute l'autre à Q (resp. P).

Exemple. $(p = 4, q = 6) \longrightarrow (p = 2, q = 7)$ ou $(p = 5, q = 4)$.
On veut calculer le nombre de façons différentes $nfg(p, q)$ permettant d'atteindre l'un des deux états gagnants à partir d'une situation où le tas P a p jetons et le tas Q en a q.

Question 1. Donner la formule de récurrence exprimant $nfg(p, q)$. 19 - Q 1

Question 2. Proposer une structure tabulaire associée au calcul de nfg ainsi qu'une évolution de son remplissage. 19 - Q 2

Question 3. En déduire un algorithme itératif de calcul du nombre de façons différentes permettant d'atteindre l'un des deux états gagnants du jeu. 19 - Q 3

Question 4. Appliquer cet algorithme au calcul de la valeur $nfg(4, 2)$. 19 - Q 4

Question 5. Montrer que tout élément $nfg(i, j)$, tel que $|i - j|$ est multiple de 3, prend la valeur 0 (à l'exclusion de $nfg(0, 0)$, qui n'a pas de sens). 19 - Q 5

Exercice 20. Les pièces jaunes ◦ ⁝

Cet exercice pose un problème apparemment simple, le dénombrement des différentes façons de former un euro avec des pièces jaunes. On montre que la démarche intuitive visant à établir une récurrence à un indice n'est pas appropriée et qu'il est judicieux de recourir à une récurrence à deux indices. La démarche retenue ici trouve un prolongement dans l'exercice 147, page 384.

On souhaite connaître le nombre de façons *différentes* de former la somme d'un euro avec des pièces jaunes, autrement dit en utilisant des pièces de un, deux, cinq, dix, 20 et 50 centimes.

Question 1. Raisonnons d'abord pour former non pas un euro, mais toute somme allant de un à neuf centimes. Pour former un centime, il y a une seule solution. Pour deux centimes, on peut prendre une pièce de deux centimes, ou une pièce de un centime et il reste à former un centime. Pour former la somme s de trois ou quatre centimes, on prend une pièce de un centime et il reste à former la somme $(s-1)$, ou on prend une pièce de deux centimes et il reste à former la somme $(s-2)$. Pour former cinq centimes, on prend une pièce de cinq centimes, ou une pièce de un centime et il reste à former quatre centimes, ou une pièce de deux centimes et il reste à former trois centimes. Enfin, pour former la somme s de six à neuf centimes, on prend une pièce de cinq centimes et il reste à former la somme $(s-5)$, ou une pièce de un centime et il reste à former la somme $(s-1)$, ou une pièce de deux centimes et il reste à former la somme $(s-2)$. On en déduit la récurrence : 20 - Q 1

$$
\begin{array}{ll}
nbf(1) = 1 & \\
nbf(2) = 1 + nbf(1) & \\
nbf(i) = nbf(i-1) + nbf(i-2) & 3 \leqslant i \leqslant 4 \\
nbf(5) = 1 + nbf(4) + nbf(3) & \\
nbf(i) = nbf(i-5) + nbf(i-2) + nbf(i-1) & 6 \leqslant i \leqslant 9.
\end{array}
$$

Expliquer pourquoi cette récurrence ne convient pas.

20 - Q 2 **Question** 2. Proposer une récurrence *à deux indices* calculant le nombre de façons de former un euro avec les pièces jaunes.

20 - Q 3 **Question** 3. Vérifier qu'il y a deux façons de former trois centimes avec l'ensemble des pièces jaunes.

20 - Q 4 **Question** 4. Écrire le programme itératif effectuant le calcul du nombre de façons de former un euro avec les pièces jaunes.

20 - Q 5 **Question** 5. Quel est le nombre de façons différentes de former un euro avec les pièces jaunes ?

20 - Q 6 **Question** 6. Prouver que le nombre de façons de former un montant m avec *l'ensemble des pièces jaunes* croit avec m.

Exercice 21. Mélange de mots ◦ ⁝

On se définit une opération de mélange de deux mots et on veut d'une part déterminer le nombre de mélanges possibles de deux mots donnés, d'autre part décider si un mot est ou non un mélange de deux autres.

On se donne trois mots : u de longueur m, v de longueur n et w de longueur $(m+n)$. Le mot w est appelé *mélange* des mots u et v s'il est formé en mélangeant les lettres de u et de v tout en préservant l'ordre des lettres de u et de v. Par exemple, pour $u = \textit{lait}$ et $v = \textit{cafe}$, le mot $w = \textit{calfaite}$ est un mélange de u et de v, alors que le mot *aclfatie* n'en est pas un.

21 - Q 1 **Question** 1. Donner une relation de récurrence pour le calcul du nombre $\mathrm{nbmlg}(m,n)$ de mélanges différents que l'on peut construire à partir de u et v, ou plus généralement à partir de tout couple de mots de longueurs m et n. Calculer $\mathrm{nbmlg}(5,4)$.

21 - Q 2 **Question** 2. Montrer par récurrence que $\mathrm{nbmlg}(m,n) = (m+n)!/(m! \cdot n!)$. Aurait-on pu trouver directement ce résultat ?

21 - Q 3 **Question** 3. Donner un algorithme permettant de décider si un mot w est un mélange des mots u et v. L'appliquer au cas des mots $u = \textit{abc}$, $v = \textit{db}$ et $w = \textit{dabbc}$.

CHAPITRE 2

Complexité d'un algorithme

> Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.
>
> *(Alan Perlis)*

2.1 Rappels

2.1.1 Algorithme

Rappelons d'abord que l'on définit un *algorithme* comme un ensemble de règles permettant de résoudre un problème sur des données d'entrée. Cet ensemble de règles définit avec précision une séquence d'opérations qui se termine dans un temps fini.

Par exemple, soit le problème consistant à trouver le résultat de la multiplication de deux nombres entiers positifs écrits en base 10. Nous avons tous appris un certain algorithme pour résoudre ce problème, fondé sur une suite précise d'opérations élémentaires : l'utilisation de la table de multiplication, l'addition, l'écriture de la retenue, etc. Nous savons que cet algorithme permet de calculer le produit de tout couple de nombres en un temps fini, fonction de la longueur des nombres.

Il est à remarquer que la technique de multiplication à l'aide d'un boulier produit le même résultat avec un procédé assez proche, mais non identique. Il existe encore d'autres algorithmes pour résoudre le même problème, parfois fondés sur des techniques bien différentes, comme la multiplication dite *à la russe*.

Ces algorithmes sont assez simples pour être appris par le plus grand nombre, et assez efficaces, au sens où ils fournissent le résultat cherché rapidement et ne nécessitent ni une grande mémoire, ni l'écriture de grandes quantités de chiffres. En effet, comparons-les avec l'algorithme naïf suivant : pour multiplier un nombre n par un nombre m, écrire n lignes, de chacune m croix, les unes sous les autres, puis compter le nombre de croix. Ce dernier procédé ne demande que de savoir compter, mais il est terriblement coûteux en temps et en place.

2.1.2 Algorithmique, complexité d'un algorithme

Nous venons de voir qu'il existe plusieurs méthodes (ou algorithmes) pour effectuer la multiplication de deux nombres. C'est généralement le cas pour de nombreux problèmes, et il est utile de pouvoir comparer les divers algorithmes qui, équivalents sur le plan fonctionnel, peuvent être bien différents au niveau de leur efficacité. L'*algorithmique*,

science de la production des algorithmes, a pour but principal, pour un problème donné, de trouver un algorithme aussi efficace que possible.

Il est naturel de quantifier l'efficacité d'un algorithme en mesurant ce que l'on appelle sa complexité *en temps* (ou *temporelle*) et sa complexité *en espace* mémoire (ou *spatiale*). Cependant, parce que la taille de la mémoire des ordinateurs a énormément augmenté au cours des années, la complexité en espace est un paramètre de moins en moins critique. C'est pourquoi, dans la suite de cet ouvrage, nous considèrerons le plus souvent uniquement la complexité en temps.

Compte tenu de cette remarque, pour l'essentiel, le but de l'algorithmique devient donc de chercher, pour un problème donné, un algorithme le plus rapide possible. La question légitime qui se pose est donc de *mesurer* cette rapidité.

Un premier point à noter est qu'il est souhaitable de mesurer la complexité d'un algorithme indépendamment à la fois du langage dans lequel il est codé et de la machine sur laquelle il est exécuté. On va donc analyser la complexité selon une ou plusieurs *opération(s) élémentaire(s)*, qui varie(nt) selon les problèmes, mais est (sont) représentative(s) de ce que fait l'algorithme considéré. Ainsi, pour un algorithme de recherche dans un tableau, l'opération élémentaire sera la comparaison, alors que pour un algorithme de tri, on aura deux opérations élémentaires à considérer : la comparaison et l'échange de deux données. Enfin, pour un produit de deux matrices, la multiplication de deux nombres réels constituera l'opération élémentaire.

Un second point est que la complexité d'un algorithme s'exprime en fonction de la *taille des données* qu'il a à traiter. Cette taille est un nombre entier caractéristique du problème. Pour un algorithme de recherche dans un tableau ou de tri d'un tableau, ce sera le nombre d'éléments (ou taille) du tableau ; pour le produit de deux matrices carrées (forcément de même taille), ce sera le nombre de lignes ou de colonnes des matrices.

Il est souvent impossible de donner la complexité exacte d'un algorithme comme le nombre d'opérations élémentaires en fonction de la taille des données. En effet, la complexité dépend la plupart du temps des données elles-mêmes. Par exemple, certains algorithmes de tri fonctionnent très rapidement si le tableau est déjà à peu près trié et beaucoup plus lentement si le tableau est trié à l'envers. C'est pour cette raison que l'on doit souvent renoncer à calculer une complexité exacte, mais que l'on s'intéresse à une complexité *minimale* ou *au mieux* (quand les données sont favorables) et une complexité *maximale* ou *au pire*. Il est tentant de définir une complexité *moyenne*, mais c'est en général difficile et cela nécessite de faire des hypothèses statistiques parfois arbitraires sur la répartition des données.

Une autre analyse en moyenne est celle de la complexité amortie, pour laquelle nous renvoyons, comme pour la complexité moyenne, aux ouvrages [17] et [35].

Une autre manière de faire l'analyse en temps d'un algorithme est d'utiliser la notion d'*ordre de grandeur de complexité*, qui exprime le comportement de la complexité quand la taille des données devient « grande ».

Dans les deux sections suivantes, nous présentons les notions de complexité minimale et maximale, puis l'analyse par ordre de grandeur de complexité.

2.1.3 Complexité minimale et maximale d'un algorithme

Nous allons traiter ces notions avec un exemple, celui de la recherche dans un dictionnaire. Supposons que nous voulions rechercher un mot dans un dictionnaire, pour en connaître la définition s'il s'y trouve, ou pour conclure qu'il n'est pas dans ce dictionnaire.

Appelons n la taille des données, c'est-à-dire le nombre de mots dans le dictionnaire. Convenons que l'opération élémentaire est la comparaison de deux mots et appelons x le mot recherché.

Un premier algorithme consiste à parcourir les mots du dictionnaire du début à la fin dans l'ordre, jusqu'à la terminaison du processus. Cette terminaison peut prendre deux formes : soit on trouve le mot et on aura effectué un nombre de comparaisons inférieur ou égal à n, soit on ne le trouve pas et il aura fallu consulter tout le dictionnaire. Pour matérialiser ce second cas, une technique possible est celle de la *sentinelle* : on allonge le dictionnaire du mot cherché x. Avec cette technique, on trouve finalement toujours x dans le dictionnaire et on sait quand on le trouve à la $(n+1)^e$ comparaison qu'il n'est pas dans le dictionnaire original.

L'algorithme *RechDict1* ci-dessous prend en donnée le dictionnaire sous la forme du tableau DIC[1 .. n] de mots tous différents, classés par ordre lexicographique croissant, et un mot x. Si x est dans DIC, l'algorithme donne en résultat l'indice où x se trouve, sinon il produit la valeur $(n+1)$.

```
1.  constantes
2.     n ∈ ℕ1 et n = ... et x ∈ chaîne et x = ...
3.  variables
4.     DIC ∈ 1 .. n+1 → chaîne et DIC = [...]
5.  début
6.     DIC[n+1] ← x ; i ← 1 ;
7.     tant que DIC[i] ≠ x faire
8.        i ← i+1
9.     fin tant que ;
10.    si i < n+1 alors
11.       écrire(le mot , x, est en position , i, du dictionnaire)
12.    sinon
13.       écrire(le mot , x, n'est pas dans le dictionnaire)
14.    fin si
15. fin
```

Quelle est la complexité pratique de cet algorithme ? On sait déjà qu'au pire elle est de $(n+1)$ comparaisons, le cas le pire étant quand x n'est pas dans DIC. De même, la complexité minimale est d'une comparaison, si par bonheur le mot cherché est le premier du dictionnaire. La complexité en moyenne peut être calculée sous des hypothèses statistiques simples et il s'avère qu'elle n'est pas égale à $n/2$ (voir exercice 27, page 60).

Puisqu'un dictionnaire est trié, comme chacun sait, il existe une méthode plus rapide pour y chercher un mot : la recherche dichotomique. Il en existe plusieurs variantes (voir par exemple l'exercice 87, page 249) ; nous proposons la version itérative suivante. On prend le mot situé « au milieu » du dictionnaire et on le compare à x. Si x est plus grand que (resp. plus petit que ou égal à) l'élément du milieu au sens de l'ordre lexicographique, on recommence l'opération dans le demi-dictionnaire supérieur (resp. inférieur) jusqu'à atteindre une partie de dictionnaire ne contenant qu'un mot. On conclut positivement si celui-ci est le mot x cherché, sinon x n'est pas dans le dictionnaire.

Le programme itératif *RechDict2* ci-après réalise ce qui vient d'être décrit :

```
1.  constantes
2.     x ∈ chaîne et x = ... et n ∈ ℕ1 et n = ...
3.  variables
4.     deb ∈ ℕ1 et fin ∈ ℕ1 et fin ⩾ deb et mil ∈ ℕ1 et mil ⩾ deb et
5.     mil ⩽ fin et DIC ∈ 1 .. n → chaîne et DIC = [...]
6.  début
7.     deb ← 1 ; fin ← n ;
8.     tant que deb ≠ fin faire
9.        mil ← ⌊(deb + fin) / 2⌋ ;
10.       si x > T[mil] alors
11.          deb ← mil + 1
12.       sinon
13.          fin ← mil
14.       fin si
15.    fin tant que ;
16.    si x = T[deb] alors
17.       écrire(le mot , x, est en position , deb, du dictionnaire)
18.    sinon
19.       écrire(le mot , x, n'est pas dans le dictionnaire)
20.    fin si
21. fin
```

Avec cet algorithme, il n'y a ni cas favorable (associé à la complexité minimale ou au mieux), ni cas défavorable (correspondant à la complexité maximale ou au pire). En effet, le nombre de comparaisons de mots ne dépend que de la taille n du dictionnaire DIC et non de son contenu. Puisque l'on divise systématiquement la taille de la zone de recherche par deux, on peut admettre (nous y reviendrons plus loin) que le nombre de comparaisons de mots est de l'ordre de $\lfloor \log_2(n) \rfloor$. Si l'on admet que la complexité en moyenne de *RechDict1* est linéaire, *RechDict2* effectuant un nombre de comparaisons logarithmique est de ce point de vue meilleur, comme escompté.

Pour un dictionnaire classique de la langue française, n se situe autour de 30 000 et le nombre de comparaisons de l'algorithme *RechDict2* ($\lfloor \log_2(n) \rfloor$) vaut donc 15 ou 16.

Notons que la variante de *RechDict2* dans laquelle l'alternative distinguerait le cas de l'égalité entre x et T[mil] (et provoque alors l'arrêt) a une complexité minimale d'une comparaison. Cependant, il convient de remarquer que : i) la configuration de données favorable n'est pas la même que pour *RechDict1*, ii) on effectue une comparaison supplémentaire par pas d'itération et le nombre de comparaisons de mots dans le pire cas passe donc à $2 \cdot \lfloor \log_2(n) \rfloor$.

2.1.4 Ordres de grandeur de complexité

La manière la plus courante d'analyser le temps que prend un algorithme est d'utiliser la notion d'*ordre de grandeur* de complexité (on parle aussi de *classe de complexité*). On note $f_{\mathcal{A}}(n)$ la fonction qui exprime la complexité maximale d'un algorithme $\mathcal{A}$ en fonction de la taille n des données qu'il traite. L'idée de départ est de majorer $f_{\mathcal{A}}(n)$ par une

fonction à la croissance connue. À cet effet, on considère un certain nombre de fonctions de référence dont la croissance est connue :

- 1 (complexité constante),
- $\log_2(n)$ (complexité logarithmique),
- n (complexité linéaire),
- $n \cdot \log_2(n)$ (complexité quasi linéaire),
- n^2 (complexité quadratique),
- n^p (avec $p > 2$) (complexité polynomiale),
- 2^n (complexité exponentielle),
- $n!$,
- n^n.

On pourrait chercher à écrire une formule du type $f_{\mathcal{A}}(n) \in \mathcal{O}(n^2)$ pour signifier (avec des précautions que nous allons préciser) que $f_{\mathcal{A}}(n)$ ne croît pas plus vite que n^2 quand n augmente. Cependant, cette première approche serait très restrictive puisque, par exemple, $f_{\mathcal{A}}(n) = n^2/2+3n/2+1$ ou $f_{\mathcal{A}}(n) = 2n^2-1$ n'obéiraient pas à la définition. On révise donc l'acception de $f_{\mathcal{A}}(n) \in \mathcal{O}(n^2)$ de sorte que les deux exemples précédents soient acceptables. Pour le premier, on va dire que $f_{\mathcal{A}}(n)$ ne croît pas plus vite que la fonction de référence (ici n^2) quand n augmente, *à partir d'un certain rang* n_0. Ainsi, on peut écrire $(n^2/2 + 3n/2 + 1) \in \mathcal{O}(n^2)$, puisqu'à partir de $n_0 = 4$ il est vrai que $(n^2/2 + 3n/2 + 1) < n^2$. Cependant, nous ne pouvons toujours pas écrire que $(2n^2 - 1) \in \mathcal{O}(n^2)$, puisque pour tout $n > 1 : (2n^2 - 1) > n^2$. Pourtant, l'ordre de grandeur de croissance de $(2n^2 - 1)$ est visiblement du même type que celui de n^2, et il semble consistant de dire que $(2n^2 - 1)$ et n^2 croissent de la même manière. Nous dirons finalement que $f_{\mathcal{A}}(n) \in \mathcal{O}(n^2)$, s'il existe une constante C et un entier n_0 au-dessus duquel l'inégalité $f_{\mathcal{A}}(n) \leqslant C \cdot n^2$ est toujours vraie. Finalement, nous arrivons à la définition suivante. Soit une fonction $g : \mathbb{N} \to \mathbb{R}_+$. On appelle *ordre de grandeur maximal de complexité* de g l'ensemble :

$$\mathcal{O}(g) \mathrel{\widehat{=}} \{t : \mathbb{N} \to \mathbb{R}_+ \mid \exists(C, n_0) \cdot (C \in \mathbb{R}_+ \text{ et } n_0 \in \mathbb{N} \text{ et } \forall n \cdot ((n \in \mathbb{N} \text{ et } n \geqslant n_0) \Rightarrow t(n) \leqslant C \cdot g(n)))\}.$$

Avec cette définition, on a bien $(n^2/2+3n/2+1) \in \mathcal{O}(n^2)$, mais aussi $(2n^2-1) \in \mathcal{O}(n^2)$. Puisque $(2n^2 - 1) \in \mathcal{O}(n^2)$, on a aussi $(2n^2 - 1) \in \mathcal{O}(n^3)$. En pratique, on cherche la fonction de référence *la plus proche* de $f_{\mathcal{A}}(n)$.

Une fois défini l'ensemble $\mathcal{O}(g)$ des fonctions qui croissent au plus aussi vite que g (avec un sens désormais précis), il est logique de définir de même l'ensemble $\Omega(g)$ des fonctions qui croissent au moins aussi vite que g, ce qui permet en association avec $\mathcal{O}(g)$ d'encadrer le comportement d'un algorithme.

On appelle *ordre de grandeur minimal de complexité* de g l'ensemble :

$$\Omega(g) \mathrel{\widehat{=}} \{t : \mathbb{N} \to \mathbb{R}_+ \mid \exists(D, n_0) \cdot (D \in \mathbb{R}_+ \text{ et } n_0 \in \mathbb{N} \text{ et } \forall n \cdot (n \geqslant n_0 \Rightarrow t(n) \geqslant D \cdot g(n)))\}.$$

Pour finir, on appelle *ordre de grandeur exact de complexité* de g l'ensemble $\Theta(g)$ défini par l'intersection des deux ordres de grandeur maximal et minimal :

$$t \in \Theta(g) \mathrel{\widehat{=}} t \in \mathcal{O}(g) \text{ et } t \in \Omega(g)$$

ou encore :

$$\Theta(g) \mathrel{\widehat{=}} \{t : \mathbb{N} \to \mathbb{R}_+ \mid \exists(C, D, n_0) \cdot (C \in \mathbb{R}_+ \text{ et } D \in \mathbb{R}_+ \text{ et } n_0 \in \mathbb{N} \text{ et } \forall n \cdot (n \geqslant n_0 \Rightarrow D \cdot g(n) \leqslant t(n) \leqslant C \cdot g(n)))\}.$$

$f \in \mathcal{O}(g)$ (resp. $\Omega(g), \Theta(g)$) se lit souvent « f est en grand o de g » (resp. en Ω de g, en Θ de g).

Résultat intéressant Soit $\mathcal{A}$ un algorithme constitué de deux parties consécutives $\mathcal{A}_1$ et $\mathcal{A}_2$, dont les complexités respectives sont telles que $f_{\mathcal{A}_1}(n) \in \mathcal{O}(f_1(n))$ et $f_{\mathcal{A}_2}(n) \in \mathcal{O}(f_2(n))$; alors $f_{\mathcal{A}}(n) \in \max(\{\mathcal{O}(f_1(n), \mathcal{O}(f_2(n))\})$. Autrement dit, l'ordre de grandeur de complexité maximale d'un algorithme est donné par celui de sa partie d'ordre de grandeur de complexité maximale le plus élevé (voir exercice 23, page 58).

Remarques

1. $f \in \mathcal{O}(1)$ signifie qu'à partir d'un certain rang n_0 il existe une constante C telle que $f(n) \leqslant C$.
2. La fonction de complexité $f_{\mathcal{A}}(n)$ de tout algorithme $\mathcal{A}$ est telle que $f_{\mathcal{A}}(n) \in \Omega(1)$.
3. Pour tout entier $a \geqslant 2$, on a : $\mathcal{O}(\log_a(n)) = \mathcal{O}(\log_2(n))$ et $\Theta(\log_a(n)) = \Theta(\log_2(n))$ puisque $\log_a(n) = \log_2(a) \cdot \log_2(n)$. En général, on prend la fonction de référence $\log_2(n)$.
4. La formule de Stirling
$$n! \approx \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n$$
permet d'écrire (en prenant le logarithme de chaque membre) que $\log_2(n!) \in \Theta(n \cdot \log_2(n))$.
5. Un ordre de grandeur de complexité n'est pas une limite. Par exemple, $f(n) = n^2 \cdot (2 + \sin^2(n)) \in \Theta(n^2)$ mais n'a pas de limite quand n croît vers l'infini. De plus, les fonctions traitées par les ordres de grandeur de complexité ne sont pas de $\mathbb{R}$ dans $\mathbb{R}$ mais de $\mathbb{N}$ dans $\mathbb{R}_+$ puisque la taille des données d'un programme est une valeur entière et qu'une complexité est par nature positive.

2.1.5 Quelques exemples de calcul de complexité

La complexité des algorithmes mis en évidence sera étudiée de façon systématique dans chacun des chapitres suivants. On va donc se limiter à l'illustrer avec quelques-uns des algorithmes rencontrés jusqu'ici. Il est aisé de statuer sur les algorithmes suivants :

- le calcul de factorielle (voir page 13) dont la complexité en nombre de multiplications est en $\Theta(n)$,
- le calcul du nombre d'arbres binaires ayant n nœuds (voir page 13) dont la complexité en nombre de multiplications est donnée par :
$$\sum_{i=1}^{n} \sum_{j=0}^{i-1} 1 = \frac{n \cdot (n+1)}{2}$$
donc en $\Theta(n^2)$,
- le calcul du nombre de Delannoy d'ordre (m, n) donné par la récurrence de la page 14, dont la complexité en nombre d'additions est exprimée par :
$$\sum_{i=1}^{n} \sum_{j=1}^{m} 2 = 2 \cdot m \cdot n$$
donc en $\Theta(m \cdot n)$,
- le calcul du nombre de partitions à p blocs d'un ensemble à n éléments (voir exercice 16, page 44) dont la complexité en nombre d'additions (ou de multiplications) est donnée par :

$$\sum_{j=2}^{n}\sum_{i=2}^{j-1} 1 = \frac{(n-1)(n-2)}{2}$$

donc en $\Theta(n^2)$,

— l'algorithme *RechDict1* qui est en $\mathcal{O}(n)$ comparaisons,

— la multiplication de deux matrices carrées qui est en $\Theta(n^3)$ multiplications de deux réels.

Pour la variante de l'algorithme *RechDict2*, il faut déterminer $nbpmax(n)$, le nombre maximal de pas de la boucle. Celui-ci est donné par la récurrence :

$$\begin{aligned} &nbpasmax(1) = 1 \\ &nbpasmax(n) = 1 + nbpasmax\left(\left\lfloor \frac{n}{2} \right\rfloor\right) && n > 1 \end{aligned}$$

dont la solution est $nbpasmax(n) = \lfloor \log_2(n) \rfloor$. Le nombre maximal de comparaisons de mots est donc $2 \cdot \lfloor \log_2(n) \rfloor + 1$; cet algorithme est en $\mathcal{O}(\log_2(n))$.

Ce dernier exemple met en évidence le rôle souvent central des relations de récurrence pour le calcul de la complexité. Celles-ci seront utilisées de façon intensive, notamment pour établir la complexité des algorithmes récursifs dans les chapitres 4 et 8.

2.1.6 À PROPOS DES OPÉRATIONS ÉLÉMENTAIRES

Dans la suite et en particulier dans les exercices, on s'intéresse à la complexité temporelle asymptotique d'algorithmes et parfois à leur complexité exacte. Il est fréquent que la nature même de l'algorithme conduise au choix d'une opération élémentaire unique, mais parfois plusieurs opérations significatives apparaissent. On distingue alors deux situations :

— celles-ci sont corrélées et on en choisit une (plus ou moins arbitrairement), comme par exemple dans le produit de matrices où additions et multiplications sont liées, mais aussi dans le calcul de la somme des éléments d'un tableau où l'addition (au cœur de la somme) et la comparaison (liée au contrôle de la boucle) sont en nombre ne différant que d'une unité,

— les opérations sont indépendantes et, soit on les dénombre séparément, soit on se focalise sur celle ayant le plus grand nombre d'occurrences.

Dans les exercices proposés et/ou traités, l'(les) opération(s) élémentaire(s) est (sont) le plus souvent explicitée(s) dans l'énoncé. Cet aspect est quelquefois laissé volontairement dans l'ombre et il importe alors de le développer de façon appropriée dans la réponse.

Il existe de nombreux algorithmes de tri qui, dans l'ensemble, font appel à deux opérations élémentaires : condition et échange (voir par exemple l'exercice 33 page 82). Dans certains exercices, un tri est une étape nécessaire et on pourra admettre que l'on utilise l'un des plus efficaces au pire, dont la complexité temporelle asymptotique est quasi linéaire ($n \cdot \log_2(n)$) en nombre de comparaisons et/ou échanges [1].

Il arrive que l'on souhaite comparer la complexité temporelle de plusieurs algorithmes résolvant un même problème. Il importe alors de choisir une (des) opération(s) élémentaire(s) identique(s) pour chacun d'eux afin de rendre la comparaison possible.

L'évaluation de conditions, simples (on parle alors de comparaisons) ou complexes, explicites (cas des alternatives ou des boucles **tant que**) ou implicites (boucles **pour**), joue

1. voir fr.wikipedia.org/wiki/Algorithme_de_tri

souvent un rôle prépondérant dans nombre d'algorithmes. Dès lors, elle peut être choisie à juste titre comme opération élémentaire, d'autant plus lorsque aucune autre opération ne s'impose pour l'algorithme considéré.

2.1.7 Temps de calcul pratique

Chaque case du tableau ci-dessous indique la taille approximative des données que peut traiter un algorithme dont la complexité exacte est en colonne, dans le temps donné en ligne, pour un temps élémentaire d'instruction de 1µs. Par exemple, en une heure, un algorithme en n^3 peut traiter un problème de taille 1500. Une taille « astronomique » est supérieure au nombre estimé d'atomes dans l'univers (10^{80}).

taille	$\log_{10}(n)$	n	n^2	n^3	2^n	$n!$
durée = 1 s	astronomique : 10^{10^6}	10^6	10^3	10^2	19	10
1 mn	astronomique	6.10^7	8.10^3	4.10^2	25	11
1 h	astronomique	4.10^8	2.10^4	1500	31	13
1 j	astronomique	9.10^{10}	10^5	4400	36	16

Pour le même temps élémentaire d'instruction de 1µs, le tableau suivant donne le temps qu'il faut à un algorithme de complexité exacte indiquée en colonne pour traiter des données de la taille donnée en ligne. Par exemple, un programme en n^2 peut traiter un problème de taille 1000 en une seconde. Un temps « astronomique » est supérieur à un milliard de milliard d'années.

complexité	$\log_2(n)$	n	$n\log n$	n^2	2^n
taille = 10	3µs	10µs	30µs	100µs	1000µs
100	7µs	100µs	700µs	1/100s	10^{14} siècles
1000	10µs	1000µs	1/100s	1s	astronomique
10000	13µs	1/100s	1/7s	1,7mn	astronomique
100000	17µs	1/10s	2s	2,8h	astronomique

2.1.8 Problèmes pseudo-polynomiaux

Rappelons que nous évaluons l'ordre de grandeur de complexité en fonction de la taille n du problème, qui est indépendante des données sur lesquelles travaille l'algorithme. Ainsi, le produit de deux matrices réelles $(n \times n)$ se calcule par un algorithme en $\Theta(n^3)$ multiplications de nombres réels, quelles que soient les valeurs des coefficients des matrices.

Il existe cependant certains algorithmes dont on évalue la complexité en tenant compte des données. Sans rentrer dans les détails (ce sera fait au chapitre 9), le problème dit du *petit commerçant* (exercice 147, page 384) propose de chercher à totaliser une somme de N centimes en utilisant un nombre minimal de pièces de monnaie. Par exemple, pour $N = 6$, la meilleure manière est de rendre une pièce de un centime et une pièce de cinq centimes, soit au total deux pièces, plutôt que – par exemple – trois pièces de deux centimes. Si l'on dispose de n pièces de valeur différente (en quantités illimitées), il s'avère que l'on ne connaît pas de méthode polynomiale en n qui fonctionne pour tout système monétaire, mais seulement une méthode exponentielle, donc en $\mathcal{O}(\in^{\backslash})$, méthode que nous appellerons $\mathcal{A}_1$.

En revanche, on connaît un algorithme (appelons-le $\mathcal{A}_2$) qui calcule l'optimum cherché en temps $\Theta(N \cdot n)$. Mais, attention, la valeur N est une donnée, pas une taille du problème. Par conséquent, le temps de traitement, obtenu de cette façon, dépend des caractéristiques de chaque occurrence du problème, ce qui est en principe contraire aux critères d'évaluation de la complexité d'un algorithme.

On pourrait croire que parvenir à l'expression $\Theta(N \cdot n)$ prouve que le problème est de complexité linéaire (polynomiale de degré 1) en n, grâce à l'algorithme $\mathcal{A}_2$. Mais c'est faux, car N n'est pas une constante, et ne peut pas être majoré *a priori* par une constante. Après tout, on pourrait demander à $\mathcal{A}_2$ de trouver la façon de décomposer de manière optimale la somme $N = 2^{2^n}$ ou $n!$. Dans ces cas, cet algorithme serait beaucoup moins performant que $\mathcal{A}_1$. En revanche, si N est une somme raisonnable (ce qui est le cas dans la plupart des applications pratiques), $\mathcal{A}_2$ est plus efficace que $\mathcal{A}_1$.

Les algorithmes comme $\mathcal{A}_2$, dont la complexité est du type $\Theta(N \cdot P(n))$, avec $P(n)$ un polynôme en n et N fonction des données, sont appelés *pseudo-polynomiaux*. Il n'est évidemment pas interdit de les étudier et de les utiliser, mais il faut garder à l'esprit que leur temps de calcul, contrairement ce que l'on attend en général d'un algorithme, dépend de manière directe et cruciale des données du problème traité.

Remarque finale Pour certains problèmes, la taille des données peut s'évaluer en fonction de plusieurs paramètres et non pas d'un seul. Par exemple, nous verrons au chapitre 9 le problème de « distribution de skis » qui consiste à choisir pour chaque skieur (parmi n) la meilleure paire de skis (parmi m) à lui affecter. Les valeurs de n et de m sont indépendantes, à la condition près que $m \geqslant n$. L'ordre de grandeur de complexité du meilleur algorithme connu pour ce problème est $\mathcal{O}(n \cdot m)$. Il est donc impossible de caractériser cette complexité en fonction d'un seul paramètre : dans un cas comme celui-ci, la taille du problème se mesure par une quantité fondamentalement multi-dimensionnelle et l'ordre de grandeur de complexité s'exprime comme un polynôme à plusieurs variables.

2.2 Exercices

Exercice 22. À propos de quelques fonctions de référence ⚷ •

Cet exercice permet de savoir si des fonctions de référence « voisines » caractérisent (ou non) le même ordre de grandeur de complexité.

Dire si les affirmations suivantes sont vraies, pour toute fonction f de $\mathbb{N}$ dans $\mathbb{R}_+$:

1. $2^{n+1} \in \mathcal{O}(2^n)$
2. $(n+1)! \in \mathcal{O}(n!)$
3. $f(n) \in \mathcal{O}(n) \Rightarrow (f(n))^2 \in \mathcal{O}(n^2)$
4. $f(n) \in \mathcal{O}(n) \Rightarrow 2^{f(n)} \in \mathcal{O}(2^n)$
5. $n^n \in \mathcal{O}(2^n)$.

Exercice 23. Propriété des ordres de grandeur $\mathcal{O}$ et Θ

Dans cet exercice, on établit une propriété intéressante relative à la somme de deux fonctions dont on connaît l'ordre de grandeur maximal ou exact. Ce résultat est utile en pratique pour décider de l'ordre de grandeur maximal ou exact d'un programme obtenu par composition séquentielle de plusieurs composants.

Soit $f_1(n)$ et $f_2(n)$ deux fonctions de $\mathbb{N}$ dans $\mathbb{R}_+$ telles que :

$$\forall n \cdot (n \geqslant n_0 \Rightarrow f_1(n) \leqslant f_2(n)),$$

comme $f_1(n) = \log_2(n)$ et $f_2(n) = n$ ou encore $f_1(n) = n^2$ et $f_2(n) = 2^n$.

23 - Q 1 **Question 1.** Montrer que si $g(n) \in \mathcal{O}(f_1(n))$ et $h(n) \in \mathcal{O}(f_2(n))$ alors $g(n) + h(n) \in \mathcal{O}(f_2(n))$.

23 - Q 2 **Question 2.** Montrer que si $g(n) \in \Theta(f_1(n))$ et $h(n) \in \Theta(f_2(n))$ alors $g(n) + h(n) \in \Theta(f_2(n))$.

Exercice 24. Variations sur les ordres de grandeur $\mathcal{O}$ et Θ

Le but principal de cet exercice est d'attirer l'attention sur le fait que l'on ne peut pas tirer de conclusion hâtive lors de la manipulation des ordres de grandeur.

Statuer sur les deux affirmations suivantes :

1. $f \in \Theta(s)$ et $g \in \Theta(s) \Rightarrow f - g \in \Theta(s)$
2. $f \in \mathcal{O}(s)$ et $g \in \mathcal{O}(r) \Rightarrow f - g \in \mathcal{O}(s - r)$.

Exercice 25. Ordre de grandeur : polynômes

Il est fréquent que la fonction de complexité d'un algorithme soit donnée par un polynôme. Dans cet exercice, on montre qu'une version simplifiée suffit pour exprimer l'ordre de grandeur de complexité d'un tel algorithme.

On considère un algorithme $\mathcal{A}$ dont la complexité s'exprime par un polynôme. On cherche à situer la complexité de $\mathcal{A}$ en termes de classe de complexité par un polynôme « simplifié ». On va d'abord considérer deux cas particuliers avant de passer au cas général.

Question 1. On considère les fonctions f de $\mathbb{N}_1$ dans $\mathbb{R}_+$ et g de D dans $\mathbb{R}_+$ (avec $D = \mathbb{N} - 0\,..\,2$) suivantes : a) $f(n) = n^3 + 3n^2 + 6n + 9$, b) $g(n) = n^3 - 3n^2 + 6n - 9$. Montrer que f et g appartiennent toutes deux à $\Theta(n^3)$. 25 - Q 1

Question 2. Soit $f(n) = a_p \cdot n^p + a_{p-1} \cdot n^{p-1} + \cdots + a_1 \cdot n + a_0$ une fonction de D dans $\mathbb{R}_+$, avec $a_i \in \mathbb{N}$ et D l'ensemble $\mathbb{N}$ éventuellement privé de ses premiers éléments pour lesquelles f ne prend pas une valeur positive. Prouver que pour $a_p > 0$, $f(n) \in \Theta(n^p)$. 25 - Q 2

Question 3. On suppose a, b et k entiers. Démontrer l'inégalité : 25 - Q 3

$$a^k + b^k \geqslant \left(\frac{a+b}{2}\right)^k. \tag{2.1}$$

En déduire que pour $k \in \mathbb{N}$ et $n \in \mathbb{N}_1$, on a :

$$f(n,k) = 1^k + 2^k + \cdots + n^k \in \Theta(n^{k+1}).$$

Exercice 26. Ordre de grandeur : paradoxe ? ◦ •

Dans l'exercice 23, page 58, on a vu une propriété de la somme de deux fonctions de complexité. L'étendre à une somme multiple est légitime, encore faut-il prendre garde à distinguer image d'une fonction et somme de fonctions.

Question 1. Montrer que : 26 - Q 1

$$\sum_{i=1}^{n} i = (1 + 2 + \cdots + n) \in \mathcal{O}(n^2).$$

Question 2. On considère le raisonnement suivant. On sait que : 26 - Q 2

$$\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}.$$

Or :

$$\sum_{i=1}^{n} i \in \mathcal{O}(1 + 2 + \cdots + n)$$

et

$$\mathcal{O}(1 + 2 + \cdots + n) = \mathcal{O}(\max(\{1, 2, \ldots, n\})) = \mathcal{O}(n).$$

Donc :

$$\frac{n \cdot (n+1)}{2} \in \mathcal{O}(n).$$

Où est l'erreur ?

Exercice 27. Un calcul de complexité en moyenne

Cet exercice a pour but de procéder à un calcul de complexité en moyenne. On y met en évidence un résultat qui, sans être inattendu, n'est pas celui qu'annoncerait l'intuition.

L'algorithme ci-dessous, voisin de celui vu pour la recherche séquentielle dans un dictionnaire, recherche la valeur x dans un tableau d'entiers T dont les n premiers éléments sont tous différents, le dernier jouant le rôle de sentinelle (auquel sera affectée la valeur x recherchée). Si x est dans $T[1\,..\,n]$, le résultat est l'indice où x se trouve, sinon la valeur $(n+1)$ est retournée.

```
1.  constantes
2.     x ∈ ℕ1 et x = ... et n ∈ ℕ1 et n = ...
3.  variables
4.     T ∈ 1..n+1 → ℕ1 et T = ... et i ∈ ℕ1
5.  début
6.     i ← 1 ; T[n+1] ← x ;
7.     tant que T[i] ≠ x faire
8.        i ← i+1
9.     fin tant que ;
10.    écrire(i)
11. fin
```

27 - Q 1 **Question 1.** Donner les complexités minimale et maximale de cet algorithme, en nombre de comparaisons.

27 - Q 2 **Question 2.** On fait l'hypothèse probabiliste selon laquelle : i) les entiers du tableau sont tirés équi-probablement sans remise entre 1 et N, avec $N \geqslant n$, et ii) x est tiré équi-probablement entre 1 et N. Quelle est la complexité en moyenne de l'algorithme ?

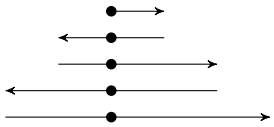
Exercice 28. Trouver un gué dans le brouillard

Cet exercice, qui traite d'un problème plus courant que sa présentation peut le laisser supposer, montre comment deux stratégies a priori *analogues peuvent produire des algorithmes de complexités différentes. On cherche ultimement une solution de complexité linéaire, mais aussi une borne sur la constante de proportionnalité, ce qui n'est pas usuel.*

Vous êtes devant une rivière, dans un épais brouillard. Vous savez qu'il y a un gué dans les parages, mais vous ignorez s'il est à gauche ou à droite, et à quelle distance. Avec ce brouillard, vous verrez l'entrée du gué seulement quand vous serez juste devant. Comment faire pour traverser la rivière ?

Il faut explorer successivement à droite, à gauche, à droite, etc. en augmentant à chaque fois la distance. Commencer par la gauche ou par la droite n'a pas d'importance, mais il faut traiter les deux côtés de manière « équilibrée » et augmenter la distance régulièrement afin d'éviter de faire trop de chemin du côté où le passage ne se trouve pas.

Une première méthode consiste à faire un pas à droite, revenir au point de départ, un pas à gauche, revenir au point de départ, deux pas à droite, revenir au point de départ, deux pas à gauche, revenir au point de départ, trois pas à droite, et ainsi de suite jusqu'au succès selon le schéma ci-après :



Supposons que le gué se trouve à gauche à 15 pas. Pour le trouver, le nombre de pas effectués est

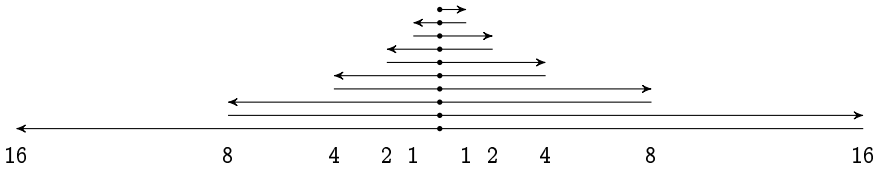
$$1 + (1+1) + (1+2) + (2+2) + (2+3) + \cdots + (14+15) + (15+15)$$

soit au total 465 pas, plus de 30 fois les 15 pas strictement nécessaires.

De façon générale, appelons n la distance en pas entre le point de départ et le gué, c'est-à-dire le nombre minimal de pas nécessaires pour atteindre le gué.

Question 1. Donner le nombre de pas effectués avec la méthode proposée si le gué est situé à n pas à gauche (resp. à droite) du point de départ. Quelle est la classe de complexité de cette méthode en termes de nombre de pas ? 28 - Q 1

Question 2. On souhaite trouver une méthode fondamentalement plus rapide (au sens de la classe de complexité) permettant de garantir que le gué est trouvé en moins de $10 \cdot n$ pas. Le défaut de la précédente réside dans une progression trop « prudente ». Augmenter d'un pas à chaque fois (progression arithmétique) coûte finalement cher et on envisage de *doubler* le nombre de pas à chaque passage (progression géométrique), ce que traduit le schéma ci-dessous : 28 - Q 2



Donner le nombre de pas effectués selon que le gué se trouve à neuf (resp. 15) pas à gauche ou à droite du point de départ. Conclure.

Question 3. La seconde méthode a permis de faire mieux que la première, mais elle reste un peu « rustique ». La modifier pour atteindre l'objectif visé, à savoir un nombre de pas inférieur à $10 \cdot n$. 28 - Q 3

CHAPITRE 3

Spécification, invariants, itération

Le bonheur est désir de répétition.

(M. Kundera)

Ce chapitre est une illustration du principe de construction *raisonnée* de programmes (algorithmes) *a priori* simples que sont les boucles. La démarche proposée est appliquée à une variété de problèmes permettant de révéler l'intérêt de son caractère systématique et rigoureux. Plusieurs exercices du chapitre 8 font appel à la construction de boucles par invariant et complètent *de facto* ceux proposés ici. Par ailleurs, l'objectif principal est de fournir les bases nécessaires à la construction de boucles par invariant sans entrer finement dans les détails.

3.1 Principe de la construction de boucles par invariant

De façon générale, construire rationnellement un programme (algorithme) consiste à le voir comme un mécanisme faisant passer un « système » d'une situation initiale appelée « précondition » à une situation finale nommée « postcondition » ou « but ». Ceci vaut en particulier dans le cas des boucles qui constituent la classe particulière de programmes traitée dans ce chapitre. Le couple (précondition, postcondition) est appelé *spécification* du programme. On cherche à construire un programme à partir de sa spécification. On utilise la notation prec prog postc signifiant que le programme prog s'arrête et fait passer de la situation prec à la situation postc. En d'autres termes, prog étant un programme, prec et postc des prédicats, prec prog postc signifie que si l'exécution de prog démarre dans un état satisfaisant prec alors elle se termine et l'état atteint satisfait le prédicat postc.

Afin de faciliter le raisonnement permettant le développement du programme prog, on en systématise la conception à l'aide de cinq constituants (outre la précondition et la postcondition) :

1. **Invariant** C'est un prédicat représentant une propriété caractéristique du problème.
2. **Condition d'arrêt** C'est également un prédicat.
3. **Progression** Il s'agit d'un fragment de programme.
4. **Initialisation** Il s'agit aussi d'un fragment de programme.
5. **Terminaison** C'est une expression à valeur entière.

qui entretiennent les relations décrites maintenant.

A) D'abord, la conjonction de l'invariant et de la condition d'arrêt conduit au but recherché, qui se formalise par :

$$(\texttt{invariant et condition d'arrêt}) \Rightarrow \texttt{postcondition}.$$

B) Ensuite, la progression doit :

   a) *conserver* l'invariant. Plus précisément, la progression est un fragment de programme défini par la précondition « invariant **et non** condition d'arrêt » et la postcondition « invariant ». On affirme ici que l'invariant est vrai avant et après l'exécution de la progression. Il ne faut bien sûr pas en conclure que l'invariant est vrai pendant toute l'exécution de la progression.

   b) faire décroître *strictement* l'expression de terminaison. Plus précisément, en situation de bouclage (c'est-à-dire lorsque le prédicat « invariant **et non** condition d'arrêt » est satisfait), la valeur (entière) de l'expression de terminaison après un pas de progression est positive ou nulle et strictement inférieure à sa valeur précédente. Ainsi, la condition d'arrêt sera atteinte au bout d'un temps fini, ce qui garantit que le programme qui est élaboré *se termine*.

C) L'initialisation doit *instaurer* l'invariant. Il s'agit donc d'un fragment de programme dont la précondition est celle du problème à résoudre et la postcondition est l'invariant.

Il importe de noter que les deux premiers composants (l'invariant et la condition d'arrêt) portent sur des *situations* tandis que les deux suivants (la progression et l'initialisation) concernent des *actions*. À la lecture des points A, B et C, il apparaît clairement que l'*invariant* est une pièce centrale puisqu'apparaissant dans les éléments A, B-a, B-b et C. Il constitue la clé de voûte de la conception des boucles, en d'autres termes la colle qui lie les autres constituants entre eux. C'est par son identification que va débuter la construction d'une boucle.

La figure 3.1, page 65, reprend et résume les différents points abordés précédemment.

Le codage de la boucle générique correspondante se présente comme suit :

```
1. initialisation ;
2. tant que non condition d'arrêt faire
3.     progression
4. fin tant que
```

Cette notation « minimale » ne fait pas apparaître dans le code la précondition, la postcondition, l'invariant et l'expression de terminaison. Les deux derniers étant assez largement développés dans la phase de construction, dans la suite nous ne ferons généralement figurer dans le code que les deux premiers sous forme de commentaires.

La complexité d'une boucle simple s'exprime en termes de conditions à évaluer ou éventuellement d'une opération caractéristique apparaissant dans son corps. La complexité est alors linéaire, sauf en cas d'appel de procédure ou fonction. En présence de boucles imbriquées, la complexité est généralement polynomiale. Cependant, il est fréquent qu'une itération ne soit que l'un des constituants de la solution en cours de construction et l'évaluation de conditions (liée au contrôle de la boucle) sera bien souvent retenue comme opération élémentaire.

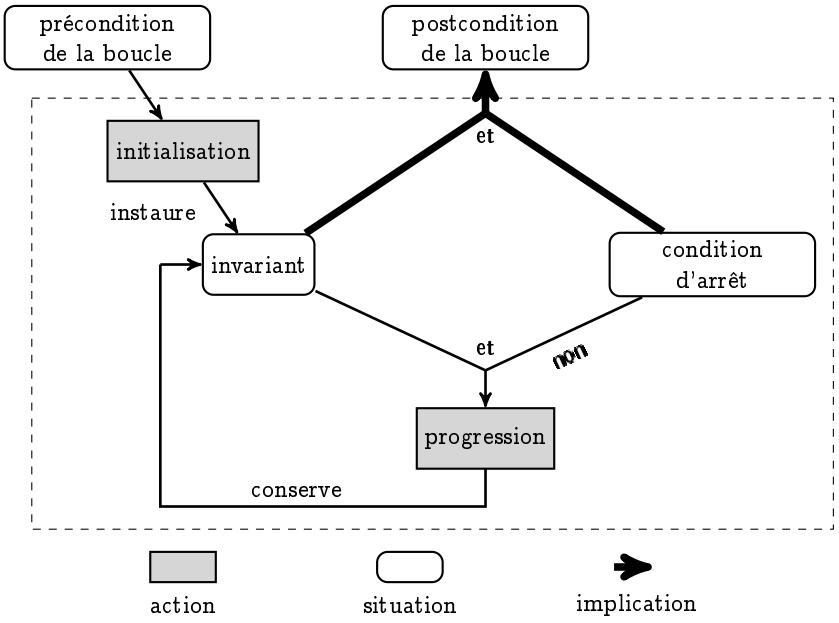


Fig. 3.1 – *Articulation invariant/condition d'arrêt/progression/initialisation*

3.2 Un exemple introductif : la division euclidienne

On considère le problème classique de la division euclidienne. On cherche le quotient de la division « entière » de l'entier A par l'entier B ($A \in \mathbb{N}, B \in \mathbb{N}_1$), étant entendu que l'on ne dispose pas de l'opération de division. La spécification du programme à réaliser est donnée par le couple :

Précondition : $(A \in \mathbb{N})$ **et** $(B \in \mathbb{N}_1)$.

Postcondition : q est le quotient et r est le reste de la division de A par B.

Afin de rendre la postcondition « utilisable » (cet aspect sera développé ultérieurement), on la reformule à partir de la définition de la division euclidienne en :

$$(A = B \cdot q + r) \quad \textbf{et} \quad (0 \leqslant r < B).$$

Partant de cette expression, on prend comme invariant :

$$(A = B \cdot q + r) \quad \textbf{et} \quad (0 \leqslant r)$$

et comme condition d'arrêt : $r < B$. Il reste à exprimer les actions liées à la progression et l'initialisation, ainsi que l'expression de terminaison. Concernant la progression, il suffit d'incrémenter q de 1 et de décrémenter r de la valeur B pour rétablir l'invariant (ce qui est possible puisque la condition d'arrêt n'étant pas satisfaite, on a $r \geqslant B$). L'invariant est instauré à partir de la précondition en affectant 0 à q et A à r. Quant à l'expression de terminaison, on observe que l'expression $(A - q \cdot B)$ vaut A après l'initialisation et décroît de B tout en restant positive après chaque pas de la progression (elle est éventuellement nulle après la dernière itération). À partir des cinq constituants mis en évidence et du

code générique, on déduit le programme ci-après :

```
1.  constantes
2.     A ∈ ℕ et A = ... et B ∈ ℕ1 et B = ...
3.  variables
4.     q ∈ 0..A et r ∈ 0..B − 1
5.  début
6.     /% PRE : (A ∈ ℕ) et (B ∈ ℕ1) %/
7.     q ← 0 ; r ← A ;
8.     tant que non(r < B) faire
9.        q ← q + 1 ; r ← r − B
10.    fin tant que ;
11.    /% POST : (A = q · B + r) et (r ⩾ 0) et (r < B) %/
12.    écrire(le quotient de la division de , A, par , B, est , q, et son reste est ,
       r)
13. fin
```

On notera que si $A = 0$ on n'entre pas dans la boucle « tant que » (c'est le seul cas d'ailleurs). De plus, si A est un multiple de B le reste r vaut 0.

3.3 Techniques auxquelles recourir si besoin

Si, dans l'exemple précédent, le problème était suffisamment simple pour être traité sans difficulté, il n'en va pas toujours ainsi. Il faut assez souvent reformuler la précondition et/ou la postcondition, et ceci doit se faire de façon à garantir la correction du programme construit *in fine*. Nous passons en revue quelques techniques permettant de faire des transformations *légales*.

3.3.1 Composition séquentielle

Nous avons vu que construire un programme fondé sur une boucle se fait en décomposant le problème en cinq sous-problèmes. Il existe une autre forme de décomposition fréquemment utilisée, à savoir celle fondée sur la règle de la séquentialité qui stipule que si $\{P\}\ prog_1\ \{R\}$ et $\{R\}\ prog_2\ \{S\}$ alors $\{P\}\ prog_1; prog_2\ \{S\}$. Dans cette règle, le « ; » est l'opérateur de séquentialité ; $\{R\}$ est appelée situation intermédiaire. Cette règle peut également être appliquée pour dire que si l'on recherche un programme spécifié par le couple (P, S), il suffit de trouver une situation intermédiaire R et deux programmes $prog_1$ et $prog_2$ tels que : i) $\{P\}\ prog_1\ \{R\}$, et ii) $\{R\}\ prog_2\ \{S\}$. La composition séquentielle de $prog_1$ et de $prog_2$ est une solution au problème initial spécifié par (P, S).

Illustrons ce procédé avec l'exemple suivant dans lequel on souhaite savoir si l'élément d'indice K du tableau T est son unique minimum. On veut construire le programme spécifié par :

Précondition : (T est un tableau constant de N entiers naturels) et $(N \geqslant 1)$ et (K constant) et $(K \in 1..N)$.

Postcondition : $infk =$ (T[K] est strictement inférieur à tous les autres éléments).

On souhaite élaborer une solution ne parcourant pas systématiquement le tableau T et ne comportant qu'une seule boucle.

Une analyse sommaire du problème montre que deux cas peuvent se présenter :

— le tableau T ne contient qu'un exemplaire de son minimum qui se trouve être T[K], ce qui nécessite de parcourir T dans sa totalité,

— T[K] n'est pas le minimum du tableau T (on rencontre une valeur strictement inférieure à T[K] et il n'est pas utile de poursuivre l'examen de T) ; ou T[K] est bien le minimum et on rencontre une valeur T[i] égale à T[K] avec $i \neq K$, et là encore l'examen de T peut être arrêté).

Afin de faciliter la conception de la solution, on va dans un premier temps ignorer la variable infk et considérer la nouvelle postcondition :

Postcondition : i est le plus petit indice de l'intervalle $1..N$ différent de K tel que $T[i] \leqslant T[K]$ s'il existe ou sinon $(N+1)$

qui représente une situation intermédiaire (au sens de la composition séquentielle) ; on envisage alors la démarche de résolution suivante :

1. On construit la boucle (BOUCLE) spécifiée par :

   Précondition : (T est un tableau constant de N entiers naturels) **et** $(N \geqslant 1)$ **et** (K constant) **et** $(K \in 1..N)$

   Postcondition : i est l'indice de l'intervalle $1..N$ différent de K tel que $T[i] \leqslant T[K]$ (s'il existe un tel i) ou sinon $(N+1)$.

2. On construit la partie complémentaire (PARTCOMP) spécifiée par :

   Précondition : i est l'indice de l'intervalle $1..N$ différent de K tel que $T[i] \leqslant T[K]$ (s'il existe un tel i) ou sinon $N+1$.

   Postcondition : $\mathit{infk} =$ (T[K] est inférieur à tous les autres éléments).

Le code associé à PARTCOMP consiste à affecter à infk la valeur de l'expression relationnelle $(i = N+1)$. Pour ce qui est de celui correspondant à BOUCLE, il s'appuie sur les cinq composants suivants :

1. **Invariant** $(i \in 1..N)$ **et** $(\forall j \cdot (j \in ((1..i-1) - \{K\}) \Rightarrow T[j] > T[K]))$
2. **Condition d'arrêt** $(i = N+1)$ **ou sinon** $((i \neq K)$ **et** $(T[i] \leqslant T[K]))$
3. **Progression** $i \leftarrow i+1$
4. **Initialisation** $i \leftarrow 1$
5. **Terminaison** $N+1-i$

Il est aisé de vérifier que : i) la conjonction de l'invariant et de la condition d'arrêt implique la postcondition, ii) la progression conserve l'invariant, iii) l'expression de terminaison décroît à chaque pas tout en restant positive ou nulle, et iv) l'initialisation instaure l'invariant, autrement dit que les relations A, B-a, B-b et C sont satisfaites. Au final, on aura construit le programme :

1. **constantes**
2. $\quad N \in \mathbb{N}_1$ **et** $N = \ldots$ **et** $T \in 1..N \rightarrow \mathbb{N}$ **et** $T = [\ldots]$ **et** $K \in 1..N$ **et** $K = \ldots$
3. **variables**
4. $\quad i \in 1..N+1$ **et** $\mathit{infk} \in \mathbb{B}$
5. **début**

```
6.  /% première partie : BOUCLE %/
7.  /% PRE : (T est un tableau constant de N entiers naturels) et (N ⩾
    1) et (K constant) et (K ∈ 1 .. N) %/
8.  i ← 1 ;
9.  tant que non((i = N + 1) ou sinon ((i ≠ K) et (T[i] ⩽ T[K]))) faire
10.    i ← i + 1
11. fin tant que ;
12. /% POST : i est l'indice de l'intervalle 1 .. N différent de K tel que
    T[i] ⩽ T[K] (s'il existe un tel i) ou sinon (N + 1) %/
13. /% deuxième partie : PARTCOMP %/
14. infk ← (i = N + 1) ;
15. si infk alors
16.    écrire(l'élément d'indice, K, du tableau, T, est son unique minimum)
17. sinon
18.    écrire(l'élément d'indice, K, du tableau, T, n'est pas son unique
19.          minimum)
20. fin si
21. fin
```

3.3.2 Renforcement et affaiblissement de prédicats

La notion de renforcement du prédicat pred s'entend souvent dans un sens logique, c'est-à-dire que l'on considère un prédicat pred′ (renforçant pred) car satisfaisant la propriété : pred′ ⇒ pred. Cette démarche peut se révéler particulièrement intéressante lorsque l'on se trouve face à la spécification (P, Q) et qu'il est plus facile de construire le programme prog′ tel que {P} prog′ {Q′} avec Q′ ⇒ Q (notamment si Q′ est défini par l'ajout d'un conjoint à Q), que le programme prog tel que {P} prog {Q}.

De façon duale, on peut parler d'affaiblissement de prédicat en considérant un prédicat pred″ (affaiblissant pred) vérifiant la propriété : pred ⇒ pred″. Cette démarche présente un intérêt, quand face à la spécification (P, Q), il est plus facile de construire le programme prog″ tel que {P″} prog″ {Q} avec P ⇒ P″ que le programme prog tel que {P} prog {Q}. C'est notamment le cas si P″ est obtenu par suppression d'un conjoint de P.

3.3.3 Renforcement par introduction de variables de programmation

On considère ici un second type de renforcement d'un prédicat dans lequel l'espace d'états du problème (l'ensemble de ses variables) s'enrichit d'(au moins) une variable. Par exemple, considérons un tableau T[1 .. N] (N ⩾ 1) d'entiers naturels et le prédicat P ≙ « s représente la somme des éléments du tableau T ». Une formulation équivalente de P est P′ ≙ « (s représente la somme des i premiers éléments du tableau T) et (i ∈ 1 .. N) et (i = N) ». La variable i a été introduite, et on a un renforcement du prédicat P initial. Le principal intérêt de cette démarche réside dans le fait qu'elle introduit au moins une conjonction qui va pouvoir être exploitée pour découvrir un invariant pour le problème considéré (voir par exemple l'exercice 31, page 81).

3.4 Heuristiques pour la découverte d'invariants

La programmation est une activité dirigée par le but. La recherche de l'invariant d'une boucle n'échappe pas à cette règle, et les trois heuristiques proposées ci-dessous se fondent sur la relation qui lie postcondition et invariant. Nous étudions successivement trois techniques de base concurrentes. La première, l'éclatement de la postcondition, fait l'hypothèse que la postcondition se présente sous la forme d'une conjonction. Comme c'est rarement le cas, il est fréquemment nécessaire de procéder à un renforcement préalable de la postcondition de façon à exhiber une ou plusieurs conjonctions. La seconde, l'hypothèse du travail réalisé en partie, conduit de par sa nature à un renforcement implicite de la postcondition par l'introduction de variables. La dernière heuristique concerne le renforcement de la postcondition suite à celui de l'invariant.

3.4.1 Éclatement de la postcondition

Compte tenu de la propriété associée à la relation A (voir page 64), il est légitime, lorsque la postcondition s'exprime comme une conjonction, d'identifier l'un des conjoints à l'invariant et le second à la condition d'arrêt. La démarche est identique dans le cas où la postcondition s'exprime par *plusieurs* conjonctions. On peut alors chercher d'une part à isoler un ou plusieurs conjoints pour constituer la condition d'arrêt, d'autre part à mettre de côté le(s) conjoint(s) restant(s) pour former l'invariant :

Postcondition : B_1 **et** B_2 **et** ... **et** B_n
Condition d'arrêt : B_i
Invariant : B_1 **et** B_2 **et** ... **et** B_{i-1} **et** B_{i+1} **et** ... **et** B_n

En supposant que l'on n'isole qu'un seul conjoint, comment va-t-on choisir parmi les n possibilités ? Il n'y a pas de réponse systématique à cette question. On peut cependant proposer les heuristiques suivantes :

1. Il est préférable que la condition d'arrêt puisse s'exprimer syntaxiquement dans un langage de programmation habituel afin que sa négation puisse constituer la condition de bouclage. Ceci exclut des prédicats avec quantificateur tels que $\forall j \cdot (j \in \ldots \Rightarrow \ldots)$.
2. Il est préférable que l'invariant soit facile à instaurer à partir de la précondition. Une fois le choix de l'invariant effectué, il faut évaluer la difficulté que représente son instauration (l'initialisation).
3. Toutes les variables de la postcondition doivent apparaître dans l'invariant (autrement dit, la condition d'arrêt ne doit pas contenir de variables « orphelines »). Le cas échéant la progression n'affecterait pas ces variables, qui apparaîtraient donc comme inutiles.

On illustre la démarche avec l'exemple suivant, où l'on recherche le premier zéro d'un tableau de nombres. On veut construire le programme spécifié par :

Précondition : (T est un tableau d'entiers naturels constant de N éléments) **et** (il existe au moins un zéro dans le tableau T).

Postcondition : La variable i désigne le zéro de T ayant le plus petit indice (le zéro le plus à « gauche »).

On remarque que, puisqu'il existe au moins un zéro dans T, c'est donc que $N \geqslant 1$. Puisque cet algorithme exige de parcourir tout ou partie du tableau, on va construire une itération. De plus, la postcondition ne se présente pas sous forme conjonctive. On va la renforcer dans cet objectif en remplaçant une *expression* exp (ici N) par une *variable* v (ici i), puis en ajoutant le conjoint $(v = exp)$. Dans la suite, ce traitement est dénommé « mise sous forme constructive ». On reformule la postcondition initiale en :

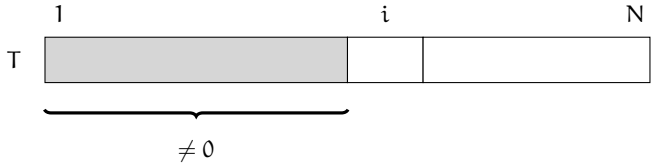
Postcondition : $(i \in 1\,..\,N)$ **et** (T[1 .. i − 1] ne contient pas de zéro) **et** $(T[i] = 0)$.

Il est alors possible d'appliquer la méthode de l'éclatement de la postcondition et les heuristiques présentées ci-dessus pour diviser cette postcondition en deux parties, la première constituant l'invariant et la seconde la condition d'arrêt :

1. **Invariant** Les deux premiers conjoints sont faciles à établir, et on les prend donc pour constituer l'invariant, soit :

   $(i \in 1\,..\,N)$ **et** (T[1 .. i − 1] ne contient pas de zéro)

   que l'on peut illustrer graphiquement par :

   Le conjoint $(T[i] = 0)$ a donc été écarté. C'est effectivement un prédicat difficile à établir puisqu'il faudrait localiser un zéro !

2. **Condition d'arrêt** Il suffit de prendre le conjoint écarté, soit : $T[i] = 0$. On note que ce prédicat ne contient pas de quantificateurs ; sa négation pourra donc être intégrée au programme final.

3. **Progression** La progression est spécifiée par :

   Précondition : $(i \in 1..N)$ **et** (T[1..i−1] ne contient pas de zéro) **et non** $(T[i] = 0)$

   PROGRESSION

   Postcondition : $(i \in 1\,..\,N)$ **et** (T[1 .. i − 1] ne contient pas de zéro).

   Une solution pour la progression consiste à déplacer i d'une position vers la droite par l'affectation : $i \leftarrow i + 1$. L'invariant est bien satisfait par la nouvelle situation puisqu'il n'y a pas de zéro dans T[1 .. i − 1] et qu'il doit y en avoir un après (selon la précondition) donc $i \leqslant N$ (d'où $i \in 1\,..\,N$).

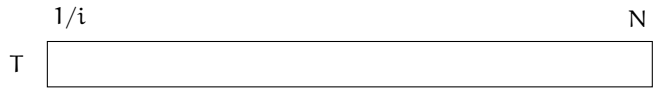
4. **Initialisation** L'initialisation est spécifiée par :

   Précondition : (T est un tableau d'entiers naturels constant de N éléments) **et** (il existe au moins un zéro dans le tableau T)

   INITIALISATION

   Postcondition : $(i \in 1\,..\,N)$ **et** (T[0 .. i − 1] ne contient pas de 0).

Passer de la précondition à l'invariant, c'est atteindre la situation suivante :



qui satisfait bien l'invariant puisque, si $i = 1$, celui-ci s'instancie en : $(1 \in 1\,..\,N)$ **et** $(T[1\,..\,0]$ ne contient pas de zéro). Ceci s'obtient par l'affectation : $i \leftarrow 1$.

5. **Terminaison** L'expression $(N - i)$ convient, car elle reste toujours positive ou nulle et décroît à chaque pas de progression.

Au final, on a construit le programme suivant :

```
1.  constantes
2.     N ∈ ℕ1 et T ∈ 1..N → ℕ et T = [...]
3.  variables
4.     i ∈ 1..N
5.  début
6.     /% PRE : (T est un tableau d'entiers naturels constant de N éléments)
       et (il existe au moins un zéro dans le tableau T) %/
7.     i ← 1;
8.     tant que non(T[i] = 0) faire
9.        i ← i + 1
10.    fin tant que;
11.    /% POST : (i ∈ 1..N) et (T[1..i−1] ne contient pas de zéro) et (T[i] = 0)
       %/
12.    écrire(l'indice du zéro le plus à gauche dans, T, est, i)
13. fin
```

Dans le meilleur des cas, cet algorithme est en $\Theta(1)$ et au pire en $\Theta(N)$ en nombre de comparaisons.

Dans la section 3.7, plusieurs exercices sont proposés dans lesquels la postcondition initiale n'est pas exprimée sous forme conjonctive. Dans ce type de situation très fréquent en pratique, la postcondition doit d'abord être mise sous forme « constructive », c'est-à-dire exprimée comme une conjonction qui pourra être éclatée.

3.4.2 Hypothèse du travail réalisé en partie

L'une des caractéristiques de l'invariant est qu'il s'agit d'une propriété qui se retrouve identique à elle-même à chaque pas de boucle. Il est légitime d'exploiter cette particularité pour rechercher un invariant. En faisant l'hypothèse qu'une partie du travail a déjà été réalisée, on peut se poser la question :

« Dans quelle situation se trouve-t-on ? »

dont la réponse est souvent l'invariant recherché.

Cette technique est illustrée avec l'exemple « classique » et simple dit « du drapeau monégasque », dont l'énoncé est le suivant. La situation initiale est un tableau $T[1\,..\,N]$ $(N \geqslant 0)$ dont les éléments sont de couleur blanche ou rouge et forment le sac S. La situation

finale est un tableau formant le même sac S, dans lequel les éléments blancs occupent la partie gauche et les éléments rouges la partie droite. On s'impose de passer de la situation initiale à la situation finale en une seule boucle, qui parcourt le tableau de gauche à droite. Les seuls changements du tableau T s'effectuent par l'opération d'échange des éléments d'indice i et j de T (*Échanger*(i, j)), ce qui garantit la préservation du sac de valeurs S initial.

On va chercher un invariant en admettant que le travail est partiellement effectué et que l'on a atteint la configuration décrite dans la figure 3.2, page 72.

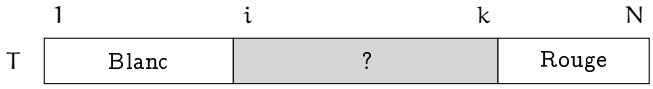


Fig. 3.2 – *Situation dans laquelle le travail est réalisé en partie.*

Formellement, cette configuration se définit par la formule :

$$(\forall u \cdot (u \in 1\,..\,i-1 \Rightarrow T[u] = \text{Blanc})) \textbf{ et } (1 \leqslant i \leqslant k+1 \leqslant N+1) \textbf{ et}$$
$$(\forall u \cdot (u \in k+1\,..\,N \Rightarrow T[u] = \text{Rouge}))$$

Cette expression couvre l'ensemble des cas suivants :

i) La situation initiale dans laquelle le contenu de T est inconnu (ni élément blanc, ni élément rouge localisé), avec $i = 1$ et $k = N$. En effet, la formule ci-dessus qui s'écrit alors :

$$(\forall u \cdot (u \in 1\,..\,0 \Rightarrow T[u] = \text{Blanc})) \textbf{ et } (1 \leqslant 1 \leqslant N+1 \leqslant N+1) \textbf{ et}$$
$$(\forall u \cdot (u \in N+1\,..\,N \Rightarrow T[u] = \text{Rouge}))$$

s'évalue à **vrai** et exprime que l'on ne sait rien sur la couleur de chacun des éléments de T,

ii) La situation finale où tous les éléments sont localisés (les blancs à gauche, les rouges à droite), avec $k = i - 1$,

iii) Une situation où il y a au moins un élément rouge et aucun élément blanc, avec $i = 1$ et $0 \leqslant k < N$,

iv) Une situation où il y au moins un élément blanc et aucun élément rouge, avec $1 < i \leqslant N+1$ et $k = N$,

v) Le cas où T est vide ($N = 0$).

Il apparaît donc qu'elle constitue un invariant pour le problème à résoudre.

À partir de la situation de départ (précondition) et de celle d'arrivée (postcondition), on va maintenant définir les cinq éléments de la boucle.

1. **Invariant** On reprend telle quelle l'expression associée à la figure 3.2, à savoir :

$$(\forall u \cdot (u \in 1\,..\,i-1 \Rightarrow T[u] = \text{Blanc})) \textbf{ et } (1 \leqslant i \leqslant k+1 \leqslant N+1) \textbf{ et}$$
$$(\forall u \cdot (u \in k+1\,..\,N \Rightarrow T[u] = \text{Rouge})).$$

2. **Condition d'arrêt** La boucle est terminée quand $k = i - 1$. On a mis en évidence (voir item ii) qu'alors la conjonction de l'invariant et de la condition d'arrêt implique la postcondition.

3. **Progression** L'action à réaliser dans la progression dépend de la couleur de T[i]. Si T[i] est blanc, il suffit d'incrémenter i de 1 (ce qui étend la zone des éléments blancs) et si T[i] est rouge, on l'échange avec T[k] et on décrémente k de 1 (ce qui agrandit la zone des éléments rouges).

4. **Initialisation** Les opérations relatives à l'initialisation ont été évoquées dans l'item i, précédemment, et consistent donc à faire les deux affectations : $i \leftarrow 1$ et $k \leftarrow N$.

5. **Terminaison** L'expression $(k - i + 1)$ convient puisqu'elle vaut N après l'initialisation, décroît de 1 à chaque pas d'itération et atteint 0 quand la condition d'arrêt est satisfaite.

Au final, on aboutit au programme ci-après :

```
1.  constantes
2.    N ∈ ℕ et N = ... et T ∈ 1 .. N → ℕ et T = [...]
3.  variables
4.    i ∈ 1 .. N et k ∈ 1 .. N
5.  début
6.    /% PRE : T[1 .. N] est un tableau dont les éléments sont de couleur
      blanche ou rouge %/
7.    i ← 1 ; k ← N ;
8.    tant que k ≠ i − 1 faire
9.      si T[i] = Blanc alors
10.       i ← i + 1
11.     sinon
12.       Échanger(i, k) ; k ← k − 1
13.     fin si
14.   fin tant que ;
15.   /% POST : les éléments blancs de T occupent la partie gauche et les
      éléments rouges de T la partie droite %/
16.   écrire(T)
17. fin
```

Cet algorithme est en $\Theta(N)$ comparaisons (N comparaisons portant sur les valeurs de T et N pour le contrôle de la boucle). Il demande autant d'échanges qu'il y a d'éléments de couleur rouge (quelle que soit leur position initiale) ; il est donc en $\mathcal{O}(N)$ échanges.

Comme dans la majorité des problèmes, ici l'invariant n'est pas unique. Le lecteur intéressé pourra réfléchir à celui qui dérive de la situation du travail réalisé en partie donnée dans la figure 3.3.

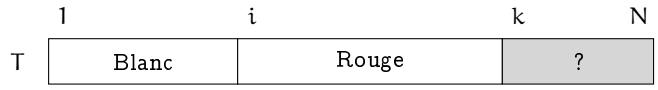


Fig. 3.3 – *Autre situation dans laquelle le travail est réalisé en partie.*

3.4.3 Renforcement de l'invariant

Nous avons vu que la conjonction de l'invariant P et de la condition d'arrêt B doit impliquer la postcondition R :

$$(P \textbf{ et } B) \Rightarrow R.$$

Il a été dit qu'il pouvait être pertinent de reformuler la postcondition en introduisant une (ou plusieurs) variable(s) afin de faire apparaître une conjonction. Ce faisant, on a renforcé la postcondition initiale par une postcondition auxiliaire qui *l'implique*. Il est cependant des cas où la nécessité, ou simplement l'occasion, de renforcer la postcondition n'apparaît qu'en cours de développement. Le renforcement de la postcondition s'exprime alors souvent *indirectement* par un renforcement de l'invariant. On va alors adjoindre à l'invariant « naturel » un conjoint supplémentaire P′ afin, soit d'assurer la progression, soit de formuler une condition d'arrêt. Ceci s'exprime formellement par :

$$\begin{array}{ll} & (P \textbf{ et } P' \textbf{ et } B) \Rightarrow (P \textbf{ et } B) \\ \Rightarrow & \qquad\qquad \text{on a soit } (P \textbf{ et } B) = R, \text{ soit au pire } (P \textbf{ et } B) \Rightarrow R \\ & (P \textbf{ et } P' \textbf{ et } B) \Rightarrow R. \end{array}$$

La démarche que nous allons adopter se décompose en quatre parties :

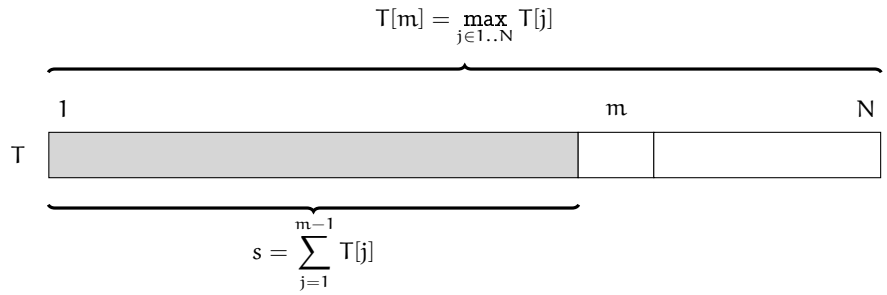
1. On développe la boucle avec l'invariant « naturel ».
2. On constate que la valeur d'une expression manque pour poursuivre la construction de la progression.
3. On fait l'hypothèse que cette valeur est disponible (dans une variable introduite à cette fin). Ceci conduit à adjoindre un nouveau conjoint à l'invariant existant.
4. On reprend la construction de la boucle à partir du nouvel invariant.

Nous allons développer cette démarche pour calculer la somme des éléments d'un tableau situés « à gauche » de sa valeur maximale (supposée unique). On veut construire le programme constitué d'une seule boucle spécifié par :

Précondition : Soit (T un tableau d'entiers relatifs constant injectif (toutes les valeurs de T sont différentes) de N éléments) **et** ($N \geqslant 1$).

Postcondition : m est la position du maximum de T et s est la somme des éléments de $T[1 \,..\, m-1]$.

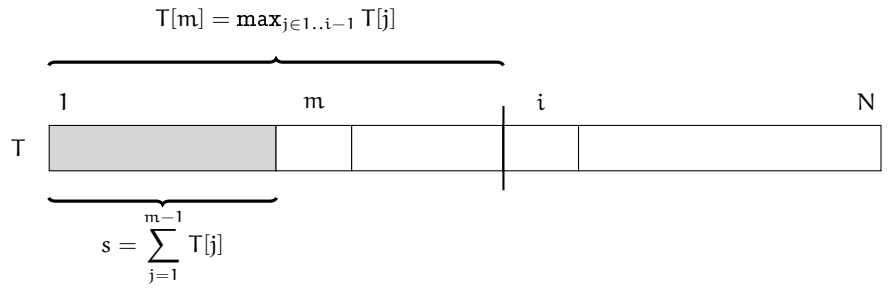
Si m est la position du maximum de $T[1 \,..\, N]$, on souhaite, à l'issue du programme, être dans la situation suivante :



soit plus formellement, et en ajoutant les domaines de variation des variables, la postcondition :

$$(m \in 1\,..\,N) \textbf{ et } (T[m] = \max_{j \in 1..N} T[j]) \textbf{ et } (s \in \mathbb{Z}) \textbf{ et } \left(s = \sum_{j=1}^{m-1} T[j]\right).$$

Première tentative. On renforce la postcondition de façon à obtenir une conjonction exploitable :



Étudions d'abord le domaine de variation des variables i, m et s. Pour que m puisse prendre une valeur, il faut que le sous-tableau $T[1..i-1]$ ne soit pas vide, ce qui impose que i ne prenne pas la valeur 1. Donc, i varie sur l'intervalle $2\,..\,N+1$, tandis que $m \in 1\,..\,i-1$ et $s \in \mathbb{Z}$. En ajoutant le domaine de variation de ces variables, on obtient :

Postcondition (version issue du renforcement de la précédente) : $(i \in 2\,..\,N+1)$ **et** $(m \in 1\,..\,i-1)$ **et** $(T[m] = \max\limits_{j \in 1..i-1} T[j])$ **et** $(s \in \mathbb{Z})$ **et** $(s = \sum_{j=1}^{m-1} T[j])$ **et** $(i = N+1)$.

Cette postcondition permet d'effectuer une première tentative d'éclatement.

1. **Invariant** Les cinq premiers conjoints sont faciles à établir, on les retient pour constituer l'invariant :

   $$(i \in 2\,..\,N+1) \textbf{ et } (m \in 1\,..\,i-1) \textbf{ et } (T[m] = \max_{j \in 1..i-1} T[j]) \textbf{ et } (s \in \mathbb{Z}) \textbf{ et } \left(s = \sum_{j=1}^{m-1} T[j]\right).$$

2. **Condition d'arrêt** On retient le conjoint écarté : $i = N+1$.

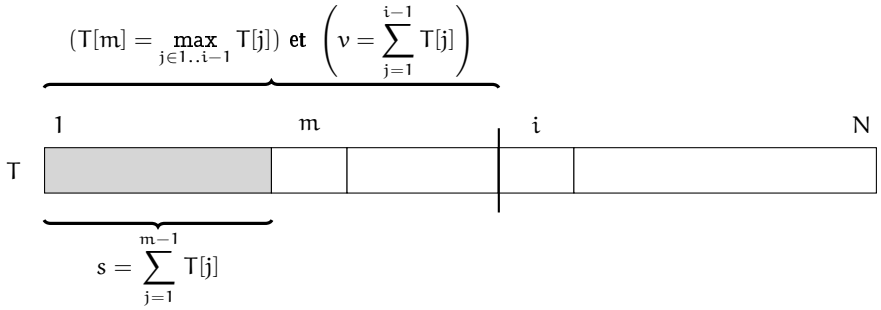
3. **Progression** Comment passer de la situation où l'invariant est vrai, ainsi que **non**$(i = N+1)$, à la situation où l'invariant est vrai, tout en progressant vers la postcondition de l'énoncé ? Considérons l'élément $T[i]$. Si i n'est pas la position du maximum de $T[1\,..\,i]$ il n'y a rien d'autre à faire que d'incrémenter i. Sinon, une solution consiste à calculer dans s, par une boucle, la somme des éléments de $T[1\,..\,i-1]$. Il est cependant facile de s'apercevoir que des sommes risquent d'être recalculées. L'idée est de profiter des calculs déjà réalisés, et pour ce faire, de considérer que cette somme est disponible dans la variable v. Mettre à jour s n'exige alors pas de boucle, s prend simplement la valeur de v. Retenir cette hypothèse, c'est renforcer l'invariant par le conjoint $v = \sum_{j=1}^{i-1} T[j]$. On doit donc reconstruire la boucle à partir du nouvel invariant.

Seconde tentative. L'invariant (et indirectement la postcondition) se renforce par l'ajout du conjoint $v = \sum_{j=1}^{i-1} T[j]$.

1. **Invariant** Il devient donc :

$$(i \in 2\,..\,N+1) \text{ et } (m \in 1\,..\,i-1) \text{ et } (T[m] = \max_{j\in 1..i-1} T[j]) \text{ et}$$
$$(s \in \mathbb{Z}) \text{ et } \left(s = \sum_{j=1}^{m-1} T[j]\right) \text{ et } (v \in \mathbb{Z}) \text{ et } \left(v = \sum_{j=1}^{i-1} T[j]\right)$$

soit, sous forme graphique :



2. **Condition d'arrêt** On conserve la même condition d'arrêt : $i = N + 1$.

3. **Progression** La progression se spécifie par :

Précondition : $(i \in 2\,..\,N+1)$ et $(m \in 1\,..\,i-1)$ et $(T[m] = \max_{j\in 1..i-1} T[j])$ et $(s \in \mathbb{Z})$ et $(s = \sum_{j=1}^{m-1} T[j])$ et $(v \in \mathbb{Z})$ et $(v = \sum_{j=1}^{i-1} T[j])$ et **non**$(i = N+1)$

PROGRESSION

Postcondition : $(i \in 2\,..\,N+1)$ et $(m \in 1\,..\,i-1)$ et $(T[m] = \max_{j\in 1..i-1} T[j])$ et $(s \in \mathbb{Z})$ et $(s = \sum_{j=1}^{m-1} T[j])$ et $(v \in \mathbb{Z})$ et $(v = \sum_{j=1}^{i-1} T[j])$.

Si i est la position du nouveau maximum dans le tableau $T[1\,..\,i]$, s prend la valeur de v. Les variables m et v sont ensuite actualisées. Dans tous les cas, on intègre T[i] à v et i augmente de 1.

4. **Initialisation** L'initialisation est spécifiée par :

Précondition : (T est un tableau injectif constant de N éléments) et $(N \geqslant 1)$

INITIALISATION

Postcondition : $(i \in 2\,..\,N+1)$ et $(m \in 1\,..\,i-1)$ et $(T[m] = \max_{j\in 1..i-1} T[j])$ et $(s \in \mathbb{Z})$ et $(s = \sum_{j=1}^{m-1} T[j])$ et $(v \in \mathbb{Z})$ et $(v = \sum_{j=1}^{i-1} T[j])$.

On donne la valeur 2 à i, ce qui impose aux autres variables les contraintes suivantes (on peut le vérifier par substitution) :

$$(s = T[1]) \text{ et } (v = T[1]) \text{ et } (m = 1).$$

4. **Terminaison** La variable i est incrémentée à chaque pas de progression. L'expression $(N + 1 - i)$ convient pour assurer la terminaison.

Finalement, on a construit le programme suivant :

```
1.  constantes
2.     N ∈ ℕ1 et N = ... et T ∈ 1 .. N → ℤ et T = [...]
3.  variables
4.     i ∈ 1 .. N + 1 et m ∈ 1 .. N et v ∈ ℤ et s ∈ ℤ
5.  début
6.     /% PRE : (T est un tableau injectif constant de N éléments) et (N ⩾ 1))
       %/
7.     i ← 2 ; s ← T[1] ; v ← T[1] ; m ← 1 ;
8.     tant que non(i = N + 1) faire
9.        si T[i] > T[m] alors
10.          s ← v ; m ← i
11.       fin si ;
12.       v ← v + T[i] ; i ← i + 1
13.    fin tant que ;
14.    /% POST : m est la position du maximum de T et s est la somme des
       éléments de T[1 .. m − 1] %/
15.    écrire(la somme des éléments précédant celui d'indice , m, est , s)
16. fin
```

Cette solution exige au pire et au mieux N itérations et 2N comparaisons.

Remarque La précondition impose que les valeurs de T soient toutes différentes. En levant cette contrainte, on obtiendrait une spécification légèrement modifiée. Le cas où le maximum serait présent plusieurs fois conduirait à un programme qui calculerait *l'une des sommes* entre l'indice 1 et la position de *l'un des maxima*.

3.5 À propos de recherche linéaire bornée

La recherche linéaire est une classe d'algorithmes où, étant donné un prédicat P(i) tel qu'il existe un ensemble non vide de valeurs le satisfaisant, on recherche le plus petit élément de cet ensemble.

La recherche linéaire *bornée* s'apparente à la recherche linéaire, mais ici l'ensemble des solutions peut être vide. En général, le résultat fourni en cas d'échec est soit une valeur conventionnelle située en dehors du domaine de définition de P, soit un booléen qui signale l'échec.

3.5.1 Un exemple

On veut déterminer si la somme des éléments du tableau T[1 .. N] est ou non supérieure à une valeur entière donnée S.

On veut construire le programme spécifié par :

Précondition : (T est un tableau de N entiers naturels) et ($N \geqslant 0$) et (S constant) et ($S \in \mathbb{N}_1$).

Postcondition : $sss = (\sum_{j=1}^{N} T[j] > S)$.

La postcondition n'étant pas sous forme constructive, dans la quantification on remplace l'expression $(N+1)$ par la variable i, d'où :

Postcondition : $(i \in 1\,..\,N+1)$ **et** $(sss = (\sum_{j=1}^{i-1} T[j] > S))$ **et** $(i = N+1)$.

La présence d'une quantification laisse augurer un problème quant à son évaluation telle quelle ; on procède à un second renforcement en introduisant la variable sp calculant la somme partielle des $(i-1)$ premiers éléments de T, d'où :

Postcondition : $(i \in 1\,..\,N+1)$ **et** $(sp = \sum_{j=1}^{i-1} T[j])$ **et** $(sss = (sp > S))$ **et** $(i = N+1)$.

On peut maintenant débuter la construction de l'algorithme.

1. **Invariant** Les trois premiers conjoints sont faciles à établir. On les conserve pour constituer l'invariant :

$$(i \in 1\,..\,N+1) \text{ et } \left(sp = \sum_{j=1}^{i-1} T[j]\right) \text{ et } (sss = (sp > S)).$$

2. **Condition d'arrêt** On reprend le conjoint écarté : $(i = N+1)$.

3. **Progression** La progression est spécifiée par :

   $(i \in 1\,..\,N+1)$ **et** $(sp = \sum_{j=1}^{i-1} T[j])$ **et** $(sss = (sp > S))$ **et non**$(i = N+1)$

   PROGRESSION

   $(i \in 1\,..\,N+1)$ **et** $(sp = \sum_{j=1}^{i-1} T[j])$ **et** $(sss = (sp > S))$.

   Une solution à cette spécification est :

   1. $sp \leftarrow sp + T[i]$;
   2. $sss \leftarrow sss$ **ou** $(sp > S)$;
   3. $i \leftarrow i + 1$

4. **Initialisation** L'initialisation est spécifiée par :

   Précondition : (T est un tableau de N entiers naturels) **et** $(N \geqslant 0)$ **et** $(S \in \mathbb{N}_1)$

   INITIALISATION

   Postcondition : $(i \in 1\,..\,N+1)$ **et** $(sp = \sum_{j=1}^{i-1} T[j])$ **et** $(sss = (sp > S))$.

   L'affectation de 1 à i, de 0 à sp et de **faux** à sss répond à cette spécification.

5. **Terminaison** $(N+1-i)$ est une expression de terminaison convenable.

Il est facile de constater que l'arrêt pourrait avoir lieu dès que sss prend la valeur **vrai**, ce qui serait susceptible de limiter le nombre de passages dans la boucle et donc la complexité au pire. Cette remarque conduit à modifier la condition d'arrêt et la progression de la construction précédente. On va modifier la condition d'arrêt en conséquence dans la construction d'une nouvelle boucle.

1. **Invariant** Il est inchangé.

2. **Condition d'arrêt** On prend : $(i = N+1)$ **ou** sss.

3. **Progression** La seconde affectation de la progression est $(sss$ **ou** $(sp > S))$, dont l'évaluation se fait dans le contexte de la précondition de la progression (impliquant que sss est **faux**), d'où :

$$\begin{array}{ll} & sss \text{ ou } (sp > S) \\ \Leftrightarrow & \qquad\qquad \text{non}(sss) \text{ et calcul propositionnel} \\ & sp > S \end{array}$$

4. Initialisation Elle est inchangée.

5. Terminaison Elle est elle aussi inchangée.

Au final, on obtient le programme suivant :

1. **constantes**
2. $N \in \mathbb{N}$ **et** $N = \ldots$ **et** $T \in 1\,..\,N \to \mathbb{N}$ **et** $T = [\ldots]$ **et** $S \in \mathbb{N}_1$ **et** $S = \ldots$
3. **variables**
4. $i \in 1\,..\,N+1$ **et** $sss \in \mathbb{B}$ **et** $sp \in \mathbb{N}$
5. **début**
6. /% *PRE :* (T *est un tableau de* N *entiers naturels*) **et** ($N \geqslant 0$) **et** (S *constant*) **et** ($S \in \mathbb{N}_1$) %/
7. $i \leftarrow 1$; $sss \leftarrow$ **faux** ; $sp \leftarrow 0$;
8. **tant que non**$((i = N+1)$ **ou** $sss)$ **faire**
9. $sp \leftarrow sp + T[i]$; $sss \leftarrow (sp > S)$; $i \leftarrow i+1$
10. **fin tant que** ;
11. /% *POST* : $sss = \left(\sum_{j=1}^{N} T[j] > S\right)$ %/
12. écrire(sss)
13. **fin**

En termes de comparaisons, la complexité de cet algorithme est en $\mathcal{O}(N)$.

3.5.2 Cas particulier et patron associé

Il arrive fréquemment que l'on soit confronté à des problèmes ressemblant au précédent, et il est intéressant de dégager un patron de programmation pour la recherche linéaire bornée pouvant être instancié plutôt que de procéder à une construction *ex nihilo*.

On se place dans le cadre de la recherche du premier élément (peu importe la structure utilisée, ensemble, sac, tableau, etc.) parmi N rendant vraie une propriété P ; j contient l'identité de cet élément ou $(N+1)$ s'il n'existe pas. On suppose de plus que la propriété P, notée $PL(j)$, est « locale » à l'élément j, au sens où son évaluation ne dépend pas des éléments examinés avant lui. Enfin, il peut arriver que, contrairement à l'exemple précédent, la propriété PL ne puisse être évaluée pour $j = N+1$. En conséquence, la condition d'arrêt utilise alors une disjonction « court-circuit ». On a donc le modèle de programme suivant. Ce patron est utilisé dans les exercices 34, page 84, et 41, page 92.

1. **constantes**
2. $N \in \mathbb{N}$ **et** $N = \ldots$
3. **variables**
4. $j \in 1\,..\,N+1$ **et** $PL \in \mathbb{B}$ **et** ...
5. **début**
6. /% *PRE* : ... %/
7. $j \leftarrow 1$; ...
8. **invariant**
9. ...
10. **terminaison**
11. $N+1-j$
12. **tant que non**$((j = N+1)$ **ou sinon** $PL)$ **faire**
13. $j \leftarrow j+1$

```
14.   fin faire ;
15.   /% POST : ... %/
16.   si j = N + 1 alors
17.     écrire(pas d'élément ...)
18.   sinon
19.     écrire(l'élément , j, ...)
20.   fin si
21. fin
```

3.6 Ce qu'il faut retenir pour construire une boucle

Un premier point clé de la construction d'une boucle est qu'elle débute par la recherche d'un invariant autour duquel s'articulent les autres composants de la boucle : initialisation, condition d'arrêt, progression et terminaison. La notion d'invariant précise les relations qu'entretiennent les variables impliquées dans la boucle.

Il n'existe pas de recette permettant de trouver à coup sûr un invariant, mais nous recommandons de le « dériver » de la postcondition du programme à construire, exprimant la situation dans laquelle on souhaite se trouver à l'issue de l'exécution de la boucle. La conjonction de l'invariant et de la condition d'arrêt *doit* impliquer logiquement la postcondition. Par suite, une heuristique de découverte d'un invariant consiste à éclater la postcondition afin de faire apparaître d'une part un invariant, de l'autre une condition d'arrêt. On peut renforcer la postcondition par l'introduction de nouvelles variables afin de faciliter un tel éclatement. Dans la mesure où un invariant constitue une propriété satisfaite à chaque pas de la boucle, on peut également le trouver en supposant le travail réalisé en partie et en formalisant la situation atteinte.

Pour sa part, la précondition de la boucle spécifie la situation avant que la boucle ne démarre. Le rôle de l'initialisation est d'instaurer l'invariant et donc de faire passer de la précondition à l'invariant. La progression doit préserver l'invariant tout en faisant « avancer » la résolution du problème. Enfin, pour s'assurer de la terminaison de la boucle, une expression entière positive ou nulle est attachée à l'évolution de la boucle dont on montre la décroissance à chaque pas.

3.7 Exercices

Exercice 29. Les haricots de Gries

Cet exercice met en évidence le fait qu'il est possible de raisonner sur un algorithme (ou un programme) donné en identifiant son invariant.

Une boîte de conserve contient une certain nombre B de haricots blancs et un certain nombre R de haricots rouges ($B + R \geqslant 1$). On dispose d'une réserve de haricots rouges suffisante pour réaliser l'opération suivante tant que c'est possible :

> On prend deux haricots au hasard dans la boîte
> si ils sont de la même couleur alors
> on les jette ; on remet un haricot rouge dans la boîte

sinon
on jette le haricot rouge ; on replace le haricot blanc dans la boîte
fin si

Question 1. Pourquoi s'arrête-t-on ? 29 - Q 1

Question 2. Que peut-on dire de la couleur du dernier haricot restant dans la boîte ? 29 - Q 2

Exercice 30. On a trouvé dans une poubelle ... ∞ •

Cet exercice met en évidence l'importance d'une construction correcte.

On a trouvé le texte suivant dans une poubelle d'une école d'informatique :

Précondition : (T un tableau constant de N entiers naturels) **et** $(N \geqslant 1)$.

Postcondition : La variable sup contient la plus grande valeur de T.

1. **Invariant** : $(i \in 1\,..\,N)$ **et** $(\forall j \cdot (j \in 1\,..\,i-1 \Rightarrow T[j] \leqslant \sup))$.
2. **Condition d'arrêt** : $(i = N)$.
3. **Progression** :
   1. **si** $T[i] > \sup$ **alors**
   2. $\quad \sup \leftarrow T[i]$
   3. **fin si** ;
   4. $i \leftarrow i + 1$
4. **Initialisation** : $i \leftarrow 2$; $\sup \leftarrow T[1]$
5. **Terminaison** : $(N + 1 - i)$

Question 1. Quelle(s) erreur(s) justifie(nt) ce rejet ? 30 - Q 1

Question 2. Fournir une version correcte. 30 - Q 2

Exercice 31. Somme des éléments d'un tableau ○ •

Cet exercice est une application simple du principe d'éclatement de la postcondition pour trouver un invariant, une fois cette dernière convenablement reformulée pour la mettre sous forme constructive.

Question 1. Construire le programme spécifié par : 31 - Q 1

Précondition : (T est un tableau d'entiers naturels constant de longueur N) **et** $(N \geqslant 0)$.

Postcondition : s représente la somme des N éléments du tableau T.

31 - Q 2 **Question** 2. Quelle en est la complexité temporelle en nombre d'additions ?

Exercice 32. Recherche dans un tableau à deux dimensions ⁘ •

Cet exercice s'intéresse au problème « élémentaire » de recherche d'une valeur dans un tableau à deux dimensions. Si une solution « naturelle » utilise deux boucles, on montre qu'il est aisé et élégant de procéder avec une seule. La démarche proposée s'étend sans difficulté à un nombre de dimensions plus élevé. On utilise l'hypothèse du travail réalisé en partie.

On considère la spécification suivante :

Précondition : (T est un tableau constant d'entiers naturels ayant L lignes et C colonnes) **et** $(L > 0)$ **et** $(C > 0)$ **et** (V est un entier naturel présent dans T).

Postcondition : (i, j) désigne une occurrence de V dans T, c'est-à-dire que $T[i, j] = V$.

32 - Q 1 **Question** 1. Proposer les éléments d'une boucle unique répondant à cette spécification, en appliquant l'hypothèse du travail réalisé en partie.

32 - Q 2 **Question** 2. En déduire le programme associé et préciser sa complexité en termes de comparaisons.

Remarque Nous invitons le lecteur à construire le programme équivalent composé de deux boucles imbriquées afin de le comparer au précédent, en particulier quant à la facilité de conception.

Exercice 33. Tri par sélection simple ∘ ⁝

On présente maintenant un programme réalisant un tri. S'il ne figure pas parmi les tris « efficaces », il n'en demeure pas moins intéressant au plan pédagogique. De plus, à la différence de l'exercice précédent, il s'appuie sur l'imbrication de deux boucles avec une démarche se révélant ici simple et progressive.

On souhaite construire le programme spécifié par :

Précondition : (T est un tableau de N entiers naturels) **et** (S est le sac des valeurs contenues dans T) **et** $(N \geqslant 1)$.

Postcondition : (T est trié par ordre croissant) **et** (S est le sac des valeurs contenues dans T).

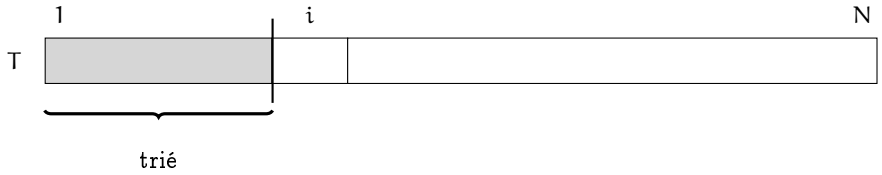
Le second conjoint de la postcondition exprime que globalement les valeurs présentes dans le tableau ne changent pas. On va en assurer le respect en ne modifiant T qu'avec la

procédure *Échanger*(i, j) qui échange les éléments de T d'indices respectifs i et j. De cette façon, on peut définitivement abandonner ce conjoint.

La postcondition n'étant pas sous forme constructive, on la renforce :

Postcondition : $(i \in 1\,..\,N+1)$ **et** (T[1 .. i − 1] est trié) **et** $(i = N+1)$.

Si l'on suppose le travail réalisé en partie, on est dans la situation représentée ci-dessous :



On entrevoit déjà que la progression vise à faire en sorte que T[1 .. i] soit trié. On a le choix entre : i) insérer T[i] à sa place dans T[1 .. i], et ii) ne pas modifier T[1 .. i − 1] en supposant toutes les valeurs de T[i .. N] supérieures ou égales à celles de T[1 .. i − 1]. La première option correspond au *tri par insertion* ; nous allons choisir la seconde, appelée *tri par sélection*, car il faut *sélectionner* la plus petite valeur de T[i .. N].

Question 1. Donner la postcondition associée à cette option, puis les constituants de la boucle en résultant, en ne spécifiant pas la partie relative à l'identification de la position m du minimum de T[i .. N]. 33 - Q 1

Question 2. Il reste maintenant à construire la boucle correspondant à l'identification de la position m du minimum du sous-tableau T[i .. N]. Ce fragment de programme est spécifié par : 33 - Q 2

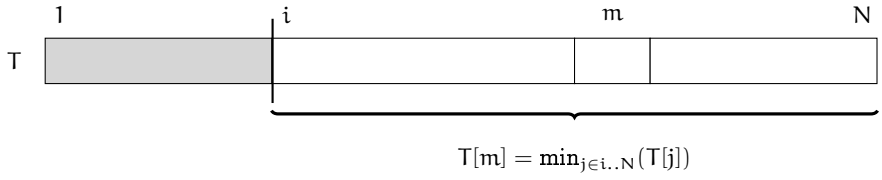
Précondition : (i constant) **et** $(i \in 1\,..\,N)$ **et** (T[i .. N] est un tableau d'entiers non vide)

IDENTIFIER LA POSITION m DU MINIMUM DU SOUS-TABLEAU T[i .. N]

Postcondition : $(m \in i\,..\,N)$ **et** $(T[m] = \min_{j \in i..N}(T[j]))$.

Le tableau T[i .. N] n'est pas vide puisque dans le contexte d'exécution de cette boucle, on a : $i \in 1\,..\,N+1$ et $i \neq N+1$. On est donc certain d'y trouver une valeur minimale.

La postcondition peut se schématiser par :



Renforcer cette postcondition afin de la mettre sous forme constructive en introduisant une variable k dans la quantification du minimum en lien avec N.

Question 3. Construire la boucle associée. 33 - Q 3

Question 4. Écrire le programme réalisant le tri par sélection du tableau T[1 .. N]. 33 - Q 4

Question 5. Quelle en est la complexité en nombre d'échanges ? 33 - Q 5

Exercice 34. Ésope reste ici et se repose ○ •

On illustre ici l'utilisation du patron proposé pour la recherche linéaire bornée (voir section 3.5, page 77) sur un exemple simple.

Soit T[1 .. N] ($N \geqslant 0$), une chaîne de caractères donnée. T est un palindrome si le mot (ou phrase sans espace) qu'elle représente s'écrit de la même façon de gauche à droite et de droite à gauche.

34 - Q 1 **Question** 1. Montrer que, si T représente un palindrome, on a la propriété :

$$\forall j \cdot ((1 \leqslant j \leqslant N) \Rightarrow (T[j] = T[N + 1 - j])).$$

34 - Q 2 **Question** 2. Préciser pourquoi et comment on peut résoudre ce problème en adaptant le patron présenté en section 3.5.2, page 79.

34 - Q 3 **Question** 3. Écrire le programme qui détermine si T représente ou non un palindrome.

34 - Q 4 **Question** 4. En déterminer la complexité en nombre de conditions évaluées.

Exercice 35. Drapeau hollandais revisité ⁘ •

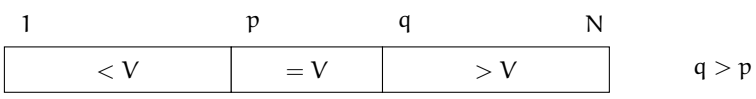
*Cet exercice illustre l'utilisation de l'hypothèse du travail réalisé en partie. Le drapeau hollandais est un problème classique, ainsi dénommé par son auteur, E.W. Dijkstra, parce que la version originale consiste à reconstituer les couleurs du drapeau de son pays. La version traitée ici est légèrement différente puisque sont considérés des entiers naturels et non des couleurs. Le fait que cet algorithme soit au cœur de l'algorithme de tri rapide (*quick sort*) en fait tout l'intérêt.*

On souhaite construire le programme spécifié par :

Précondition : (T est un tableau de N entiers naturels) **et** (S est le sac des valeurs contenues dans T) **et** ($N \geqslant 1$) **et** ($V = T[1]$).

Postcondition : (S est le sac des valeurs contenues dans T) **et** (T se compose de trois parties, à gauche les valeurs inférieures à V, à droite les valeurs supérieures à V et au centre toutes les occurrences de la valeur V).

Schématiquement, la postcondition se présente comme suit (la zone du milieu n'est pas vide puisque qu'elle contient au moins un exemplaire de V) :



Toutes les modifications de T vont s'effectuer par des échanges entre des valeurs de T au moyen de la procédure *Échanger*(i, j) où i et j désignent l'indice d'un élément de T. Par

conséquent, comme dans l'exercice 33 page 82, on peut abandonner le premier conjoint de la postcondition. La complexité de la (des) solution(s) se fait en dénombrant les appels à la procédure *Échanger*. Tout d'abord, formalisons l'expression de la postcondition :

Postcondition (version formalisée) :
$\exists(p, q) \cdot ((p \in 1\,..\,N)$ **et** $(q \in p+1\,..\,N+1)$ **et** $(\forall j \cdot (j \in 1\,..\,p-1 \Rightarrow T[j] < V))$ **et**
$(\forall j \cdot (j \in p\,..\,q-1 \Rightarrow T[j] = V))$ **et** $(\forall j \cdot (j \in q\,..\,N \Rightarrow T[j] > V))$

Cette version de la postcondition n'est pas utilisable telle quelle pour un éclatement. On la transforme (renforce) en remplaçant les variables de quantification existentielle p et q par les variables de programmation b et w (voir section 3.3.3, page 68), d'où :

Postcondition (seconde version) :
$(b \in 1\,..\,N)$ **et** $(w \in b+1\,..\,N+1)$ **et** $(\forall j \cdot (j \in 1\,..\,b-1 \Rightarrow T[j] < V))$ **et**
$(\forall j \cdot (j \in b\,..\,w-1 \Rightarrow T[j] = V))$ **et** $(\forall j \cdot (j \in w\,..\,N \Rightarrow T[j] > V))$.

À ce stade, on ne peut toujours pas procéder à un éclatement. On envisage d'introduire une variable r en association avec l'une des variables b, w et N afin de faire apparaître un nouveau conjoint.

Question 1. Exprimer la postcondition résultant de l'association de r à w, puis l'invariant dont on donnera une représentation graphique. 35 - Q 1

Question 2. Poursuivre la construction de la boucle en identifiant les cas ne demandant aucun échange et de sorte que l'on effectue au plus un échange dans un pas de progression. 35 - Q 2

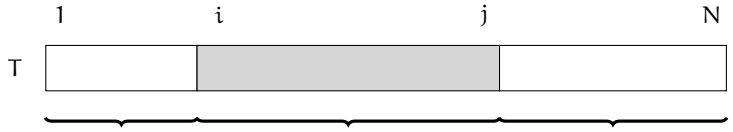
Question 3. Donner le programme résultant de cette construction et préciser sa complexité en nombre d'échanges. 35 - Q 3

Exercice 36. Les sept et les vingt-trois ○ •

Cet exercice est un autre exemple d'utilisation de l'hypothèse du travail réalisé en partie. Ici, la condition d'arrêt mérite une attention particulière.

Soit T[1 .. N] un tableau d'entiers, avec $N \geqslant 0$. On veut construire un programme qui permet d'obtenir dans T une permutation des valeurs initiales de sorte que tous les 7 soient situés avant les 23. Cette permutation ne doit différer de la configuration initiale que par la position des valeurs 7 et 23. Comme dans l'exercice précédent, la procédure *Échanger*(i, j) est supposée disponible.

Question 1. Compléter le schéma ci-après afin d'exhiber un invariant sur la base de l'hypothèse du travail réalisé en partie. 36 - Q 1



36 - Q 2 **Question** 2. Que constate-t-on si $i = j$? Peut-on prendre ce prédicat comme condition d'arrêt ?

36 - Q 3 **Question** 3. Donner les trois autres constituants de la boucle.

36 - Q 4 **Question** 4. En déduire la complexité du programme associé, à la fois en termes de comparaisons et d'échanges.

Exercice 37. Le M^e zéro ∘ ⁝

Dans cet exercice, on illustre un cas où un renforcement de la postcondition est effectué par introduction de variable en présence d'une quantification de dénombrement.

On souhaite construire le programme spécifié par :

Précondition : (M constant) **et** ($M \in \mathbb{N}_1$) **et** (T est un tableau constant de N entiers) **et** (T contient au moins M zéros).

Postcondition : i désigne la position du M^e zéro dans T.

Notons tout d'abord que la précondition implique que $N \geqslant M$. Le quantificateur # dénote le dénombrement. Ainsi :

$$\#j \cdot ((j \in 1\,..\,N) \text{ et } (T[j] = 0))$$

dénombre les zéros présents dans T. La postcondition se formalise alors de la manière suivante :

Postcondition : $(i \in 1\,..\,N)$ **et** $(\#j \cdot ((j \in 1\,..\,i-1) \text{ et } (T[j] = 0)) = M - 1)$ **et** $(T[i] = 0)$.

Bien que cette postcondition soit sous forme constructive, on peut penser que la présence de la quantification de dénombrement va être gênante pour identifier un invariant efficace.

37 - Q 1 **Question** 1. Donner une version renforcée de la postcondition dans laquelle l'expression quantifiée est identifiée à une variable p.

37 - Q 2 **Question** 2. Construire la boucle sur la base de cette nouvelle postcondition.

37 - Q 3 **Question** 3. En déduire le programme associé à la boucle et en donner la complexité en termes de comparaisons.

Exercice 38. Alternance pair – impair ○ ⁝

Cet exercice illustre la méthode d'éclatement de la postcondition après une succession de renforcements dont un de nature logique.

On considère un tableau contenant autant de nombres pairs que de nombres impairs et on veut placer chacun des nombres pairs (resp. impairs) en position d'indice pair (resp. impair). Notons que si tous les nombres pairs sont correctement placés, il en est de même des nombres impairs (et réciproquement), d'où la spécification suivante :

Précondition : (T est un tableau de 2N entiers naturels) **et** (S est le sac des valeurs contenues dans T) **et** ($N \geqslant 0$) **et** (T contient N entiers pairs et N entiers impairs).

Postcondition : (S est le sac des valeurs contenues dans T) **et** ((les positions d'indice pair contiennent les valeurs paires) **ou** (les positions d'indice impair contiennent les valeurs impaires)).

Comme dans l'exercice précédent, les évolutions du tableau T s'opèrent exclusivement au moyen de la procédure *Échanger*, ce qui permet de ne plus se préoccuper du premier conjoint de la postcondition. Celle-ci ne se présente pas sous forme conjonctive, mais il est facile de la renforcer en préservant sa symétrie :

Postcondition (deuxième version qui implique la première) : (($p \in 2\,..\,2N+2$) **et** (p est pair) **et** (les positions d'indice pair de l'intervalle $2\,..\,p-2$ contiennent des valeurs paires) **et** ($p = 2N+2$)) **ou** (($i \in 1\,..\,2N+1$) **et** (i est impair) **et** (les positions d'indice impair de l'intervalle $1\,..\,i-2$ contiennent des valeurs impaires) **et** ($i = 2N+1$)).

Cependant, il ne s'agit toujours pas d'une forme conjonctive, et il faut à nouveau renforcer cette version. Pour ce faire, posons :

$$P \mathrel{\widehat{=}} \left(\begin{array}{l}(p \in 2\,..\,2N+2) \text{ et } (p \text{ est pair}) \text{ et}\\ (\text{les positions d'indice pair de l'intervalle } 2\,..\,p-2\\ \text{contiennent des valeurs paires})\end{array}\right)$$

$$Q \mathrel{\widehat{=}} \left(\begin{array}{l}(i \in 1\,..\,2N+1) \text{ et } (i \text{ est impair}) \text{ et}\\ (\text{les positions d'indice impair de l'intervalle } 1\,..\,i-2\\ \text{contiennent des valeurs impaires})\end{array}\right)$$

ce qui permet de réécrire la postcondition en :

Postcondition (deuxième version réécrite) :

$$(P \text{ et } (p = 2N)) \text{ ou } (Q \text{ et } (i = 2N+1)).$$

Question 1. Montrer que : 38 - Q 1

$$A \text{ et } B \text{ et } (C \text{ ou } D) \Rightarrow (A \text{ et } C) \text{ ou } (B \text{ et } D).$$

En déduire une nouvelle version de la postcondition sous forme constructive impliquant la seconde.

Question 2. Donner les éléments de la boucle construite à partir de cette nouvelle postcondition. 38 - Q 2

Question 3. En déduire le programme réalisant le travail demandé. 38 - Q 3

38 - Q 4 **Question** 4. Quelle en est la complexité en nombre de conditions évaluées et d'échanges ?

38 - Q 5 **Question** 5. Justifier le fait que l'on peut aussi construire une boucle à partir de la post-condition :

> $(p \in 2..2N)$ **et** (p est pair) **et** (les positions d'indice pair de l'intervalle $2..p-2$ contiennent des valeurs paires) **et** $(i \in 1..2N+1)$ **et** (i est impair) **et** (les positions d'indice impair de l'intervalle $1..i-2$ contiennent des valeurs impaires) **et** $(p = 2N)$ **et** $(i = 2N+1)$.

38 - Q 6 **Question** 6. Expliciter la progression qui en résulte.

Exercice 39. Plus longue séquence de zéros ◦ ⁝

Cet exercice illustre le cas d'un double renforcement, à savoir de la postcondition d'une part (ce qui est classique), de l'invariant exigé par la progression de l'autre. De plus, on y met en évidence le fait que la démarche de construction proposée ne s'oppose pas à des modifications visant à améliorer l'efficacité du programme obtenu.

On souhaite construire le programme spécifié par :

Précondition : (T un tableau constant d'entiers de N éléments) **et** $(N \geqslant 0)$.

Postcondition : lg est la longueur de la plus longue succession de zéros consécutifs dans T.

La postcondition n'est pas sous forme conjonctive ; pour y parvenir, on peut introduire la variable i de la manière suivante :

Postcondition (version issue du renforcement de la précédente) : (lg est la longueur de la plus longue succession de zéros contenue dans le sous-tableau $T[1..i-1]$) **et** $(i \in 1..N+1)$ **et** $(i = N+1)$.

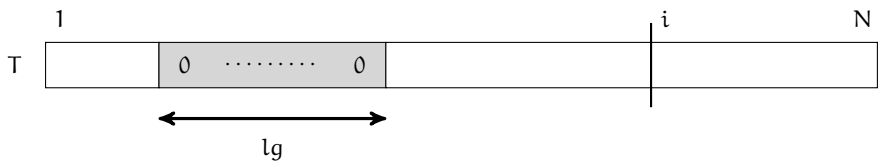
Première tentative

Cette formulation suggère un invariant et une condition d'arrêt :

1. **Invariant** Les deux premiers conjoints sont faciles à établir, on les conserve pour constituer l'invariant :

   > (lg est la longueur de la plus longue succession de zéros contenue dans le sous-tableau $T[1..i-1]$) **et** $(i \in 1..N+1)$.

   Sous forme graphique, on obtient :



2. **Condition d'arrêt** On retient le conjoint écarté : $(i = N + 1)$.

3. **Progression** Il faut rechercher un fragment de programme spécifié par :

   Précondition : (lg est la longueur de la plus longue succession de 0 contenue dans le sous-tableau $T[1\,..\,i-1]$) **et** $(i \in 1\,..\,N+1)$ **et non**$(i = N+1)$

   PROGRESSION

   Postcondition : (lg est la longueur de la plus longue succession de zéros contenue dans le sous-tableau $T[1\,..\,i-1]$) **et** $(i \in 1\,..\,N+1)$.

   La valeur $T[i]$ existe puisque $i \neq N+1$. Si $T[i] \neq 0$, il n'y a rien à faire d'autre qu'à incrémenter i. Sinon ($T[i] = 0$), il faut exprimer que $T[i]$ peut faire partie de la plus longue chaîne de zéros du sous-tableau $T[1\,..\,i]$ (avant de rétablir l'invariant), ce qui peut se faire en calculant, par une boucle rétrograde, la longueur p de la plus longue chaîne de zéros consécutifs s'achevant en i. Mais on remarquera que tous les éléments de $T[1\,..\,i-1]$ ont déjà été examinés et qu'il serait dommage de les examiner à nouveau (même en partie). On va supposer cette longueur p déjà connue, ce qui conduit à un nouvel invariant, issu du précédent par renforcement, en lui adjoignant cette hypothèse. Le prix à payer est la reconstruction de la boucle sur la base du nouvel invariant.

Seconde tentative

On effectue la construction fondée sur la suggestion précédente.

Question 1. Expliciter le nouvel invariant. 39 - Q 1

Question 2. En prenant comme condition d'arrêt $(i = N+1)$, poursuivre la construction de la boucle (progression, initialisation et expression de terminaison). 39 - Q 2

Question 3. Écrire le programme. Donner sa complexité en nombre de comparaisons. 39 - Q 3

Question 4. Proposer et justifier une autre condition d'arrêt susceptible d'améliorer l'efficacité de la boucle. 39 - Q 4

Exercice 40. Élément majoritaire

Cet exercice sur la recherche d'un élément majoritaire dans un sac est également abordé dans le chapitre « Diviser pour Régner » (voir exercice 104, page 271). La solution développée ici fait appel à une technique originale. En effet, on exploite fréquemment l'heuristique éprouvée qui consiste à renforcer *la postcondition afin d'obtenir une bonne efficacité temporelle. Ici, au contraire, on va* affaiblir *la postcondition afin d'obtenir un algorithme simple (mais ne fournissant qu'une solution* possible *devant être « confirmée »). La solution obtenue est concise, efficace et élégante.*

Si V est un tableau ou un sac, l'expression $\text{mult}(x, V)$ représente la multiplicité (c'est-à-dire le nombre d'occurrences) de la valeur x dans V. On considère un sac S de cardinal N ($N \geqslant 1$) d'entiers strictement positifs. S est dit *majoritaire* s'il existe un entier x tel que :

$$\text{mult}(x, S) \geqslant \left\lfloor \frac{N}{2} \right\rfloor + 1;$$

x est alors appelé *élément majoritaire* de S (il est unique). Le problème posé est celui de la recherche d'un élément majoritaire dans S. On cherche à construire un programme dont la spécification est :

Précondition : (S est un sac de N valeurs) **et** ($N \geqslant 1$).

Postcondition : x contient la valeur de l'élément majoritaire de S s'il existe, -1 sinon.

La détermination d'un candidat ayant obtenu la majorité absolue lors d'un scrutin ou la conception d'algorithmes tolérants aux fautes sont des applications possibles de ce problème.

Un algorithme naïf

Il est aisé de spécifier un algorithme naïf, au mieux en $\Theta(N)$ et au pire en $\Theta(N^2)$ comparaisons résolvant ce problème, en considérant la comparaison entre éléments de S comme opération élémentaire. On prend un élément quelconque de S et on compte son nombre d'occurrences dans S. S'il n'est pas majoritaire, on prend un autre élément de S, et ainsi de suite jusqu'à trouver un élément majoritaire ou avoir traité les éléments de la moitié de S sans en avoir trouvé.

Une seconde approche

Afin d'améliorer la complexité au pire, on envisage une solution en trois temps : tout d'abord un raffinement de S par un tableau T suivi d'un tri de T, puis la recherche d'une séquence de valeurs identiques de longueur supérieure à $\lfloor N/2 \rfloor$. La première étape est en $\Theta(N)$, la seconde en $\mathcal{O}(N \cdot \log_2(N))$ et la troisième de complexité linéaire, d'où une complexité au pire en $\mathcal{O}(N \cdot \log_2(N))$.

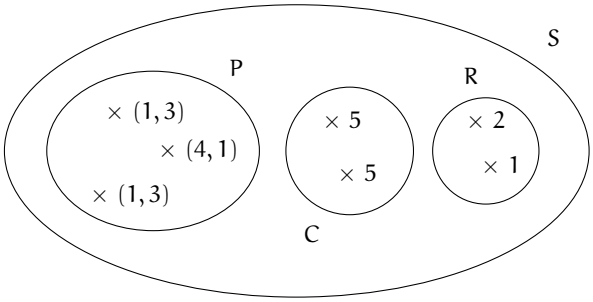
Une solution itérative efficace

La solution envisagée maintenant se fonde sur une structure de données *abstraite* constituée de quatre sacs. Après avoir proposé un invariant pour l'itération, certaines propriétés de cet invariant sont mises en évidence afin de permettre la construction d'un programme correct. On effectue ensuite un raffinement de la structure de données abstraite en éliminant trois des quatre sacs initiaux et en n'utilisant qu'un tableau constant et des variables scalaires dans l'algorithme final.

Invariant Soit S le sac de valeurs pour lequel on recherche un éventuel élément majoritaire et sa partition multiensembliste (hypothèse du travail réalisé en partie) composée de :

- R le sac des valeurs « restant à traiter »,
- P un sac des paires de valeurs tel que, pour toute paire, les deux valeurs sont différentes,
- C un sac dénommé « sac des célibataires » dont tous les éléments ont la même valeur.

Exemple Soit $S = [\![1,3,4,1,1,3,5,2,5,1]\!]$ et la configuration possible ci-dessous :



La partition de S (notamment dans la situation décrite ci-dessus) satisfait les propriétés suivantes :

Propriété 2 :
$|S| = 2 \cdot |P| + |C| + |R|$.

Cette propriété est triviale, et nous ne la démontrons pas.

Propriété 3 :
Si les sacs R *et* C *sont vides, alors* S *n'est pas majoritaire.*

Propriété 4 :
Si le sac R *est vide mais* C *ne l'est pas (appelons* c *la valeur présente dans* C*) alors :*

1. *si* S *possède un élément majoritaire, c'est la valeur* c*,*
2. *si* S *ne possède pas d'élément majoritaire, on ne peut rien affirmer à propos de* c *(pas même que* c *est l'élément qui possède la plus grande multiplicité dans* S*).*

Question 1. Démontrer les propriétés 3 et 4. 40 - Q 1

Question 2. On va construire une boucle dont l'invariant correspond à une situation où S est partitionné en P, C et R et dans laquelle la condition d'arrêt survient quand R est vide. Quelle conclusion peut-on tirer alors ? 40 - Q 2

Question 3. Préciser la précondition et la postcondition de la boucle abstraite construite sur l'invariant proposé. Comment peut-on répondre au problème initial à partir de cette boucle ? 40 - Q 3

Question 4. Compléter la construction de la boucle abstraite (progression, initialisation, terminaison). 40 - Q 4

Question 5. Comment peut-on raffiner la structure de données sur la base de variables scalaires et d'un tableau ? En déduire un algorithme itératif résolvant le problème de la recherche de l'élément majoritaire d'un sac S. Quelle en est la complexité en nombre de comparaisons ? 40 - Q 5

Exercice 41. Cherchez la star

De façon analogue à l'exercice relatif à l'élément majoritaire (voir exercice 40, page 89), on cherche une solution en affaiblissant la postcondition. La solution retenue fait, pour partie, appel à la recherche linéaire bornée (voir section 3.5, page 77). La solution obtenue se révèle élégante et de complexité linéaire.

Dans un groupe de N personnes, une *star* est une personne que tout le monde connaît et qui ne connaît personne. On considère un groupe de N ($N \geqslant 1$) personnes et on cherche une star dans le groupe, s'il y en a une. Par convention, un groupe d'une seule personne a cette personne pour star. La seule opération autorisée, notée *Connaît*(i, j) est de choisir une personne i et de lui demander si elle connaît la personne j (différente d'elle-même). L'opération *Connaît*(i, j) rend **vrai** ou **faux** en fonction de la réponse (obligatoire et sincère) de la personne interrogée.

Comme il y a $N(N-1)/2$ paires de personnes, le problème peut être à coup sûr résolu par au plus $N(N-1)$ opérations. Cependant, on cherche à effectuer moins d'opérations, si possible un (petit) multiple de N.

41 - Q 1 **Question 1.** Montrer qu'un groupe a au plus une star.

41 - Q 2 **Question 2.** Montrer qu'en l'absence d'autre information quand on effectue l'opération *Connaît*(i, j) :

a) si la réponse est **vrai**, alors j peut être la star, mais pas i,

b) si la réponse est **faux**, alors i peut être la star, mais pas j.

41 - Q 3 **Question 3.** Donner la précondition et la postcondition du programme résolvant le problème posé.

41 - Q 4 **Question 4.** Spécifier les constituants d'une boucle identifiant une personne susceptible d'être la star du groupe et appelée star « potentielle ».

41 - Q 5 **Question 5.** Spécifier les autres composants du programme trouvant la star du groupe ou prouvant l'absence de star dans le groupe et donner le programme résolvant le problème initial.

41 - Q 6 **Question 6.** Quelle est la complexité de ce programme en prenant *Connaît* (i, j) comme opération élémentaire ?

Exercice 42. Affaiblissement de la précondition ○ ⁝

Dans cet exercice, on s'intéresse à l'affaiblissement de la précondition. Cette démarche qui est légitime, consiste à résoudre un problème en se reportant à un problème plus général dont on connaît la solution. L'affaiblissement est réalisé par élimination d'une hypothèse et on étudie l'impact de ce choix sur les performances à travers deux exemples.

On a vu en section 3.3.2 qu'il est légal que, cherchant à résoudre le problème spécifié par $\{P\}\ prog\ \{Q\}$, on résolve le problème spécifié par $\{P'\}\ prog'\ \{Q\}$ avec $P \Rightarrow P'$. En d'autres termes, on affaiblit la précondition et on peut s'interroger sur l'impact de cette démarche au plan des performances. Deux exemples sont traités pour éclairer cette question.

Tri croissant d'un tableau trié par ordre décroissant

On souhaite construire le programme spécifié par :

Précondition : (T est un tableau de N entiers naturels) **et** (S est le sac des valeurs contenues dans T) **et** ($N \geqslant 1$) **et** (T est trié par ordre décroissant).

Postcondition : (T est trié par ordre croissant) **et** (S est le sac des valeurs contenues dans T).

Deux stratégies s'offrent alors : i) on développe un algorithme spécifique tenant compte du fait que T est trié par ordre décroissant, ou ii) on *affaiblit la précondition* en supprimant le conjoint qui précise que le tableau est trié.

Question 1. Proposer le principe d'une solution de complexité linéaire en nombre d'échanges dans le cadre de la première option. 42 - Q 1

Question 2. Quelle complexité atteint-on si l'on utilise l'algorithme proposé dans l'exercice 33, page 82 ? Peut-on espérer mieux en recourant à un algorithme de tri « classique » ? Conclure. 42 - Q 2

Tableau à variation contrainte

On considère le programme dont la spécification est :

Précondition : (T est un tableau d'entiers naturels constant de N éléments) **et** (il existe au moins un zéro dans le tableau T) **et** (l'écart entre deux éléments consécutifs de T est d'au plus 1).

Postcondition : La variable i désigne le zéro de plus petit indice de T.

Cette spécification diffère de celle de l'exemple traité page 69, puisque l'on dispose d'une propriété sur la variation des nombres présents dans T. Ici encore, deux stratégies sont envisageables : programme spécifique ou utilisation de l'algorithme proposé page 71. Cette seconde option correspond à l'affaiblissement de la précondition du problème posé dont on ignore le troisième conjoint.

Question 3. Montrer que si T est tel que l'écart entre deux éléments consécutifs de T est d'au plus 1, alors : 42 - Q 3

$$(T[i] > 0) \Rightarrow \forall j \cdot (j \in i\,..\,i + T[i] - 1 \Rightarrow T[j] > 0).$$

42 - Q 4 **Question 4.** Développer la solution spécifique fondée sur une boucle dont on donnera les constituants.

42 - Q 5 **Question 5.** Que dire de l'impact du choix en termes de complexité ?

42 - Q 6 **Question 6.** On considère maintenant le programme spécifié par :

Précondition : (T est un tableau d'entiers naturels constant de N éléments) **et** (il existe au moins un zéro dans le tableau T) **et** (l'écart entre deux éléments consécutifs de T est d'au moins 1).

Postcondition : La variable i désigne le zéro de plus petit indice de T.

Commenter l'affaiblissement de la précondition en :

Précondition : (T est un tableau d'entiers naturels constant de N éléments) **et** (il existe au moins un zéro dans le tableau T).

0 - R 7 **Question 7.** Comparer les trois situations d'affaiblissement précédentes.

Exercice 43. Meilleure division du périmètre d'un polygone ○ ⁝

Le problème traité ici est une variante assez sophistiquée de la recherche séquentielle. Habituellement, dans un problème de ce type, l'emploi de la technique « d'arrêt au plus tôt » n'a pas d'incidence sur la complexité asymptotique. Ce n'est pas le cas ici, où elle est la clé de voûte de l'amélioration d'efficacité obtenue par rapport à une méthode naïve. Par ailleurs, ce problème trouve une formulation dans un espace à deux dimensions généralement résolue par deux boucles imbriquées. Ici, comme dans l'exercice 32 page 82, l'utilisation d'une seule boucle rend la construction plus simple.

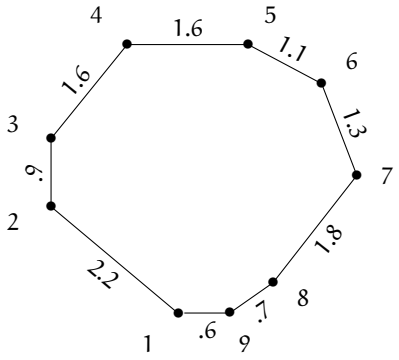
On considère un polygone quelconque d'ordre N fini ($N > 2$) dont les sommets sont étiquetés de 1 à N dans le sens des aiguilles d'une montre. On recherche l'une quelconque des cordes qui partage le périmètre p du polygone « le plus exactement possible », c'est-à-dire telle que la valeur absolue de la différence entre la somme des longueurs des côtés délimités par la corde soit minimale. La complexité est évaluée en nombre de sommets visités.

Définitions – Notations – Représentation

Dans la suite, $|a|$ dénote la valeur absolue de a. Le côté qui a comme origine le sommet i est noté (i). La corde qui joint les sommets i et j est notée $\overline{(i,j)}$. L'arc qui va du sommet i au sommet j est noté $\overset{\frown}{(i,j)}$. La longueur d'un arc a est notée $\|a\|$.

Un polygone d'ordre N est représenté par un tableau d ($d \in 1\,..\,N \to \mathbb{R}_+$) tel que $d[i]$ est la longueur du côté (i). La figure 3.4, illustre le cas d'un polygone d'ordre 9.

Plus formellement, si l'opérateur $\cdot$ dénote la concaténation d'arcs adjacents, un arc (non vide) se définit de façon inductive de la manière suivante comme une succession de côtés.



$$\begin{array}{lcl} d[1] & = & 2.2 \\ d[2] & = & 0.9 \\ d[3] & = & 1.6 \\ d[4] & = & 1.6 \\ d[5] & = & 1.1 \\ d[6] & = & 1.3 \\ d[7] & = & 1.8 \\ d[8] & = & 0.7 \\ d[9] & = & 0.6 \end{array}$$

Fig. 3.4 – *Exemple d'un polygone d'ordre 9 et de sa représentation par le tableau* d

Définition 20 (Arc) :
Cas de base :

$$\begin{cases} (i, \overset{\frown}{i+1}) & = (i) & i \in 1\,..\,N-1 \\ (\overset{\frown}{N,1}) & = (N). \end{cases}$$

Cas inductif :

$$\begin{cases} (\overset{\frown}{i,j}) & = (i, \overset{\frown}{i+1}) \cdot (\overset{\frown}{i+1}, j) & j \neq i+1 \text{ et } i \neq N \\ (\overset{\frown}{N,j}) & = (\overset{\frown}{N,1}) \cdot (\overset{\frown}{1,j})) & j \neq 1. \end{cases}$$

De même, la longueur d'un arc se définit comme suit.

Définition 21 (Longueur d'un arc) :
Cas de base :

$$\begin{cases} \|(i, \overset{\frown}{i+1})\| & = d[i] & i \in 1\,..\,N-1 \\ \|(\overset{\frown}{N,1})\| & = d[N]. \end{cases}$$

Cas inductif :

$$\begin{cases} \|(\overset{\frown}{i,j})\| & = d[i] + \|(\overset{\frown}{i+1}, j)\| & j \neq i+1 \text{ et } i \neq N \\ \|(\overset{\frown}{N,j})\| & = d[N] + \|(\overset{\frown}{1,j}))\| & j \neq 1. \end{cases}$$

Définition 22 (Corde localement optimale) :
La corde $\overline{(j,k)}$ *est localement optimale par rapport à son origine* j *si :*

$$\left| \|(\overset{\frown}{j,k})\| - \|(\overset{\frown}{k,j})\| \right| = \min_{i \in 1..N-\{j\}} \left(\left| \|(\overset{\frown}{j,i})\| - \|(\overset{\frown}{i,j})\| \right| \right).$$

Définition 23 (Corde globalement optimale) :
La corde $\overline{(j,k)}$ *est globalement optimale si :*

$$\left|\|\overset{\frown}{(j,k)}\| - \|\overset{\frown}{(k,j)}\|\right| = \min_{\substack{u \in 1\,..\,N \text{ et} \\ v \in 1\,..\,N \text{ et} \\ u \neq v}} \left(\left|\|\overset{\frown}{(u,v)}\| - \|\overset{\frown}{(v,u)}\|\right|\right). \tag{3.1}$$

Le problème

L'objectif de l'exercice est de trouver une corde globalement optimale (le problème possède au moins une solution). Une méthode consiste à évaluer la formule 3.1 et à retenir l'un des couples de sommets qui la satisfait. L'algorithme correspondant est en $\Theta(N^2)$. Il est également possible d'exploiter l'identité suivante afin d'éviter des calculs inutiles (p est le périmètre du polygone) :

$$\left|\|\overset{\frown}{(i,j)}\| - \|\overset{\frown}{(j,i)}\|\right| = 2 \cdot \left|\|\overset{\frown}{(i,j)}\| - \frac{p}{2.0}\right|. \tag{3.2}$$

Les couples de sommets (j,k) qui satisfont la formule (3.1) sont également ceux qui vérifient :

$$\frac{\left|\|\overset{\frown}{(j,k)}\| - \|\overset{\frown}{(k,j)}\|\right|}{2.0} = \min_{\substack{u \in 1\,..\,N \text{ et} \\ v \in 1\,..\,N \text{ et} \\ u \neq v}} \left|\|\overset{\frown}{(u,v)}\| - \frac{p}{2.0}\right|.$$

Mais, puisque $|a-b| = |b-a|$, pour visiter tous les arcs, la variable v n'a pas besoin de parcourir la totalité de l'ensemble $1\,..\,N - \{u\}$. Il suffit qu'elle prenne ses valeurs dans l'intervalle $u+1\,..\,N \cup \{1\}$, d'où :

$$\frac{\left|\|\overset{\frown}{(j,k)}\| - \|\overset{\frown}{(k,j)}\|\right|}{2.0} = \min_{\substack{u \in 1\,..\,N \text{ et} \\ v \in u+1\,..\,N \cup \{1\}}} \left|\|\overset{\frown}{(u,v)}\| - \frac{p}{2.0}\right|.$$

Si $\overline{(j,k)}$ est une corde *localement* optimale issue de j, la valeur $\left(\left|\|\overset{\frown}{(j,k)}\| - p/2.0\right|\right)$ est notée δ_j. Les longueurs des arcs intervenant dans le calcul peuvent être rassemblées dans la matrice triangulaire suivante (appelée triangle des longueurs dans la suite) :

$$
\begin{array}{ccccccc}
\|\overset{\frown}{(1,2)}\| & \|\overset{\frown}{(1,3)}\| & \cdots\cdots\cdots & & \|\overset{\frown}{(1,1)}\| \\
 & \|\overset{\frown}{(2,3)}\| & \cdots\cdots\cdots & & \|\overset{\frown}{(2,1)}\| \\
 & & \vdots & & \\
 & \|\overset{\frown}{(i,i+1)}\| & \cdots\cdots & & \|\overset{\frown}{(i,1)}\| \\
 & & \ddots & & \vdots \\
 & & & \ddots & \vdots \\
 & & & & \vdots \\
 & & & & \|\overset{\frown}{(N,1)}\|
\end{array}
$$

On notera que le coin supérieur droit du triangle donne la valeur du périmètre $p = \|\overset{\frown}{(1,1)}\|$ du polygone. Dans le cas du polygone de la figure 3.4 page 95, $p = 11.8$, et ce triangle s'instancie de la manière suivante :

	2	3	4	5	6	7	8	9	1
1	2.2	3.1	4.7	6.3	7.4	8.7	10.5	11.2	11.8
2		0.9	2.5	4.1	5.2	6.5	8.3	9.0	9.6
3			1.6	3.2	4.3	5.6	7.4	8.1	8.7
4				1.6	2.7	4.0	5.8	6.5	7.1
5					1.1	2.4	4.2	4.9	5.5
6						1.3	3.1	3.8	4.4
7							1.8	2.5	3.1
8								0.7	1.3
9									0.6

Cependant, ce type de triangle contient $(N(N+1))/2$ éléments et les évaluer tous conduit à nouveau à une solution en $\Theta(N^2)$. Ce résultat peut être amélioré sur la base des quatre observations suivantes :

1. Pour un sommet j donné, il existe soit une, soit deux cordes optimales locales.
2. Si la corde $\overline{(j,k)}$ est la première[1] corde optimale locale issue de j, alors aucune des cordes $\overline{(j+1,j+2)}$, $\overline{(j+1,j+3)}$,...,$\overline{(j+1,k-1)}$ n'est globalement aussi bonne que $\overline{(j,k)}$.
3. Soit $\overline{(j,k)}$ la première corde optimale locale issue de j. Si elle existe, la corde $\overline{(j+1,k)}$ peut être meilleure que $\overline{(j,k)}$.
4. Lorsque, pour le sommet j, on a découvert une corde localement optimale $\overline{(j,k)}$, il est inutile d'évaluer l'optimalité de $\overline{(j,k+1)}$,..., $\overline{(j,N+1)}$ (principe classique de « l'arrêt au plus tôt »).

Il convient de noter que, pour ce qui concerne l'observation 3, d'autres cordes issues du sommet $j+1$ peuvent surclasser la corde $\overline{(j+1,k)}$.

Question 1. Démontrer l'identité 3.2, page 96. 43 - Q 1

Question 2. Dans quel cas existe-t-il deux cordes optimales locales issues d'un même sommet j ? Proposer un exemple. 43 - Q 2

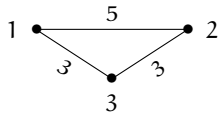
1. la première *rencontrée* en effectuant le parcours selon le sens des aiguilles d'une montre.

43 - Q 3 **Question** 3. Démontrer la proposition relative à la seconde observation ci-dessus.

43 - Q 4 **Question** 4. Démontrer la proposition relative à la troisième observation ci-dessus.

43 - Q 5 **Question** 5. Dans l'exemple de la figure 3.4, page 95, quels sont les arcs dont la longueur intervient dans le calcul ? Quelle corde globalement optimale est-elle découverte ?

43 - Q 6 **Question** 6. Considérons l'exemple suivant :



Fournir, sous la forme d'un triangle des longueurs, les arcs dont la longueur intervient dans le calcul. Que constate-t-on ?

43 - Q 7 **Question** 7. On décide de construire une solution algorithmique sur la base d'*une seule boucle*. Proposer un invariant pour cette boucle.

43 - Q 8 **Question** 8. Compléter la construction de la boucle (condition d'arrêt, progression, initialisation, terminaison) et produire le code de cette solution. Quelle est sa complexité (en nombre de sommets visités) ?

CHAPITRE 4

Diminuer pour résoudre, récursivité

Le caméléon n'a la couleur du caméléon que lorsqu'il est posé sur un autre caméléon.

(F. Cavanna)

4.1 Quelques rappels sur la récursivité

De nombreux problèmes ont une solution s'exprimant naturellement et efficacement de façon récursive. Il importe de s'assurer de la correction des algorithmes (ou procédures) associés ainsi que de leur terminaison. Dans les cas les plus simples, de façon similaire à une récurrence simple comportant un cas terminal et un terme général, une procédure récursive s'écrit comme une alternative, avec d'une part un cas d'arrêt, de l'autre un appel récursif. Cependant, si ce type de construction est souhaitable, il n'est garant ni de la terminaison, ni de la correction de la procédure. Considérons la fonction suivante :

1. **fonction** $Calc(n)$ **résultat** $\mathbb{Z}$ **pré**
2. $\quad n \in \mathbb{Z}$
3. **début**
4. $\quad$ **si** $n > 100$ **alors**
5. $\quad\quad$ **résultat** $n - 10$
6. $\quad$ **sinon**
7. $\quad\quad$ **résultat** $Calc(Calc(n + 11))$
8. $\quad$ **fin si**
9. **fin**

Quand le paramètre d'appel est strictement supérieur à 100, la terminaison est évidente, mais qu'en est-il sinon ? Le lecteur pourra vérifier à l'exercice 44, page 109, que dans cette hypothèse cette fonction se termine toujours, et que le résultat retourné est la valeur 91.

Nous donnons maintenant un exemple de problème pour lequel une approche récursive permet d'élaborer une solution simple et élégante. Soit une suite de nombres entiers positifs terminée par un marqueur sous forme du nombre négatif -1 ou -2. Si le marqueur est -1, on doit faire la somme des valeurs entrées, dans le cas contraire leur produit. On ne veut lire la suite de nombres qu'une seule fois et ne pas la stocker explicitement (dans un tableau par exemple). En fait, on va utiliser la récursivité pour « engranger » chacun des nombres lus dans la variable locale nb. L'opération appropriée (somme ou produit) associée au marqueur (variable globale $marq$) sera effectuée en retour de récursivité, et l'on aura bien au final la valeur voulue (variable globale res) grâce à la procédure suivante :

```
1.  procédure SomProd pré
2.      nb ∈ ℕ1 ∪ {−1, −2}
3.  début
4.      lire(nb) ;
5.      si nb > 0 alors
6.          SomProd ;
7.          si marq = −1 alors
8.              res ← res + nb
9.          sinon
10.             res ← res · nb
11.         fin si
12.     sinon
13.         marq ← nb ;
14.         si marq = −1 alors
15.             res ← 0
16.         sinon
17.             res ← 1
18.         fin si
19.     fin si
20. fin
```

Ici, aucune structure de mémorisation explicite n'est utilisée pour stocker les nombres à additionner ou multiplier, ceux-ci sont mémorisés dans la pile d'exécution liée à la gestion de la récursivité. Aucune « économie d'espace » n'est donc réalisée. On remarquera que la procédure *SomProd* ne se résume pas à une alternative, même si elle en comporte bien une gérant la distinction entre cas d'arrêt et cas récursif.

4.2 Relation de récurrence et récursivité

Au chapitre 1, on a vu que la solution d'un certain nombre de problèmes fait appel à une relation de récurrence. La mise en œuvre associée s'est traduite par une procédure itérative visant à remplir une structure tabulaire. Une alternative à ce choix consisterait à implanter une procédure récursive. Si une telle approche est parfaitement légitime, elle se révèle en général moins efficace.

Prenons comme exemple la récurrence associée au calcul du nombre d'arbres binaires ayant n nœuds :

$$\begin{aligned} &nbab(0) = 1 \\ &nbab(n) = \sum_{i=0}^{n-1} nbab(i) \cdot nbab(n-i-1) \qquad n \geqslant 1. \end{aligned}$$

La procédure récursive associée $nbArBin(n)$ est :

```
1.  fonction NbArBin(n) résultat ℕ1 pré
2.      n ∈ ℕ1 et som ∈ ℕ
3.  début
4.      si n = 0 alors
5.          résultat 1
```

```
6.    sinon
7.       som ← 0 ;
8.       pour i parcourant 0 .. n − 1 faire
9.          som ← som + NbArBin(i) · NbArBin(n − 1 − i)
10.      fin pour ;
11.      résultat som
12.   fin si
13. fin
```

et pour calculer le nombre d'arbres binaires ayant 15 nœuds, on effectue l'appel :

```
1. constantes
2.    n = 15
3. début
4.    écrire(le nombre d'arbres binaires ayant , n, nœuds est , NbArBin(n))
5. fin
```

qui rend la même valeur que celle retournée par le programme itératif de la page 13.

Cependant, chaque nombre $nbab$ est ici calculé de multiples fois. Pour s'en rendre compte, il suffit d'observer ce qui se passe avec $n = 5$. L'appel $NbArBin(5)$ provoque deux appels à $NbArBin(0)$, $NbArBin(1)$, $NbArBin(2)$, $NbArBin(3)$ et $NbArBin(4)$. Ceux-ci, à l'exception de $NbArBin(0)$, provoquent à leur tour des paires de nouveaux appels (par exemple $NbArBin(4)$ appelle à son tour $NbArBin(0)$, $NbArBin(1)$, $NbArBin(2)$ et $NbArBin(3)$), et ainsi de suite. De façon plus précise, le nombre $nbm(n)$ de multiplications nécessaires au calcul de $nbab(n)$ est donné par la récurrence :

$$\begin{aligned} &nbm(0) = 0 \\ &nbm(n) = \sum_{i=0}^{n-1} (1 + nbm(i) + nbm(n-i-1)) = 1 + 2 \cdot nbm(n-1) \qquad n \geqslant 1 \end{aligned}$$

dont la solution est $nbm(n) = (3^n - 1)/2$ pour tout $n \geqslant 1$ (résultat pouvant être démontré par récurrence forte). Le nombre de multiplications exigé par cet algorithme est donc en $\Theta(3^n)$, alors que celui de la page 13 est en $\Theta(n^2)$.

Une autre illustration de l'intérêt d'utiliser une mise en œuvre itérative du calcul d'une grandeur définie par une relation de récurrence est donnée dans l'exercice 45 en page 109.

4.3 Diminuer pour résoudre et sa complexité

Dans cette section, on s'intéresse à une famille particulière de procédures récursives résolvant un problème de taille donnée n, en faisant appel à la résolution de problèmes identiques de taille $(n - 1)$. Le chapitre 8 traite du cas plus général où les problèmes auxquels on fait appel ne sont pas toujours de même taille.

4.3.1 Présentation

On considère un problème donné unidimensionnel de taille fixée n que l'on note $Pb(n)$. Pour le résoudre, on fait appel à a sous-problèmes de taille $(n - 1)$ (d'où le terme « diminuer ») de même nature que le problème initial et à une fonction complémentaire f. Cette dernière vise d'une part à générer les sous-problèmes, d'autre part à « composer »

leurs solutions afin d'obtenir le résultat global. Il doit exister une taille n_0 (0 ou 1 le plus souvent en pratique) pour laquelle on sait résoudre le problème directement (on parle alors de *problème élémentaire*), c'est-à-dire sans recourir aux sous-problèmes. On a donc le schéma de résolution par diminution (aussi appelé modèle de diminution) :

$$\begin{array}{|ll|}\hline \text{Pb}(0) \text{ ou } \text{Pb}(1) \text{ élémentaire} & \\ \text{Pb}(n) \to a \cdot \text{Pb}(n-1) + f(n) & n > 0 \text{ ou } n > 1. \\ \hline \end{array}$$

Quand $a = 1$, on a une récursion « classique » qui peut très souvent être transformée de façon simple en une itération (voir chapitre 1). Dans le cas général, la structure du programme est :

```
1.  procédure DiminuerPourRésoudre(n) pré
2.      n ∈ ℕ1
3.  début
4.      si n = ... alors
5.          Résoudre le problème élémentaire associé
6.      sinon
7.          DiminuerPourRésoudre(n − 1) ;
8.                  ...          /% a occurrences de l'appel %/
9.          DiminuerPourRésoudre(n − 1) ;
10.         Rassembler
11.     fin si
12. fin
```

où la procédure générique *Rassembler* a comme objectif de composer le résultat final à partir des résultats partiels élaborés en réponse à chacun des sous-problèmes engendrés. L'appel se fait par la séquence :

```
1.  constantes
2.      m ∈ ℕ1 et m = ...
3.  début
4.      DiminuerPourRésoudre(m)
5.  fin
```

La complexité d'un tel programme s'exprime en fonction d'une opération élémentaire liée au problème spécifique considéré, celle-ci intervenant dans le cas général du modèle de diminution (le cas échéant dans la fonction f) et/ou dans le problème élémentaire. En supposant que la complexité de f soit en $\Theta(n^k)$ (le plus souvent avec $k = 0$, c'est-à-dire une fonction f de complexité constante), l'équation de complexité s'écrit :

$$\begin{array}{|ll|}\hline C(1) = d \quad (d \in \Theta(1)) & \\ C(n) = a \cdot C(n-1) + c \cdot n^k & n > 1 \\ \hline \end{array}$$

dont la solution générale est $C(n) = a^{n-1} \cdot d + c \cdot \sum_{i=0}^{n-2} a^i \cdot (n-i)^k$, solution qui peut être obtenue par la méthode des facteurs sommants. En effet, on a :

$$\begin{aligned} C(n) &= a \cdot C(n-1) + c \cdot n^k \\ C(n-1) &= a \cdot C(n-2) + c \cdot (n-1)^k \end{aligned}$$

$$
\begin{aligned}
&C(n-2) = a \cdot C(n-3) + c \cdot (n-2)^k \\
&\ldots \\
&C(2) = a \cdot C(1) + c \cdot (2)^k \\
&C(1) = d
\end{aligned}
$$

d'où on tire en multipliant la seconde ligne par a, la troisième par a^2, ..., l'avant-dernière par a^{n-2}, la dernière par a^{n-1} et en sommant les termes de gauche d'une part et de droite d'autre part :

$$
\begin{aligned}
&C(n) + a \cdot C(n-1) + a^2 \cdot C(n-2) + \cdots + a^{n-2} \cdot C(2) + a^{n-1} \cdot C(1) = \\
&a \cdot C(n-1) + c \cdot n^k + a^2 \cdot C(n-2) + a \cdot c \cdot (n-1)^k + a^3 \cdot C(n-3) + a^2 \cdot c \cdot (n-2)^k + \cdots + a^{n-1} \cdot C(1) + a^{n-2} \cdot c \cdot (2)^k + a^{n-1} \cdot d.
\end{aligned}
$$

En simplifiant, on obtient :

$$C(n) = c \cdot n^k + a \cdot c \cdot (n-1)^k + \cdots + a^2 \cdot c \cdot (n-2)^k + \cdots + a^{n-2} \cdot c \cdot (2)^k + a^{n-1} \cdot d,$$

soit finalement :

$$C(n) = a^{n-1} \cdot d + c \cdot \sum_{i=0}^{n-2} a^i \cdot (n-i)^k.$$

On en déduit les solutions particulières :

$$
\begin{aligned}
&k = 0 \ \text{ et } \ a = 1 \longrightarrow C(n) = d + c \cdot \sum_{i=0}^{n-2} 1, \ \text{ soit } C(n) \in \Theta(n) \\
&k = 0 \ \text{ et } \ a > 1 \longrightarrow C(n) = a^{n-1} \cdot d + c \cdot \sum_{i=0}^{n-2} a^i, \ \text{ soit } C(n) \in \Theta(a^n) \\
&k = 1 \ \text{ et } \ a = 1 \longrightarrow C(n) = d + c \cdot \sum_{i=0}^{n-2} (n-i), \ \text{ soit } C(n) \in \Theta(n^2).
\end{aligned}
$$

Afin de s'assurer de la correction de la solution proposée, on va s'appuyer de façon systématique sur une construction inductive (généralement récurrence simple sur $\mathbb{N}$ ou $\mathbb{N}_1$) à trois constituants : 1) la *base* dans laquelle on explicite le cas élémentaire et la solution associée, 2) l'*hypothèse d'induction* où il est supposé que l'on sait résoudre le problème de taille $(n-1)$ pour $(n-1)$ supérieur ou égal à la valeur de la base, et 3) l'*induction* à proprement parler où on démontre comment résoudre le problème $Pb(n)$ de taille n en utilisant l'hypothèse d'induction. On omettra un quatrième élément apparaissant généralement dans ce type de preuve, à savoir la *terminaison*, qui, ici, est garantie pour autant que l'on ait une taille de problème entière au moins égale à la valeur spécifiée pour la base (puisque partant d'une valeur entière positive supérieure ou égale à celle de la base, on obtient la valeur de la base par soustraction successive de la valeur 1).

Au travers de deux exemples, on illustre maintenant les cas $a = 1$ et $a = 2$ pour $k = 0$.

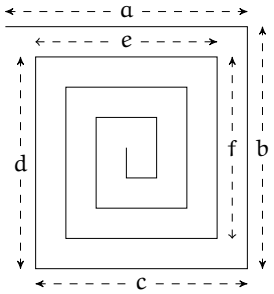


Fig. 4.1 – *Une spirale pour* $n = 4$

4.3.2 Exemple 1 : le dessin en spirale

On souhaite réaliser le dessin « en spirale » illustré par la figure 4.1, par une méthode de type « Diminuer pour Résoudre ». On dira que l'on a une spirale de taille n si la figure est composée de n « pseudo carrés » imbriqués ($n = 4$ dans la figure 4.1). On fixe $a = b, c = d = a - \alpha, e = a - 2\alpha$, avec α donné. De plus, le dessin se fait de l'extérieur vers l'intérieur ; il débute donc par le segment horizontal supérieur de longueur a. Enfin, on dispose de deux primitives pour effectuer les dessins : $placer(x, y)$ qui positionne la plume baissée au point de coordonnées (x, y), et $tracer(x, y)$ qui trace le segment allant du point courant au point (x, y).

Pour poursuivre le dessin, il faut d'abord s'interroger sur la valeur que doit prendre f (voir figure 4.1). Puisqu'à l'étape 1, on a $a = b, c = d = a - \alpha$, pour que le sous-problème auquel il va être fait appel soit de même nature que le problème initial, il faut que f égale e, soit $f = a - 2\alpha$.

Compte tenu de la nature du dessin, n, a et α doivent satisfaire une précondition afin que le tracé du dessin en spirale de taille n puisse être réalisé. En effet, le tracé impose de pouvoir retrancher $(n - 1)$ fois 2α à a ; on doit donc avoir : $a > 2 \cdot \alpha \cdot (n - 1)$.

Ces préliminaires étant posés, on va préciser le modèle de résolution par diminution pour ce problème, puis proposer une procédure récursive effectuant ce dessin, d'en-tête $DessinEnSpirale(n, x, y, a, \alpha)$ où n est le nombre de « pseudo carrés » à dessiner, x et y sont les coordonnées du point de départ du dessin, a et α correspondent aux symboles définis auparavant. La construction de ce dessin obéit au schéma inductif suivant.

Base On sait réaliser le dessin en spirale de taille 0 qui consiste à ne rien tracer.

Hypothèse d'induction On admet que, pour autant que a soit supérieur à $2 \cdot \alpha \cdot (n-1)$, on sait réaliser le dessin en spirale de taille $(n - 1)$ $(n - 1 \geqslant 0)$ ayant un segment horizontal supérieur de longueur l égale à celle de son segment vertical droit et ses segments horizontal inférieur et vertical gauche de longueur $l - \alpha$.

Induction On spécifie comment réaliser le dessin en spirale de taille n en supposant a supérieur à $2 \cdot \alpha \cdot (n - 1)$. On commence par tracer le segment horizontal supérieur de longueur a, puis le segment vertical de droite de même longueur. Ensuite, on trace le segment horizontal inférieur de longueur $(a - \alpha)$, puis le segment vertical de gauche de même longueur. On trace ensuite le dessin en spirale de taille $(n - 1)$ dont le segment horizontal supérieur aura la longueur $(a - 2\alpha)$, ce que l'on sait réaliser d'après l'hypothèse d'induction puisque :

$$a > 2 \cdot \alpha \cdot (n-1) \quad \Rightarrow \quad a > 2 \cdot \alpha \cdot (n-2),$$

on a ainsi réalisé le dessin désiré.

Le modèle de résolution par diminution pour ce problème est donc le suivant :

$$\begin{array}{l} \text{DessSpirale}(0, a) \text{ élémentaire (ne rien faire)} \\ \text{DessSpirale}(n, a) \rightarrow \left\{ \begin{array}{ll} \text{DessSpirale}(n-1, a-2\alpha) & \\ + & n > 0. \\ \text{tracé du pseudo-carré extérieur} & \end{array} \right. \end{array}$$

On réalise le tracé grâce à la procédure ci-après :

```
1.  procédure DessinEnSpirale(n, x, y, a, α) pré
2.      n ∈ ℕ et x ∈ ℝ*+ et y ∈ ℝ*+ et a ∈ ℝ et α ∈ ℝ*+ et a > 2·α·(n−1)
3.  début
4.      si n > 0 alors
5.          tracer(x + a, y);
6.          tracer(x + a, y − a);
7.          tracer(x + α, y − a);
8.          tracer(x + α, y − α);
9.          DessinEnSpirale(i − 1, x + α, y − α, a − 2α, α)
10.     fin si
11. fin
```

dont l'appel ci-après :

```
1.  constantes
2.      x0 ∈ ℝ*+ et x0 = ... et y0 ∈ ℝ*+ et y0 = ...
3.  début
4.      DessinEnSpirale(7, x0, y0, 24.0, 1.5)
5.  fin
```

provoque le tracé d'une spirale de taille 7 dont le côté supérieur initial débutant au point de coordonnées (x_0, y_0) est de longueur 24 cm, le suivant 21 cm, etc. Cet appel respecte bien la précondition imposée puisque $24 > 2 \cdot 1.5 \cdot 6$ $(= 18)$.

Remarque On a ici une récursivité « terminale » et donc la mise en œuvre effective passera plutôt par une procédure itérative et non pas récursive (voir chapitre 1).

Pour terminer, on va calculer la longueur du tracé en fonction de n, a et α en supposant la précondition $a > 2 \cdot \alpha \cdot (n-1)$ satisfaite. La longueur du tracé est la solution de la récurrence :

$$\begin{array}{ll} \text{LG}(0, -, -) = 0 & \\ \text{LG}(n, a, \alpha) = 2a + 2(a - \alpha) + \text{LG}(n-1, a-2\alpha, \alpha) & n > 0 \end{array}$$

soit par sommation :

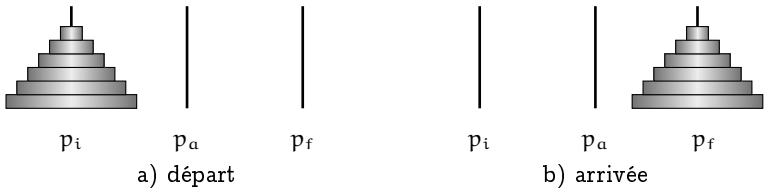


Fig. 4.2 – *Les configurations initiale et finale pour* $n = 6$

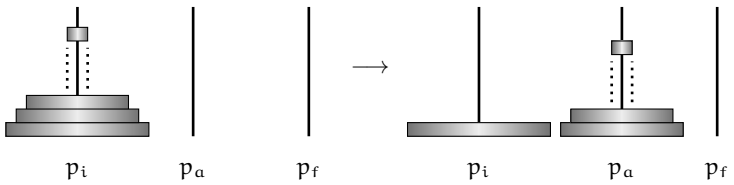


Fig. 4.3 – *La première étape du processus inductif de déplacement d'une tour de Hanoï*

$$LG(n, a, \alpha) = \sum_{i=0}^{2n-1} 2(a - i\alpha) = \sum_{i=0}^{2n-1} 2a - 2\alpha \sum_{i=0}^{2n-1} i = 4an - 4\alpha n^2 + 2\alpha n.$$

Pour $n = 7$, on obtient $LG(7, a, \alpha) = 28a - 182\alpha$, soit 98 cm pour $a = 10.0$ cm et $\alpha = 1.0$ cm.

4.3.3 Exemple 2 : les tours de Hanoï

Le problème des tours de Hanoï consiste à « reconstruire » une pagode (ou tour de Hanoï) initiale composée de l'empilement de n disques de diamètres décroissants de la base au sommet. On dispose de trois piliers : i) p_i celui où se trouve la pagode initiale, ii) p_f celui où devra se trouver la pagode finale, et iii) un pilier auxiliaire p_a hébergeant des empilements intermédiaires de disques.

Les règles suivantes doivent être respectées :

— on peut déplacer un seul disque à la fois grâce à l'opération déplacer(p_1, p_2), où p_1 désigne le pilier dont on extrait le disque supérieur et p_2 celui au sommet duquel est posé ce disque,

— on ne peut placer un disque que sur un autre plus grand que lui ou sur un pilier vide.

Un exemple de configuration de départ et d'arrivée est donné dans la figure 4.2, pour $n = 6$. On va construire une solution pour déplacer toute pagode de n disques ($n \geqslant 1$) selon le schéma inductif ci-après.

Base On sait déplacer une pagode (tour) composée d'un seul disque d'un pilier vers un autre pilier vide.

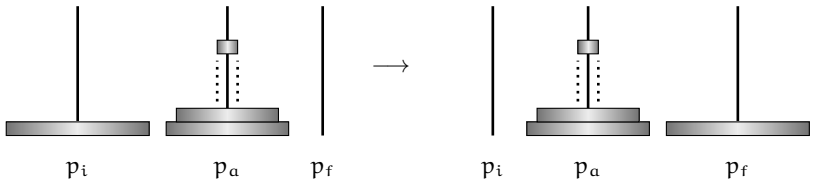


Fig. 4.4 – *La seconde étape du processus inductif de déplacement d'une tour de Hanoï*

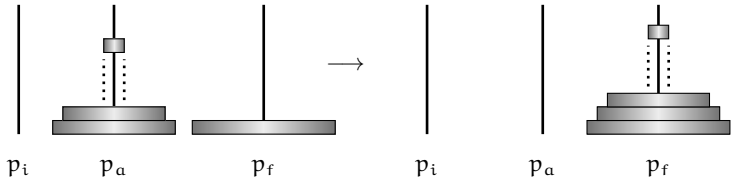


Fig. 4.5 – *La dernière étape du processus inductif de déplacement d'une tour de Hanoï*

Hypothèse d'induction On suppose que l'on sait déplacer une pagode (tour) composée de $(n-1)$ $(n-1 \geqslant 1)$ disques de diamètres décroissants d'un pilier vers un autre vide ou hébergeant déjà le plus grand disque en utilisant le pilier auxiliaire vide au départ.

Induction On va montrer que l'on sait déplacer une pagode (tour) composée de n disques de diamètres décroissants du pilier p_i vers le pilier p_f vide au départ en utilisant le pilier auxiliaire p_a. Pour ce faire : i) on déplace la tour composée des $(n-1)$ disques supérieurs de p_i vers p_a (voir figure 4.3 page 106), ce que l'on sait faire d'après l'hypothèse d'induction puisque le pilier auxiliaire p_a est vide au départ, ii) puis on déplace le dernier disque (le plus grand) de p_i vers p_f (voir figure 4.4), ce qui correspond à un déplacement élémentaire que l'on sait réaliser et qui est légal puisque le pilier p_f est vide, iii) enfin on déplace la tour composée des $(n-1)$ disques se trouvant en p_a vers le pilier p_f (voir figure 4.5), ce que l'on sait faire d'après l'hypothèse d'induction en notant que, bien que non vide, le pilier d'arrivée p_f contient à sa base le plus grand disque.

On en déduit le modèle de diminution :

$$\begin{array}{l} \mathrm{hanoi}(1, p_i, p_f, p_a) \text{ élémentaire } (\mathrm{déplacer}(p_i, p_f)) \\ \mathrm{hanoi}(n, p_i, p_f, p_a) \rightarrow \left\{ \begin{array}{l} \mathrm{hanoi}(n-1, p_i, p_a, p_f) \\ + \\ \mathrm{déplacer}(p_i, p_f) \\ + \\ \mathrm{hanoi}(n-1, p_a, p_f, p_i) \end{array} \right. \qquad n > 1. \end{array}$$

Ici, la fonction générique f consiste à déplacer le plus grand disque de p_i vers p_f. En prenant comme opération élémentaire le déplacement d'un disque, on va calculer $\mathrm{NDD}(n)$, le nombre de déplacements de disques nécessaires au déplacement d'une tour de n disques. Le nombre $\mathrm{NDD}(n)$ vérifie la relation de récurrence :

$$\begin{aligned} NDD(1) &= 1 \\ NDD(n) &= 2 \cdot NDD(n-1) + 1 \qquad n > 1 \end{aligned}$$

dont la solution est $NDD(n) = 2^n - 1$; on a donc une complexité exponentielle.

On va maintenant montrer que cette solution est optimale. Appelons $minNDD(n)$ le nombre minimum de déplacements de disques nécessaires pour une pagode de n disques. Le plus grand des disques doit nécessairement être déplacé au moins une fois. Au moment du premier mouvement de ce grand disque, il doit être seul sur son emplacement, son emplacement de destination doit être vide ; tous les autres disques doivent donc être rangés, dans l'ordre, sur le troisième emplacement. Donc, avant le premier mouvement du grand disque, on aura dû déplacer une pagode de taille $(n-1)$. Il en est de même après le dernier mouvement du grand disque. On a donc :

$$minNDD(n) \geqslant 2 \cdot minNDD(n-1) + 1.$$

Puisque l'on a trouvé un algorithme dont le nombre de mouvements vérifie

$$NDD(n) = 2 \cdot NDD(n-1) + 1,$$

cet algorithme est optimal.

4.4 Ce qu'il faut retenir pour résoudre par diminution

Résoudre un problème de taille n par la méthode « Diminuer pour résoudre » suppose tout d'abord de l'exprimer dans le cas général comme la composition d'un certain nombre de (sous-)problèmes de même nature que le problème initial, mais de taille $(n-1)$. Pour la taille 0 ou 1, le problème doit avoir une solution directe, c'est-à-dire ne faisant pas appel à des sous-problèmes. Dans le schéma de résolution par diminution ainsi identifié, apparaît également une fonction complémentaire engendrant les sous-problèmes et dont on doit spécifier comment elle compose les résultats des sous-problèmes pour former le résultat du problème initial. La preuve de correction du schéma de résolution par diminution envisagé se fait par induction, avec comme base la résolution du cas ayant une solution directe et comme hypothèse d'induction le fait que l'on sait résoudre le (sous-)problème de taille $(n-1)$. On peut alors d'une part établir la complexité temporelle de l'algorithme associé (celle-ci peut être linéaire, polynomiale ou même exponentielle), d'autre part rédiger cet algorithme sous la forme d'une procédure récursive calquée sur le schéma de résolution par diminution.

4.5 Exercices

Exercice 44. Double appel récursif ○ •

Cet exercice porte sur la récursivité. Son intérêt réside dans la façon dont on prouve la terminaison de la procédure récursive considérée.

On considère la fonction *Calc* dont le code figure en début de chapitre. On va montrer que cette fonction retourne la valeur 91 pour tout paramètre d'appel strictement inférieur à 102.

Question 1. Établir que $Calc(100) = Calc(101) = 91$. 44 - Q 1

Question 2. Montrer par induction (récurrence simple sur $k \in \mathbb{N}$) que, pour tout n de l'intervalle $(90 - 11k)\,..\,(100 - 11k)$, l'appel $Calc(n)$ retourne la valeur 91. 44 - Q 2

Question 3. Que dire de la terminaison de cette fonction ? 44 - Q 3

Exercice 45. Complexité du calcul récursif de la suite de Fibonacci ○ •

Cet exercice a pour but principal d'illustrer le bien-fondé d'une mise en œuvre non récursive du calcul d'une grandeur définie par une relation de récurrence.

On note $\mathcal{F}(n)$ le terme courant de la suite de Fibonacci définie pour n strictement positif. Par définition :

$$\begin{aligned} &\mathcal{F}(1) = 1 \\ &\mathcal{F}(2) = 1 \\ &\mathcal{F}(n) = \mathcal{F}(n-1) + \mathcal{F}(n-2) \qquad n \geqslant 2. \end{aligned}$$

Question 1. Écrire une fonction récursive $FiboR(n)$ calquée sur la définition récurrente pour calculer $\mathcal{F}(n)$. 45 - Q 1

Question 2. Donner le nombre total d'additions effectuées pour $n = 6$. 45 - Q 2

Question 3. Calculer le nombre d'additions engendrées par l'appel $FiboR(n)$. 45 - Q 3

Exercice 46. Le point dans ou hors polygone ○ •

Cet exercice s'inscrit dans le domaine de la géométrie euclidienne et ne présente aucune difficulté notable, si ce n'est que, la valeur associée à la base du raisonnement inductif utilisé, n'est pas 0 ou 1, cas annoncés comme habituels dans la présentation de ce chapitre.

On veut déterminer si un point est inclus (au sens large) dans un polygone convexe de n ($n \geqslant 3$) côtés. On se place dans un repère orthonormé xOy du plan.

46 - Q 1 **Question 1.** Étant donnés deux points A et B de coordonnées (x_A, y_A) et (x_B, y_B), donner le principe d'une fonction booléenne d'en-tête :

$\mathit{MêmeCôté}(x_A, y_A, x_B, y_B, x_C, y_C, x_D, y_D)$ **résultat** $\mathbb{B}$

décidant si les points C et D, de coordonnées (x_C, y_C) et (x_D, y_D), sont du même côté de la droite (AB). Quelle en est la complexité en termes de conditions à évaluer ?

46 - Q 2 **Question 2.** Énoncer une propriété d'inclusion d'un point dans un triangle et écrire une fonction d'en-tête :

$\mathit{DansTriangle}(x_A, y_A, x_B, y_B, x_C, y_C, x_P, y_P)$ **résultat** $\mathbb{B}$

décidant si le point P de coordonnées (x_P, y_P) est inclus (au sens large) dans le triangle (ABC). Quelle est sa complexité en nombre de conditions évaluées ?

46 - Q 3 **Question 3.** Déduire une fonction de type « Diminuer pour résoudre » d'en-tête :

$\mathit{DansPolygone}(n, x_P, y_P)$ **résultat** $\mathbb{B}$

décidant si le point P est inclus dans le polygone convexe à n sommets de coordonnées (x_1, y_1), (x_2, y_2), …, (x_n, y_n) avec $n \geqslant 3$.

46 - Q 4 **Question 4.** Donner la complexité temporelle au pire de cette fonction en termes de nombre de conditions évaluées.

46 - Q 5 **Question 5.** Pourquoi le polygone doit-il être convexe ?

Exercice 47. Dessin en doubles carrés imbriqués ○ ⁝

L'intérêt principal de cet exercice réside dans l'élaboration du tracé à réaliser, qui demande en particulier l'identification de son point de départ.

On veut tracer le dessin de la figure 4.6, page 111, dans lequel le motif de base est constitué de deux carrés imbriqués. On dispose des deux primitives habituelles $\mathit{placer}(x, y)$ et $\mathit{tracer}(x, y)$ de l'exemple 1 page 104, et le dessin doit être effectué sous les contraintes suivantes :

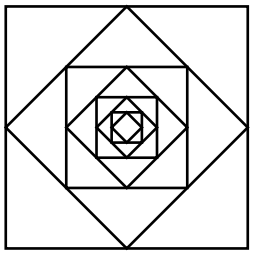


Fig. 4.6 – *Un dessin ayant quatre doubles carrés imbriqués*

— à la fin du tracé, la plume se trouve au point de départ,

— la plume ne doit pas être levée en cours de dessin et aucun trait ne doit être fait plusieurs fois (ni temps mort, ni travail inutile).

On cherche à réaliser le dessin grâce à une procédure de type « Diminuer pour résoudre » respectant ces contraintes. Ici, la taille n du problème ($n \in \mathbb{N}$) correspond au nombre de doubles carrés que l'on souhaite imbriquer.

Question 1. Identifier les points de départ possibles du tracé. 47 - Q 1

Question 2. Proposer une stratégie de type « Diminuer pour résoudre » qui permet de réaliser ce dessin. En déduire le modèle de résolution par diminution associé. 47 - Q 2

Question 3. Écrire la procédure réalisant ce dessin, dont l'appel se fait par la séquence : 47 - Q 3

1. **constantes**
2. $m \in \mathbb{N}_1$ **et** $m = \ldots$ **et** $x_0 \in \mathbb{R}$ **et** $x_0 = \ldots$ **et** $y_0 \in \mathbb{R}$ **et** $y_0 = \ldots$ **et**
3. $c_0 \in \mathbb{R}_+$ **et** $c_0 = \ldots$
4. **début**
5. $placer(x_0, y_0)$;
6. $DessDbCarrImb(m, x_0, y_0, c_0)$
7. **fin**

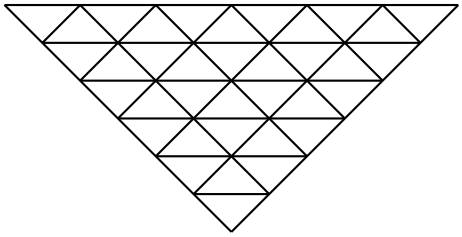
où x_0 et y_0 désignent les coordonnées du point de départ du tracé, m le nombre effectif de doubles carrés à dessiner et c_0 la longueur du côté du carré « extérieur ».

Question 4. Quelle en est la complexité en termes de nombre d'appels à la fonction *tracer* et quelle est la longueur du tracé ? 47 - Q 4

Exercice 48. Dessin en triangles ○ ⁝

Cet exercice vise lui aussi à effectuer le tracé d'un dessin. Il met en évidence la construction d'une solution optimale passant par une première approche qui ne l'est pas, mais qui « met le pied à l'étrier ». La solution finale utilise deux procédures de type « Diminuer pour résoudre ».

On veut tracer la figure suivante :



appelée « grand triangle inversé ». Chaque triangle élémentaire est rectangle isocèle (avec l'angle droit en bas), de hauteur issue de l'angle droit d ; sa base (horizontale) a donc pour longueur $2d$. Le schéma ci-dessus représente six « couches » de tels triangles. La couche du bas comporte un triangle, celle de dessus deux, et ainsi de suite jusqu'à six. Comme on le voit, un assemblage de n couches forme un grand triangle semblable au triangle élémentaire, mais de côtés n fois plus grands. Une telle figure est dite de taille n. Pour un triangle élémentaire, si le sommet associé à l'angle droit a pour coordonnées (x, y), les deux autres sommets ont pour coordonnées $(x - d, y + d)$ et $(x + d, y + d)$. On dispose ici aussi des primitives *placer*(x, y) et *tracer*(x, y) définies dans l'exemple 1 page 104. On veut que le tracé démarre et termine au sommet associé à l'angle droit auquel on assigne les coordonnées (x_0, y_0).

Remarque Le nombre minimum de triangles élémentaires qu'il faut tracer pour obtenir un triangle de taille n est $n(n+1)/2$.

48 - Q 1 **Question 1.** Dans un premier temps, proposer le principe d'une solution de type « Diminuer pour résoudre », avec la seule contrainte de ne pas lever la plume en cours de tracé. Préciser le modèle de résolution par diminution utilisé.

48 - Q 2 **Question 2.** Donner le code de la procédure correspondante dont l'appel est fait par la séquence :

```
1. constantes
2.    m ∈ ℕ1 et m = ... et x0 ∈ ℝ et x0 = ... et y0 ∈ ℝ et y0 = ... et
3.    d0 ∈ ℝ+ et d0 = ...
4. début
5.    placer(x0, y0) ;
6.    Triangle1(m, x0, y0, d0)
7. fin
```

où m est le nombre de couches de triangles, (x_0, y_0) représente les coordonnées du sommet associé à l'angle droit (bas du dessin) et d_0 est la hauteur du triangle élémentaire.

Question 3. Quelle est la complexité de cette procédure en termes d'appels à la fonction *tracer* et de nombre de triangles élémentaires tracés ? 48 - Q 3

Question 4. Afin de gagner en efficacité en évitant des tracés inutiles, on demande de réviser la stratégie précédente et de proposer le principe d'une solution dans laquelle la plume n'est pas levée en cours de dessin et tout segment n'est tracé qu'une seule fois. 48 - Q 4

Question 5. Donner le code de la nouvelle procédure d'en-tête *Triangle2*(n, x, y, d) et montrer qu'elle est optimale. 48 - Q 5

Exercice 49. Parcours exhaustif d'un échiquier ○ ⁝

Cet exercice met en évidence une famille d'échiquiers qui peuvent être parcourus par un cavalier de façon exhaustive et simple, en visitant chaque case une seule fois. Il est à rapprocher de l'exercice 53 page 149 du chapitre 5.

On considère un échiquier de côté $c = 4m + 1$ (avec $m \geqslant 0$). On va montrer qu'un cavalier peut en effectuer le parcours exhaustif en ne visitant chaque case qu'une seule fois. Notons qu'il est possible que ce problème ait une solution pour certains échiquiers de côté autre, mais ceci reste hors du champ de cet exercice. Le cavalier respecte ses règles de déplacement au jeu d'échecs, c'est-à-dire que s'il est sur la case (i, j), il peut aller sur les huit cases :

$$\begin{array}{cccc} (i-2, j-1) & (i-2, j+1) & (i+2, j-1) & (i+2, j+1) \\ (i-1, j-2) & (i-1, j+2) & (i+1, j-2) & (i+1, j+2) \end{array}$$

pour autant qu'il ne sorte pas de l'échiquier. Dans cet exercice, i désigne l'indice de ligne et j l'indice de colonne.

Question 1. Proposer un parcours exhaustif d'un échiquier de côté $c = 5$ $(m = 1)$ en partant de la case $(1, 1)$. 49 - Q 1

Question 2. Généraliser ce parcours au cas d'un échiquier de côté $c = 4m + 1$ $(m \geqslant 1)$, toujours en partant de la case $(1, 1)$. 49 - Q 2
Indication. On pourra mettre en évidence le parcours exhaustif de la couronne « extérieure » de largeur 2 bordant l'échiquier.

Question 3. Spécifier le modèle de résolution par diminution utilisé. 49 - Q 3

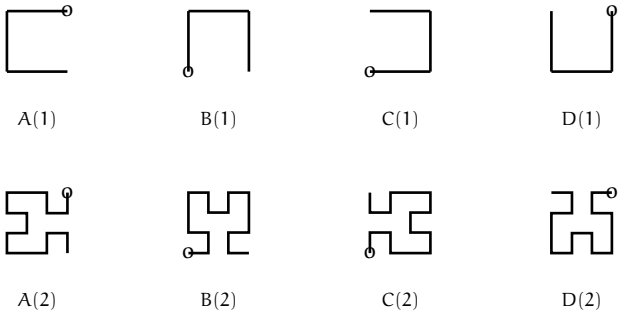


Fig. 4.7 – *Les courbes* $A(1)$ *à* $D(1)$ *(*$\alpha = 1$*) et* $A(2)$ *à* $D(2)$ *(*$\alpha = 1/3$*)*

Exercice 50. Courbes de Hilbert et W-courbes ○ ⁝

Cet exercice met en lumière deux problèmes de tracés de courbes dans lesquels la construction de la solution déborde du cadre strict d'un procédé « Diminuer pour résoudre ». En fait, le tracé de ces deux familles de courbes est fondé sur quatre procédures très semblables dans leur structure, chacune de type « Diminuer pour résoudre » croisées.

Les courbes de Hilbert

On considère les courbes de la figure 4.7, dont les points de départ sont représentés par un petit cercle. On dispose de la fonction *tracer* définie dans l'exemple 1 page 104.

50 - Q 1 **Question** 1. En partant de ces courbes, donner l'expression de $A(i+1)$, $B(i+1)$, $C(i+1)$ et $D(i+1)$ en fonction de $A(i)$, $B(i)$, $C(i)$, $D(i)$ et des quatre tracés élémentaires « trait vers la gauche » noté g, « trait vers la droite » noté d, « trait vers le haut » noté h, « trait vers le bas » noté b, tous quatre de longueur α fixée.

50 - Q 2 **Question** 2. Dessiner la courbe $A(3)$ appelée courbe de Hilbert de niveau 3.

50 - Q 3 **Question** 3. Donner le code d'une procédure effectuant le tracé de la courbe de Hilbert de niveau n correspondant à $A(n)$.

50 - Q 4 **Question** 4. Quelle est la complexité $C_A(n)$ du tracé de $A(n)$ en nombre d'appels à la fonction *tracer* ?

50 - Q 5 **Question** 5. Déterminer la hauteur et la largeur du tracé réalisé en fonction de n et de α.

Les W-courbes

On considère les courbes définies comme suit :

$$A(1) = d \quad B(1) = b \quad C(1) = g \quad D(1) = h$$

et pour $i \geqslant 1$:

$$\begin{array}{lcllllllllllll}
A(i+1) & = & A(i) & b & d & B(i) & d & D(i) & d & & h & A(i) & & \\
B(i+1) & = & B(i) & g & b & C(i) & b & A(i) & b & & d & B(i) & & \\
C(i+1) & = & C(i) & h & g & D(i) & g & B(i) & g & & b & C(i) & & \\
D(i+1) & = & D(i) & d & h & A(i) & h & C(i) & h & & g & D(i) & & \\
W(i) & = & A(i) & b & d & B(i) & g & b & C(i) & h & g & D(i) & d & h
\end{array}$$

où b, d, g, h représentent les tracés élémentaires de longueur α du début de l'énoncé.

Question 6. Tracer les courbes $W(1)$ et $W(2)$. 50 - Q 6

Question 7. Établir la complexité du tracé de la courbe $W(n)$ $(n \geqslant 1)$ en termes d'appels à la procédure *tracer* en fonction de n et α. 50 - Q 7

Remarque On peut montrer que la courbe $W(n)$ s'inscrit dans un carré de côté $\alpha \cdot (2^{n+1} - 1)$.

Complément Le lecteur intéressé pourra étudier le tracé des courbes de Sierpinski (voir par exemple `http://aesculier.fr/fichiersMaple/sierpinski2D/sierpinski2D.html`), qui fait lui aussi appel au mécanisme « Diminuer pour résoudre ».

CHAPITRE 5

Essais successifs

> Though patience be a tired mare,
> yet she will plod.
>
> *(William Shakespeare)*

Les techniques appliquées dans ce chapitre supposent une bonne assimilation du contenu de la section 1.4.5, page 19 (chapitre 1 intitulé « Mathématiques et informatique : quelques notions utiles »). La plupart des exemples et exercices proposés dans ce chapitre entrent dans la classe des problèmes NP-complets (voir par exemple [17] pour une introduction à la problématique des classes de complexité), ce qui signifie que l'on ne connaît pas d'algorithme polynomial pour les résoudre.

5.1 Rappels

5.1.1 Principe

Essais successifs : un problème de recherche

La programmation par essais successifs (encore appelée du type *générer et tester* ou par *backtracking*) est une méthode de recherche de solutions parmi un ensemble fini de candidats. À l'instar du paradigme classique de recherche (dont un exemple typique est celui de la recherche séquentielle dans un tableau), le principe de la programmation par essais successifs consiste en la donnée d'un *ensemble fini de candidats* (ensemble défini en extension ou en compréhension) et d'un *prédicat de sélection* qui permet de déterminer si un candidat est ou non solution du problème considéré. On est alors à même de déterminer le sous-ensemble des candidats qui sont des solutions (c'est-à-dire des candidats qui satisfont le prédicat de sélection). Dans la programmation par essais successifs comme dans la recherche classique, la technique algorithmique de base consiste à examiner successivement tous les candidats.

Dans l'hypothèse où l'ensemble des solutions n'est pas vide, une première variante possible consiste à rechercher seulement l'*une quelconque* des solutions au problème et non pas toutes. Une seconde variante se caractérise par la recherche de la meilleure solution (dans un sens à définir selon le problème).

La principale caractéristique de la programmation par essais successifs porte sur le fait que, contrairement à la recherche classique, l'ensemble C des candidats est défini en compréhension (il est « calculé », « construit », « engendré » au fil de la recherche, à partir de ses propriétés). Cet ensemble peut se révéler très grand, au point de rendre cette technique inexploitable telle quelle en pratique.

Premier exemple

En développant un exemple simple « en largeur », cette section se présente sous la forme d'un tutoriel qui aborde, sans les approfondir, la plupart des aspects qui sont développés plus loin. Elle vise également à ébaucher une démarche méthodologique destinée à être affinée dans le reste de cette introduction, puis à être appliquée dans les exercices de ce chapitre.

Étant donné un ensemble E de n ($n > 0$) éléments de $\mathbb{Z}$, on cherche tous les sous-ensembles de E dont la somme des éléments est nulle. L'ensemble C des candidats est $\mathbb{P}(E)$, l'ensemble des parties de E ; le prédicat de sélection est « la somme des éléments d'un candidat est nulle ». Il faut donc examiner une fois et une seule (énumérer sans répétition) les 2^n sous-ensembles de E et calculer pour chacun la somme de ses éléments. On retient les sous-ensembles de E qui résolvent le problème (ici, il existe au moins une solution : le sous-ensemble vide $\varnothing$, mais dans le cas général il peut ne pas y en avoir).

Concrètement, on représente l'ensemble E par un tableau T de n éléments. Par exemple, pour $n = 6$ et $E = \{10, 0, -3, -7, 5, -5\}$, $T = [10, 0, -3, -7, 5, -5]$. Chaque candidat est matérialisé par son vecteur caractéristique. Ici, ces 2^6 vecteurs sont $[0, 0, 0, 0, 0, 0]$ (pour l'ensemble vide), $[1, 0, 0, 0, 0, 0]$ (pour l'ensemble $\{10\}$), ..., $[1, 1, 1, 1, 1, 1]$ (pour l'ensemble E). On note que chacun de ces vecteurs est une fonction de l'intervalle $1..6$ dans l'intervalle $0..1$. La solution recherchée s'obtient en calculant la somme des éléments de T filtrés par le vecteur caractéristique pour ne retenir que les sous-tableaux qui ont 0 comme somme.

Squelette Dans la résolution de ce problème, la première difficulté qui se présente est de produire successivement tous les candidats. Une façon d'y parvenir consiste à parcourir l'arbre (dit *arbre de récursion*) dont les feuilles sont les vecteurs caractéristiques en question. Pour $n = 3$, l'arbre de récursion obtenu est celui de la figure 5.1.

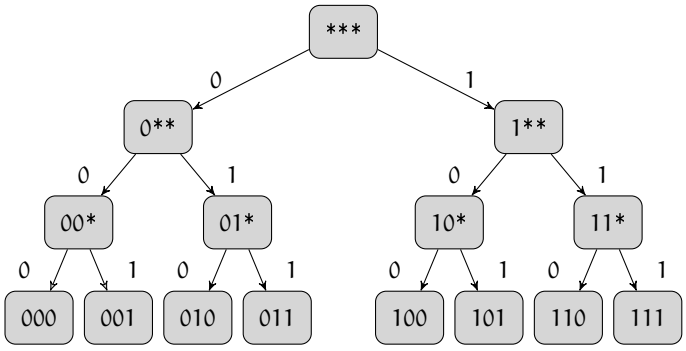


Fig. 5.1 – *Arbre de récursion pour la construction de toutes les fonctions totales de l'intervalle* $1..3$ *dans l'intervalle* $0..1$. *Ces fonctions permettent de retrouver l'ensemble des parties de l'ensemble* $\{1, 2, 3\}$.

L'algorithme suivant répond à l'objectif et effectue un parcours en profondeur d'abord de l'arbre de récursion,

1. **procédure** *PartEns1*(i) **pré**
2. $\quad i \in 1..n$
3. **début**

```
4.    pour j parcourant 0 .. 1 faire
5.       X[i] ← j ;
6.       si i = n alors
7.          écrire(X)
8.       sinon
9.          PartEns1(i + 1)
10.      fin si
11.   fin pour
12. fin
```

à condition d'être appelé dans le contexte suivant :

```
1. constantes
2.    n ∈ ℕ1 et n = ... et T ∈ 1 .. n → ℤ et T = [...]
3. variables
4.    X ∈ 1 .. n → 0 .. 1
5. début
6.    PartEns1(1)
7. fin
```

Cet algorithme est la première étape vers la solution de notre problème. C'est le *squelette* de la solution. X est appelé *vecteur d'énumération*.

La démonstration formelle de la correction de cet algorithme peut se faire en appliquant l'axiomatique de Hoare étendue aux procédures récursives [38]. Elle se fonde sur l'invariant de récursivité (voir [49]) suivant : « $X[1 .. i-1]$ est constant, c'est une fonction totale de $1 .. i-1$ dans $0 .. 1$ et, si un nœud de l'arbre de récursion satisfaisant la condition $(i = n)$ a été imprimé, alors tous les nœuds satisfaisant cette condition ont été imprimés ». Ainsi par exemple, si, dans le schéma de la figure 5.1, on considère le sous-arbre ayant comme racine $X = [1,*,*]$ $(i = 2)$, on entre et sort de la procédure avec cette valeur de X, et si une seule des feuilles de ce sous-arbre à été imprimée, alors les quatre feuilles ont été imprimées.

Sélection Pour l'instant, nous n'avons fait que produire toutes les fonctions totales de $1 .. n$ dans $0 .. 1$. Notre objectif final est plus ambitieux puisqu'il s'agit d'identifier les sous-ensembles de somme nulle. Il faut donc *sélectionner* les candidats. Supposons que l'on dispose de la fonction booléenne *SomNulle* qui vérifie que la somme des éléments de T, filtrés par X, est nulle. Pour obtenir une solution à notre problème, il suffit d'adjoindre à la condition $(i = n)$ de l'invariant ci-dessus la condition *SomNulle*. Nous obtenons alors la version *PartEns2* suivante :

```
1. procédure PartEns2(i) pré
2.    i ∈ 1 .. n
3. début
4.    pour j parcourant 0 .. 1 faire
5.       X[i] ← j ;
6.       si i = n [et SomNulle] alors
7.          écrire(X)
8.       sinonsi i ≠ n alors
9.          PartEns2(i + 1)
10.      fin si
11.   fin pour
12. fin
```

L'appel récursif doit se faire uniquement quand $i \neq n$. La condition de la ligne 8 est donc bien nécessaire. Le code nouvellement introduit est encadré et ce procédé est utilisé de façon systématique dans la suite.

Remarque En toute rigueur, la condition de la ligne 6 peut se limiter à la condition *SomNulle*, sous réserve de compléter le vecteur X par des 0. Cependant, cette solution est abandonnée pour des raisons didactiques.

Optimisation La version *PartEns2* est correcte mais guère efficace. On constate en effet que l'appel de la fonction *SomNulle* conduit à des calculs redondants. Les éviter passe par un renforcement de l'invariant entraîné par l'introduction de la variable fraîche SomCour. L'invariant de récursivité devient alors : « $X[1..i-1]$ est constant, c'est une fonction totale de $1..i-1$ dans $0..1$, $\text{SomCour} = \sum_{k=1}^{i-1} T[k]$ et si un nœud de l'arbre de récursion satisfaisant la condition $i = n$ a été imprimé, alors tous les nœuds satisfaisant cette condition ont été imprimés ». Il faut alors, à la ligne 6, mettre à jour la variable SomCour et, puisque le retour de récursivité n'annule pas cette mise à jour, défaire son effet, en soustrayant de SomCour la valeur $j \cdot T[i]$ qui y a été ajoutée à la ligne 5. On obtient alors la version suivante :

```
1.  procédure PartEns3(i) pré
2.      i ∈ 1..n
3.  début
4.      pour j parcourant 0..1 faire
5.          X[i] ← j ; | SomCour ← SomCour + j · T[i] ;
6.          si i = n | et SomCour = 0 | alors
7.              écrire(X)
8.          sinonsi i ≠ n alors
9.              PartEns3(i + 1)
10.         fin si ;
11.         | SomCour ← SomCour − j · T[i] |
12.     fin pour
13. fin
```

qui est appelée dans le contexte suivant :

```
1.  constantes
2.      n ∈ ℕ1 et n = ... et T ∈ 1..n → ℤ et T = [...]
3.  variables
4.      X ∈ 1..n → 0..1 et | SomCour ∈ ℤ |
5.  début
6.      | SomCour ← 0 ; |
7.      PartEns3(1)
8.  fin
```

Complexité et élagage Le nombre de fonctions totales de $1..n$ dans $0..1$ est aussi le nombre de *feuilles* que possèdent les arbres comme celui de la figure 5.1 page 118, soit 2^n. Le nombre d'appels à la fonction *PartEns* (quelle que soit la version retenue) correspond quant à lui au nombre de nœuds *internes*, soit $\sum_{i=0}^{n-1} 2^i$, soit encore $2^n - 1$. La complexité exponentielle qui en résulte nous amène naturellement à nous interroger sur la nécessité de

visiter toutes les feuilles de l'arbre et à nous demander par conséquent si la configuration de certains nœuds ne permettrait pas d'éviter le parcours exhaustif (encore appelé recherche par *force brute*) de tous les sous-arbres situés sous ces nœuds. Si c'était le cas, on pourrait ainsi améliorer l'efficacité de la solution (sans toutefois être certain de réduire l'ordre de grandeur de la complexité).

De manière générale, il n'est en effet pas toujours nécessaire d'examiner toutes les possibilités de solutions : certaines peuvent être éliminées avant leur construction complète. C'est le principe de l'*élagage*. Celui-ci permet, en testant chaque candidat partiel, d'éviter, quand c'est possible, de développer les candidats qui se trouvent en-dessous du nœud où l'on se trouve dans l'arbre, parce que l'on constate que la solution partielle ne mènera jamais à une solution. Dans le cas de la recherche d'une solution optimale, il est souvent possible de tirer parti du critère d'optimalité. En effet, lorsque la fonction qui lui est associée est monotone croissante (resp. décroissante), on tentera de faire une sous-estimation (resp. sur-estimation) de sa valeur finale afin de procéder à un élagage quand cette valeur n'atteint pas (resp. dépasse) l'optimal courant.

Dans notre exemple, pour la version *PartEns3*, un élagage se fonde sur l'observation suivante. Si, à un instant donné du calcul, la valeur absolue de SomCour est supérieure à l'extremum de T multiplié par le nombre de valeurs de T restant à traiter, il est inutile de développer le sous-arbre courant [1]. La version suivante met en œuvre cet élagage :

```
1.  procédure PartEns4(i) pré
2.     i ∈ 1..n
3.  début
4.     pour j parcourant 0..1 faire
5.        si |SomCour| ⩽ (n − i + 1) · AbsExtT alors
6.           X[i] ← j ; SomCour ← SomCour + j · T[i] ;
7.           si i = n et SomCour = 0 alors
8.              écrire(X)
9.           sinonsi i ≠ n alors
10.             PartEns4(i + 1)
11.          fin si ;
12.          SomCour ← SomCour − j · T[i]
13.       fin si
14.    fin pour
15. fin
```

à la condition que l'appel se fasse dans le contexte suivant (la constante AbsExtT est la valeur absolue de l'extremum de T) :

```
1.  constantes
2.     n ∈ ℕ1 et n = ... et T ∈ 1..n → ℤ et T = [...] et
3.     AbsExtT ∈ ℤ et AbsExtT = max({|max(codom(T))|, |min(codom(T))|})
4.  variables
5.     X ∈ 1..n → 0..1 et SomCour ∈ ℤ
6.  début
7.     SomCour ← 0 ;
8.     PartEns4(1)
9.  fin
```

1. Cette remarque suffit pour réaliser un élagage grossier. Il serait possible de l'affiner de différentes façons, en distinguant par exemple le cas où $SomCour > 0$ du cas où $SomCour < 0$.

Bien que, pour une taille donnée du problème à traiter, l'effet d'un élagage soit souvent impressionnant, tant sur le plan temporel que sur celui du nombre d'appels, l'expérience montre que le gain ainsi obtenu ne permet de traiter que des problèmes de tailles légèrement supérieures à la taille initialement considérée. L'exercice 52, page 147, en est un exemple parmi d'autres.

Conclusion Dans cette section, nous avons ébauché la démarche que nous préconisons pour aborder un problème selon l'approche « essais successifs ». Les deux sections suivantes affinent cette approche pour aboutir à un répertoire de « patrons » à même de s'appliquer dans un large éventail de situations.

5.1.2 Fonctions et squelettes associés

Dans la section précédente, nous avons vu comment « habiller » le squelette d'un algorithme énumérant toutes les fonctions totales de intervalle $1..n$ dans l'intervalle $0..1$, afin de résoudre le problème des sous-ensembles à somme nulle. Dans cette section, nous nous focalisons sur cette notion de squelette, en élargissant notre étude à une vaste gamme de fonctions telles que les fonctions totales, les fonctions totales surjectives, les fonctions partielles, les fonction partielles injectives, etc. afin de couvrir tous les problèmes abordés dans les exercices de la section 5.3 page 145. Dans la suite de la présente section, on se place dans la situation où l'on recherche les fonctions de $I = 1..n$ dans $F = 1..m$.

Le cas des fonctions totales

Cas général : énumération des fonctions totales entre deux intervalles I et F Ce cas généralise le cas étudié dans l'exemple de la section 5.1.1, page 118. Nous recherchons un algorithme capable de produire toutes les fonctions totales de $I = 1..n$ dans $F = 1..m$. Le squelette :

```
1.  procédure T(i) pré
2.      i ∈ 1..n
3.  début
4.      pour j parcourant 1..m faire
5.          X[i] ← j ;
6.          si i = n alors
7.              écrire(X)
8.          sinon
9.              T(i+1)
10.         fin si
11.     fin pour
12. fin
```

répond à cet objectif à condition d'être appelé dans le contexte suivant :

```
1.  constantes
2.      n ∈ ℕ et n = ... et m ∈ ℕ et m = ...
3.  variables
4.      X ∈ 1..n → 1..m
5.  début
6.      T(1)
7.  fin
```

Le nombre de fonctions totales de $1..n$ dans $1..m$ est aussi le nombre de *feuilles* que possède l'arbre de récursion, soit m^n. Le nombre d'appels à la fonction T correspond quant à lui au nombre de nœuds *internes*, soit $\sum_{i=0}^{n-1} m^i$, soit encore $(m^n-1)/(m-1)$. La nature exponentielle de cette fonction (de n) conduit en général à des algorithmes dont la complexité temporelle limite l'intérêt pratique. C'est également le cas des autres squelettes étudiés dans cette section.

L'exemple développé à la section 5.1.4, page 136, concerne le cas où I est un sac. La procédure T s'applique sans inconvénient particulier ; cependant elle produit, toujours sous la forme d'un vecteur caractéristique, tous les sous-sacs de I (avec des doublons de sous-sacs en général).

Le cas où l'ensemble de départ est un produit cartésien Comment peut-on adapter la procédure T au cas d'un ensemble I, produit cartésien de deux intervalles $1..n$ et $1..q$? Le principe est simple. Il suffit de transformer le *vecteur* d'énumération X en une *matrice* d'énumération et de réécrire la procédure T de la manière suivante (qui remplit X de gauche à droite et de haut en bas) :

```
1.  procédure T2D(l, c) pré
2.      l ∈ 1..n et c ∈ 1..q
3.  début
4.      pour j parcourant 1..m faire
5.          X[l, c] ← j ;
6.          si l = n et c = q alors
7.              écrire(X)
8.          sinon
9.              si c = q alors
10.                 T2D(l + 1, 1)
11.             sinon
12.                 T2D(l, c + 1)
13.             fin si
14.         fin si
15.     fin pour
16. fin
```

Le programme appelant devient alors :

```
1.  constantes
2.      n ∈ ℕ et n = ... et q ∈ ℕ et q = ... et m ∈ ℕ et m = ...
3.  variables
4.      X ∈ 1..n × 1..q → 1..m
5.  début
6.      T2D(1, 1)
7.  fin
```

La transposition aux cas des fonctions autres que les fonctions totales se fait sans difficulté.

Énumération des fonctions totales surjectives L'invariant de récursivité se formule par : « $X[1..i-1]$ est constant. C'est une fonction totale de $1..i-1$ dans $1..m$ et, si un nœud de l'arbre de récursion satisfaisant la condition $(i = n)$ a été imprimé, alors tous les nœuds satisfaisant cette condition ont été imprimés ». On note que $X[1..i-1]$ n'est en général pas une fonction surjective. La production de toutes les fonctions surjectives s'obtient

en aménageant la procédure T, en lui appliquant une technique similaire à celle utilisée dans l'optimisation de la procédure *PartEns3*, page 120, de la manière suivante. Avant d'écrire une solution, il faut vérifier qu'elle est bien surjective. Pour ce faire, on tient à jour un tableau booléen B global des valeurs de j enregistrées dans X. La vérification de la surjectivité se spécifie par la quantification universelle présente dans la condition de l'alternative (ligne 7 du squelette TS). Cette partie de la condition peut se raffiner par une recherche séquentielle. La mise à jour de B, à la ligne 6, doit être annulée une fois le traitement effectué ; ce qui est permis grâce à l'utilisation de la variable locale $Sauv$ dans laquelle on préserve l'ancienne valeur.

```
1.  procédure TS(i) pré
2.     i ∈ 1 .. n et Sauv ∈ 𝔹
3.  début
4.     pour j parcourant 1 .. m faire
5.        X[i] ← j ;
6.        Sauv ← B[j] ; B[j] ← vrai ;
7.        si i = n et alors ∀k · (k ∈ 1 .. m ⇒ B[k]) alors
8.           écrire(X)
9.        sinonsi i ≠ n alors
10.          TS(i + 1)
11.       fin si ;
12.       B[j] ← Sauv
13.    fin pour
14. fin
```

L'appel suivant, qui initialise le tableau B, permet d'obtenir le résultat attendu :

```
1.  constantes
2.     n ∈ ℕ et n = ... et m ∈ ℕ et m = ...
3.  variables
4.     X ∈ 1 .. n → 1 .. m et B ∈ 1 .. m → 𝔹
5.  début
6.     B ← (1 .. m) × {faux} ;
7.     TS(1)
8.  fin
```

Pour $n = 3$ et $m = 2$, l'arbre de récursion est présenté à la figure 5.2, page 125.

Dans le squelette TS, l'énumération de toutes les fonctions totales surjectives passe tout d'abord par celle de toutes les fonctions totales. Par conséquent, le nombre d'appels à la fonction TS est encore de $((m^n - 1)/(m - 1))$.

Dans les exercices à venir, nous rencontrerons des cas où la vérification de la surjectivité n'est pas nécessaire, autrement dit, des cas où la quantification universelle de la ligne 7 du squelette TS peut être omise (voir par exemple l'exercice 51, page 146). Le squelette à retenir est alors T dans sa totalité.

Énumération des fonctions totales injectives et bijectives, production des permutations Une extension intéressante de la procédure T consiste à produire toutes les fonctions totales *injectives* de $I = 1 .. n$ dans $F = 1 .. m$. L'invariant de récursivité est « $X[1 .. i - 1]$ est constant, c'est une fonction totale *injective* de $1..i-1$ dans $1..m$ et, si un nœud de l'arbre de récursion satisfaisant la condition $(i = n)$ a été imprimé, alors tous les nœuds satisfaisant cette condition ont été imprimés ». L'injectivité s'obtient en excluant de l'intervalle

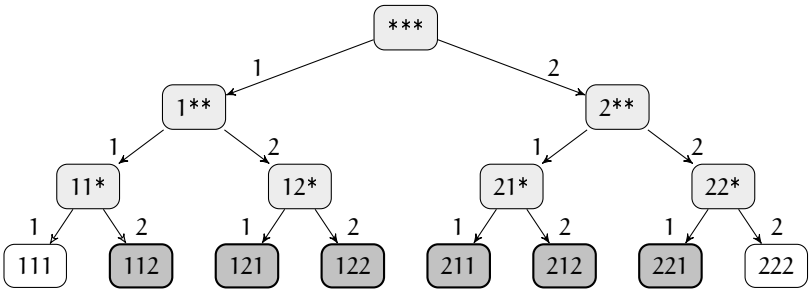


Fig. 5.2 – *Arbre de récursion pour la construction de toutes les fonctions surjectives de* $1..3$ *dans* $1..2$. *Les feuilles en blanc signalent les fonctions qui ne sont pas surjectives. Les feuilles en gris foncé symbolisent les fonctions surjectives.*

$1..m$ sur lequel la boucle prend ses valeurs, les éléments déjà présents dans $X[1..i-1]$. C'est ce que fait le squelette *TI* suivant :

```
1.  procédure TI(i) pré
2.     i ∈ 1..n
3.  début
4.     pour j parcourant (1..m − codom(X[1..i − 1])) faire
5.        X[i] ← j ;
6.        si i = n alors
7.           écrire(X)
8.        sinon
9.           TI(i + 1)
10.       fin si
11.    fin pour
12. fin
```

Cette procédure est appelée par le programme principal suivant :

```
1. constantes
2.    n ∈ ℕ et n = ... et m ∈ ℕ et m = ...
3. variables
4.    X ∈ 1..n → 1..m
5. début
6.    TI(1)
7. fin
```

Puisque le nombre d'injections d'un ensemble de n éléments dans un ensemble de k éléments est égal à A_k^n (nombre d'arrangements de k objets parmi n), le nombre de feuilles visitées par l'algorithme est de $A_m^n = m!/(m-n)!$, si $m \geqslant n$. Quant au nombre d'appels, (la démonstration est laissée aux soins du lecteur), il est égal à :

$$\sum_{i=0}^{n-1} A_m^i \tag{5.1}$$

La figure 5.3 fournit l'arbre de récursion parcouru par la procédure pour $I = 1..3$ et $F = 1..4$. Lorsque $F = I$, cette procédure produit l'ensemble des *bijections* de $I = 1..n$

vers I et permet donc d'obtenir les *permutations* d'un ensemble de valeurs. La vérification de la surjectivité n'est pas nécessaire puisqu'une fonction totale injective dans elle-même est une bijection. La formule $\sum_{i=0}^{m-1} A_m^i$ fournit le nombre d'appels réalisés.

La plupart des langages de programmation n'offrent pas de constructions permettant de coder directement la boucle ci-dessus. Alors, comment raffiner ce programme ? Une solution (dont le principe est similaire à la solution appliquée pour construire la procédure *EnsPart3*, page 120) consiste à gérer (sous la forme d'un tableau pour des raisons d'efficacité) une fonction L (pour « Libre ») définie sur l'intervalle $F = 1\,..\,m$ et à valeurs booléennes (ou parfois, par commodité, dans l'intervalle $0\,..\,1$) telle que $L[k]$ signifie que k est absent de $X[1\,..\,i-1]$. Le schéma ci-dessus se raffine alors de la manière suivante :

```
1.  procédure TI2(i) pré
2.     i ∈ 1 .. n
3.  début
4.     pour j parcourant 1 .. m faire
5.        si L[j] alors
6.           X[i] ← j ; L[j] ← faux ;
7.           si i = n alors
8.              écrire(X)
9.           sinon
10.             TI2(i + 1)
11.          fin si ;
12.          L[j] ← vrai
13.       fin si
14.    fin pour
15. fin
```

L'appel se fait alors comme suit :

```
1. constantes
2.    n ∈ ℕ et n = ... et m ∈ ℕ et m = ...
3. variables
4.    X ∈ 1 .. n → 1 .. m et L ∈ 1 .. m → 𝔹
5. début
6.    L ← (1 .. m) × {vrai} ;
7.    TI2(1)
8. fin
```

Dans la suite, en général, ce raffinement ne sera pas explicité.

Le cas des fonctions partielles

Cas général : énumération des fonctions partielles entre deux intervalles I et F Nous recherchons un algorithme capable de produire toutes les fonctions partielles de $I = 1..n$ dans $F = 1..m$. Partant du squelette *T*, il suffit d'ajouter une valeur fantôme à F (0 par exemple) telle que tout couple ayant comme extrémité 0 sera considéré comme inexistant du point de vue de la fonction (voir figure 5.4, page 128).

Le squelette suivant :

```
1.  procédure P(i) pré
2.     i ∈ 1 .. n
3.  début
```

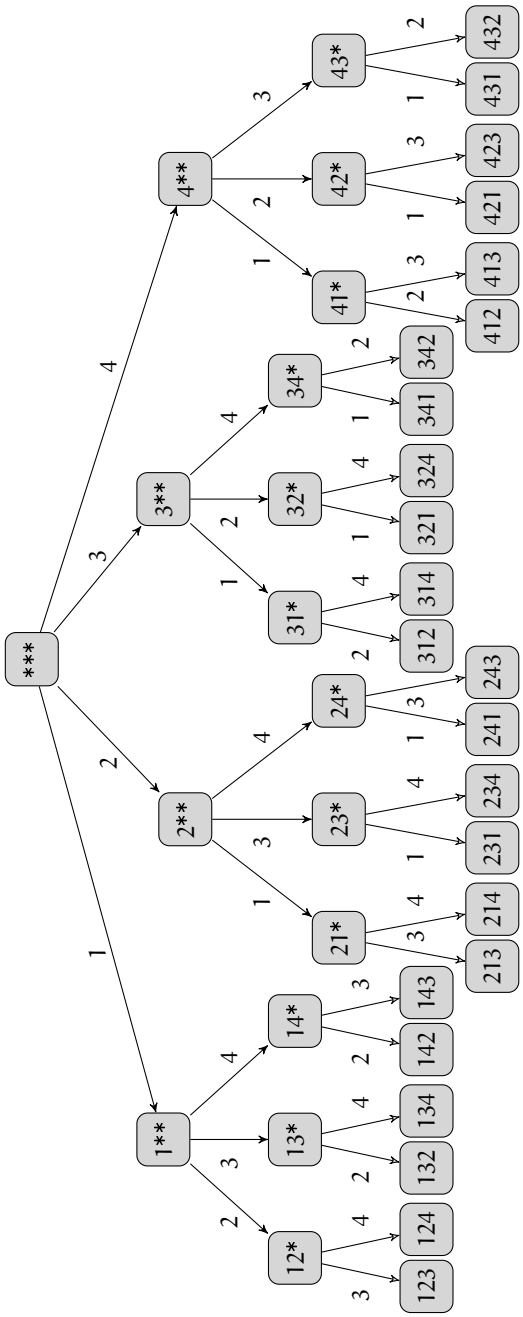


Fig. 5.3 – *Arbre de récursion pour la construction de toutes les injections totales entre les intervalles* 1..3 *et* 1..4

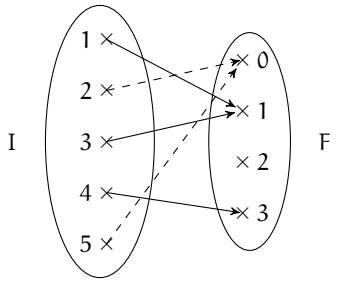


Fig. 5.4 – *Fonction partielle de* $I = 1..5$ *dans* $F = 1..3$. *Le caractère partiel de la fonction est obtenu en créant une fonction totale de* I *vers l'intervalle* $0..3$ *puis en restreignant le codomaine à l'intervalle* $1..3$.

```
4.    pour j parcourant {0} ∪ 1 .. m faire
5.       X[i] ← j ;
6.       si i = n alors
7.          écrire(X)
8.       sinon
9.          P(i + 1)
10.      fin si
11.   fin pour
12. fin
```

répond à cet objectif. L'invariant « $X[1..i-1]$ est constant. C'est une fonction totale de $I = 1..i-1$ dans $F' = 0..m$ et, si un nœud de l'arbre de récursion satisfaisant la condition $(i = n)$ a été imprimé, alors tous les nœuds satisfaisant cette condition ont été imprimés » s'interprète comme « $X[1..i-1]$ est une fonction partielle de l'intervalle $1..i-1$ dans $F = 1..m$ et, si un nœud de l'arbre de récursion satisfaisant la condition $(i = n)$ a été imprimé, alors tous les nœuds satisfaisant cette condition ont été imprimés » à condition d'ignorer les entrées de X contenant 0. Le contexte d'appel est similaire à celui de T. Pour $I = 1..2$ et $F = 1..2$, la figure 5.5page 129 fournit l'arbre de récursion parcouru par ce squelette.

Il existe $(m+1)^n$ fonctions partielles de l'intervalle $1..n$ dans $1..m$, et l'énumération de toutes ces fonctions exige $((m+1)^n - 1)/((m+1) - 1)$ appels.

Énumération des fonctions partielles injectives entre deux intervalles I et F L'obtention des seules fonctions partielles injectives passe également, comme pour le cas des simples fonctions partielles, par l'introduction d'une valeur fantôme 0. Compte tenu de son statut particulier, cette valeur ne doit pas être supprimée de l'intervalle de parcours de la boucle, d'où le squelette suivant :

```
1. procédure PI(i) pré
2.    i ∈ 1 .. n
3. début
4.    pour j parcourant ((1 .. m) − (codom(X[1 .. i − 1])) ∪ {0}) faire
5.       X[i] ← j ;
6.       si i = n alors
```

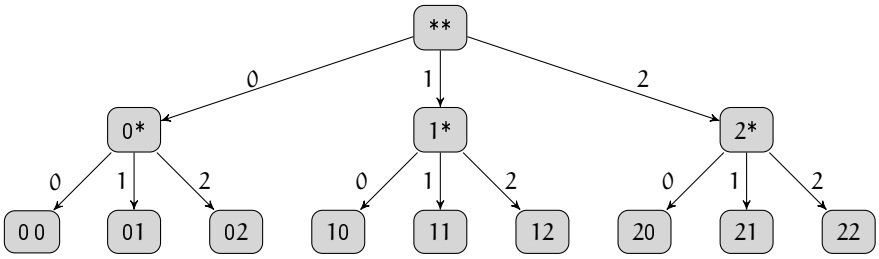


Fig. 5.5 – *Arbre de récursion pour la construction de toutes les fonctions partielles entre* $I = 1..2$ *et* $F = 1..2$. *Le* 0 *dans un nœud représente l'absence de couple pour la position considérée.*

```
7.          écrire(X)
8.       sinon
9.          PI(i + 1)
10.      fin si
11.   fin pour
12. fin
```

La figure 5.6 page 130 fournit l'arbre de récursion pour $I = 1..3$ et $F = 1..2$. Le 0 matérialise un couple absent. Le nombre de fonctions injectives partielles entre $1..n$ et $1..m$ est donné par l'équation récurrente suivante :

$$\begin{aligned} &H(m,0) = 1 \\ &H(1,n) = n+1 && n > 0 \\ &H(m,n) = H(m,n-1) + m \cdot H(m-1,n-1) && n > 0 \text{ et } m > 1. \end{aligned}$$

On reconnaît dans cette récurrence la relation qui fournit le nombre de Stirling de seconde espèce, c'est-à-dire le nombre de partitions en n sous-ensembles d'un sous-ensemble de m éléments (voir également exercice 16, page 44). Il est facile de montrer par induction que l'on a $H(m,n) = m^n + P_{n-1}(m)$, où m est l'indéterminée et $P_{n-1}(m)$ est un polynôme de degré inférieur ou égal à $(n-1)$. Le nombre d'appels au squelette *PI* est donné par l'expression (la démonstration est laissée aux soins du lecteur) :

$$\sum_{i=1}^{n-1} H(m,i).$$

Le raffinement de la boucle dans un langage classique s'obtient de la manière suivante :

```
1. procédure PI2(i) pré
2.    i ∈ 1 .. n
3. début
4.    pour j parcourant {0} ∪ (1 .. m) faire
5.       X[i] ← j ;
6.       si j ≠ 0 alors
7.          L[j] ← faux
8.       fin si ;
9.       si i = n alors
```

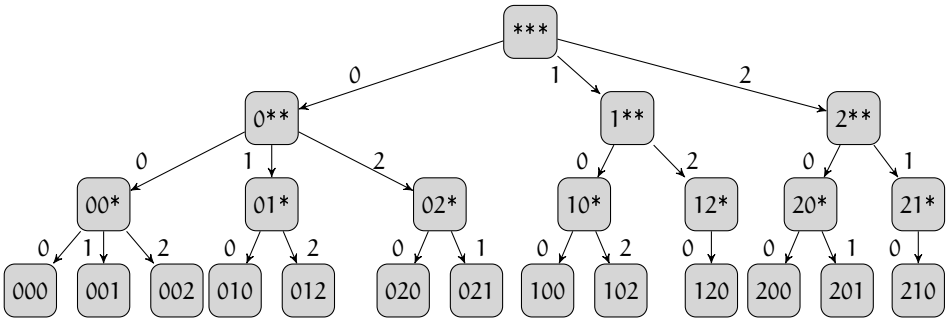


Fig. 5.6 – *Arbre de récursion pour la construction des fonctions partielles injectives entre les intervalles* $1..3$ *et* $1..2$

```
10.          écrire(X)
11.       sinon
12.          PI2(i + 1)
13.       fin si ;
14.       L[j] ← vrai
15.    fin pour
16. fin
```

De même que dans le cas des fonctions totales injectives (squelette *TI2*), L est un tableau de booléens, mais défini cette fois sur l'intervalle $0..m$. L'appel se fait alors comme suit :

```
1. constantes
2.    n = ... et m = ...
3. variables
4.    X ∈ 1..n → 1..m et L ∈ 0..m → 𝔹
5. début
6.    L ← (0..m) × {vrai} ;
7.    PI2(1)
8. fin
```

Comme on vient de le voir, quel que soit le squelette retenu, le nombre de candidats à construire (et comparer) est exponentiel. La complexité de l'algorithme résultant l'est donc elle aussi, indépendamment de l'opération élémentaire retenue. En conséquence, dans la suite de ce chapitre, nous ne nous attarderons sur l'aspect complexité que pour mettre l'accent sur la limitation (algorithmiquement simple et donc peu coûteuse) du nombre de candidats engendrés.

5.1.3 Patrons pour les essais successifs

Jusqu'à présent, nous nous sommes intéressés uniquement à la recherche de *toutes* les solutions à un problème donné. Deux variantes méritent cependant notre attention. La première s'appuie sur un critère de qualité portant sur les solutions, pour ne conserver que la plus performante. Appliquée à l'exemple développé dans la section 5.1.1, page 117, consistant à sélectionner les sous-ensembles d'un ensemble de relatifs E dont la somme est nulle, cette variante pourrait consister par exemple à rechercher la (l'une des) solution(s)

dont le cardinal est le plus grand possible. La seconde est celle qui consiste à rechercher *une seule* solution (si bien sûr il en existe au moins une), celle qui est découverte en premier par exemple.

Il s'agit à présent de « croiser » les squelettes développés dans la section 5.1.2 avec les trois versions présentées ci-dessus (appelées respectivement *ToutesSolutions*, *SolutionOptimale* et *UneSolution*). Ceci va nous conduire à envisager autant de patrons qu'il y a de croisements possibles : un croisement de *ToutesSolutions* avec le squelette T pour obtenir le patron TT, puis avec le squelette TS, pour obtenir le patron TTS, etc. Cependant, tous les croisements ne sont pas développés ci-dessous : certains ne sont pas utilisés, tandis que d'autres font l'objet d'exemples ou d'exercices.

Les patrons dérivés de « ToutesSolutions »

Seuls trois cas sont détaillés ici. Ils résultent du croisement de « ToutesSolutions » et des squelettes T, TS et TI, donnant naissance respectivement aux trois patrons TT, TTS et TTI présentés à la figure 5.7, page 132.

Dans cette figure, les procédures ou fonctions *Satisfaisant*, *Faire*, *Défaire* et *SolutionTrouvée* sont communes aux trois patrons. Ainsi que nous l'avons vu aux sections 5.1.1 page 118, et 5.1.2 page 122, la condition *Satisfaisant* est utilisée pour introduire un élagage, une optimisation ou un raffinement. Les procédures *Faire* et *Défaire* vont par paire. Elles sont nécessaires pour effectuer le retour-arrière quand des variables globales (autres que la structure d'énumération X) ont été introduites, en général pour des raisons d'élagage, d'optimisation ou de raffinement. La condition *SolutionTrouvée* (qui peut ne pas être nécessaire) vient s'ajouter à la condition $(i = n)$ (qui traduit le fait que la structure d'énumération X est complète) pour exprimer que X satisfait aux contraintes propres au problème. La condition $(i \neq n)$ n'est nécessaire que quand la condition *SolutionTrouvée* est présente. Elle caractérise une structure d'énumération incomplète. La recherche de solution doit alors être poursuivie.

Les patrons TP et TPI correspondant aux croisements de « ToutesSolutions » avec les squelettes respectifs P (fonctions partielles) et PI (fonctions partielles injectives) ne sont pas étudiées : ils ne sont pas utilisés dans la suite.

Les patrons dérivés de « SolutionOptimale »

La version *SolutionOptimale* fait l'hypothèse qu'il existe une relation d'ordre total sur les candidats. Y est une variable globale qui reçoit les solutions optimales courantes successives. La procédure *ConserverContexteCour* préserve dans des variables globales les informations permettant de comparer les solutions entre elles. Le patron OPI (solution optimale lorsque le vecteur d'énumération représente une fonction partielle injective) apparaît à la figure 5.8, page 133. Le patron OT (solution optimale lorsque le vecteur d'énumération représente une fonction totale) est quant à lui présenté dans l'exemple de la section 5.1.4, à la page 137.

Une variante de la procédure *SolutionOptimale* consisterait à obtenir *toutes* les solutions optimales. La démarche adoptée pour obtenir toutes les solutions (voir patron TT) ne peut cependant se transposer ici puisque l'on ne peut écrire Y au fil de l'exécution. Il faut attendre la fin du traitement pour effectuer l'affichage, à condition d'avoir enregistré préalablement toutes les solutions optimales dans une structure de données *ad hoc*. Cette version n'est mentionnée que pour mémoire, elle n'est pas utilisée dans la suite.

```
1. procédure TT(i) pré
2.    i ∈ 1 .. n
3. début
4.    pour j parcourant 1 .. m faire
5.       si Satisfaisant(i, j) alors
6.          X[i] ← j ; Faire(i, j) ;
7.          si ( i = n et alors
               SolutionTrouvée(i, j) ) alors
8.             écrire(X)
9.          sinonsi i ≠ n alors
10.            TT(i + 1)
11.         fin si ;
12.         Défaire(i, j)
13.      fin si
14.   fin pour
15. fin
```

```
1. procédure TTS(i) pré
2.    i ∈ 1 .. n et Sauv ∈ 𝔹
3. début
4.    pour j parcourant 1 .. m faire
5.       si Satisfaisant(i, j) alors
6.          X[i] ← j ;
7.          Sauv ← B[j] ; B[j] ← vrai ;
8.          Faire(i, j) ;
9.          si ( i = n et alors
               ∀ k · (k ∈ 1 .. m ⇒ B[k])
               et alors SolutionTrouvée(i, j) )
            alors
10.            écrire(X)
11.         sinonsi i ≠ n alors
12.            TTS(i + 1)
13.         fin si ;
14.         B[j] ← Sauv ; Défaire(i, j)
15.      fin si
16.   fin pour
17. fin
```

```
1. procédure TTI(i) pré
2.    i ∈ 1 .. n
3. début
4.    pour j parcourant ( (1 .. m)−
                          codom(X[1 .. i − 1]) )
      faire
5.       si Satisfaisant(i, j) alors
6.          X[i] ← j ;
7.          Faire(i, j) ;
8.          si ( i = n et alors
               SolutionTrouvée(i, j) ) alors
9.             écrire(X)
10.         sinonsi i ≠ n alors
11.            TTI(i + 1)
12.         fin si ;
13.         Défaire(i, j)
14.      fin si
15.   fin pour
16. fin
```

Fig. 5.7 – *Patrons d'algorithmes de type « essais successifs », dans le cas où l'on recherche* toutes *les solutions et où celles-ci s'expriment comme une fonction totale de* $I = 1 .. n$ *vers* $F = 1 .. m$. *Le patron* TT *concerne les fonctions totales. Le patron* TTS *se rapporte aux fonctions totales surjectives. Le patron* TTI *s'applique aux fonctions totales injectives. Le patron* TTB *(cas des bijections) n'est pas développé, il correspond au patron* TTI *lorsque* $F = I$ *(c'est-à-dire lorsque* $m = n$*). Le cas de la production des vecteurs caractéristiques de tous les sous-ensembles de l'intervalle* $1 .. n$ *s'obtient à partir du patron* TT *en prenant* $F = 0 .. 1$. *Pour les trois patrons ci-dessus, les séquences d'appel sont similaires à celles rencontrées pour les squelettes respectifs.*

```
1. procédure OPI(i) pré
2.    i ∈ 1..n
3. début
4.    pour j parcourant ((1..m)−codom(X[1..
      i − 1])) ∪ {0} faire
5.       si Satisfaisant(i, j) alors
6.          X[i] ← j ; Faire(i, j) ;
7.          si ( i = n et alors
               SolutionTrouvée(i, j) ) alors
8.             Trouvé ← vrai ;
9.             si SolutionMeilleure alors
10.               Y ← X ;
11.               ConserverContexteCour
12.            fin si
13.         sinonsi i ≠ n alors
14.            OPI(i + 1)
15.         fin si ;
16.         Défaire(i, j)
17.      fin si
18.   fin pour
19. fin
```

```
1. constantes
2.    n ∈ ℕ1 et n = ... et
3.    m ∈ ℕ1 et m = ...
4. variables
5.    X ∈ 1..n → 0..n et
6.    Y ∈ 1..n → 0..n et
7.    Trouvé ∈ 𝔹
8. début
9.    Trouvé ← faux ;
10.   OPI(1) ;
11.   si Trouvé alors
12.      écrire(Y)
13.   fin si
14. fin
```

Fig. 5.8 – *Patron et séquence d'appel d'algorithmes de type « essais successifs », dans le cas où l'on recherche une solution* optimale *qui s'exprime comme une fonction* partielle injective *de* $I = 1..n$ *vers* $F = 1..m$. *En cas d'échec (c'est-à-dire s'il n'existe aucune solution), rien n'est produit.*

```
1.  procédure UT(i) pré
2.     i ∈ 1 .. n et j ∈ 1 .. m + 1
3.  début
4.     j ← 1 ;
5.     tant que non(j = m+1 ou Trouvé) faire
6.        si Satisfaisant(i, j) alors
7.           X[i] ← j ; Faire(i, j) ;
8.           si ( i = n et alors
                  SolutionTrouvée(i, j) ) alors
9.              Trouvé ← vrai ;
10.             écrire(X)
11.          sinonsi i ≠ n alors
12.             UT(i + 1)
13.          fin si ;
14.          Défaire(i, j)
15.       fin si ;
16.       j ← j + 1
17.    fin tant que
18. fin
```

```
1.  procédure UTI(i) pré
2.     i ∈ 1 .. n et d ∈ ListeTriée(1 .. m)
3.  début
4.     d ← DomVar(1 .. m − codom(X[1 ..
       i − 1])) ;
5.     tant que non(d = ⟨⟩ ou Trouvé) faire
6.        si Satisfaisant(i, j) alors
7.           X[i] ← d.val ; Faire(i, j) ;
8.           si ( i = n et alors
                  SolutionTrouvée(i, j) ) alors
9.              Trouvé ← vrai ;
10.             écrire(X)
11.          sinonsi i ≠ n alors
12.             UTI(i + 1)
13.          fin si ;
14.          Défaire(i, j)
15.       fin si ;
16.       d ← d.svt
17.    fin tant que
18. fin
```

Fig. 5.9 – *Patrons d'algorithmes de type « essais successifs », dans le cas où l'on recherche* la première *solution et où celle-ci s'exprime comme une fonction totale de* $I = 1..n$ *vers* $F = 1..m$. *Le patron* UT *concerne les fonctions totales. Le patron* UTI *se rapporte aux fonctions totales injectives. Dans* UTI, *la déclaration* $d \in$ ListeTriée$(1..m)$ *définit* d *comme une liste triée de valeurs de l'intervalle* $1..m$ *à prendre en compte. La fonction* DomVar *invoquée à la ligne 4 construit la liste triée des valeurs de l'ensemble qui est passé en paramètre. Par convention, le champs* val *(voir ligne 7) donne accès à la valeur placée en tête de liste, tandis que le champs* svt *(voir ligne 14) désigne le reste de la liste (voir section 1.4.7, page 21). Le patron* UTB *(cas des bijections) n'est pas développé, il correspond au patron* UTI *lorsque* $F = I$ *(c'est-à-dire lorsque* $m = n$*).*

Les patrons dérivés de « UneSolution »

Le troisième et dernier cas, nommément *UneSolution*, se code avec une boucle **tant que**, qui permet l'arrêt dès que la première solution a été découverte. Nous nous limitons aux deux patrons *UT* (« UneSolution » pour les fonctions *totales* de $1..n$ dans $1..m$) et *UTI* (« UneSolution » pour les fonctions totales *injectives* – et bijectives si $m = n$ – de $1..n$ dans $1..m$). Ces patrons sont présentés à la figure 5.9, page 134. L'appel à une instance de ces patrons doit être précédé de l'initialisation à **faux** de la variable booléenne Trouvé.

Le patron *UTI* présente une difficulté technique sur laquelle nous devons nous arrêter. Alors que, dans le patron *UT*, la variable j parcourt le début ou la totalité de l'intervalle $1..m$; ce n'est plus le cas dans le patron *UTI*, où cet intervalle doit être privé des valeurs présentes dans la tranche $X[1..i-1]$. La méthode utilisée ici consiste à construire dans la variable locale d une liste triée des valeurs possibles pour j et à parcourir cette liste dans la boucle.

```
1.  procédure UTI2(i) pré
2.     i ∈ 1 .. n et j ∈ 1 .. m + 1 et L ∈ 1 .. m → 𝔹
3.  début
4.     L ← (1 .. m − codom(X[1 .. i − 1]) × {vrai}) ∪ codom(X[1 .. i − 1]) × {faux} ;
5.     j ← 1 ;
6.     tant que non(j = m + 1 ou Trouvé) faire
7.        si L[j] et Satisfaisant(i, j) alors
8.           X[i] ← j ; Faire(i, j) ;
9.           ...
10.       fin si ;
11.       j ← j + 1
12.    fin tant que
13. fin
```

Fig. 5.10 – *Raffinement du patron* UTI *obtenue en utilisant un tableau de booléens. La ligne 9 est à remplacer par les lignes 8 à 14 du patron* UTI.

Le patron *UTI* peut se raffiner comme le montre la figure 5.10, en remplaçant la liste d par un tableau de booléens L, défini sur l'intervalle 1 .. m. L'initialisation de la boucle doit alors calculer la valeur à donner à L (**vrai** si la position est à prendre en compte, **faux** sinon).

Conclusion

Dans cette section, nous avons montré comment on peut aborder un problème selon l'approche « essais successifs ». La démarche que nous préconisons se décline selon trois étapes :

1. **Patron** Identifier le type de patron imposé par le problème à résoudre. Rappelons que le choix de ce patron se fonde sur le croisement entre un squelette (c'est-à-dire un type de fonction) et les caractéristiques des solutions recherchées (une, toutes, la meilleure).
2. **Instanciation – élagage** Ce patron doit alors être instancié afin de répondre aux contraintes propres au problème considéré. En premier lieu, il s'agit de déterminer le prédicat *SolutionTrouvée* qui sélectionne les solutions parmi les candidats produits. On peut également, par l'intermédiaire du prédicat *Satisfaisant*, introduire des élagages qui vont permettre d'éviter de développer certaines branches stériles. Cette étape d'élagage peut exiger le renforcement de l'invariant de récursivité, associé à l'introduction de variables fraîches. Celles-ci sont initialisées avant le premier appel et mises à jour et restituées dans leur état initial par le couple de procédures Faire et Défaire.
3. **Optimisation** Il s'agit d'une étape optionnelle. En général, il est possible d'améliorer la version obtenue au point 2 en renforçant l'invariant de récursivité comme nous l'avons fait dans l'exemple introductif. Le plus souvent, il est alors nécessaire de revoir les procédures et fonctions auxiliaires *Satisfaisant*, Faire, etc.

Le tableau 5.1 répertorie les patrons possibles et fournit le numéro de page où, s'il y a lieu, le code est développé.

L'approche adoptée dans ce chapitre, avec notamment l'utilisation intensive de variables globales, n'est pas la seule possible. Il peut exister des solutions se démarquant plus ou

	Toutes solutions	Solution optimale	Une solution
Totales	*TT* (p. 132)	*OT* (p. 137)	*UT* (p. 134)
Totales surjectives	*TTS* (p. 132)	*OTS*	*UTS*
Totales injectives	*TTI* (p. 132)	*OTI*	*UTI* (p. 134)
Partielles	*TP*	*OP*	*UP*
Partielles surjectives	*TPS*	*OPS*	*UPS*
Partielles injectives	*TPI*	*OPI* (p. 133)	*UPI*

Tab. 5.1 – *Répertoire des 18 patrons pour les essais successifs et références des pages où sont situés les codes. Le cas des fonctions bijectives s'obtient à partir des fonctions injectives.*

moins des patrons présentés dans cette section. Par principe, les solutions proposées pour les exercices de ce chapitre ne s'écartent cependant pas de ces schémas d'algorithmes.

Dans un registre différent, un paradigme déclaratif tel que la programmation logique permettrait de s'exonérer de l'utilisation de variables globales et surtout de la gestion explicite du retour arrière (puisque c'est le comportement par défaut de ces langages). Dans la programmation par contraintes, la modélisation se fait à travers l'ensemble des relations qu'entretiennent les variables du problème. Ces relations sont activées et propagées lors de la résolution du problème. Ce paradigme de programmation s'applique particulièrement bien aux problèmes traités par la technique des essais successifs. Le lecteur est invité à appliquer ces différents styles de programmation aux exemples et exercices proposés dans ce chapitre.

5.1.4 Un exemple : la partition optimale de tableau

Le problème de base : partition optimale de tableau

Prenons comme exemple un problème de recherche d'une solution optimale. On considère un tableau $T[1..n]$ ($n \geqslant 0$) d'entiers positifs. On désire obtenir une partition de T en deux sacs S_1 et S_2 de sorte que, en notant $Somme1 = \sum_{i \in S_1} i$ et $Somme2 = \sum_{i \in S_2} i$, on ait $Somme2 - Somme1 \geqslant 0$ et $(Somme2 - Somme1)$ minimal.

Par exemple, pour $n = 6$ et T défini de la sorte :

i	1	2	3	4	5	6	7
T[i]	6	8	2	7	9	4	1

une solution est de construire un sac $S_1 = [\![8, 9, 1]\!]$, composé à partir de T[2], T[5] et T[7], de somme $Somme1 = 18$, et un sac $S_2 = [\![6, 2, 7, 4]\!]$ composé à partir de T[1], T[3], T[4] et T[6], de somme $Somme2 = 19$. Le sac S_1 (resp. S_2) peut se représenter par le vecteur caractéristique $\overline{S_1} = [0, 1, 0, 0, 1, 0, 1]$ (resp. $\overline{S_2} = [1, 0, 1, 1, 0, 1, 0]$).

On note que si l'on pose $Somme = \sum_{i \in dom(T)} T[i]$:

$$\begin{aligned} & Somme2 - Somme1 \\ = & \qquad \text{arithmétique} \\ & Somme2 - Somme1 + Somme - Somme \\ = & \qquad \text{propriété de } Somme1 \text{ et } Somme2 \end{aligned}$$

```
1.  procédure OT(i) pré
2.      i ∈ 1 .. n
3.  début
4.      pour j parcourant 1 .. m faire
5.          si Satisfaisant(i, j) alors
6.              X[i] ← j ; Faire(i, j) ;
7.              si i = n et SolutionTrouvée(i, j) alors
8.                  si SolutionMeilleure alors
9.                      Y ← X ;
10.                     ConserverContexteCour
11.                 fin si
12.             sinonsi i ≠ n alors
13.                 OT(i + 1)
14.             fin si ;
15.             Défaire(i, j)
16.         fin si
17.     fin pour
18. fin
```

Fig. 5.11 – *Patron d'algorithmes de type « essais successifs » dans le cas où l'on recherche une solution* optimale *qui s'exprime comme une fonction* totale *de* $I = 1 .. n$ *vers* $F = 1 .. m$. *Cette version présuppose qu'il existe au moins une solution.*

$$\begin{array}{ll} & \text{Somme2} - \text{Somme1} + \text{Somme} - (\text{Somme1} + \text{Somme2}) \\ = & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{arithmétique} \\ & \text{Somme} - 2 \cdot \text{Somme1}. \end{array}$$

Minimiser $(\text{Somme2} - \text{Somme1})$ revient donc à minimiser $(\text{Somme} - 2 \cdot \text{Somme1})$.

Nous allons chercher l'ensemble des sous-sacs du sac représenté par T. Il suffit de considérer l'intervalle $1 .. n$ et de prendre l'ensemble des parties de $1 .. n$. Les vecteurs caractéristiques obtenus donneront indirectement accès au résultat recherché. Le patron qui s'applique ici est *OT* (Solution optimale avec l'ensemble des fonctions totales). Saisissons l'occasion qui s'offre à nous pour présenter ce patron, en complément de ceux développés à la section 5.1.3. C'est l'objet de la figure 5.11. L'existence d'une solution est garantie : il est inutile de gérer le booléen Trouvé comme dans le patron de la figure 5.8, page 133.

Ce patron *OT* s'instancie avec $I = 1 .. n$ et $F = 0 .. 1$ (afin d'obtenir les vecteurs caractéristiques de tous les sous-ensembles). Soit *ÉcartMini1* cette instance.

1. La première ébauche que nous développons est la suivante :

```
1.  procédure ÉcartMini1(i) pré
2.      i ∈ 1 .. n
3.  début
4.      pour j parcourant 0 .. 1 faire
5.          ...
6.      fin pour
7.  fin
```

2. Passons à la condition *Satisfaisant* vue comme une condition d'élagage. Pouvons-nous arrêter l'exploration de l'arbre avant d'atteindre une feuille, en considérant le candi-

dat partiel que nous sommes en train de construire ? La réponse est positive. Soit respectivement SomCour1 et SomOpt la valeur courante de Somme1 et la valeur de l'optimal courant (SomCour1 et SomOpt sont initialisées à 0 avant l'appel principal). Quand la valeur que nous sommes sur le point d'ajouter à SomCour1 l'augmente au-delà de la moitié de Somme (rappelons que le tableau T est composé de nombres entiers *positifs*), on peut élaguer les sous-arbres à cet endroit et explorer une autre branche. On obtient la seconde ébauche :

1. **procédure** *ÉcartMini1*(i) **pré**
2. $i \in 1..n$
3. **début**
4. **pour** j **parcourant** $0..1$ **faire**
5. **si** $\boxed{\text{SomCour1} + j \cdot T[i] \leqslant \left\lfloor \frac{\text{Somme}}{2} \right\rfloor}$ **alors**
6. ...
7. **fin si**
8. **fin pour**
9. **fin**

3. Passons à ce qui correspond aux lignes 6 et 12 du patron *OT*. L'instance de la procédure générique *Faire* consiste à tenir à jour la variable globale SomCour1 ; celle de la procédure *Défaire* annule ce qui a été fait par *Faire* :

1. **procédure** *ÉcartMini1*(i) **pré**
2. $i \in 1..n$
3. **début**
4. **pour** j **parcourant** $0..1$ **faire**
5. **si** $\text{SomCour1} + j \cdot T[i] \leqslant \left\lfloor \frac{\text{Somme}}{2} \right\rfloor$ **alors**
6. $X[i] \leftarrow j$; $\boxed{\text{SomCour1} \leftarrow \text{SomCour1} + j \cdot T[i]\,;}$
7. ...
8. $\boxed{\text{SomCour1} \leftarrow \text{SomCour1} - j \cdot T[i]}$
9. **fin si**
10. **fin pour**
11. **fin**

4. *SolutionTrouvée* vaut **vrai** si l'on est arrivé à une feuille de l'arbre, ce qui s'exprime simplement par $i = n$, d'où découle le raffinement suivant :

1. **procédure** *ÉcartMini1*(i) **pré**
2. $i \in 1..n$
3. **début**
4. **pour** j **parcourant** $0..1$ **faire**
5. **si** $\text{SomCour1} + j \cdot T[i] \leqslant \left\lfloor \frac{\text{Somme}}{2} \right\rfloor$ **alors**
6. $X[i] \leftarrow j$; SomCour1 $\leftarrow$ SomCour1 $+ j \cdot T[i]$;
7. **si** $\boxed{i = n}$ **alors**
8. ...
9. **sinon**
10. *ÉcartMini1*$(i+1)$
11. **fin si** ;

```
12.          SomCour1 ← SomCour1 − j · T[i]
13.        fin si
14.      fin pour
15.  fin
```

5. *SolutionMeilleure* teste si la solution courante est meilleure que la solution optimale trouvée jusqu'alors. Si c'est le cas, cette solution est préservée dans la variable globale Y, tandis que la procédure *ConserverContexteCour* s'instancie en mettant à jour la variable globale SomOpt.

```
1.  procédure ÉcartMini1(i) pré
2.    i ∈ 1 .. n
3.  début
4.    pour j parcourant 0 .. 1 faire
5.      si SomCour1 + j · T[i] ⩽ ⌊Somme/2⌋ alors
6.        X[i] ← j ; SomCour1 ← SomCour1 + j · T[i] ;
7.        si i = n alors
8.          si [SomCour1 > SomOpt] alors
9.            Y ← X ; [SomOpt ← SomCour1]
10.         fin si
11.       sinon
12.         ÉcartMini1(i + 1)
13.       fin si ;
14.       SomCour1 ← SomCour1 − j · T[i]
15.     fin si
16.   fin pour
17. fin
```

L'appel initial de la procédure *ÉcartMini1* s'effectue dans le contexte suivant :

```
1.  constantes
2.    n ∈ ℕ et n = ... et
3.    T ∈ 1 .. n → ℕ et T = [...] et
4.    Somme ∈ ℕ et Somme = Σ_{i=1}^{n} T[i]
5.  variables
6.    X ∈ 1 .. n → 0 .. 1 et Y ∈ 1 .. n → 0 .. 1 et
7.    SomCour1 ∈ ℕ et SomOpt ∈ ℕ et
8.    S1 ∈ sac(ℕ) et S2 ∈ sac(ℕ)
9.  début
10.   SomOpt ← 0 ; SomCour1 ← 0 ;
11.   ÉcartMini1(1) ;
12.   écrire(Y)
13.   /% Construction de S1 et de S2 %/
14. fin
```

La construction effective de S_1 et de S_2 à partir de Y est facile à réaliser. Il suffit de ventiler les valeurs de T dans S_1 ou S_2 selon la valeur correspondante de Y. Cette construction n'est pas réalisée ici.

Notons que, quand il existe plusieurs solutions optimales, cet algorithme enregistre, dans Y, la première dans l'ordre d'énumération lexicographique. Dans le cas de l'exemple

de la page 136, la meilleure solution trouvée est $Y = [0,0,1,1,1,0,0]$, et non pas $Y = [0,1,0,0,1,0,1]$ (qui correspond à celle que nous avons proposée plus haut).

Le problème initial, avec une précondition et une postcondition renforcées

Nous imposons maintenant que n soit pair et que les deux sacs S_1 et S_2 aient obligatoirement la même taille $n/2$. Comment cette nouvelle contrainte peut-elle s'introduire dans le programme sans le bouleverser ?

Une première solution consiste à conserver le programme précédent et à générer de la même manière toutes les feuilles de l'arbre, sauf celles qui sont élaguées par *Satisfaisant*. On ajoutera simplement à *SolutionTrouvée* la condition que S_1 soit de taille $n/2$. Pour ce faire, il faut introduire une mémorisation de la taille courante de S_1 : on va gérer une nouvelle variable globale Taille1. Cette variable sera incrémentée dans *Faire* et décrémentée dans *Défaire*.

```
1.  procédure ÉcartMini2(i) pré
2.     i ∈ 1 .. n
3.  début
4.     pour j parcourant 0 .. 1 faire
5.        si SomCour1 + j · T[i] ⩽ ⌊Somme/2⌋ alors
6.           X[i] ← j ; SomCour1 ← SomCour1+j·T[i] ; Taille1 ← Taille1 + j ;
7.           si i = n et Taille1 = n/2 alors
8.              si SomCour1 > SomOpt alors
9.                 Y ← X ; SomOpt ← SomCour1
10.             fin si
11.          sinonsi i ≠ n alors
12.             ÉcartMini2(i + 1)
13.          fin si ;
14.          SomCour1 ← SomCour1 − j · T[i] ; Taille1 ← Taille1 − j
15.       fin si
16.    fin pour
17. fin
```

L'appel initial devient :

```
1.  variables
2.     /% Cf. lignes de 2 à 6 de l'appel de ÉcartMini1 %/
3.     ⌊n/2⌋ · 2 = n et
4.     SomCour1 ∈ ℕ et SomOpt ∈ ℕ et Taille1 ∈ ℕ et
5.     S1 ∈ sac(ℕ) et S2 ∈ sac(ℕ)
6.  début
7.     SomOpt ← 0 ; SomCour1 ← 0 ; Taille1 ← 0 ;
8.     ÉcartMini2(1) ;
9.     écrire(Y)
10.    /% Construction de S1 et de S2 %/
11. fin
```

Cette solution est certes correcte, mais elle présente l'inconvénient de ne tirer parti de

la nouvelle condition (sur la taille des sacs S_1 et S_2) que lorsqu'un candidat est totalement construit.

La clé d'une amélioration réside dans une nouvelle version de *SolutionTrouvée*. Il est en effet possible de ne pas attendre la fin de la construction du vecteur X pour vérifier que l'égalité des tailles est satisfaite. Il suffit, dès que l'un des deux sacs atteint la taille de $n/2$, de compléter le vecteur X en question sans finir de parcourir l'arbre de récursion. Notons Taille1, Taille2 et SomCour2 respectivement la taille du sac S_1, celle du sac S_2 et la somme partielle de S_2.

1. La première ébauche de la procédure *ÉcartMini3* ne change pas par rapport à *ÉcartMini1* :

```
1. procédure ÉcartMini3(i) pré
2.    i ∈ 1..n
3. début
4.    pour j parcourant 0..1 faire
5.       si SomCour1 + j·T[i] ⩽ ⌊Somme/2⌋ alors
6.          ...
7.       fin si
8.    fin pour
9. fin
```

2. Passons à ce qui correspond aux lignes 6 et 15 du patron *OT*. L'instance de la procédure générique *Faire* consiste à tenir à jour les variables globales SomCour1, SomCour2, Taille1 et Taille2 ; celle de la procédure *Défaire* annule ce qui a été fait par *Faire* :

```
1. procédure ÉcartMini3(i) pré
2.    i ∈ 1..n
3. début
4.    pour j parcourant 0..1 faire
5.       si SomCour1 + j·T[i] ⩽ ⌊Somme/2⌋ alors
6.          X[i] ← j ;
7.          SomCour1 ← SomCour1 + j·T[i] ;
8.          SomCour2 ← SomCour2 + (1 − j)·T[i] ;
9.          Taille1 ← Taille1 + j ; Taille2 ← Taille2 + (1 − j) ;
10.         ...
11.         SomCour1 ← SomCour1 − j·T[i] ;
12.         SomCour2 ← SomCour2 − (1 − j)·T[i] ;
13.         Taille1 ← Taille1 − j ; Taille2 ← Taille2 − (1 − j)
14.      fin si
15.   fin pour
16. fin
```

3. *SolutionTrouvée* signifie que l'on peut construire directement une solution qui mène à une feuille et que le vecteur correspondant est une solution (pas forcément optimale). Il existe deux possibilités : soit $\text{Taille1} = n/2$, soit $\text{Taille2} = n/2$.

4. *SolutionMeilleure* et *ConserverContexteCour* consistent à enregistrer le vecteur solution que l'on vient de trouver s'il est meilleur que l'optimal courant. Il faut également finir de construire les solutions partielles pour lesquelles l'un des deux tableaux est de taille inférieure à $n/2$.

5. *SolEncorePossible*, le prédicat qui permet de poursuivre l'énumération, se limite à $(i \neq n)$.

Au final, l'algorithme obtenu est :

```
1.  procédure ÉcartMini3(i) pré
2.     i ∈ 1 .. n
3.  début
4.     pour j parcourant 0 .. 1 faire
5.        si SomCour1 + j · T[i] ⩽ ⌊Somme/2⌋ alors
6.           X[i] ← j ;
7.           SomCour1 ← SomCour1 + j · T[i] ;
8.           SomCour2 ← SomCour2 + (1 − j) · T[i] ;
9.           Taille1 ← Taille1 + j ; Taille2 ← Taille2 + (1 − j) ;
10.          si Taille1 = n/2 alors
11.             si SomCour1 > SomOpt alors
12.                Y[1 .. i] ← X[1 .. i] ; Y[i + 1 .. n] ← (i + 1 .. n) × {0};
13.                SomOpt ← SomCour1
14.             fin si
15.          sinonsi Taille2 = n/2 alors
16.             si SomCour1 > SomOpt alors
17.                Y[1 .. i] ← X[1 .. i] ; Y[i + 1 .. n] ← (i + 1 .. n) × {1};
18.                SomOpt ← SomCour1
19.             fin si
20.          sinonsi i ≠ n alors
21.             EcartMini3(i + 1)
22.          fin si ;
23.          SomCour1 ← SomCour1 − j · T[i] ;
24.          SomCour2 ← SomCour2 − (1 − j) · T[i] ;
25.          Taille1 ← Taille1 − j ; Taille2 ← Taille2 − (1 − j)
26.       fin si
27.    fin pour
28. fin
```

Une séquence d'appel à la procédure *ÉcartMini3* se présente comme suit :

```
1.  variables
2.     /% Cf. lignes de 2 à 6 de l'appel de ÉcartMini1 %/
3.     SomOpt ∈ ℕ et SomCour1 ∈ ℕ et SomCour2 ∈ ℕ et
4.     Taille2 ∈ ℕ et S1 ∈ sac(ℕ) et S2 ∈ sac(ℕ)
5.  début
6.     SomOpt ← 0 ;
7.     SomCour1 ← 0 ; Taille1 ← 0 ; SomCour2 ← 0 ; Taille2 ← 0 ;
8.     ÉcartMini3(1) ;
9.     écrire(Y)
10.    /% Construction de S1 et de S2 %/
11. fin
```

Le problème initial, avec une précondition renforcée

Nous reprenons le problème précédent, avec une précondition renforcée par la proposition suivante : le tableau T est trié par ordre croissant. La théorie (voir [36] et chapitre 3) nous apprend que si l'on dispose d'un programme correct pour une précondition R donnée, alors ce même programme est encore correct pour une précondition P plus forte (c'est-à-dire où $P \Rightarrow R$). Cependant, en n'exploitant pas la nouvelle précondition, on risque de passer à côté d'une version plus efficace.

On pourrait penser que l'hypothèse du tri autorise à sortir du cadre de la programmation par essais successifs et qu'il existe un algorithme de complexité $\mathcal{O}(n)$ (ou polynomiale) qui résolve le problème. En réalité, les algorithmiciens estiment, sans en être sûrs, qu'un tel algorithme (qui serait sans doute un algorithme *glouton*, voir chapitre 7) n'existe pas.

Nous admettons donc que, pour tenter d'obtenir une meilleure version que *ÉcartMini3*, il faut aborder le problème sous l'angle des essais successifs. En quoi le fait que T soit trié peut-il autoriser de nouveaux élagages ?

Nous conservons les variables globales $Taille1$, $SomCour1$, $Taille2$ et $SomCour2$ de la version précédente. Quand T n'est pas trié, la condition d'élagage se présente de la sorte :

```
...
si SomCour1 + j · T[i] ⩽ ⌊Somme/2⌋ alors
   ...
fin si
...
```

Elle se contente de vérifier que la valeur $j \cdot T[i]$ ajoutée à $SomCour1$ ne dépasse pas $\lfloor Somme/2 \rfloor$. Elle peut être raffinée en exploitant le fait que T est trié. Si, en minorant $Somme1$ mise à jour par anticipation en lui ajoutant $j \cdot T[i]$, on dépasse $\lfloor Somme/2 \rfloor$, ou si de manière duale, en majorant $Somme2$ mise à jour par anticipation en lui ajoutant $(1-j) \cdot T[i]$, on reste en deçà de $\lfloor Somme/2 \rfloor$, on n'a aucune chance de satisfaire la condition qui veut que $Somme1 \leqslant Somme2$.

En effet, à ce stade :

— il reste $(n/2 - Taille1 - j)$ valeurs à mettre dans S_1 et $(n/2 - Taille2 - (1-j))$ valeurs à mettre dans S_2,

— $((SomCour1 + j \cdot T[i]) + (n/2 - Taille1 - j) \cdot T[i+1])$ est un minorant de $Somme1$, et $((SomCour2 + (1-j) \cdot T[i]) + (n/2 - Taille2 - (1-j)) \cdot T[n])$ majore $Somme2$.

En conséquence, la condition d'élagage *Satisfaisant* peut se réécrire :

```
...
si ( (SomCour1 + j · T[i]) + (n/2 − Taille1 − j) · T[i+1] ⩽ ⌊Somme/2⌋ et
   ( (SomCour2 + (1 − j) · T[i]) + (n/2 − Taille2 − (1 − j)) · T[n] ⩾ ⌊Somme/2⌋ )
alors
   ...
fin si
...
```

Le reste du programme est inchangé. Un élagage complémentaire peut être apporté : on peut vérifier à chaque étape que $SomOpt$ peut être amélioré. En effet, il reste à ajouter $(n/2 - Taille1 - j)$ valeurs à S_1 ; et pour avoir une chance de faire mieux que l'optimal

courant, il faut qu'en prenant $(n/2 - Taille1 - j)$ fois la plus grande valeur, on dépasse strictement SomOpt. On peut donc compléter *Satisfaisant* de la façon suivante :

```
...
si ( (SomCour1 + j·T[i]) + (n/2 − Taille1 − j)·T[i+1] ⩽ ⌊Somme/2⌋ et
   ( (SomCour2 + (1−j)·T[i]) + (n/2 − Taille2 − (1−j))·T[n] ⩾ ⌊Somme/2⌋
   ( (SomCour1 + j·T[i]) + (n/2 − Taille1 − j)·T[n] > SomOpt )
alors
   ...
fin si
...
```

On peut encore raffiner cet algorithme en remarquant que, dans la solution proposée, on se base sur la plus petite ($T[i+1]$) ou la plus grande ($T[n]$) des valeurs restantes. Au prix d'un calcul plus coûteux, dans la première condition, on pourrait ajouter à $(SomCour1 + j \cdot T[i])$ la somme $T[i+1] + \cdots + T[i + (n/2 - Taille1 - j) - 1]$ au lieu de $(n/2 - Taille1 - j) \cdot T[i+1]$. De même, dans la seconde condition, on pourrait ajouter à $(SomCour2 + (1-j) \cdot T[i])$ la somme $T[n/2 + Taille2 - (1-j) + 1] + \cdots + T[n]$ plutôt que $(n/2 - Taille2 - (1-j)) \cdot T[n]$. Enfin, dans la troisième condition, on pourrait ajouter à $(SomCour1 + j \cdot T[i])$ la somme $T[n/2 + Taille1 - j + 1] + \cdots + T[n]$ au lieu de $(n/2 - Taille1 - j) \cdot T[n]$.

Remarque En l'absence de l'hypothèse de tri de T, on pourrait malgré tout sous-estimer la valeur future de *Somme1* à $SomCour1 + j \cdot T[i] + (n/2 - Taille1 - j) \cdot 1$. En revanche, aucune sur-estimation de *Somme2*, ni de SomOpt ne serait possible sans information sur les éléments de la tranche $T[i+1 \,..\, n]$.

5.2 Ce qu'il faut retenir des essais successifs

La démarche adoptée dans ce chapitre au regard des essais successifs est originale. Ainsi que le montre le schéma de développement en trois étapes présenté à la section 5.1.3, page 135, elle consiste à établir une frontière claire entre l'aspect purement routinier du développement (l'identification d'un patron de développement) et l'aspect créatif (la recherche de techniques d'élagage et d'optimisation). Ce dernier aspect exige souvent des trésors d'ingéniosité pour découvrir des solutions plus efficaces que la force brute. En général, il est toutefois difficile de quantifier le gain en performance qui en résulte.

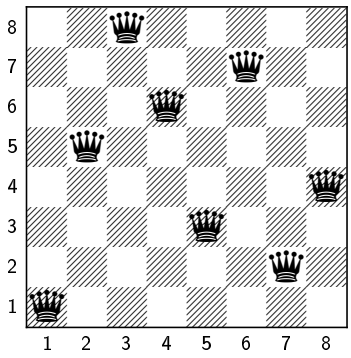
Le style de développement préconisé dans ce chapitre est à rapprocher de ce que, en génie logiciel, il est convenu d'appeler « design pattern » (ou « patron de conception »). Le principal avantage de la démarche réside dans le fait qu'il n'est pas nécessaire de construire une solution *ex nihilo* pour chaque nouveau problème. En revanche, par définition, le problème considéré doit pouvoir se reformuler pour se conformer au « patron » choisi.

5.3 Exercices

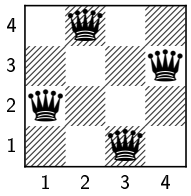
Exercice 51. Le problème des n reines ○ ⁝

Il s'agit d'un exercice classique de programmation, traité ici selon le paradigme « essais successifs ». Le problème consiste à tenter de placer n reines sur un échiquier $n \times n$, sans qu'aucune de ces pièces ne soit en prise. Deux solutions sont étudiées. La première solution utilise comme structure d'énumération une matrice carrée destinée à représenter toutes les fonctions totales de l'échiquier vers les booléens. Cette solution fait l'objet d'une optimisation. La seconde solution utilise comme structure d'énumération un vecteur destiné à représenter toutes les bijections de l'intervalle $1..n$ sur $1..n$.

Jouez-vous aux échecs ? Savez-vous comment mettre huit reines sur un échiquier de sorte qu'elles ne se menacent pas mutuellement ? Rappelons qu'une reine, au jeu d'échecs, met en prise toute autre pièce située sur la même colonne, la même ligne ou l'une de ses deux diagonales (à condition qu'elle ne soit pas masquée par une pièce intermédiaire). On peut poser ce problème de manière plus générale : comment mettre n reines ($n \geqslant 4$) [2] sur un échiquier de taille $n \times n$ de sorte qu'aucune ne soit attaquée par une autre ? Par exemple, le schéma suivant fournit une solution pour $n = 8$ (partie (a)) et pour $n = 4$ (partie (b)) :



(a)



(b)

Dans la suite, on souhaite obtenir toutes les configurations de l'échiquier répondant au problème posé.

Question 1. Soit un échiquier de taille $n \times n$ et deux reines R_1 et R_2 situées sur les cases respectives (l_1, c_1) et (l_2, c_2). À quelle condition nécessaire et suffisante R_1 et R_2 sont-elles situées sur une même diagonale montante (sud-ouest – nord-est) ? Sur une même diagonale descendante (nord-ouest – sud-est) ? 51 - Q 1

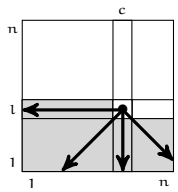
2. Pour $n = 1$, il existe une seule solution, pour $n = 2$ ainsi que pour $n = 3$, il n'existe pas de solution, et pour $n \geqslant 4$ il existe plusieurs solutions.

Une première version

Dans cette première version, on choisit comme structure d'énumération le tableau X, défini sur le produit cartésien $1 \,..\, n \times 1 \,..\, n$ et à valeur dans $\mathbb{B}$ (**faux** (resp. **vrai**) signifiant l'absence (resp. la présence) d'une reine sur la position considérée). Si X est une solution, X est une fonction totale (chaque cellule de X est l'origine d'un couple) surjective (il y a n couples ayant comme extrémité **vrai** et $(n^2 - n)$ couples ayant comme extrémité **faux**). Un tableau X en cours d'élaboration est une fonction totale *non surjective a priori* puisqu'il peut n'y avoir aucune cellule qui désigne **faux**. Cependant, si X est une solution, X satisfait la contrainte propre qui précise que les n reines attaquent toutes les cases vides de l'échiquier mais qu'elles ne sont pas elles-mêmes en prise. Cette contrainte entraîne la surjectivité de X. Il est donc inutile de vérifier la surjectivité (comme le ferait le patron *TTS*, page 132), celle-ci étant un sous-produit de la solution. Le patron approprié est donc *TT* (toujours page 132), aménagé à partir du squelette *T2D* (page 123) afin de tenir compte du caractère bidimensionnel du domaine de définition de la structure d'énumération.

51 - Q 2 **Question** 2. Identifier les élagages possibles et tracer une portion significative de l'arbre de récursion pour le cas $n = 4$.

51 - Q 3 **Question** 3. Pour cette version, on suppose disponibles les quatre fonctions booléennes *LigneLibre*(l, c), *ColonneLibre*(l, c), *DiagMontLibre*(l, c) et *DiagDescLibre*(l, c) (l et c représentent respectivement le numéro de ligne et le numéro de colonne de la case où l'on envisage de poser une reine), qui délivrent **vrai** si et seulement si la portion de ligne, de colonne ou de diagonale mentionnée dans le schéma suivant ne contient pas de reines.



La procédure *NReines1*(l, c) tente de placer une reine dans la case (l, c) sachant que la partie de l'échiquier délimitée par les lignes allant de 1 à $l - 1$ constitue une solution partielle. Écrire cette procédure en précisant comment s'instancie chacune des opérations génériques du patron *TT*.

51 - Q 4 **Question** 4. Les appels des quatre fonctions booléennes mentionnées ci-dessus conduisent à des calculs redondants, qu'il est possible d'éviter en remplaçant ces fonctions par des *tableaux* de booléens. Comment ? Fournir la procédure *NReines2* qui tient compte de cette remarque.

Une seconde version

On souhaite améliorer la version précédente en raffinant la structure de données de la manière suivante (voir aussi [35]). Si l'on considère la fonction X utilisée ci-dessus, on peut constater que l'on ne perd pas d'information en ne conservant que les couples du domaine de X qui sont en relation avec la valeur **vrai**. Puisqu'une solution, si elle existe, doit comporter une reine et une seule par ligne, cette relation, X, est une bijection de l'intervalle $1 \,..\, n$ sur lui-même (ou si l'on préfère, une permutation de l'intervalle $1 \,..\, n$). Nous sommes alors dans la situation où le patron approprié est *TTI* de la page 132,

appliqué au cas où le domaine et le codomaine de X sont $1..n$. Il nous faut donc produire, non plus toutes les fonctions totales comme précédemment, mais uniquement toutes les bijections de $1..n$ sur $1..n$. Par construction, rapportée à leur interprétation échiquéenne, ces bijections représentent une configuration d'échiquier comprenant une reine par ligne et une reine par colonne. Reste cependant à réaliser le filtrage qui se rapporte aux diagonales. Pour ce faire, on propose de reprendre la technique des tableaux booléens utilisée dans la question 4 ci-dessus.

Question 5. Identifier les élagages possibles et dessiner une portion significative de l'arbre de récursion pour le cas $n = 4$. 51 - Q 5

Question 6. Mettre en œuvre cette solution à travers la procédure *NReines3*(l) dans laquelle le paramètre l est l'indice de remplissage du vecteur d'énumération X. 51 - Q 6

Question 7. Comparer expérimentalement les trois solutions pour quelques valeurs de n. 51 - Q 7

Exercice 52. Les sentinelles ∘ ⁝

Frère jumeau du problème des n *reines (voir exercice 51), qu'il est conseillé de réaliser au préalable, le présent exercice s'en démarque sur deux points. D'une part, le vecteur d'énumération ne représente pas cette fois une bijection, mais une injection partielle, et d'autre part il s'agit de rechercher une solution* optimale *et non* toutes *les solutions.*

Deux versions sont étudiées. La première est la transposition directe des algorithmes présentés dans l'introduction du chapitre : la seule structure de données est la structure d'énumération. Dans la seconde, par application de la stratégie classique du renforcement de l'invariant de récursivité, on adjoint au vecteur d'énumération une structure de données ad hoc *destinée à améliorer l'efficacité de l'algorithme.*

On veut placer un nombre *minimal* de reines sur un échiquier $n \times n$ de sorte que :

1. elles ne sont pas en prise,
2. l'ensemble des cases de l'échiquier est sous leur contrôle.

On rappelle qu'une reine contrôle les cases de la colonne et de la ligne sur laquelle elle se trouve, ainsi que les cases des deux diagonales passant par la case où elle est située. L'échiquier (a) de la figure 5.12, page 148, illustre une première configuration avec six reines contrôlant toutes les cases d'un échiquier 8×8 sans qu'aucune d'elles ne soit attaquée. L'échiquier (b) montre une autre configuration, à cinq reines cette fois.

La configuration (b) est meilleure que la (a) du point de vue du critère d'optimalité considéré (le nombre de reines présentes sur l'échiquier), mais on ne sait pas si elle est optimale. En fait, elle l'est car il n'existe pas de configuration à quatre reines contrôlant toutes les cases de l'échiquier. Le lecteur pourra vérifier expérimentalement cette affirmation à l'issue de l'exercice.

Question 1. Pour un échiquier $n \times n$, donner un minorant au nombre de reines nécessaires pour résoudre le problème. 52 - Q 1

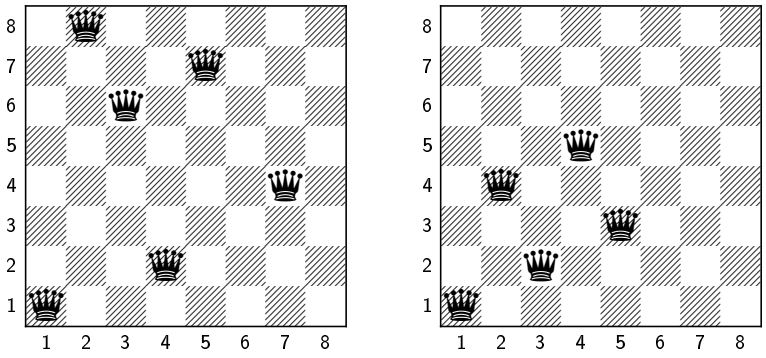


Fig. 5.12 – *Deux exemples d'échiquiers avec sentinelles*

Dans l'exercice des n reines (voir exercice 51, page 145), nous avons étudié deux solutions, la première fondée sur une matrice d'énumération (voir questions 2 et 3, page 146), dont l'efficacité est perfectible, et la seconde, fondée sur un vecteur d'énumération représentant une fonction totale injective (voir question 6, page 147), qui améliore sensiblement la première solution, comme le montrent les résultats de la question ?? page ??. Ici, nous écartons d'emblée la première solution pour ne retenir que l'homologue de la seconde solution. Cependant, compte tenu de la nature du problème et outre le fait qu'il s'agit de rechercher la meilleure solution, le patron à instancier est *OPI* (voir figure 5.8, page 133). En effet, puisqu'il peut exister des lignes privées de reines (voir figure 5.12, page 148), certaines cellules du vecteur d'énumération X peuvent ne pas désigner de colonne. X représente donc une fonction partielle. Elles est injective puisqu'il ne peut y avoir plus d'une reine par colonne. Deux variantes sont étudiées : la première n'utilise comme structure de données que la structure d'énumération X et la solution optimale Y, la seconde adjoint à X, à des fins d'efficacité, des structures de données redondantes par rapport à X.

52 - Q 2 **Question** 2. Déterminer les élagages possibles. Construire une partie de l'arbre de récursion incluant au moins une situation d'élagage et une situation de succès, pour un échiquier 5×5.

52 - Q 3 **Question** 3. On s'inspire du patron *OPI* (figure 5.8, page 133) pour définir la procédure *Sentinelles1*. Préciser comment s'instancient les opérations génériques auxiliaires de *OPI*. Fournir le code, ainsi qu'un exemple d'appel de la procédure *Sentinelles1*. La contrainte que l'on s'impose pour cette solution (voir ci-dessus) a comme conséquence que le couple d'opérations *Faire* et *Défaire* est sans objet ici.

52 - Q 4 **Question** 4. La solution précédente conduit à refaire plusieurs fois les mêmes calculs (comme le dénombrement des cellules libres dès l'ajout d'une nouvelle reine). Nous souhaitons raffiner cette solution en adjoignant à X les éléments suivants.

— Le tableau Prise, défini sur $1..n \times 1..n$ et à valeur dans $0..n$, qui pour une configuration donnée de X, comptabilise, pour chaque case de l'échiquier, le nombre de fois où elle est en prise. Si la position (l, c) est occupée par une reine, $Prise[l, c] = 1$.

— La variable entière NbReinesPlacées qui, pour une configuration donnée de X, fournit le nombre de reines sur l'échiquier.

— La variable entière NbCasesLibres qui, pour une configuration donnée de X, fournit le nombre de cases qui ne sont pas contrôlées par au moins une reine.

Mises à jour de manière incrémentale, ces structures permettent d'éviter des calculs inutiles.
Répertorier et spécifier les opérations nécessaires à la gestion du tableau Prise, puis définir la procédure *Sentinelles2*, instance du patron *OPI*. Fournir son code.

Exercice 53. Parcours d'un cavalier aux échecs ○ ⁝

Encore un problème sur le thème des échiquiers. Cette fois, le résultat ne consiste pas à obtenir une configuration particulière mais à déterminer un ordre de placement de pièces. Cet exercice étudie deux variantes du célèbre « tour du cavalier », qui vise à faire parcourir toutes les cases d'un échiquier à un cavalier avant de revenir à son point de départ (voir l'exercice 49, page 113. La première partie de l'exercice recherche toutes les solutions pour aller d'une case d à une case a. Une évaluation (très grossière) de la complexité est également demandée. La seconde partie porte sur la recherche d'une solution optimale.

Détermination de tous les déplacements d'un cavalier

On considère un échiquier $n \times n$ vide ($n \geqslant 1$, typiquement $n = 8$), un cavalier, d et a deux cases différentes de l'échiquier. On recherche tous les parcours élémentaires[3] que peut emprunter le cavalier posé sur la case de départ d pour atteindre la case d'arrivée a, en respectant les règles du déplacement des cavaliers au jeu d'échec.

Dans la suite, on utilise indifféremment la notation échiquéenne (chiffres en ordonnée, lettres en abscisse, positions sous la forme lettre/chiffre) ou cartésienne pour repérer une case de l'échiquier.

Rappel du déplacement du cavalier au jeu d'échec : si le cavalier est sur la case (i, j), il peut aller sur les huit cases ci-dessous :

$$\begin{array}{llll} (i-2, j-1) & (i-2, j+1) & (i-1, j-2) & (i-1, j+2) \\ (i+1, j-2) & (i+1, j+2) & (i+2, j-1) & (i+2, j+1), \end{array} \tag{5.2}$$

à condition évidemment de ne pas sortir de l'échiquier. Le schéma (a) de la figure 5.13 représente les déplacements possibles en un coup pour un cavalier placé initialement sur la case $(4, 4)$.

3. C'est-à-dire les parcours sans circuit (voir chapitre 1 et l'exercice 54, page 152).

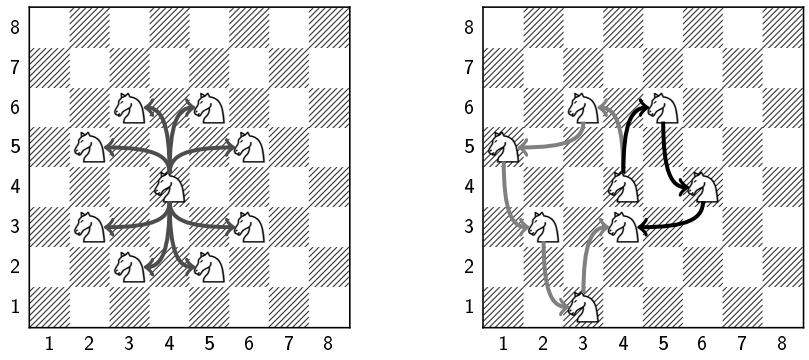


Fig. 5.13 – *Déplacement des cavaliers aux échecs.* (a) *: tous les déplacements possibles d'un cavalier.* (b) *: deux parcours d'un cavalier entre les positions* $(4,4)$ *et* $(4,3)$.

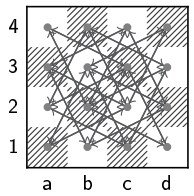


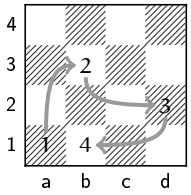
Fig. 5.14 – *Graphe de tous les déplacements d'un cavalier sur un échiquier* 4×4

Exemple Le schéma (b) de la figure 5.13 montre deux parcours pour aller de la case $(4,4)$ à la case $(4,3)$, l'un de longueur 5, l'autre de longueur 3.

Une première solution – qui n'est pas développée ici – consiste à calculer préalablement le graphe des déplacements possibles du cavalier, depuis toute case de l'échiquier. Ainsi, pour $n = 4$, on obtiendrait le graphe de la figure 5.14.

Le problème devient alors un problème de recherche de tous les chemins élémentaires d'un sommet à un autre sommet dans un graphe. Les méthodes classiques de recherche de tous les chemins élémentaires peuvent alors s'appliquer. Le principe sur lequel nous fondons notre solution est différent. Il consiste à calculer, au fur et à mesure des besoins, les cases à portée du cavalier depuis sa position courante. Une spécification possible du problème consiste à considérer que le parcours réalisé est représenté par une matrice d'énumération X définie sur le domaine $1..n \times 1..n$ et à valeurs sur $1..n \times 1..n$. La cellule $X[l,c]$ contient un couple qui désigne la case succédant à la case (l,c) dans le parcours. Une telle matrice représente une fonction *partielle* puisque toutes les cases ne sont pas forcement atteintes. De plus, elle est *injective*, puisqu'une case extrémité apparaît au plus une fois. Deux contraintes propres doivent être mentionnées : l'identité n'appartient pas à la fermeture transitive (voir définition 14, page 26) de X (afin d'exclure les circuits), mais le couple (d,a)

(case de départ et d'arrivée) appartient bien, lui, à la fermeture transitive. Un raffinement possible de cette structure d'énumération consiste à placer sur un échiquier $n \times n$ le numéro du coup effectué par le cavalier lors de son trajet. Par exemple, sur un échiquier 4×4 pour aller de a1 à b1, une possibilité serait d'avoir le trajet suivant :

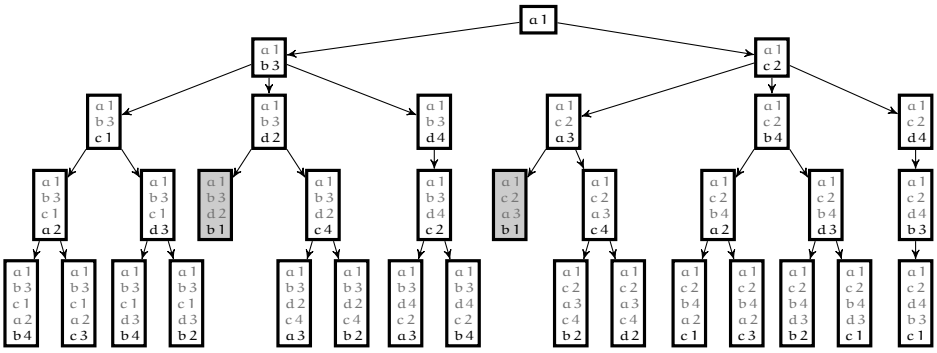


La structure d'énumération représente encore une fonction partielle (car certaines cases n'apparaissent pas dans le trajet), injective (car à une étape donnée est associée une seule case) de $1..n \times 1..n$ dans $1..n^2$. C'est *a priori* un candidat possible pour la fonction d'énumération recherchée. Cependant, compte tenu de notre expérience sur ce type de structure (voir exercice 51, page 145), nous écartons cette solution. En raison du caractère injectif de cette fonction, il est possible de prendre la fonction réciproque, qui est donc aussi une fonction partielle injective, mais cette fois de $1..n^2$ dans $1..n \times 1..n$. Il est cependant important de noter qu'un vecteur d'énumération *en cours d'élaboration* représente une fonction *totale* sur le domaine $1..i-1$, ce qui nous autorise à prendre *TTI* (voir page 132) pour patron plutôt que *TPI*. L'exemple ci-dessus se présente alors sous la forme d'un vecteur d'énumération contenant les coordonnées échiquéennes des cases atteintes :

1	2	3	4
(a, 1)	(b, 3)	(d, 2)	(b, 1)

Ce faisant, nous avons néanmoins introduit une difficulté supplémentaire, qu'il va falloir surmonter. Elle se rapporte à la structure des squelettes présentés à la section 5.1.2, page 122, dans lesquels la boucle **pour** parcourt un ensemble *scalaire*, dont les valeurs sont enregistrées dans la structure d'énumération. Ce n'est plus le cas ici puisque le vecteur d'énumération contient des *couples*. Comment résoudre ce problème ? Une solution consisterait à utiliser *deux* boucles pour parcourir toutes les cases de l'échiquier. Il y a cependant mieux à faire dans la mesure où seuls (au plus) huit emplacements sont candidats à être la prochaine étape du parcours. Au prix d'une légère entorse au patron *TTI*, il suffit alors de parcourir l'intervalle $1..8$, pour donner indirectement accès aux cases candidates, ce qui s'obtient par l'intermédiaire de la description 5.2 page 149.

Question 1. On considère un échiquier 4×4, un cavalier posé sur la case de départ a1 et destiné à atteindre la case b1. Poursuivre le développement de la branche gauche de l'arbre de recherche ci-dessous : 53 - Q 1



53 - Q 2 **Question** 2. Effectuer l'analyse du problème posé en précisant les constituants du patron *TTI*. Fournir d'une part le code de la procédure *Cavalier1*, instance du patron *TTI*, et d'autre part le code d'une séquence d'appel. Évaluer la complexité au pire en nombre de nœuds de l'arbre de récursion.

Détermination du parcours optimal d'un cavalier

On s'intéresse maintenant au problème suivant : en combien de coups minimum un cavalier peut-il aller de la case $d = (i, j)$ à la case $a = (k, l)$ (en supposant toujours ces deux cases différentes) ? Pour ce faire, on se propose de rechercher le parcours optimal (ou l'un des parcours optimaux) d'un cavalier en instanciant le patron *OPI* (voir figure 5.8, page 133). On conserve la même structure d'énumération que dans la première partie.

53 - Q 3 **Question** 3. Spécifier les divers éléments à instancier dans le patron *OPI* permettant de résoudre ce problème.

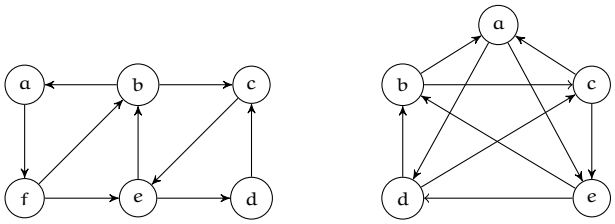
53 - Q 4 **Question** 4. Donner l'algorithme *CavalierOpt* qui permet de connaître l'un des parcours optimaux du cavalier.

Exercice 54. Circuits et chemins eulériens – tracés d'un seul trait ⚷ ⚷

Cet exercice s'intéresse aux parcours eulériens dans un graphe orienté connexe. Partant de l'algorithme pour la recherche d'un circuit eulérien dans un graphe orienté obtenu dans la troisième question, on demande de le transformer afin de rechercher un chemin eulérien dans un graphe non orienté. Une application aux tracés d'un seul trait (c'est-à-dire aux tracés pour lesquels on ne lève pas la plume et on ne repasse pas sur un trait) est étudiée.

On considère un graphe orienté $G = (N, V)$ dans lequel N est l'ensemble des sommets et V l'ensemble des arcs. Posons $n = \text{card}(V)$. On étudie tout d'abord le problème des circuits eulériens, puis celui des chemins eulériens (voir définitions 12 et 13, page 25).

54 - Q 1 **Question** 1. Pour chacun des graphes ci-dessous, fournir, s'il en existe, l'un des circuits eulériens.

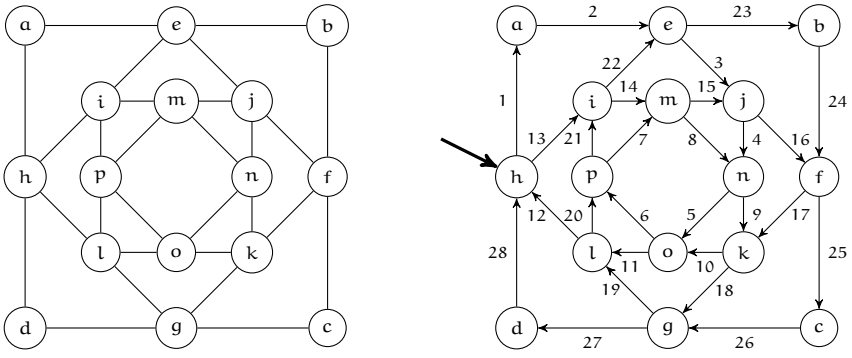


(a) (b)

Question 2. Dans le cadre de la recherche de tous les circuits eulériens pour le graphe orienté G, on choisit une structure d'énumération X contenant des *sommets*. Compléter la définition de X considérée comme une fonction, en déduire le type de patron qui doit s'appliquer. 54 - Q 2

Question 3. Instancier la procédure générique *ToutesSolutions*(i) (voir figure 5.7, page 132), afin d'obtenir un algorithme à essais successifs permettant d'afficher tous les circuits eulériens d'un graphe orienté connexe. 54 - Q 3

Question 4. Le problème du tracé sans lever la plume se pose en des termes différents de la recherche d'un circuit dans un graphe orienté. En effet, un graphe *non orienté* connexe est fourni et il s'agit de découvrir un *chemin* eulérien (c'est-à-dire une succession de sommets qui passe une et une seule fois par chacune des arêtes – soit dans un sens, soit dans un autre – sans nécessairement revenir au sommet de départ. En revanche, ce chemin peut franchir un nœud autant de fois qu'on l'estime nécessaire). 54 - Q 4
La partie (a) du schéma ci-dessous représente le dessin qu'il faut réaliser sans lever la plume et sans passer plusieurs fois sur le même trait.



(a) (b)

La partie (b) montre une solution possible. Il s'agit d'un chemin eulérien débutant au sommet h ; ce chemin constitue un graphe orienté qui se superpose au graphe initial. Chaque arc est accompagné du numéro d'ordre du parcours.

Expliquer les modifications qu'il faut apporter à l'algorithme de la troisième question pour résoudre cette variante du problème initial.

Exercice 55. Chemins hamiltoniens : les dominos ○ •

L'intérêt de cet exercice est d'étudier un algorithme de type « essais successifs » pour le problème classique de la recherche d'un chemin hamiltonien dans un graphe.

On considère le jeu suivant : six dominos portent chacun un mot de quatre lettres, tiré d'un lexique du français. On peut juxtaposer deux dominos si les deux dernières lettres du premier forment un mot de quatre lettres avec les deux premières lettres du second domino.

Dans la suite, on se base sur :

1. le lexique suivant, de 13 mots (ce lexique ne tient pas compte des accents, et on accepte les noms propres ainsi que les verbes conjugués) :

TELE	TETE	MELE	MERE	CURE	CUBE	SEVE
SETE	LESE	LEVE	MISE	MITE	MILE	

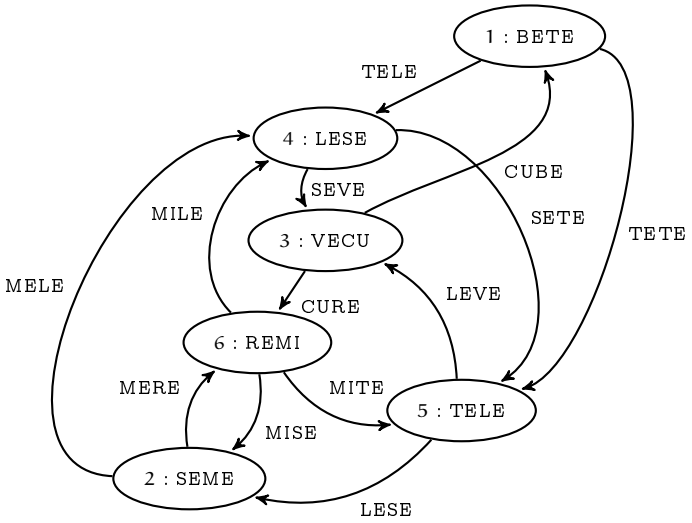
2. les six dominos suivants, numérotés de 1 à 6 :

1	2	3	4	5	6
BETE	SEME	VECU	LESE	TELE	REMI

Par exemple, le domino SEME peut être mis après le domino REMI, puisque MISE est un mot du lexique.

On représente un tel jeu par un graphe : les six dominos correspondent aux six nœuds du graphe, et un arc relie deux nœuds s'il est possible de juxtaposer le nœud origine au nœud extrémité.

Exemple Le graphe du jeu se présente comme suit :



L'étiquette placée sur chaque arc correspond au mot du lexique formé par la concaténation des mots placés aux extrémités de l'arc, privé des deux caractères de début et de fin. Le but du jeu consiste à créer un chemin utilisant les six dominos. Une solution possible est :

BETE TELE VECU REMI SEME LESE.

Les cinq mots issus de cette juxtaposition sont donc :

TETE LEVE CURE MISE MELE.

D'une manière plus générale, l'objectif de cet exercice est d'instancier l'un des patrons du tableau 5.1 page 136 afin qu'il énumère, à partir d'un graphe donné, tous les chemins qui passent une fois et une seule par chacun des nœuds, c'est-à-dire tous les chemins *hamiltoniens* (voir définition 11, page 25).

Dans l'exemple précédent, les deux mots CUVE et MIRE seraient ignorés même s'ils appartenaient au lexique, car issus d'une boucle sur respectivement les mots VECU et REMI. En effet, les boucles ne présentent pas d'intérêt dans le cas de chemins hamiltoniens.

Question 1. Cette question porte sur un traitement manuel du problème. Pour réduire l'espace de recherche, on impose de commencer par le domino numéro 1 (BETE), puis de prendre le domino numéro 5 (TELE). Trouver toutes les solutions commençant par ces deux dominos. Peut-on trouver d'autres solutions (voire toutes) à partir de celles-ci ? 55 - Q 1

Dans la suite, on suppose disponible l'ensemble M des nœuds du graphe (numérotés de 1 à n) ainsi que la fonction $Succ(s)$ (voir définition 5, page 24), qui est telle que $Succ \in 1\,..\,n \rightarrow \mathbb{P}(1\,..\,n)$ et qui, pour chaque nœud s, fournit l'ensemble Succ(s) des successeurs de s (c'est-à-dire l'ensemble des mots juxtaposables à s). Ainsi, dans l'exemple de l'énoncé, on a :

s	1	2	3	4	5	6
$Succ(s)$	{4,5}	{4,6}	{1,6}	{3,5}	{2,3}	{2,4,5}

Question 2. Dans le cadre de la recherche de tous les chemins hamiltoniens, proposer une structure d'énumération X, fournir ses propriétés et choisir un patron à instancier. 55 - Q 2

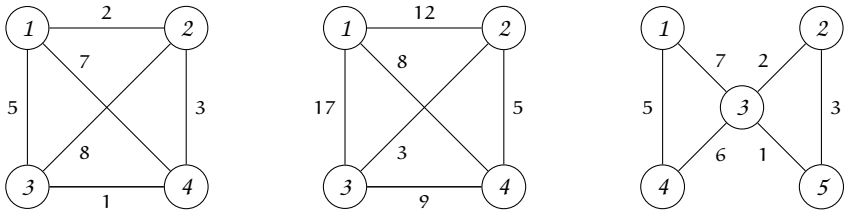
Question 3. Pour l'exemple ci-dessus, fournir l'arbre de récursion obtenu à partir du nœud 1 (le domino BETE) comme racine. 55 - Q 3

Question 4. Fournir une instance du patron *ToutesSolutions* qui trouve tous les chemins hamiltoniens dans un graphe de n nœuds. 55 - Q 4

Exercice 56. Le voyageur de commerce

Exemple classique d'application du principe de la recherche d'une solution optimale par essais successifs, cet exercice est d'un abord simple. L'existence d'algorithmes plus efficaces (comme l'algorithme de Held-Karp ou une approche de type PSEP – voir exercice 70, page 191) réduit cependant son intérêt pratique.

Un voyageur de commerce doit visiter n villes constituant les sommets d'un graphe non orienté connexe dont les arêtes sont étiquetées par la distance entre les villes qu'elles rejoignent. Le voyageur part d'une certaine ville et doit, si possible, y revenir après avoir visité toutes les autres villes une fois et une seule. La question que doit résoudre le programme à construire est : quel parcours doit-il réaliser pour effectuer le trajet le plus court possible ? Formellement, partant d'un graphe non orienté $G = (N, V)$, valué sur $\mathbb{R}_+^*$ par la fonction D (pour distance), il s'agit de trouver un cycle hamiltonien le plus court possible.



(a) (b) (c)

Fig. 5.15 – *Exemples de réseaux de villes*

Dans le graphe (a) de la figure 5.15, le trajet $\langle 1, 2, 4, 3, 1\rangle$ est le plus court, avec une longueur de 11, tandis que dans le graphe (b), 32 est la longueur du meilleur trajet, valeur atteinte pour le parcours $\langle 1, 2, 3, 4, 1\rangle$. En revanche, avec le graphe (c) (qui ne possède pas de cycle hamiltonien), le problème posé n'a pas de solution.

Remarque On peut, sans perte de généralité, choisir n'importe quel nœud du graphe comme ville de départ.

56 - Q 1 **Question 1.** En supposant que le graphe considéré est complet (c'est-à-dire qu'il existe une arête entre tout couple de villes), combien de cycles hamiltoniens existe-t-il depuis une ville donnée ?

56 - Q 2 **Question 2.** Dans le graphe (b) de la figure 5.15, le trajet $\langle 1, 2, 3, 2, 4, 1\rangle$ est bien un cycle mais il n'est pas hamiltonien (il passe deux fois par le sommet 2). Sa longueur, 31, est inférieure au meilleur trajet hamiltonien trouvé ($\langle 1, 2, 3, 4, 1\rangle$). Montrer que si, dans le graphe, l'inégalité triangulaire n'est pas respectée, il peut exister des cycles non hamiltoniens (c'est-à-dire passant plus d'une fois par un sommet) meilleurs qu'un cycle hamiltonien optimal.

Question 3. Proposer une structure d'énumération X pour résoudre le problème considéré et fournir ses propriétés. En déduire le patron d'algorithme qui convient et fournir son code générique s'il n'est pas disponible dans l'introduction. 56 - Q 3

Question 4. Fournir l'arbre de récursion pour le graphe (a) de la figure 5.15, page 156, en partant du nœud 1. Un élagage basé sur la longueur des chaînes peut facilement être appliqué. Préciser son mode opératoire et déterminer les conséquences attendues sur l'arbre de récursion. 56 - Q 4

Question 5. Dans cette question, on suppose disponibles le tableau D des longueurs des arêtes du graphe, ainsi que la fonction $Succ(s)$ qui, pour chaque nœud du graphe G, fournit l'ensemble des nœuds successeurs de s. Instancier le patron fourni en réponse à la question ?? pour produire un algorithme qui détermine dans quel ordre le voyageur doit visiter les villes afin de minimiser la longueur totale du chemin parcouru. 56 - Q 5

Exercice 57. Isomorphisme de graphes

Cet exercice porte sur des graphes orientés. L'algorithme considère deux graphes entre lesquels on recherche un isomorphisme. Il opère sur deux niveaux : il recherche une bijection des nœuds, qui sous-tend une bijection des arcs. Cette caractéristique est à l'origine d'un élagage efficace en général.

Soit $G_1 = (N_1, V_1)$ et $G_2 = (N_2, V_2)$ deux graphes orientés tels que $\text{card}(N_1) = \text{card}(N_2)$ et $\text{card}(V_1) = \text{card}(V_2)$, et B une bijection entre N_1 et N_2. G_1 et G_2 sont *isomorphes* à travers B, si B induit une bijection entre V_1 et V_2 (autrement dit si l'application de la bijection permet de réécrire le graphe G_1 en utilisant le vocabulaire du graphe G_2), plus précisément si $B^{-1} \circ V_1 \circ B = V_2$ (soit encore si $V_1 \circ B = B \circ V_2$). Par exemple, soit $G = (N_G, V_G)$ et $H = (N_H, V_H)$ les deux graphes de la figure 5.16.

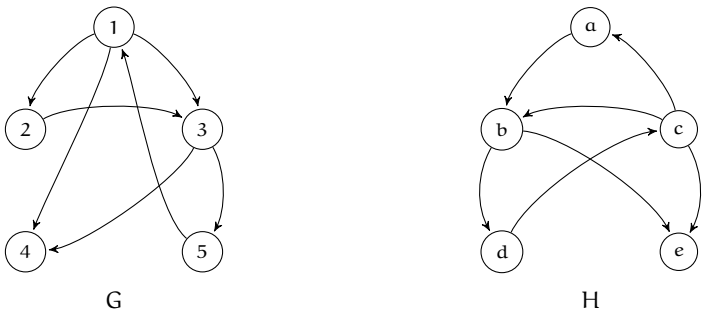


Fig. 5.16 – *Deux exemples de graphes*

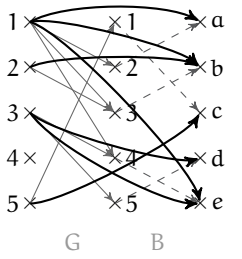
Ces deux graphes sont isomorphes à travers la bijection de sommets suivante :

$$B = \begin{array}{|c|c|c|c|c|}\hline 1 & 2 & 3 & 4 & 5 \\ \hline c & a & b & e & d \\ \hline \end{array}$$

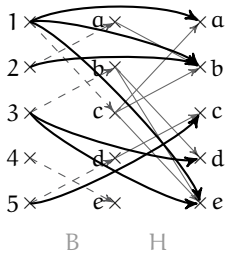
En effet, cette relation induit bien une bijection sur les arcs, que l'on peut représenter par le tableau suivant :

$(1,2)$	$(1,3)$	$(1,4)$	$(2,3)$	$(3,4)$	$(3,5)$	$(5,1)$
(c,a)	(c,b)	(c,e)	(a,b)	(b,e)	(b,d)	(d,c)

Dans les deux schémas ci-dessous, on utilise une représentation bipartite des relations. Le schéma de gauche ($V_G \circ B$) présente en gris les arcs V_G, en pointillés la relation B, et en noir la composition des deux relations. Les mêmes conventions sont utilisées pour le schéma de droite $B \circ V_H$.



$V_G \circ B$



$B \circ V_H$

On constate que les deux compositions de relations $V_G \circ B$ et $V_G \circ B$ sont identiques : les graphes G et H sont isomorphes.

Dans les exemples de la figure 5.16, page 157, le nœud 1 du graphe G a pour demi-degré extérieur 3 et pour demi-degré intérieur 1. Il en est de même du sommet c du graphe H.

57 - Q 1 **Question 1.** Donner la table des demi-degrés pour les graphes G et H de la figure 5.16.

On s'intéresse à l'écriture d'un algorithme à qui l'on fournit un couple de graphes (G_1, G_2) ayant le même nombre n de sommets et le même nombre d'arcs, et qui délivre le nombre d'isomorphismes possibles entre G_1 et G_2.

57 - Q 2 **Question 2.** On considère les deux graphes G et H de la figure 5.16, page 157. Énumérer toutes les bijections entre N_G et N_H qui se limitent à préserver l'arité des sommets. Obtient-on systématiquement des isomorphismes entre G et H ? En déduire une stratégie d'élagage.

57 - Q 3 **Question 3.** Dans le cadre du dénombrement des isomorphismes, proposer une structure d'énumération X et fournir ses propriétés.

57 - Q 4 **Question 4.** Fournir l'arbre de récursion élagué pour l'exemple de la figure 5.16, page 157.

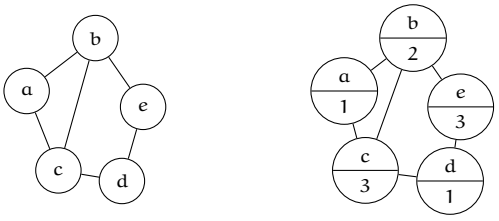
57 - Q 5 **Question 5.** Fournir un algorithme à essais successifs pour résoudre cet exercice. On s'intéressera au traitement à effectuer pour s'assurer que le vecteur d'énumération construit corresponde à un isomorphisme entre G_1 et G_2. La fonction $d^+(s)$ (resp. $d^-(s)$) délivre le demi-degré extérieur (resp. intérieur) du sommet s (voir définition 3 page 24).

Exercice 58. Coloriage d'un graphe

Cet exercice aborde le problème du coloriage de graphes. On se limite à concevoir un algorithme qui détermine si le graphe peut-être colorié avec m *couleurs.*

On considère un graphe non orienté connexe $G = (N, V)$ dans lequel N est l'ensemble des sommets et V l'ensemble des arêtes. Un tel graphe est dit *peint* si une couleur est attribuée à chaque sommet, c'est-à-dire s'il existe une fonction totale X de l'ensemble des sommets N vers un ensemble C de m couleurs. Un graphe peint est dit *colorié* par X si, et seulement si, deux sommets de la même couleur *ne sont pas* reliés par une arête.

Par exemple, avec l'ensemble des trois « couleurs » $C = \{1, 2, 3\}$, on peut colorier le graphe du schéma (a) ci-dessous comme le montre le schéma (b) :



(a) (b)

La fonction de coloriage peut être représentée par le vecteur d'énumération $X[1..5] = [1, 2, 3, 1, 3]$ dont le premier élément est la couleur du premier nœud a, le second, la couleur du second nœud b, et ainsi de suite pour les cinq nœuds.

Question 1. Pour le graphe (a) ci-dessus, proposer un coloriage X' (différent de X) obtenu par une permutation P des couleurs ($X' = P \circ X$). Proposer un second coloriage X'' qui ne soit pas obtenu par une permutation des couleurs. 58 - Q 1

Question 2. Soit $G = (N, V)$ un graphe et Z une peinture de G (c'est-à-dire une fonction totale de N vers un ensemble de couleurs). Fournir une expression ensembliste (exprimée sur la base des relations Z et V) de la condition qui exprime que Z est un *coloriage* de G. 58 - Q 2

Question 3. Définir le vecteur d'énumération X et ses propriétés pour le problème de la recherche d'un coloriage. En déduire le patron qui s'applique si l'on décide de s'arrêter au premier coloriage trouvé (voir tableau 5.1, page 136). 58 - Q 3

Question 4. Fournir l'arbre de récursion élagué parcouru pour la recherche du premier coloriage du graphe (a) ci-dessus. 58 - Q 4

Question 5. On ne connaît pas de propriété nécessaire et suffisante de coloriage d'un graphe G par les m couleurs de l'ensemble C. En conséquence, pour déterminer si un graphe donné G quelconque peut ou non être colorié à l'aide d'un ensemble de m couleurs données, il est nécessaire de renforcer l'objectif à atteindre en recherchant *explicitement* un coloriage. Fournir le code de l'algorithme et un exemple d'appel (on suppose disponible la fonction $Succ(s)$ – voir définition 5 page 24 –, qui délivre les voisins du sommet s). 58 - Q 5

Exercice 59. Élections présidentielles à l'américaine ○ ●●

Dans cet exercice, on s'intéresse aux situations où le scrutin présidentiel des États-Unis ne permet pas l'élection d'un candidat pour cause d'égalité de voix. Les dernières questions portent sur un élagage rendu possible par l'exploitation d'une symétrie.

Les élections présidentielles américaines se déroulent approximativement de la façon suivante : dans chacun des 50 états, les électeurs votent pour l'un des deux candidats à la présidence, démocrate ou républicain. Si c'est le candidat républicain qui dépasse l'autre en nombre de voix dans cet état, l'état enverra à Washington des « grands électeurs », tous républicains. Si c'est le candidat démocrate qui gagne dans cet état, l'état enverra à Washington le même nombre de grands électeurs, tous démocrates [4]. Le nombre de grands électeurs dépend de la population de l'état.

Pour l'étape finale du scrutin, les grands électeurs se retrouvent à Washington et votent conformément à leur étiquette. Le président élu est celui qui obtient le plus de voix de grands électeurs. En pratique, sur un total de 538 grands électeurs, la Californie en a 54, le Texas en compte 32, l'état de New York 33, la Floride 25, la Pennsylvanie 23, l'Illinois 22, etc.

Dans la suite, les états sont codés sur l'intervalle $1\,..\,n$ ($n \geqslant 1$) ; GE est le tableau qui associe au code de chaque état le nombre de grands électeurs attribués à cet état, et T est le total de grands électeurs sur tout le pays.

Le résultat d'une élection peut être représenté par un vecteur caractéristique défini sur l'intervalle $1\,..\,n$ et à valeur sur $0\,..\,1$. La valeur 0 (resp. 1) signifie par convention que les démocrates (resp. les républicains) sont les vainqueurs dans l'état considéré. La solution à notre problème consiste à passer en revue l'ensemble des parties de $1..n$ afin de déterminer s'il existe des cas d'égalité.

59 - Q 1 **Question 1.** Définir le vecteur d'énumération X et fournir ses propriétés. En déduire le patron qui s'applique (voir tableau 5.1, page 136).

59 - Q 2 **Question 2.** Décrire les constituants de la procédure *Élections* qui énumère toutes les configurations où les deux candidats se retrouvent avec le même nombre de voix. Quel est le code obtenu ? Fournir un exemple d'appel.

59 - Q 3 **Question 3.** Montrer que le nombre de telles configurations est pair.

59 - Q 4 **Question 4.** Proposer une modification de l'algorithme qui ne produit que l'une des deux configurations de l'appariement.

59 - Q 5 **Question 5.** Proposer une modification très simple de la constitution américaine pour que l'élection soit toujours effective, c'est-à-dire que les deux candidats ne puissent pas avoir le même nombre de voix de grands électeurs.

Exercice 60. Crypto-arithmétique ○ ●

4. On suppose qu'à ce stade du scrutin il n'y a jamais égalité des voix.

La principale originalité de cet exercice réside dans le fait qu'il s'agit de produire des injections totales *entre deux ensembles. Ce sont ces injections qui constituent les solutions potentielles. L'une des difficultés concerne le calcul de complexité. Il n'y a pas lieu de rechercher une complexité asymptotique, puisque le paramètre choisi varie sur un intervalle fini. Cependant, les calculs sous-jacents se révèlent assez difficiles.*

Soit Σ l'alphabet latin de 26 lettres : $\Sigma = \{\mathcal{A}, \mathcal{B}, \ldots, \mathcal{Z}\}$. Soit $L \subset \Sigma$ un ensemble de n lettres de l'alphabet ($n \leqslant 10$) et $+$ une addition formelle, exprimée avec ces lettres comme par exemple :

$$\mathcal{NEUF} + \mathcal{UN} + \mathcal{UN} = \mathcal{ONZE}$$

que l'on peut aussi écrire :

$$\begin{array}{cccccc} & \mathcal{N} & \mathcal{E} & \mathcal{U} & \mathcal{F} \\ + & & & \mathcal{U} & \mathcal{N} \\ + & & & \mathcal{U} & \mathcal{N} \\ \hline & \mathcal{O} & \mathcal{N} & \mathcal{Z} & \mathcal{E} \end{array}$$

Le but de l'exercice est de découvrir toutes les injections totales de L vers l'ensemble des dix chiffres décimaux, de sorte que la substitution de chaque lettre par le chiffre qui lui correspond fournit une opération arithmétique correcte en base 10.

L'exemple ci-dessus possède une solution que l'on peut représenter par la bijection partielle suivante :

$$\{\mathcal{E} \mapsto 9, \mathcal{F} \mapsto 7, \mathcal{N} \mapsto 1, \mathcal{O} \mapsto 2, \mathcal{U} \mapsto 8, \mathcal{Z} \mapsto 4\}$$

de $\{\mathcal{E}, \mathcal{F}, \mathcal{N}, \mathcal{O}, \mathcal{U}, \mathcal{Z}\}$ dans $\{1, 2, 4, 7, 8, 9\}$, puisque :

$$1987 + 81 + 81 = 2149.$$

En revanche, la solution correspondant à :

$$1988 + 81 + 81 = 2150,$$

correcte sur le plan arithmétique, n'est cependant pas acceptable. En effet, elle provient d'une fonction qui n'est pas injective, puisque les lettres $\mathcal{F}$ et $\mathcal{U}$ sont toutes deux en relation avec 8. Quant à l'addition :

$$1986 + 82 + 82,$$

elle est issue d'une relation qui n'est pas fonctionnelle : $\{\ldots, \mathcal{N} \mapsto 1, \mathcal{N} \mapsto 2, \mathcal{O} \mapsto 2, \ldots\}$.

On supposera qu'il existe une fonction *CalculExact* qui, partant d'une injection entre lettres et chiffres décimaux, et d'une représentation de l'opération formelle, rend **vrai** si, effectuée à travers l'injection, l'opération est arithmétiquement correcte et **faux** sinon.

Question 1. Donner le principe d'un algorithme permettant de trouver toutes les solutions à tout problème de crypto-arithmétique. Quel est le patron approprié (voir tableau 5.1, page 136) ? Dans un premier temps, on ne cherche pas à effectuer d'élagages. 60 - Q 1

60 - Q 2 **Question** 2. Définir les instances des différents composants du patron utilisé (*Satisfaisant*, *SolutionTrouvée*, *Faire* et *Défaire*). En déduire le code de la procédure *CryptoArith*. Fournir un exemple d'appel.

60 - Q 3 **Question** 3. Pour un problème particulier comme celui donné ci-dessus, comment améliorer la complexité temporelle en introduisant des élagages ? Peut-on découvrir des conditions générales d'élagage ?

Exercice 61. Carrés latins

L'originalité de cet exercice réside dans le fait que, bien que le vecteur d'énumération représente une fonction injective, des restrictions de celle-ci possèdent une propriété plus forte, qu'il est intéressant d'exploiter.

Un *carré latin d'ordre* n ($n \geqslant 1$) est un tableau carré dans lequel les cellules contiennent les n éléments d'un ensemble S, qui sont disposés de telle manière qu'ils apparaissent une et une seule fois dans chaque ligne et dans chaque colonne. Chacune des lignes et des colonnes est donc constituée par une permutation des n éléments.

Par exemple, pour $n = 6$ et $S = \{1, 2, 3, 4, 5, 6\}$, on a (parmi 812 851 200 solutions) les trois carrés latins suivants :

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 6 & 5 \\ 4 & 6 & 5 & 2 & 3 & 1 \\ 3 & 4 & 6 & 1 & 5 & 2 \\ 2 & 5 & 1 & 3 & 4 & 6 \\ 5 & 3 & 2 & 6 & 1 & 4 \\ 6 & 1 & 4 & 5 & 2 & 3 \end{bmatrix} \quad \begin{bmatrix} 3 & 6 & 2 & 1 & 4 & 5 \\ 1 & 3 & 4 & 6 & 5 & 2 \\ 6 & 4 & 3 & 5 & 2 & 1 \\ 2 & 1 & 5 & 3 & 6 & 4 \\ 4 & 5 & 1 & 2 & 3 & 6 \\ 5 & 2 & 6 & 4 & 1 & 3 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 5 & 3 & 6 & 4 \\ 2 & 6 & 1 & 4 & 3 & 5 \\ 5 & 4 & 3 & 2 & 1 & 6 \\ 3 & 5 & 4 & 6 & 2 & 1 \\ 6 & 1 & 2 & 5 & 4 & 3 \\ 4 & 3 & 6 & 1 & 5 & 2 \end{bmatrix}$$

Le second carré présente une particularité : chacune des deux diagonales est entièrement composée d'éléments identiques. Un tel carré latin est dit *antidiagonal*.

Le troisième carré présente une particularité différente : les éléments de S apparaissent dans le même ordre sur la première ligne et sur la première colonne. Un tel carré latin est dit *normalisé*. Il existe 96 773 760 carrés latins normalisés et 76 640 antidiagonaux d'ordre 6. Dans la suite, on limite l'étude au cas où $S = 1 \,..\, n$.

61 - Q 1 **Question** 1. Sachant que l'on recherche tous les carrés latins pour un n donné, définir la structure d'énumération X et fournir ses propriétés. En déduire le patron qui s'applique ici. Que peut-on en conclure sur dumasl'ensemble que va parcourir la variable j de la boucle ?

61 - Q 2 **Question** 2. Fournir une portion de l'arbre de récursion pour un carré latin d'ordre 3.

61 - Q 3 **Question** 3. Fournir le code de la procédure *CarréLatin*, ainsi qu'un exemple d'appel.

61 - Q 4 **Question** 4. Dans la version de la question précédente de la procédure *CarréLatin*, on exploite le fait que l'ensemble parcouru par la variable j est un intervalle à trous, sous-ensemble fini de $\mathbb{N}$ (et non un intervalle complet). Cette caractéristique n'existe pas dans

la plupart des langages de programmation classiques. Dans la procédure *PartEns3* de l'exemple introductif (page 120), nous avons vu comment contourner ce problème en renforçant l'invariant par une structure de données (SomCour pour l'exemple en question) redondante par rapport à la structure X. En s'inspirant de cet exemple, aménager la procédure *CarréLatin* de façon à disposer d'une version efficace *CarréLatin2* (on veillera à éviter des recherches séquentielles dans X).

Question 5. Démontrer qu'à l'exception du carré latin d'ordre 1, il n'existe pas de carré latin antidiagonal d'ordre impair, puis montrer comment on peut aménager la procédure *CarréLatin* afin d'obtenir la procédure *CarréLatinAntidiagonal*, cette dernière permettant d'écrire tous les carrés antidiagonaux à un ordre n quelconque. 61 - Q 5

Question 6. Donner le principe d'une procédure permettant d'écrire tous les carrés latins normalisés à un ordre n donné. 61 - Q 6

Exercice 62. Le jeu de sudoku

8

Cet exercice porte sur le jeu du sudoku. Dans la version finale (question 4), il tire son originalité du fait que la structure d'énumération n'est pas vide au démarrage de l'algorithme, puisqu'elle doit contenir les chiffres déjà placés sur la grille.

Ce jeu est une extension du jeu du carré latin. Il est conseillé de traiter l'exercice s'y rapportant (voir page 162) avant d'aborder celui-ci.

Le but de ce jeu est de remplir de chiffres un carré de neuf cases de côté, subdivisé en autant de carrés identiques de trois cases de côté, appelés régions, de façon à ce que chaque ligne, chaque colonne et chaque région contienne une fois et une seule les chiffres de 1 à 9. Au début du jeu, un certain nombre de chiffres sont déjà en place (ils sont appelés les *dévoilés*). En général, la grille de départ représente un sudoku minimal[5]. Voici un exemple de grille sudoku (la grille à compléter à gauche et sa solution à droite) :

	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Dans une première étape, on considère les grilles *sans* dévoilés. Il s'agit donc de produire toutes les grilles de sudoku possibles.

5. Une grille comportant des dévoilés est dite minimale, si d'une part la solution existe et est unique, et si d'autre part la suppression d'un dévoilé quelconque fait perdre l'unicité.

62 - Q 1 **Question** 1. Proposer une structure d'énumération X. Définir ses propriétés. Que peut-on en conclure quant à l'ensemble que va parcourir la variable j de la boucle ? Parmi la liste de patrons du tableau 5.1, page 136, quel est celui qui est approprié au cas considéré ?

62 - Q 2 **Question** 2. Fournir le code de la procédure *Sudoku1* (sans dévoilés), ainsi qu'un exemple d'appel. Il existe environ $7\ 10^{21}$ solutions. Estimer le temps de calcul de ce programme sur un processeur actuel typique.

62 - Q 3 **Question** 3. Dans la version de la question précédente de la procédure *Sudoku1*, on exploite le fait que le langage de programmation utilisé permet de parcourir un intervalle à trous, sous-ensemble fini de $\mathbb{N}$ (et non un intervalle complet). Cette caractéristique n'existe pas dans la plupart des langages classiques. Dans la procédure *PartEns3* de l'exemple introductif (page 120), nous avons vu comment contourner ce problème en renforçant l'invariant par une structure de données (SomCour pour l'exemple en question) redondante par rapport à la structure X. En s'inspirant de cet exemple, aménager la procédure *Sudoku1* de façon à disposer d'une version efficace *Sudoku2* (on veillera à éviter des recherches séquentielles dans X).

On considère à présent les grilles avec dévoilés. On ne s'intéresse qu'à la variante de *Sudoku2* (dans laquelle on utilise une structure de données auxiliaire).

62 - Q 4 **Question** 4. Aménager la procédure *Sudoku2* de façon à traiter les grilles avec dévoilés.

62 - Q 5 **Question** 5. Comment adapter le résultat de la question précédente afin de proposer une opération qui vérifie qu'une grille est bien minimale.

Exercice 63. Sept à onze

Cet exercice est un exemple particulièrement intéressant dont la solution naïve présente une complexité temporelle désastreuse, alors qu'une transformation ingénieuse, basée sur une décomposition en facteurs premiers, fournit une solution d'une efficacité acceptable.

Le magasin d'alimentation de votre quartier est ouvert de 7 h à 23 h, d'où son nom : le « 7 à 11 ». Vous y achetez un ensemble de quatre articles $\{a_1, a_2, a_3, a_4\}$. À la caisse, la note qui vous est présentée s'élève à $7,11€$. Une note de $7,11€$ au « 7 à 11 » ! Vous faites remarquer la coïncidence au caissier en lui demandant comment il est arrivé à ce montant : « Eh bien, j'ai tout simplement multiplié le prix des quatre articles ». Vous lui expliquez calmement que le montant total doit être calculé en *additionnant* le prix des articles et non en les multipliant. Il vous répond : « ça n'a aucune importance, la facture serait encore de $7,11€$ ». Il a raison. Le problème que l'on se pose est de déterminer le prix de chaque article, sachant que la première solution trouvée nous conviendra.

Question 1. Exprimée en centimes d'euros, quelle est la somme (resp. le produit) des quatre prix ? 63 - Q 1

Question 2. En supposant que le prix des articles est supérieur ou égal à deux centimes, et en se fondant uniquement sur les propriétés de la somme, sur quel intervalle les prix, exprimés en centimes, peuvent-ils varier ? 63 - Q 2

Question 3. Afin de mettre en œuvre un algorithme pour résoudre ce problème, proposer un vecteur d'énumération X et définir ses propriétés. Quel type de patron entraîne ce choix (voir tableau 5.1, page 136) sachant que l'on se limite à la recherche de la première solution ? 63 - Q 3

Question 4. En déduire la procédure *SeptAOnze1* qui affiche la première solution trouvée. Optimiser la solution en renforçant l'invariant de récursivité. 63 - Q 4

Dans l'objectif de la recherche d'une meilleure efficacité pour un algorithme, la démarche qui consiste à « prétraiter » les données se révèle souvent féconde. Alors que dans la question précédente nous avons examiné tous les quadruplets possibles de prix, nous pouvons décider de ne prendre en considération que les quadruplets dont le produit vaut 711 000 000. Répertorier de tels quadruplets passe tout d'abord par la décomposition de 711 000 000 en un produit de facteurs premiers[6] (c'est le prétraitement). Nous calculons facilement que $711\,000\,000 = 2^6 \cdot 3^2 \cdot 5^6 \cdot 79$. Si les valeurs de l'intervalle $1 \,..\, 4$ codent chacun des quatre articles, le problème se ramène à trouver une partition particulière, à quatre éléments, du multiensemble $[\![2, 2, 2, 2, 2, 2, 3, 3, 5, 5, 5, 5, 5, 5, 79]\!]$. Ainsi, par exemple, la partition $[\![[\![2, 2, 5, 5]\!], [\![2, 2, 5, 5]\!], [\![2, 3, 79]\!], [\![2, 3, 5, 5]\!]]\!]$ correspond à quatre articles aux prix respectifs de 100, 100, 474 et 150. Par construction, le produit de ces quatre prix est de 711 000 000, vérification qui devient donc inutile. Reste à trouver la (l'une des) partition(s) dont la somme vaut 711.

Question 5. Déduire de la remarque précédente une nouvelle structure d'énumération X et en fournir les propriétés. En déduire le patron qui s'applique (voir tableau 5.1, page 136) et fournir son code s'il n'apparaît pas dans l'introduction. 63 - Q 5

Question 6. En déduire une nouvelle version de la procédure *SeptAOnze* qui affiche la première solution trouvée. Que peut-on dire de la complexité de cette solution ? 63 - Q 6

Exercice 64. Décomposition d'un nombre entier ○ ⁝

Dans cet exercice, partant de la solution « force brute », plusieurs optimisations et élagages sont étudiés. C'est le principal intérêt de cet exercice, pour lequel il existe des solutions plus efficaces que celles fondées sur le principe des essais successifs.

6. La décomposition est unique à condition que 1 soit exclu. C'est pour cette raison que nous imposons des prix supérieurs ou égaux à deux centimes d'euro.

Étant donné un nombre entier n ($n \geqslant 1$), on appelle « décomposition » de n, l'ensemble des ensembles d'entiers naturels non nuls dont la somme vaut n. Par exemple, la décomposition de $n = 6$ est l'ensemble : $\{\{1,2,3\}, \{1,5\}, \{2,4\}, \{6\}\}$[7]. L'objectif de l'exercice est de fournir différentes variantes de la procédure qui produit successivement tous les éléments de la décomposition d'un entier n positif donné.

64 - Q 1 **Question 1.** Donner la décomposition de $n = 10$.

64 - Q 2 **Question 2.** Afin de résoudre ce problème par une démarche de type « essais successifs », définir une structure d'énumération X et en fournir les propriétés. En déduire le patron qui lui correspond.

64 - Q 3 **Question 3.** Dans cette question, on s'intéresse à la solution de type « force brute ». Fournir la procédure *DécompEntier1*, instance du patron choisi à la question précédente, en précisant comment s'instancie la fonction *SolutionTrouvée*.

64 - Q 4 **Question 4.** Un premier élagage est possible si l'on constate qu'il est inutile de poursuivre l'exploration d'une branche qui a déjà dépassé la valeur n. Quelles sont les modifications à apporter à la procédure *DécompEntier1* pour mettre en œuvre cette optimisation ?

64 - Q 5 **Question 5.** Pour l'instant, le calcul de la somme des nombres représentés dans le vecteur X s'effectue systématiquement lors de l'évaluation de la condition correspondant à la fonction générique *SolutionTrouvée*. Il est clair que ceci conduit à refaire plusieurs fois les mêmes calculs. L'optimisation envisagée ici consiste à éliminer ces calculs redondants. Effectuer cette amélioration en adaptant la technique du renforcement d'invariant de récursivité appliqué dans l'exemple introductif de ce chapitre (voir page 117).

64 - Q 6 **Question 6.** Un dernier élagage est possible. Il se fonde sur le fait que l'on peut arrêter la recherche dès que la somme exacte a été trouvée, que i soit ou non égal à n. Comment doit-on modifier la version précédente pour parvenir à cette version ?

Exercice 65. Madame Dumas et les trois mousquetaires ○ •

Cet exercice est un exemple typique de production de permutations sous contraintes. On tente ci-dessous, dans la mesure du possible et pour des raisons de généricité, de dissocier l'aspect lié à la production des permutations de celui de la prise en compte des contraintes.

Madame « Dumas père » organise un dîner en l'honneur de d'Artagnan et des trois mousquetaires. Les places à table sont numérotées de 1 à 5. Madame Dumas sait que :

1. Porthos préfère être à la place numéro 1,
2. Athos préfère être séparé de d'Artagnan,

7. Il existe une notion proche, celle de *partition* d'un entier, dans laquelle on recherche un ensemble de *sacs* (un entier peut apparaître plusieurs fois dans une somme). Par exemple, la partition de 6 est : $\{[\![6]\!], [\![5,1]\!], [\![4,2]\!], [\![4,1,1]\!], [\![3,3]\!], [\![3,2,1]\!], [\![3,1,1,1]\!], [\![2,2,2]\!], [\![2,2,1,1]\!], [\![2,1,1,1,1]\!], [\![1,1,1,1,1,1]\!]\}$. La combinatoire s'accroît par rapport à la décomposition puisque toute décomposition est une partition (mais toute partition n'est pas une décomposition).

3. Aramis préfère être séparé d'Athos,
4. Porthos préfère être séparé d'Athos.

Pour ce qui la concerne, Madame Dumas souhaite être séparée de d'Artagnan (préférence numéro 5). Pouvez-vous aider Madame Dumas à établir son plan de table (c'est-à-dire à répondre à la question « qui est où ? »), si possible dans le respect des préférences de chacun ? Ci-dessous, on considère que chaque participant est codé par un nombre de l'intervalle $1 \,..\, 5$, selon l'ordre alphabétique (1 : *Aramis*, 2 : *Athos*, 3 : *d'Artagnan*, 4 : *Dumas*, 5 : *Porthos*).

Dans la suite, on pourra supposer que la préférence 1 est traitée de manière *ad hoc* plutôt que par l'intermédiaire de l'algorithme générique.

Question 1. Définir le vecteur d'énumération X et fournir ses propriétés. Parmi ceux proposés dans le tableau 5.1, page 136, quel est le patron qui s'applique, sachant que l'on souhaite obtenir toutes les solutions possibles ? 65 - Q 1

Question 2. Fournir l'arbre de récursion obtenu en choisissant comme racine *Porthos*. 65 - Q 2

Question 3. Proposer une solution pour traiter les contraintes du type « $\mathcal{Y}$ préfère être séparé de $\mathcal{Z}$ ». 65 - Q 3

Question 4. Fournir l'algorithme qui affiche tous les plans de table satisfaisant les préférences exprimées. Donner un exemple d'appel. 65 - Q 4

Exercice 66. Mini Master Mind

Le Master Mind est un jeu qui a connu la célébrité dans les années 70. Deux joueurs s'affrontent, l'un passif, le codeur, propose un code, que le second, le décodeur, doit découvrir. Inspiré de ce jeu, cet exercice présente trois intérêts. D'abord, il pousse à une réflexion sur les propriétés des propositions faites par le décodeur. Une seconde solution étudie un élagage particulièrement efficace. Enfin, une troisième solution emprunte une voie prometteuse totalement différente.

Il s'agit d'un jeu à deux joueurs, le *codeur* et le *décodeur*. Le premier se fixe une permutation de n couleurs (ici $n = 5$) `blanc`, `noir`, `orange`, `rouge` et `vert`, codées respectivement `B`, `N`, `O`, `R`, `V`, que le second tente de découvrir. Pour ce faire, le décodeur propose une liste des cinq couleurs différentes et en réponse, le codeur l'informe du nombre de couleurs correctement placées. Si c'est le cas pour les cinq couleurs, la partie est terminée. Sinon, le décodeur effectue une autre proposition qui est évaluée à son tour. Le décodeur doit découvrir le code en effectuant le moins possible de propositions. Il s'aide pour cela des informations qui ont été fournies en réponse à ses précédentes propositions. Dans cet exercice, le programme à construire joue le rôle du décodeur.

La figure 5.17 montre l'historique du déroulement d'une partie de Mini Master Mind à partir du codage `[V,R,N,B,O]`. La partie se termine au bout de sept propositions.

N° Prop.	Propositions	Évaluation
1	[B,N,O,R,V]	0
2	[N,B,R,V,O]	1
3	[N,O,V,B,R]	1
4	[O,V,R,B,N]	1
5	[R,B,V,O,N]	0
6	[V,O,R,N,B]	1
7	[V,R,N,B,O]	5

Fig. 5.17 – *Historique du déroulement d'une partie pour le codage initial [V,R,N,B,O]. Les lettres soulignées correspondent aux couleurs correctement placées. Cette information n'est pas disponible pour le décodeur.*

66 - Q 1 **Question** 1. Quelle condition nécessaire une proposition du décodeur doit-elle satisfaire pour être la permutation attendue par le codeur ? Suggestion : se poser la question suivante : « dans l'hypothèse où la proposition est la solution, comme s'évalue-t-elle par rapport aux différentes entrées de l'historique ? »

Dans la suite, la liste des n couleurs à découvrir est représentée par le tableau C, et l'historique H se présente sous la forme d'une structure de données (voir figure 5.17 pour un exemple) accessible à travers les trois opérations suivantes :

- **procédure** *InitHisto* qui vide H,
- **procédure** *InsérerHisto*(P, E) qui ajoute à l'historique H la permutation P évaluée à E,
- **fonction** *CompatAvecHisto*(P) **résultat** $\mathbb{B}$ qui délivre **vrai**, si et seulement si la permutation P satisfait la condition nécessaire qui fait l'objet de la première question.

66 - Q 2 **Question** 2. Définir le vecteur d'énumération X. Lequel des patrons du tableau 5.1, page 136, s'applique-t-il ? En déduire la procédure *PermutMasterMind1*(i) qui construit, dans le vecteur d'énumération X, la prochaine proposition du décodeur. Montrer comment utiliser cette procédure pour réaliser une partie de Mini Master Mind (on suppose qu'aucun des joueurs ne commet d'erreur).

La procédure *PermutMasterMind1*(i) ne réalise aucun élagage. Il est pourtant possible d'éviter de construire une occurrence complète du vecteur X en constatant que dès qu'une permutation en cours d'élaboration, confrontée à une entrée de l'historique, produit une réponse supérieure à la réponse présente dans l'historique, il est inutile de poursuivre la construction de X. C'est le rôle dévolu à l'opération **fonction** *PossibleHisto*(c, k) **résultat** $\mathbb{B}$, qui vérifie que le sous-vecteur $X[1 \,..\, k-1]$ allongé en k par la couleur c ne produit pas d'appariements en excès par rapport aux réponses enregistrées dans l'historique. Nous nous proposons d'étudier cette stratégie dans les deux questions suivantes.

66 - Q 3 **Question** 3. Dans cette question, on suppose que $n = 4$ et que le tableau C des couleurs est C = [B,O,R,V]. On suppose par ailleurs que :

a) la permutation à découvrir est [R,O,V,B],

b) au moment qui nous intéresse, l'historique se présente de la manière suivante :

N° Prop.	Prop.	Évaluation
1	[B,O,R,V]	1
2	[B,R,V,O]	1
3	[B,V,O,R]	0
4	[O,R,B,V]	0

c) les propositions sont produites par le décodeur dans l'ordre lexicographique et qu'il arrête sa recherche dès qu'une proposition est compatible avec l'historique (au sens de la question 1).

Donner l'arbre de récursion élagué qui aboutit à la cinquième proposition.

Question 4. Fournir la procédure *PermutMasterMind2* qui met en œuvre cette stratégie. 66 - Q 4

Que l'on utilise *PermutMasterMind1* ou *PermutMasterMind2*, ces deux procédures effectuent la recherche de la prochaine proposition en partant systématiquement de la même permutation initiale. Il est clair que cette méthode conduit à repasser sur des permutations qui ont déjà échoué. Ainsi, dans l'exemple de la question 3, les procédures par essais successifs démarrent la recherche avec le code [B,O,R,V]. Celle-ci est écartée – ainsi que toutes les permutations déjà présentes dans l'historique jusqu'à [O,R,B,V] – grâce à la condition *CompatAvecHisto*. Une meilleure solution *a priori* consisterait à partir de la permutation qui suit (selon l'ordre lexicographique) la dernière ayant échoué. Pour l'exemple, on partirait de la permutation qui succède à [O,R,B,V], soit [O,R,V,B].

Dans une première étape, il s'agit de construire un algorithme qui, à partir d'une permutation donnée, produit la suivante (toujours selon l'ordre lexicographique), en supposant (précondition) qu'il en existe une. Pour faciliter la lecture, les explications sont fournies avec un code constitué des neuf chiffres de 1 à 9 ; la fonction $S(p)$ délivre la permutation qui suit p. $S([9,8,7,6,5,4,3,2,1])$ n'existe pas (la précondition n'est pas satisfaite). En revanche, $S([1,2,3,4,5,6,7,8,9]) = [1,2,3,4,5,6,7,9,8]$. En effet, le nombre (ne comportant pas de chiffres en double) qui suit 123456789 est 123456798. De même $S([5,9,8,7,6,4,3,2,1]) = [6,1,2,3,4,5,7,8,9]$, ou encore $S([6,1,9,8,4,7,5,3,2]) = [6, 1, 9, 8, 5, 2, 3, 4, 7]$.

Comment parvenir à ces résultats ? Observons tout d'abord que le passage d'une permutation à la suivante peut se faire en ne procédant que par des échanges. Le cas $[5,9,8,7,6,4,3,1,2]$ est facile à traiter : on échange simplement les chiffres 2 et 1. Considérons le cas $[5,9,8,7,6,4,3,2,1]$. Ce code privé du premier élément 5, soit $[9,8,7,6,4,3,2,1]$, ne possède pas de successeur puisque la suite de chiffres est décroissante. $S([5,9,8,7,6,4,3,2,1])$ ne peut débuter par l'un des chiffres $1, 2, 3$ ou 4 : le nombre serait inférieur au code de départ. Il ne peut non plus débuter par 5 puisque les chiffres qui suivent 5 se présentent dans l'ordre décroissant. Il doit obligatoirement débuter par le chiffre présent dans $[9,8,7,6,4,3,2,1]$ immédiatement supérieur à 5, soit 6. Échangeons 5 et 6. On obtient $[6,9,8,7,5,4,3,2,1]$. Ce n'est pas le résultat attendu car il existe plusieurs codes qui s'intercalent entre $[5,9,8,7,6,4,3,2,1]$ et $[6,9,8,7,5,4,3,2,1]$, comme $[6,9,8,5,7,4,3,2,1]$. L'échange seul ne suffit donc pas. Quelle opération doit-on réaliser à la suite de l'échange ? Il suffit d'inverser le sous-tableau décroissant qui suit le premier chiffre. Pour l'exemple, on obtient $[6,1,2,3,4,5,7,8,9]$, qui est le résultat attendu.

Cette démarche s'applique aux cas plus complexes tels que $[6,1,9,8,4,7,5,3,2]$. Il suffit d'identifier le plus long code décroissant situé sur la gauche ($[7,5,3,2]$) et, comme ci-dessus, de rechercher le successeur de $[4,7,5,3,2]$. Celui-ci débute par 5. Échangeons 4 et 5 :

$[5, 7, 4, 3, 2]$. Inversons les quatre derniers chiffres : $[5, 2, 3, 4, 7]$. Le début du code, $[6, 1, 9, 8]$, ne jouant aucun rôle dans la démarche, le résultat recherché est $[6, 1, 9, 8, 5, 2, 3, 4, 7]$. Notons enfin que cette approche s'applique uniformément sur tous les codes dotés d'un successeur.

66 - Q 5 **Question** 5. Appliquer la démarche ci-dessus pour aboutir au code de la fonction $S(p)$ délivrant la permutation qui suit p dans l'ordre lexicographique. Cette fonction a comme précondition qu'il existe bien une permutation suivante. Évaluer sa complexité.

66 - Q 6 **Question** 6. Montrer comment utiliser cette procédure pour réaliser une partie de Mini Master Mind.

66 - Q 7 **Question** 7. Selon vous, quelle stratégie est la plus efficace ? Étayez votre réponse par quelques résultats expérimentaux sur un Mini Master Mind à 12 couleurs.

Exercice 67. Le jeu des mots casés

Cet exercice s'articule autour d'une grille de mots croisés. *Il s'agit d'un exemple typique du gain que l'on peut espérer obtenir par un élagage performant. La solution obtenue en appliquant un élagage élaboré permet d'obtenir un gain substantiel par rapport à la solution où seul un élagage grossier est appliqué.*

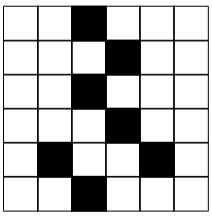
Le jeu connu sous le nom de « mots casés » est une variante des célèbres mots croisés dans laquelle on fournit au départ au joueur, d'une part une grille de mots croisés vide, de l'autre le sac des mots qui apparaîtront dans la grille résolue. Il s'agit alors pour le joueur de trouver une configuration (la première qui est découverte) où tous les mots sont placés sur la grille.

Exemple La figure 5.18, page 171, fournit un exemple avec, sur la gauche la grille vierge accompagnée du lexique de 24 mots, et sur la droite la grille complétée par une configuration possible. On note que *tous* les mots, y compris ceux réduits à une seule lettre, sont présents dans le lexique.

La première solution que nous nous proposons d'étudier se limite à un élagage grossier. Son principe consiste à remplir la grille horizontalement en ne prenant en compte, à chaque étape, que les mots dont la longueur est égale à celle de l'emplacement considéré (c'est l'élagage en question), puis, une fois la grille remplie, à vérifier que les mots qui n'ont pas été placés sont bien ceux que l'on retrouve verticalement sur la grille.

Dans la suite, on suppose que :

1. la grille traitée, Grille, comporte l lignes et c colonnes,
2. H est une constante qui représente le nombre d'emplacements horizontaux, ces emplacements étant numérotés de 1 à H (dans l'exemple de la figure 5.18, page 171, $H = 13$),
3. Dico est un tableau constant, défini sur l'intervalle $1 .. N$, qui représente le sac des N mots à placer sur la grille.



D	BI	BEC
E	CA	DIS
E	ET	FER
E	IF	MIL
M	ME	EMUE
R	RU	IBIS
T	SI	LISIER
AI	AME	MARBRE

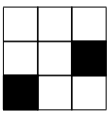
M	E	■	M	I	L
A	M	E	■	B	I
R	U	■	D	I	S
B	E	C	■	S	I
R	■	A	I	■	E
E	T	■	F	E	R

Un énoncé (la grille vierge et le lexique) Une solution possible

Fig. 5.18 – *Exemple d'énoncé et de solution pour le jeu des mots casés*

Question 1. Quelle structure d'énumération permet de mettre en œuvre cette solution ? Quelles sont ses propriétés ? Lequel des patrons du tableau 5.1, page 136, doit-il être retenu ? 67 - Q 1

Question 2. Pour le problème 3×3 de la figure 5.19, fournir l'arbre de récursion (on arrête la recherche dès la découverte de la première solution). 67 - Q 2



A	TA	TRI
I	TA	
AI	RIT	

Fig. 5.19 – *Exemple de mots casés* 3×3

Afin de faciliter le traitement, on fait les hypothèses suivantes :

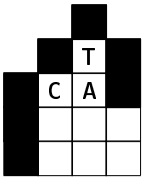
1. *LongEmplH*(i) est une fonction qui délivre la longueur de l'emplacement horizontal i.
2. *MotV* est une fonction qui, une fois la grille complète, délivre le sac des mots placés verticalement.
3. Libre est le sac des mots qui n'apparaissent pas dans la structure d'énumération X. L'union multiensembliste de Libre et des mots présents dans X constitue l'ensemble des mots de Dico.
4. La fonction *ConvSac* convertit un tableau de mots en un sac.

Question 3. Fournir la procédure *MotsCasés1*(i) qui recherche et écrit la première solution trouvée (cette procédure ayant comme précondition qu'il existe au moins une solution). En donner la complexité au pire en termes de conditions évaluées. 67 - Q 3

Cette première solution est perfectible sur le plan de l'efficacité. Nous allons à présent étudier et mettre en œuvre un élagage destiné à apporter une amélioration en termes de complexité temporelle. Pour ce faire, nous proposons de ne pas attendre la fin de la phase de génération pour effectuer une vérification verticale. Plus précisément, dès qu'un mot est candidat à un placement horizontal, on vérifie qu'il ne constitue pas un obstacle

au placement vertical de l'un des mots encore disponible en s'assurant que chacun des caractères du mot candidat est aussi un caractère possible pour un mot vertical.

Exemple Considérons la configuration suivante pour laquelle on s'apprête à tenter de placer le mot **RUE** sur l'avant-dernier emplacement horizontal



alors que le sac des mots disponibles est ⟦**CRI**, **TALC**, **EU**, **OSE**⟧. Le placement de **RUE** est compatible avec celui du mot vertical **CRI**, le **R** étant commun. En revanche, le **U** de **RUE** est incompatible avec tous les mots de quatre lettres libres puisque **TAU** n'est le début d'aucun mot libre de longueur 4. Le placement du mot **RUE** est donc abandonné, ce qui produit un élagage de l'arbre de récursion.

67 - Q 4 **Question 4.** Pour l'exemple de la figure 5.19, page 171, fournir l'arbre de récursion obtenu par l'élagage décrit ci-dessus. Conclusion ?

67 - Q 5 **Question 5.** L'élagage présenté ci-dessus exige un accès horizontal mais aussi vertical aux emplacements et aux mots de la grille. Pour cette raison, nous décidons de prendre comme structure d'énumération la grille elle-même. Spécifier les opérations qui vous semblent nécessaires à la mise en œuvre de l'élagage, puis fournir la procédure *MotsCasés2* qui met en application cet élagage.

Exercice 68. Tableaux autoréférents

L'autoréférence (c'est-à-dire la propriété, pour une entité, de faire référence à elle-même) est une notion qui se rencontre dans de nombreux domaines scientifiques comme en linguistique, en logique ou encore en mathématiques. Dans l'exercice qui suit, on cherche à produire un tableau autoréférent. Deux élagages intéressants sont appliqués.

Un tableau X de n ($n > 0$) éléments, défini sur l'intervalle $0\,..\,n-1$ et à valeurs dans l'intervalle $0\,..\,n-1$, est qualifié d'autoréférent si, pour tout indice i du tableau, $X[i]$ est le nombre d'occurrences de la valeur i dans le tableau. Formellement :

$$\forall i \cdot (i \in 0\,..\,n-1 \Rightarrow X[i] = \#\,j \cdot (j \in 0\,..\,n-1 \text{ et alors } X[j] = i))\,. \tag{5.3}$$

Rappel : # est le quantificateur de comptage.

Ainsi par exemple, pour $n = 4$, le tableau :

i	0	1	2	3
X[i]	1	2	1	0

est un tableau autoréférent : la valeur 0 existe en un exemplaire, la valeur 1 en deux exemplaires, etc. Pour $n < 7$ il est facile de montrer, par énumération, qu'il n'existe pas de solution pour $n \in \{1, 2, 3, 6\}$, et qu'il n'existe qu'une seule solution pour $n = 5$.

Question 1. Donner un second tableau autoréférent pour $n = 4$. 68 - Q 1

Question 2. Que peut-on affirmer à propos de la somme des éléments d'un tableau autoréférent ? Justifier votre réponse. 68 - Q 2

Question 3. On cherche à produire, pour un n donné, tous les tableaux autoréférents, en utilisant la démarche des essais successifs. Quel est le patron approprié parmi ceux de la liste présentée à la figure 5.1, page 136 ? Quel élagage basé sur le résultat de la question 2 peut-il s'appliquer pour l'instanciation de la fonction générique *Satisfaisant* ? Comment la fonction générique *SolutionTrouvée* peut-elle se représenter ? En déduire la procédure *TabAutoRef1*, ainsi qu'un contexte d'appel convenable. 68 - Q 3

Question 4. Un second élagage peut être mis en œuvre. Il se base sur le fait que, si dans la tranche $X[0\,..\,i-1]$ l'élément j est déja présent X[j] fois, il est inutile de tenter de placer j en position i. Ainsi, dans l'exemple suivant : 68 - Q 4

i	...	2	3	...	5	...	12	...	20	...	50
X[i]	...	5	5	...	3	...	5				

X[5] vaut 3 et la valeur 5 est justement présente 3 fois dans $X[0\,..\,19]$. Il est donc inutile de chercher à placer 5 en position 20, la tentative serait vouée à l'échec. Mettre en œuvre cet élagage à travers la procédure *TabAutoRef2*.

Question 5. Montrer que, pour $n \geqslant 7$, les tableaux conformes à la structure suivante : 68 - Q 5

i	0	1	2	3	...	$n-5$	$n-4$	$n-3$	$n-2$	$n-1$
X[i]	$n-4$	2	1	0	...	0	1	0	0	0

$n-7$ fois

sont des tableaux autoréférents.

Conjecture Les auteurs conjecturent que la condition suffisante qui fait l'objet de la question 5 est en fait une condition nécessaire et suffisante.

CHAPITRE 6

Programmation par Séparation et Évaluation Progressive (PSEP)

L'esprit d'ordre est un capital de temps.

(H.-F. Amiel)

6.1 Introduction

Il est conseillé de lire la totalité de cette introduction, ainsi que celle du chapitre 5 avant d'aborder les exercices.

6.1.1 PSEP : le principe

On considère un ensemble C [1] dont les éléments sont appelés des *candidats*. À chaque élément de C est associé un coût v constant : $v \in C \to \mathbb{R}_+$, appelé parfois « coût réel » dans la suite. On recherche l'un quelconque des candidats c minimisant $v(c)$ [2]. Ces candidats sont appelés des *solutions*. Bien entendu, si C est vide, il n'existe pas de solution [3].

Une manière classique de résoudre ce type de problème consiste à énumérer et à évaluer tous les candidats pour retenir l'un de ceux qui minimisent le coût. Lorsque la cardinalité de C est grande, l'inconvénient de cette démarche réside dans le nombre de candidats à considérer et dans la complexité temporelle prohibitive qui en résulte.

Schématiquement, la méthode PSEP (pour Programmation par Séparation et Évaluation Progressive), aussi appelée *Branch and Bound*, se décrit de la manière suivante. Soit $C_1, \ldots, C_n$ une partition de C. Tous les sous-ensembles C_i sont rassemblés dans une structure de données dénommée OPEN. À chaque C_i est associée une évaluation $f(C_i)$ qui est une sous-estimation du coût réel de tous les candidats de C_i. On choisit (c'est la phase de sélection) le sous-ensemble C_j le plus prometteur (c'est-à-dire celui qui minimise la valeur de la fonction d'évaluation f), avant de le partitionner en différents sous-ensembles non vides $C_{j_1}, \cdots, C_{j_m}$ (c'est la phase de Séparation [4]). Pour chaque sous-ensemble C_{j_k}, on calcule la valeur de $f(C_{j_k})$ (c'est l'Évaluation). Le sous-ensemble C_i est alors supprimé

1. En général C est défini en compréhension, mais les cas où C est défini en extension (voir exercice 72, page 195) se traitent de la même façon. Le cas le plus fréquent est celui d'un ensemble C fini, mais l'exercice 71 page 193 traite le cas d'un ensemble infini dénombrable.
2. Les problèmes où l'on cherche à maximiser le coût se traitent de manière identique.
3. Dans le cas d'une définition en compréhension de C, il est en général difficile de savoir *a priori* si C est vide ou non.
4. Aussi appelée éclatement, développement ou partitionnement.

d'OPEN tandis que tous les C_{j_k} y sont introduits. Le procédé est alors réappliqué sur la nouvelle configuration d'OPEN. Initialement, OPEN contient uniquement C. L'algorithme s'achève soit quand l'ensemble C_i sélectionné ne contient qu'un seul candidat, soit quand OPEN est vide. Pour que PSEP fournisse une solution correcte, plusieurs conditions, répertoriées ci-après, doivent être satisfaites (voir le théorème de la page 178). On dit alors que PSEP est *admissible*.

Dans la suite de cette introduction, on précise la nature des trois phases mentionnées ci-dessus avant de synthétiser le tout sous la forme d'un algorithme PSEP générique. On fournit et démontre une condition suffisante d'admissibilité. Un exemple est ensuite traité en détail.

L'étape de séparation

Les séparations successives peuvent se représenter sous la forme d'un arbre (appelé arbre de recherche) dont chaque nœud s'identifie à un sous-ensemble C_j. Une feuille n'est autre qu'un sous-ensemble réduit à un seul élément. Cette dualité partition/arbre se répercute jusqu'au vocabulaire, et nous l'exploitons pour assurer une meilleure compréhension. Un algorithme PSEP parcourt, sans le construire, tout ou partie de l'arbre de recherche, l'objectif visé étant de mettre à contribution l'estimation des nœuds (des sous-ensembles) et la stratégie de sélection afin de limiter la quantité de nœuds développés.

Exemple Soit l'ensemble de candidats $C = \{a, b, c, d, e, f, g, h, i, j, k\}$. La figure 6.1 montre un arbre de recherche complet possible. Les valeurs qui accompagnent chaque nœud peuvent être ignorées pour l'instant.

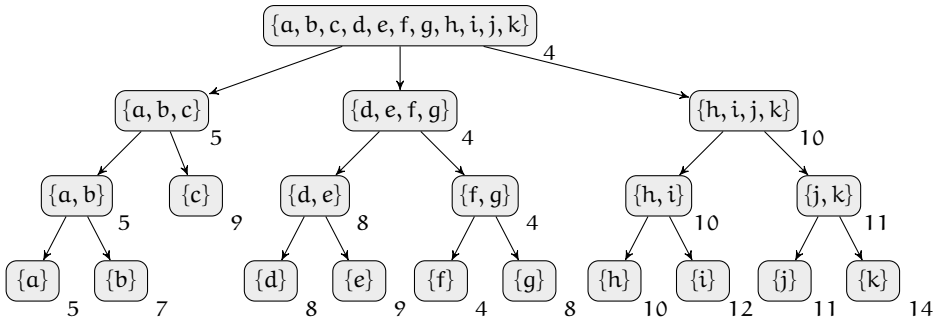


Fig. 6.1 – *Exemple d'arbre de recherche pour l'ensemble* $\{a, \ldots, k\}$. *L'ensemble des fils d'un nœud donné est une partition de ce nœud.*

Deux problèmes doivent être abordés à cette étape : i) comment effectuer la séparation ? ii) comment représenter économiquement chacun des nœuds (des sous-ensembles) ? Ces choix dépendent du problème considéré, mais il n'est pas rare de retrouver ici les solutions adoptées dans le chapitre consacré aux essais successifs (voir section 5.1, page 117) : représentation par un vecteur d'énumération X et séparation par instanciation d'une position de ce vecteur.

L'étape de sélection

L'ordre de développement des nœuds n'est ici pas fixé à l'avance : on cherche à développer en priorité les nœuds que l'on considère comme les plus prometteurs. Pour ce faire, chaque feuille de l'arbre courant est présente dans une file de priorité (voir [36] pour des mises en œuvre efficaces des files de priorité). C'est la structure OPEN mentionnée auparavant. La figure 6.2 montre une configuration possible à un instant donné de la recherche (les priorités sont placées à la droite de chaque nœud).

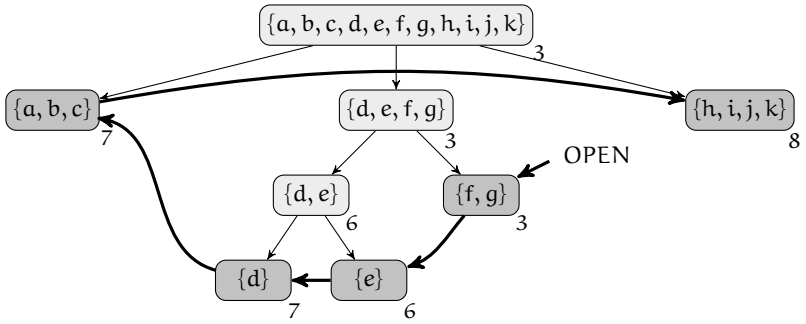


Fig. 6.2 – *Exemple de configuration. Les coûts estimés sont placés sous le coin sud-est des nœuds. La file de priorité est mise en évidence. Bien que possédant la même priorité que le nœud* $\{a, b, c\}$*, le nœud* $\{d\}$*, plus « développé », se situe avant dans la file.*

Le prochain nœud sélectionné (développé/séparé/éclaté/partitionné) est alors le nœud $\{f, g\}$, puisqu'il est affecté de la meilleure priorité. On note qu'au cours de la recherche, on rencontre trois sortes de nœuds : i) ceux présents dans OPEN (en gris foncé dans la figure), qui comprennent en particulier les nœuds susceptibles d'être développés, ii) les nœuds déjà développés (en gris clair ci-dessus), qui ne sont plus actifs, iii) les nœuds pas encore créés, qui n'apparaissent pas sur le schéma.

Cette étape met en lumière la nature essentiellement itérative des algorithmes de type PSEP, puisqu'il s'agit, à chaque pas de progression, de développer le nœud prioritaire. L'inconvénient qui en résulte, par rapport à la méthode des essais successifs, est que la file de priorité peut occuper un espace mémoire important (cependant borné par le poids maximal de l'arbre de recherche). Ceci peut se révéler rédhibitoire pour certains problèmes de grande taille. En revanche, on peut espérer développer moins de nœuds et donc atteindre une meilleure efficacité.

L'étape d'évaluation

Avant d'insérer un nœud dans la file OPEN, il faut lui attribuer une *priorité*. Définissons une fonction $f^\star$ de profil $f^\star \in (\mathbb{P}(C) - \varnothing) \to \mathbb{R}_+$ et telle que :

$$\begin{array}{ll} f^\star(\{c\}) = v(c) & \text{pour tout candidat } c \\ f^\star(\{c_1, \ldots, c_n\}) = \min_{c \in \{c_1, \ldots, c_n\}} (f^\star(\{c\})) & \text{pour } n > 1. \end{array}$$

$f^*(\{c_1, \ldots, c_n\})$ est donc le coût optimal pour tous les candidats de l'ensemble $\{c_1, \ldots, c_n\}$. Pour $E \subseteq C$, $f^*(E)$ est, par extension, appelée le coût *réel* d'un nœud E et $f^*(C)$ le coût (réel) des solutions (pour l'ensemble racine C).

La définition de la fonction f^* implique que celle-ci est croissante, dans le sens où, si $E' \subseteq E$, $f^*(E) \leqslant f^*(E')$ (la fonction f^* croît quand on parcourt une branche depuis la racine). Dans la figure 6.1, page 176, une valeur arbitraire mais possible pour une fonction f^* est placée à la droite de chaque nœud. La (seule) solution est f, de coût 4. Cette figure illustre la croissance de f^*.

Ce coût réel ne peut cependant être utilisé comme priorité pour placer les nœuds dans la file OPEN, puisqu'il est (en général) inconnu. On utilise à la place une fonction f, de même profil que f^*, appelée *fonction d'évaluation*, qui fournit un coût estimé. Cette fonction f doit satisfaire le théorème suivant :

Théorème (Condition suffisante d'admissibilité de l'algorithme PSEP) :
Si la fonction f *vérifie les deux conditions suivantes :*

1. f *est positive ou nulle et n'est jamais supérieure à* f^* *:*

$$\forall E \cdot (E \subseteq C \Rightarrow f(E) \leqslant f^*(E)), \tag{6.1}$$

2. *pour tout candidat* c*, la valeur de* f *en* $\{c\}$ *est égale à la valeur de* f^* *en* $\{c\}$ *:*

$$\forall c \cdot (c \in C \Rightarrow f(\{c\}) = f^*(\{c\})), \tag{6.2}$$

alors l'algorithme PSEP est admissible.

Ce théorème est démontré dans la section portant sur la construction de l'algorithme PSEP.

Dans la section suivante, nous nous attachons à construire une version abstraite du programme générique à la base de toute solution de type PSEP, en supposant disponible une fonction d'évaluation f dotée des deux propriétés ci-dessus. Il ne faut pas perdre de vue qu'évaluer la fonction f a un coût (comme la consultation et la maintenance de structures de données), qu'il faut limiter autant que faire se peut, pour ne pas perdre les avantages de la méthode PSEP.

6.1.2 L'algorithme générique PSEP

La file de priorité

L'algorithme générique que nous allons développer s'appuie sur une structure de données introduite à la section 1.7 page 33 : la file de priorité OPEN. Une précision doit être apportée qui concerne l'opération *Ajouter* : en cas de conflit (égalité sur les priorités), la préférence est donnée au nœud le plus développé (c'est-à-dire le plus bas dans l'arbre de recherche). Si ce critère ne permet toujours pas de départager deux sous-ensembles, la préférence va à celui qui est déjà présent dans la file.

Construction de l'algorithme générique PSEP

Nous développons cet algorithme sur la base des cinq points classiques de la construction de boucles.

Invariant La file de priorité OPEN contient une partition de l'ensemble C. Chaque élément de cette partition est accompagné de sa priorité. Certains d'entre eux ne contiennent qu'un seul candidat, les autres sont susceptibles d'être « séparés ». Dans la figure 6.2, page 177, chaque nœud E du schéma est étiqueté par la valeur de la fonction d'évaluation $f(E)$ (qui est à la fois le coût estimé et la priorité) ; la file OPEN est dessinée en gras et ses constituants sont grisés. On note que la file est triée sur les priorités croissantes. Dans cette figure, l'arbre n'est là que pour faciliter la compréhension. Il n'est pas construit par l'algorithme.

Condition d'arrêt La boucle s'arrête soit quand la file OPEN est vide, soit quand la tête de file représente un *candidat* c (sa priorité est $f(\{c\})$). Dans ce dernier cas, deux observations peuvent alors être faites.

1. Puisque $\{c\}$ est en tête de la file OPEN et que $f(\{c\}) = f^*(\{c\})$ (voir formule 6.2, page 178), il n'existe pas de meilleur candidat dans OPEN (il peut en revanche exister des candidats équivalents).
2. Soit E ($\mathrm{card}(E) > 1$) un sous-ensemble de C présent dans OPEN, avec la priorité $f(E)$ ($f(\{c\}) \leqslant f(E)$). L'invariant entraîne que $E \cap \{c\} = \varnothing$. Nous allons montrer que $f(\{c\}) \leqslant f(E)$ implique bien que c est une solution.

$$
\begin{array}{ll}
 & f(\{c\}) \leqslant f(E) \\
\Rightarrow & \text{propriété 6.1 } (f(E) \leqslant f^*(E)) \text{ et transitivité} \\
 & f(\{c\}) \leqslant f^*(E) \\
\Leftrightarrow & \text{définition de } f^*\text{, cas général, voir page 177} \\
 & f(\{c\}) \leqslant \min_{d \in E}(f^*(\{d\})) \\
\Leftrightarrow & \text{définition de min} \\
 & \forall d \cdot (d \in E \Rightarrow f(\{c\}) \leqslant f^*(\{d\})) \\
\Leftrightarrow & \text{propriété 6.2 } (f(\{c\}) = f^*(\{c\})) \text{ et substitution des égaux} \\
 & \forall d \cdot (d \in E \Rightarrow f^*(\{c\}) \leqslant f^*(\{d\}))
\end{array}
$$

   Cette démonstration peut se résumer par : pour tout d de l'ensemble E, $f(\{c\}) = f^*(\{c\}) \leqslant f^*(E)) \leqslant f^*(\{d\})$. La dernière formule de cette démonstration peut se traduire par : un sous-ensemble E ($E \subset C$), présent dans OPEN et situé après la tête de file $\{c\}$, ne peut donner naissance à un meilleur candidat que c.

Dans le cas où la file est vide, l'invariant nous affirme qu'elle contient une partition de l'ensemble C, ce dernier est donc vide lui aussi. Il n'y a pas de candidat, ni de solution. Si la file n'est pas vide, la conjonction de l'invariant et de la condition d'arrêt entraîne bien que le candidat c est une solution, ce qui démontre l'admissibilité de l'algorithme (voir page 178). En supposant disponible la fonction booléenne *EstFeuille*(E), qui détermine si le sous-ensemble E est ou non une feuille de l'arbre, la condition d'arrêt s'exprime par $EstVide(\mathrm{OPEN})$ **ou sinon** $EstFeuille(Tête(\mathrm{OPEN}).se)$ [5].

Progression La précondition de la progression précise que l'invariant est satisfait ainsi que la négation de la condition d'arrêt. On peut en déduire que la tête de la file OPEN existe et qu'il s'agit d'un ensemble de plusieurs candidats. Elle peut donc être « séparée ».

5. Le champ *se* permet d'extraire le sous-ensemble présent dans un nœud en écartant la priorité.

La progression consiste à partitionner la tête de la file, à la supprimer de la file, à calculer la priorité de chacun des éléments de la partition, avant de les introduire dans la file, selon leur priorité. Cette étape exige (en général) de construire une boucle qui va rétablir l'invariant. Cette construction n'est pas explicitée ici.

Initialisation On instaure l'invariant en plaçant C dans la file, avec sa priorité.

Terminaison Si C est fini, l'algorithme se termine puisque $\mathbb{P}(C)$ est également fini. Une fonction de terminaison possible consiste à choisir le poids maximum de l'arbre de recherche moins le nombre de nœuds déjà construits. Si l'ensemble C est infini dénombrable et qu'il ne contient pas de solution, l'algorithme ne s'achève pas.

L'algorithme générique PSEP proprement dit

Dans l'algorithme qui suit, ElmtFdp est le type des éléments à placer dans la file : c'est un couple dont le premier constituant *se* (pour sous-ensemble) est un sous-ensemble de C et le second, p, la priorité. FdP est un type (voir chapitre 1, section 1.7) qui permet de déclarer des files de priorité. Pour notre exemple, cette file contient des éléments du type ElmtFdP. On suppose disponible la fonction *Partition*(E) qui fournit, selon une stratégie à définir au coup par coup, une partition de E ne contenant pas l'ensemble vide. La fonction qui évalue le coût estimé f est également supposée disponible.

1. **constantes**
2. C ⊆ ... **et** C = {...} **et** ElmtFdP = {(se, p) | se ∈ ℙ(C) − ∅ **et** p ∈ ℕ}
3. **variables**
4. OPEN ∈ FdP(ElmtFdP) **et** E ⊂ C **et** E ≠ ∅ **et** t ∈ ElmtFdP
5. **début**
6. *InitFdP*(OPEN) ; *AjouterFdP*(OPEN, (C, f(C))) ;
7. **tant que non** (*EstVideFdP*(OPEN) **ou sinon** *EstFeuille*(*TêteFdP*(OPEN).se)) **faire**
8. t ← *TêteFdP*(OPEN) ; *SupprimerFdP*(OPEN) ;
9. **pour** E ∈ *Partition*(t.se) **faire**
10. *AjouterFdP*(OPEN, (E, f(E))) ;
11. **fin pour**
12. **fin tant que** ;
13. **si** *EstVideFdP*(OPEN) **alors**
14. écrire(*Pas de solution*)
15. **sinon**
16. écrire(*TêteFdP*(OPEN))
17. **fin si**
18. **fin**

Remarques

1. Cette version place dans la file de priorité tous les éléments de la partition, y compris ceux pour lesquels le coût estimé est égal à $+\infty$. Il peut être intéressant d'éviter d'encombrer la file de priorité avec de tels éléments, qui ne présentent pas d'intérêt. L'invariant ci-dessus doit alors être modifié afin de tenir compte du fait que les ensembles dont le coût est infini ne sont pas dans la file.
2. Une seconde heuristique (dite de nettoyage), qui va également dans le même sens que la précédente, peut être systématiquement prise en compte afin de limiter la

taille de la file de priorité. Elle est basée sur le constat suivant. Dès qu'un candidat c de coût $v(c)$ (doté d'une priorité $p = v(c)$) est introduit dans la file de priorité (p est alors, selon formule 6.2 page 178, son coût réel), il est inutile d'*introduire* ou de *conserver* dans la file tout nœud doté d'une priorité égale ou supérieure.

6.1.3 $f^\star/f$: UN CAS PARTICULIER INTÉRESSANT

Il arrive fréquemment que chaque arc de l'arbre de recherche soit valué et que le coût réel d'un candidat c résulte de la somme des coûts des arcs qui mènent de la racine C à la feuille $\{c\}$. La fonction $f^\star$ est alors telle que, pour tout nœud E, $f^\star(E)$ est le coût du chemin optimal issu de la racine C et *passant par le nœud* E. Cette version de $f^\star$ est compatible avec la version générique étudiée ci-dessus. Pour tout nœud E, $f^\star(E)$ peut se décomposer en une somme de deux fonctions :

$$f^\star(E) = g^\star(E) + h^\star(E) \tag{6.3}$$

où $g^\star(E)$ est le coût du chemin entre la racine C et le nœud E, tandis que $h^\star(E)$ est le coût du meilleur chemin du sous-arbre ayant comme racine E.

L'avantage qui résulte de ce cas de figure est que, lorsque l'on atteint le nœud E, $g^\star(E)$ est connu : c'est le coût du chemin déjà parcouru. On peut alors définir la fonction d'évaluation f par :

$$f(E) = g^\star(E) + h(E) \tag{6.4}$$

h est appelée « fonction heuristique ». L'évaluation de $h(E)$ ne porte que sur la portion de chemin restant à parcourir pour atteindre un candidat. On a alors le théorème suivant :

Théorème (Condition suffisante d'admissibilité de l'algorithme PSEP) :
Soit f *une fonction d'évaluation définie pour tout ensemble* E *par :*

$$f(E) = g^\star(E) + h(E),$$

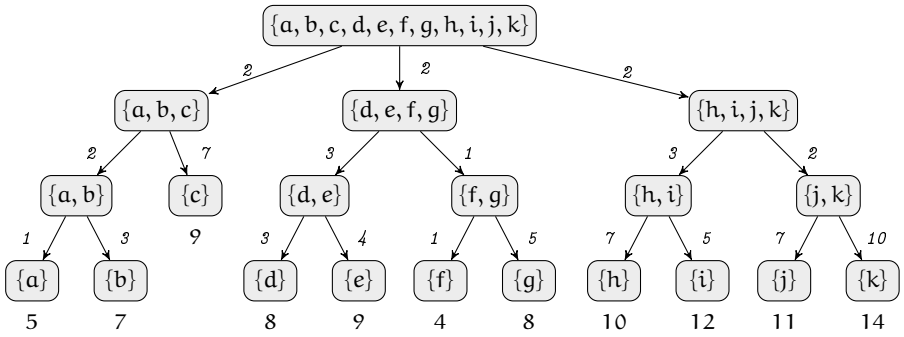
où $g^\star$ *est le coût du chemin déjà parcouru. Si la fonction heuristique* h *vérifie :*

$$0 \leqslant h(E) \leqslant h^\star(E)$$

alors l'algorithme PSEP est admissible.

Ce théorème est une conséquence directe du théorème de la page 178.

Exemple Reprenons l'exemple de la figure 6.1, page 176, en attribuant à chaque arc une valeur de $\mathbb{R}_+$. Dans l'arbre ci-dessous, la valeur notée en petits caractères et en italique représente le coût de chaque arc. Pour chaque candidat c, la valeur de $f^\star(\{c\})$ est placée sous les feuilles ; c'est la somme des valeurs de la branche correspondante.



Si l'on note $k(a, b)$ le coût de l'arc (a, b), le nouveau programme générique s'obtient en modifiant le programme de la page 180 de la manière suivante. La nouvelle ligne 2 intègre le champ $g^\star$ dans un nœud :

$$\text{ElmtFdP} \;=\; \{(se, g^\star, p) \mid se \in (\mathbb{P}(C) - \varnothing) \textbf{ et } g^\star \in \mathbb{N} \textbf{ et } p \in \mathbb{N}\}.$$

La nouvelle ligne 6 insère dans la file un élément dont le champ $g^\star$ est nul et le champ p se réduit à $h(C)$:

$$\mathit{Init}(\text{OPEN})\,;\ \mathit{Ajouter}(\text{OPEN}, (C, 0, 0 + h(C)))$$

Enfin, la nouvelle ligne 10 prend en compte la valeur de $g^\star$ calculée à partir de la valeur du père du nœud et du coût de l'arc qui les joint :

$$\mathit{Ajouter}(\text{OPEN}, (E, t.g^\star + k(t.se, E), t.g^\star + k(t.se, E) + h(E)))\,;$$

Plus h est proche (inférieurement) de $h^\star$, plus on gagne en efficacité[6]. Plus précisément, soit h_1 et h_2 deux fonctions heuristiques telles que pour tout argument E, $h_1(E) \leqslant h_2(E)$, et soit A_1 (resp. A_2) une version de PSEP utilisant la fonction h_1 (resp. h_2). À l'issue de l'exécution, si le nœud w est développé par A_2, il l'est aussi par A_1. Une solution triviale, mais peu efficace, consiste à prendre pour h la fonction nulle.

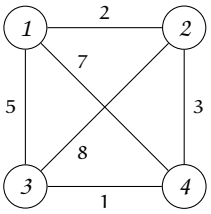
6.1.4 Un exemple : le problème du voyageur de commerce

Nous retrouvons ici le problème traité selon la méthode des essais successifs dans l'exercice 56, page 156. Partant d'un graphe non orienté[7] $G = (N, V, D)$ de n ($n \geqslant 2$) sommets, valué sur $\mathbb{R}_+$, il s'agit de découvrir, s'il existe, un cycle hamiltonien de coût minimal. Nous pouvons, sans perte de généralité, nous limiter à la recherche d'une solution débutant au nœud 1. Le graphe décrit à la figure 6.3 sert de support à notre propos. Le schéma (b) représente la matrice des distances D entre les nœuds $1, 2, 3$ et 4.

Outre qu'il vient illustrer les généralités exposées dans les deux précédentes sections, cet exemple vise un double objectif :

1. il s'agit tout d'abord de montrer que l'algorithme générique de la section précédente exige parfois des adaptations propres au problème considéré ;
2. plus essentiel, cet exemple permet de montrer l'importance du choix de la fonction heuristique h pour l'efficacité de l'algorithme.

6. En faisant abstraction du coût imputable au calcul de h.
7. Les versions proposées ci-dessous se transposent facilement au cas des graphes *orientés*.



(a)

$$\begin{bmatrix} \infty & 2 & 5 & 7 \\ 2 & \infty & 8 & 3 \\ 5 & 8 & \infty & 1 \\ 7 & 3 & 1 & \infty \end{bmatrix}$$

(b)

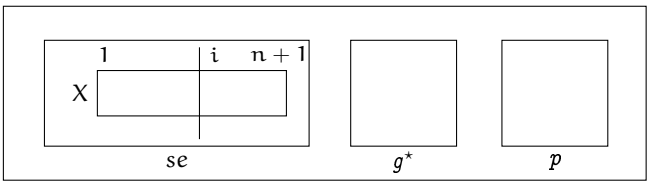
Fig. 6.3 – *Exemple de graphe non orienté valué. Le schéma* (b) *est la représentation matricielle* D *du schéma* (a). *Le cycle* $[1,3,4,2,1]$ *est une solution, dont le coût est* 11.

Choix initiaux

Comme suggéré ci-dessus, il faut tout d'abord décider de la représentation des sous-ensembles de candidats, ainsi que de la stratégie de partitionnement. Pour notre problème, l'ensemble des candidats se présente sous la forme : $\{[1,2,3,4,1],[1,3,2,4,1],\ldots,[1,4,3,2,1]\}$. Plus généralement, un candidat est donc un vecteur X de $n+1$ éléments débutant et s'achevant par la valeur 1, et tel que les n premiers éléments constituent une permutation de l'intervalle $1\,..\,n$.

L'algorithme

L'objectif est ici d'instancier l'algorithme générique afin de prendre en compte les particularités du problème. Il apparaît alors que la progression de la boucle principale doit distinguer le cas général (rechercher une permutation convenable à placer dans la tranche $X[1\,..\,n]$) du cas particulier où l'on cherche à placer 1 en $X[n+1]$. Par ailleurs, en accord avec la remarque 1 de la page 180, nous décidons d'écarter les branches présentant un coût infini. Dans cette partie, une instance – restant à définir – de la fonction h est supposée disponible. Précisons la nature des nœuds de la file OPEN. Ainsi que le montre le schéma ci-dessous, un tel nœud est constitué des trois champs se, $g^\star$ et p :



1. Le champ se : c'est le vecteur X accompagné de son indice de remplissage i. Ce dernier désigne la prochaine cellule à compléter. X possède $n+1$ cellules (qui permettent de placer une permutation de l'intervalle $1\,..\,n$ plus le 1 final). La tranche $X[1\,..\,i-1]$ contient les positions instanciées et représente le sous-ensemble de tous les candidats débutant par ces $(i-1)$ valeurs. Le partitionnement consiste alors à instancier la position i avec chacune des valeurs encore disponibles pour obtenir une permutation. Pour un X complet, nous avons $i = n+2$.
2. Le champ $g^\star$ est la valeur de la fonction éponyme de la formule 6.3, page 181.

3. Le champ p est la priorité de l'élément (c'est-à-dire la valeur de la fonction d'évaluation f de la formule 6.4).

Décrivons à présent les principaux constituants de l'algorithme présenté ci-dessous. L'ensemble PC (déclaré à la ligne 3) matérialise tous les sous-ensembles possibles de candidats (selon le choix de représentation adopté ci-dessus). L'ensemble ElmtFdP (ligne 4) représente les structures placées dans la file de priorité (voir ci-dessus). seNouv, $g^\star$Nouv et pNouv sont des variables auxiliaires utilisées pour construire les éléments de l'ensemble ElmtFdP à placer dans la file. La ligne 10 construit le champ se du nœud initial (qui représente l'ensemble C), en plaçant la valeur 1 en X[1] et 0 dans la tranche $X[2\,..\,n+1]$ et en initialisant i à 2. La ligne 11 a pour effet de placer dans la file OPEN la structure constituée des trois valeurs seNouv (qui vient d'être construite) pour le champ se, 0 pour le champ $g^\star$ et $0 + h(se\text{Nouv})$ pour le champ p. La différence de traitement entre le cas général et le cas particulier lié à la position $(n+1)$ du vecteur X est prise en compte par l'alternative débutant à la ligne 14. L'élimination des nœuds présentant un coût infini s'opère grâce aux deux alternatives des lignes 14 et 23. La séquence allant de la ligne 16 à la ligne 18 construit et place un ensemble de *un* candidat dans la file OPEN, tandis que la séquence qui va de la ligne 24 à la ligne 27 fait de même lorsque l'on est en présence d'un ensemble d'au moins deux candidats (c'est-à-dire face à une structure dont le vecteur X n'est pas complet), ce qui a pour effet de rétablir l'invariant. L'algorithme se présente comme suit :

1. **constantes**
2. $\quad n \in \mathbb{N}_1 - \{1\}$ **et** $n = \ldots$ **et**
3. $\quad PC = \{(X, i) \mid X \in 1\,..\,n+1 \to 1\,..\,n$ **et** $i \in 1\,..\,n+2\}$ **et**
4. $\quad ElmtFdP = \{(se, g^\star, p) \mid se \in PC$ **et** $g^\star \in \mathbb{N}$ **et** $p \in \mathbb{N}\}$
5. **variables**
6. $\quad OPEN \in FdP(ElmtFdP)$ **et** $t \in ElmtFdP$ **et**
7. $\quad se\text{Nouv} \in PC$ **et** $g^\star\text{Nouv} \in \mathbb{N}$ **et** $p\text{Nouv} \in \mathbb{N}$
8. **début**
9. $\quad Init(OPEN)$;
10. $\quad se\text{Nouv} \leftarrow (\{1 \mapsto 1\} \cup (2\,..\,n+1 \times \{0\}), 2)$;
11. $\quad Ajouter(OPEN, (se\text{Nouv}, 0, 0 + h(se\text{Nouv})))$;
12. $\quad$ **tant que non**$(EstVide(OPEN)$ **ou sinon** $EstFeuille(Tête(OPEN).se))$ **faire**
13. $\quad\quad t \leftarrow Tête(OPEN)$; $Supprimer(OPEN)$;
14. $\quad\quad$ **si** $t.se.i = n+1$ **et alors** $D[t.se.X[t.se.i-1], 1] \neq +\infty$ **alors**
15. $\quad\quad\quad$ /% *Cas particulier, retour vers le nœud de départ* : %/
16. $\quad\quad\quad se\text{Nouv} \leftarrow (t.se.X[1\,..\,n] \cup \{n+1 \mapsto 1\}, n+2)$;
17. $\quad\quad\quad g^\star\text{Nouv} \leftarrow t.g^\star + D[t.se.X[t.se.i-1], 1]$;
18. $\quad\quad\quad Ajouter(OPEN, (se\text{Nouv}, g^\star\text{Nouv}, g^\star\text{Nouv}))$
19. $\quad\quad$ **sinonsi** $t.se.i \neq n+1$ **alors**
20. $\quad\quad\quad$ /% *Cas général* : %/
21. $\quad\quad\quad$ **pour** j **parcourant** $(2\,..\,n) - \text{codom}(t.se.X[2\,..\,t.se.i-1])$ **faire**
22. $\quad\quad\quad\quad$ /% *Éclatement* : %/
23. $\quad\quad\quad\quad$ **si** $D[t.se.X[t.se.i-1], j] \neq +\infty$ **alors**
24. $$se\text{Nouv} \leftarrow \left(\begin{pmatrix} t.se.X[1\,..\,t.se.i-1] \cup \{i \mapsto j\} \cup \\ (t.se.i+1\,..\,n+1 \times \{0\}) \end{pmatrix}, \; t.se.i+1 \right) ;$$
25. $\quad\quad\quad\quad g^\star\text{Nouv} \leftarrow t.g^\star + D[t.se.X[t.se.i-1], j]$;

```
26.              pNouv ← g*Nouv + h(seNouv) ;
27.              Ajouter(OPEN, (seNouv, g*Nouv, pNouv))
28.            fin si
29.          fin pour
30.        fin si
31.    fin tant que ;
32.    si EstVide(OPEN) alors
33.      écrire(Pas de solution)
34.    sinon
35.      écrire( Tête(OPEN))
36.    fin si
37. fin
```

Dans la suite de cette section, nous considérons, en guise d'exemple, deux cas de figure pour h (un troisième est étudié dans l'exercice 70, page 191).

Première étude de cas : la fonction heuristique nulle

Le cas le plus simple que l'on puisse imaginer consiste à prendre pour fonction heuristique h la fonction nulle, ce qui revient à utiliser le coût du chemin déjà parcouru comme priorité, sans faire intervenir d'estimation. Le théorème de la page 181 s'applique, et cette technique peut se pratiquer quel que soit le problème traité par PSEP. Elle garantit la découverte d'une solution, s'il en existe. Le tableau 6.1 et l'arbre de recherche de la figure 6.4, page 186, offrent deux visions différentes (mais équivalentes) du résultat de l'application de l'algorithme au graphe de la figure 6.3, page 183.

Le tableau 6.1 met l'accent sur le contenu et la structure de la file de priorité. Chaque ligne correspond à une configuration de la file de priorité, les nœuds apparaissant dans l'ordre de priorité croissant (le champ f). Pour toute ligne k au-delà de la première, les cellules grisées sont les fils développés par la tête de file de la ligne $k-1$. Certains nœuds sont représentés dans une police grisée et en italique : il s'agit des nœuds qui sont écartés de la file, ou qui n'y sont pas introduits lorsque l'heuristique de nettoyage décrite dans la remarque 2, page 180, est appliquée. Ainsi, à la ligne 6 du tableau, l'introduction du candidat 12431 entraîne la suppression du nœud *132***. Dans les lignes suivantes, tous les nœuds qui suivent 12431 sont grisés et en italique : ils ne peuvent donner naissance à une solution meilleure que 12431.

L'arbre de la figure 6.4 présente des informations similaires. Les arêtes sont étiquetées par leur coût. Les branches sont associées à deux informations numériques. Celle qui apparaît sur un fond grisé fournit l'ordre de prise en compte du nœud pour un (éventuel) développement de ses fils. La seconde est la priorité associée au nœud (c'est-à-dire pour ce cas de figure, le coût du chemin déjà parcouru). Certains nœuds sont représentés dans une police grise et en italique. Il s'agit des nœuds qui sont écartés de la file ou qui n'y sont pas introduits lorsque l'heuristique décrite dans la remarque 2, page 180, est appliquée. Ainsi, lors de l'introduction du candidat 12431, le nœud*132*** est supprimé, puisqu'il ne peut engendrer de meilleure solution que 12431. Quant aux nœuds *1234**, *13421*, *1423** et *1432**, ils ne sont alors tout simplement pas introduits dans la file.

Pour l'exemple considéré, la version qui applique l'heuristique de nettoyage de la remarque 2, page 180, développe treize nœuds (et en supprime un déjà présent) avec une

taille maximale de la file égale à cinq. La version de base (sans nettoyage) développe 17 nœuds pour une taille maximale de 6.

	nœud	f	nœud	f	nœud	f	nœud	f	nœud	f	nœud	f
1	1****	0										
2	12***	2	13***	5	14***	7						
3	124**	5	13***	5	14***	7	123**	10				
4	13***	5	1243*	6	14***	7	123**	10				
5	1243*	5	134**	6	14***	7	123**	10	132**	13		
6	134**	6	14***	7	123**	10	12431	11	*132**	*13*		
7	14***	7	1342*	9	123**	10	12431	11	*132**	*13*		
8	143**	8	1342*	9	123**	10	142**	10	12431	11	*132**	*13*
9	1342*	9	123**	10	142**	10	12431	11	*132**	*13*	*1432**	*16*
10	123**	10	142**	10	12431	11	*13421*	*11*	*132**	*13*	*1432**	*16*
11	142**	10	12431	11	*13421*	*11*	*1234**	*11*	*132**	*13*	*1432**	*16*
12	12431	11	*13421*	*11*	*1243**	*11*	*132**	*13*	*1432**	*16*	*1423**	*18*

Tab. 6.1 – *États successifs de la file de priorité* OPEN *pour le traitement du graphe de la figure 6.3, page 183, pour le cas* $h = 0$. *Pour chaque ligne* k, *les cellules en gris représentent les fils de la tête de la file de la ligne* $(k-1)$. *Les cellules en police grise et italique sont les nœuds qui peuvent soit être supprimés soit être ignorés en raison de l'introduction du candidat 12431.*

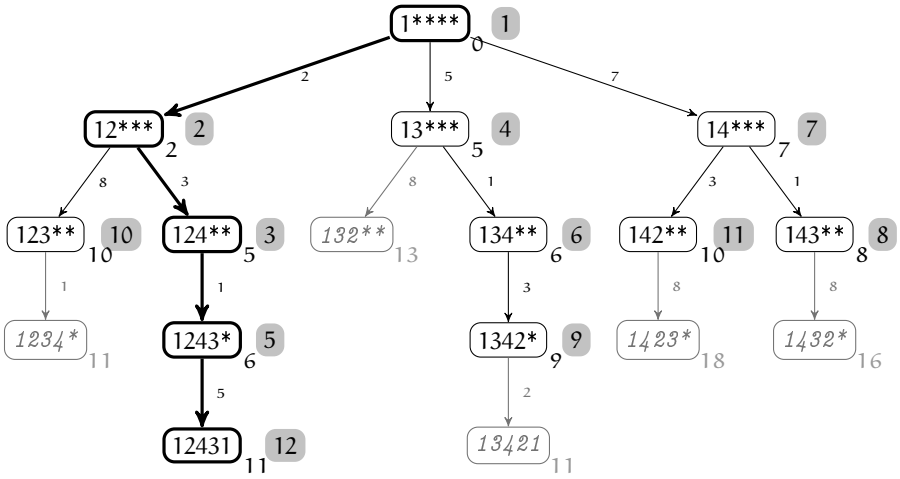


Fig. 6.4 – *Arbre de recherche pour le traitement du graphe de la figure 6.3 page 183, pour le cas* $h = 0$. *Les * correspondent à des positions non instanciées du vecteur* X. *L'ordre d'apparition d'un nœud en tête de la file de priorité est le numéro placé sur fond gris. Le coût du chemin menant au nœud est placé sous le coin sud-est de chaque nœud. Les nœuds en police grise et italique sont les nœuds qui peuvent soit être supprimés soit être ignorés suite à l'introduction du candidat 12431.*

Seconde étude de cas : le coût uniforme

En utilisant une fonction heuristique h plus fine, on peut espérer une meilleure efficacité. C'est ce que nous allons tenter de montrer en choisissant la fonction h suivante, dite du coût uniforme. Pour fixer les idées, considérons le graphe (a) de la figure 6.5, page 188. Formulons l'hypothèse que la chaîne déjà parcourue est la chaîne 146. La fonction $g^*([146*****])$ qui détermine le coût de cette chaîne vaut donc $3+4=7$ (voir schéma (b) de la figure). Soit G′ le sous-graphe obtenu à partir de G en écartant les sommets de la chaîne déjà parcourue. Pour notre exemple, ce sous-graphe est constitué des sommets $2, 3, 5, 7$ et 8, qui apparaissent sous la ligne pointillée dans le schéma (b) de la figure. La fonction h de coût uniforme se définit comme suit :

1. On choisit l'arête de coût minimal dans G′ (3 ici pour l'arête $(3,7)$) et on applique uniformément ce coût aux chemins hamiltoniens que l'on pourrait construire sur G′. Dans notre exemple, un chemin hamiltonien sur G′ comprend quatre arêtes. Au total, si toutes les arêtes valaient 3, ce coût s'élèverait à $4 \cdot 3 = 12$.
2. On prend le coût minimal nécessaire pour rabouter la chaîne déjà parcourue avec une chaîne hamiltonienne sur G′ afin d'obtenir un cycle hamiltonien. Pour ce faire, on choisit l'arête de coût minimal entre le sommet 1 et le graphe G′ (soit l'arête $(1,3)$ de coût 1) et l'arête de coût minimal entre le sommet 6 et le graphe G′ (soit l'arête $(6,2)$ de coût 2). Au total le coût de cette opération s'élève à $1+2=3$.

$h([146******])$ vaut donc $12+3=15$.

La fonction h ainsi définie est, pour tout nœud, inférieure à h^*. Elle satisfait la condition du théorème de la page 181 ce qui garantit l'admissibilité de l'algorithme.

Appliquons cette fonction au graphe de la figure 6.3, page 183. Le principe est similaire au cas de la fonction heuristique nulle. Les résultats sont présentés, sous deux formes complémentaires, d'une part dans le tableau 6.2, page 187, et d'autre part dans l'arbre de la figure 6.6, page 189.

Le tableau 6.2, page 187, met l'accent sur le nombre d'itérations nécessaires à l'obtention du résultat : 5. Tout en adoptant les mêmes conventions que celles du tableau 6.1, page 186, il détaille également le calcul de la fonction d'évaluation f.

	nœud			nœud			nœud			nœud		
	g*	h	f	g*	h	f	g*	h	f	g*	h	f
1	1****											
	0	2+2+2·1	6									
2	12***			13***			14***					
	2	5+3+1·1	11	5	2+1+3·1	11	7	2+1+8·1	18			
3	124**			13***			123**			14***		
	5	5+1+0	11	5	2+1+3·1	11	10	7+1+0	18	7	2+1+8·1	18
4	1243*			13***			123**			14***		
	6	5	11	5	2+1+3·1	11	10	7+1+0	18	7	2+1+8·1	18
5	12431			*13****			*123***			*14****		
	11	0	11	*5*	*2+1+3·1*	*11*	*10*	*7+1+0*	*18*	7	*2+1+8·1*	*18*

Tab. 6.2 – *États successifs de la file de priorité* OPEN *pour le traitement du graphe de la figure 6.3 page 183, par la fonction heuristique du coût uniforme. Voir aussi la légende du tableau 6.1, page 186.*

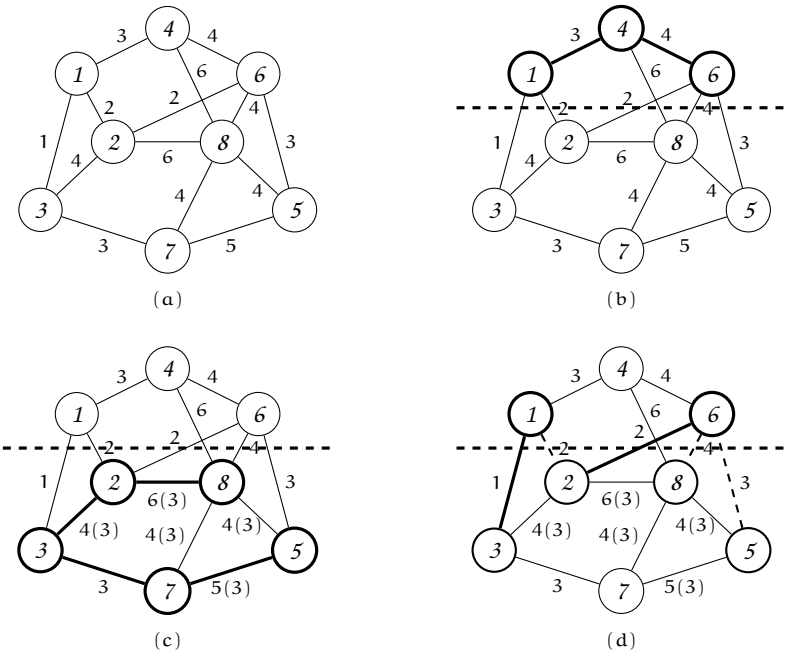


Fig. 6.5 – *Exemple de calcul de la valeur de* $f([146******])$ *par la méthode du coût uniforme. Le schéma* (a) *montre un graphe non orienté valué, de huit nœuds. Le schéma* (b) *met en évidence, en gras, le coût de* $g^*([146******])$, *soit* $3+4=7$. *Le schéma* (c) *permet de connaître le coût uniforme d'une chaîne hamiltonienne sur le sous-graphe* G' *incluant les nœuds* $2,3,5,7$ *et* 8. *Le coût uniforme (le coût minimum des arêtes du sous-graphe* G', *soit* 3, *pour l'arête* $(3,7)$*) est noté entre parenthèses. On obtient un coût uniforme total de* $4 \cdot 3 = 12$. *Enfin, le schéma* (d) *montre comment se raboutent les chaînes hamiltoniennes du schéma* (c) *et la chaîne du schéma* (b), *en prenant, pour les arêtes quittant les nœuds* 1 *et* 6 *et rejoignant une chaîne hamiltonienne construite sur les nœuds* $2,3,5,7$ *et* 8, *l'arête de coût minimal (respectivement* 1 *et* 2*). Au total* $f([146******]) = 7+12+(1+2) = 22$.

Au regard de l'arbre (voir figure 6.6, page 189), l'expression apparaissant sous chaque nœud sous la forme $a+(b+c+d)$ dénote le coût de la fonction d'évaluation f. Plus précisément, a est la valeur de g^*, et $(b+c+d)$ la valeur de h. b (resp. c) est le coût minimum pour l'arête qui quitte le sommet 1 (resp. le dernier sommet de la chaîne déjà parcourue), d est une borne minorante du coût d'une chaîne hamiltonienne sur G'. Sur cet exemple, les trois nœuds en police grisée et en italique sont des nœuds qui sont supprimés de la file lorsque le candidat 12431 y est introduit (en vertu de l'heuristique de nettoyage décrite dans la remarque 2, page 180). Cette version développe donc huit nœuds (à comparer aux treize nœuds nécessaires dans le cas de la fonction heuristique nulle). Il faut cependant insister sur le fait qu'évaluer la complexité uniquement en termes de nombre de nœuds développés peut occulter le surcoût engendré par les évaluations d'une fonction heuristique sophistiquée.

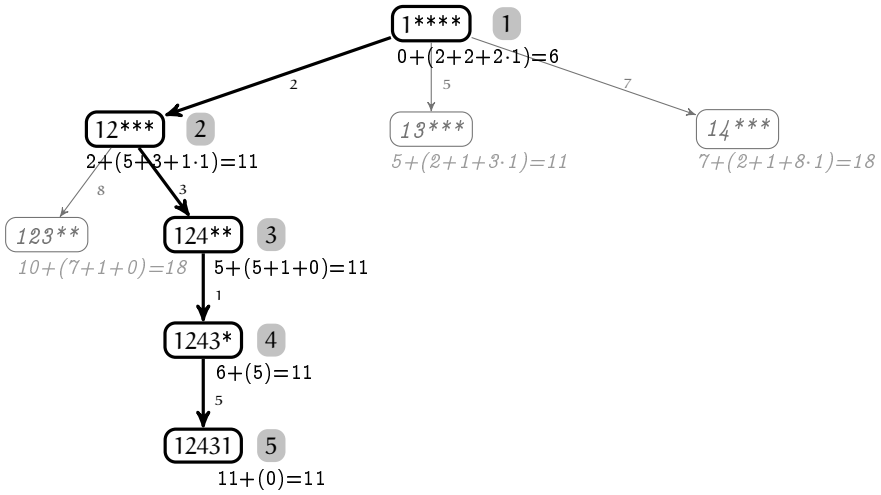


Fig. 6.6 – *Arbre de recherche pour le traitement du graphe de la figure 6.3 page 183, par la fonction heuristique du coût uniforme. Voir aussi la légende de la figure 6.4, page 186. La branche en gras est celle qui mène à la solution.*

6.2 Ce qu'il faut retenir de la démarche PSEP

Le problème de recherche d'une solution minimisant un certain coût peut être abordé sous l'angle de la méthode PSEP aux conditions suivantes.

1. Il existe un ensemble fini de candidats C, chaque candidat c étant doté d'un coût $v(c)$ non négatif.
2. Un ensemble d'au moins deux candidats peut être partitionné en plusieurs sous-ensembles non vides.
3. Un sous-ensemble de candidats possède un coût (réel) $f^*(\{c\})$ tel que : i) $f^*(\{c\}) = v(c)$, ii) f^* est une fonction croissante au sens large.
4. Il est possible d'attribuer à tout sous-ensemble de candidats un coût estimé f tel que : i) pour tout candidat c, $f^*(\{c\}) = f(\{c\}) = v(c)$, ii) $f \leqslant f^*$.

Par conséquent, les deux premières actions à entreprendre, avant d'instancier le programme générique de la section 6.1.2 page 179, consistent à choisir :

1. une représentation pour les sous-ensembles de candidats,
2. une stratégie de partitionnement pour tout ensemble de candidats.

En revanche, le choix d'une fonction d'évaluation f fournissant le coût estimé d'un ensemble de candidats peut en général être différé. Il n'est cependant pas rare qu'il existe une interdépendance entre ces trois aspects.

Comme les essais successifs, la méthode PSEP a intrinsèquement une complexité exponentielle due au nombre de candidats engendrés. Nous nous intéresserons donc principalement à la mise en évidence de fonctions d'évaluation efficaces (notamment au constituant h de la fonction f).

6.3 Exercices

Exercice 69. Assignation de tâches

Dans cet exercice, quatre fonctions d'évaluation f *sont étudiées. L'enseignement que l'on en retire est qu'il faut s'assurer avec beaucoup d'attention que le théorème de la page 181 (sur une condition suffisante d'admissibilité) s'applique bien.*

On considère n agents, qui doivent effectuer n tâches, chaque agent se voyant assigner exactement une tâche. Le problème est que tous les agents ne sont pas également efficaces sur toutes les tâches. Si l'agent i effectue la tâche j, le coût (par exemple en temps) de cette assignation vaut D[i, j]. Étant donnée une matrice $D[1 \,..\, n, 1 \,..\, n]$ des coûts, on cherche à minimiser le coût de l'affectation, obtenu par addition des coûts sur chaque agent. Dans la suite, les agents sont notés en italique et les tâches en police droite. Le terme « affectation » est synonyme, pour cet exercice, du terme générique « candidat » employé dans l'introduction.

Par exemple, pour les quatre agents *1*, *2*, *3* et *4* et les tâches 1, 2, 3, et 4, la matrice de coûts D est la suivante :

	1	2	3	4
1	8	13	4	5
2	11	7	1	6
3	7	8	6	8
4	11	6	4	9

Ainsi, l'affectation $\{1 \to 4, 2 \to 3, 3 \to 2, 4 \to 1\}$ attribue la tâche 4 à l'agent *1*, la tâche 3 à l'agent *2*, la tâche 2 à l'agent *3* et la tâche 1 à l'agent *4*. Elle a pour coût (réel) :

$$f^*(\{1 \to 4, 2 \to 3, 3 \to 2, 4 \to 1\}) = D[1,4] + D[2,3] + D[3,2] + D[4,1] = 25.$$

L'objectif de l'exercice est donc de construire, selon la démarche PSEP, un algorithme qui produit l'une quelconque des affectations présentant un coût minimal.

69 - Q 1 **Question 1.** Quel est l'ensemble C de tous les candidats ? Quel est son cardinal ? Proposer une représentation et un procédé de séparation pour un ensemble de candidats.

69 - Q 2 **Question 2.** On recherche à présent une fonction d'évaluation f. L'exercice se prête bien à la décomposition de f en la somme des deux fonctions $g^\star$ (pour le coût réel de la partie de l'affectation déjà réalisée) et h pour une estimation optimiste du coût du reste de l'affectation. Une stratégie de coût uniforme consiste, pour h, à considérer le plus petit des coûts encore disponibles. Ainsi, pour l'exemple ci-dessus, si l'ensemble des candidats courant est représenté par le vecteur $[3, *, *, *]$, l'estimation retenue pour h est la plus petite des valeurs de D présente quand on supprime la première ligne et la troisième colonne, multipliée par le nombre de tâches restant à affecter, soit 6 (pour $D[2,4]$ ou $D[4,2]$, voir tableau 6.3) multiplié par 3 (il reste trois tâches à affecter). Montrer que h satisfait bien la condition d'admissibilité du théorème de la page 181. Fournir l'arbre de recherche PSEP pour cette fonction d'évaluation et pour la matrice D ci-dessus.

	1	2	3	4
1	8	13	4	5
2	11	7	1	6
3	7	8	6	8
4	11	6	4	9

Tab. 6.3 – *Tableau des coûts* D. *Les zones en gris clair sont les valeurs de* D *qui deviennent indisponibles lorsque la tâche* 3 *est affectée à l'agent 1.*

Question 3. Une meilleure solution (*a priori*) serait, pour chaque agent i qui reste à affecter, de prendre pour fonction heuristique h le plus petit coût encore disponible sur la ligne correspondante de D. Ainsi, pour le même exemple que dans la question précédente, pour l'agent *2* (resp. *3* et *4*), on prendrait 6 (resp. 7 et 6). Refaire la seconde question en appliquant cette fonction heuristique. 69 - Q 3

Question 4. Dans le but d'améliorer à nouveau la fonction heuristique h, on reprend la démarche de la question précédente en recherchant successivement le minimum pour chaque ligne restant à traiter, mais cette fois on supprime des recherches futures la colonne qui a produit ce minimum. Que peut-on dire de la fonction f définie à partir de cette fonction heuristique ? 69 - Q 4

Question 5. On cherche à appliquer une quatrième stratégie définie de la manière suivante. Pour les agents restant à affecter à une tâche, on recherche le minimum sur l'ensemble des cases de D encore disponibles (et non plus, comme dans la question précédente, le minimum sur la ligne). On supprime des recherches futures la ligne et colonne qui ont produit ce minimum. On réitère tant que cela reste possible. Montrer, à l'aide d'un contre-exemple, que le théorème de la page 181 ne s'applique pas. 69 - Q 5

Exercice 70. Le voyageur de commerce (le retour)

Cet exercice reprend, avec les mêmes hypothèses, l'exemple du voyageur de commerce traité dans l'introduction. Une troisième fonction d'évaluation est étudiée. Les résultats sont comparés avec ceux des précédentes solutions.

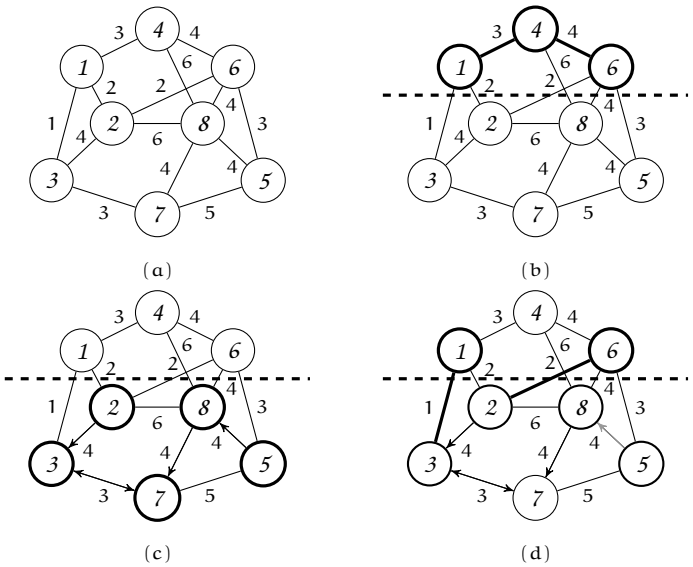
Le problème du voyageur de commerce a été traité dans l'exercice 56, page 156, du chapitre « Essais successifs ». Il a également été pris en exemple et développé de deux manières différentes (en appliquant successivement deux fonctions d'évaluation) dans l'introduction de ce chapitre. Dans le présent exercice, nous définissons une nouvelle fonction d'évaluation fondée sur une fonction heuristique f, plus fine que les deux autres.

On rappelle (voir section 6.1.4, page 182) que l'on part d'un graphe non orienté $G = (N, V, D)$ de n ($n \geqslant 2$) sommets, valué sur $\mathbb{R}_+$, et qu'il s'agit de découvrir, s'il en existe, un cycle hamiltonien de coût minimal débutant et aboutissant au sommet 1.

Dans la méthode du coût uniforme, (voir section 6.1.4, page 187) la valeur de la fonction d'évaluation f est constituée : i) du coût g^* de la chaîne déjà parcourue, ii) de la valeur de h, elle-même composée, d'une part d'un minorant du coût des chaînes hamiltoniennes éventuelles pour le sous-graphe G′ formé par les sommets n'apparaissant pas dans la chaîne déjà parcourue, d'autre part du coût du raboutement. Le coût du minorant est obtenu en appliquant uniformément le coût de l'arête la moins coûteuse à toute arête de G′ moins une (pour faire en sorte que les cycles soient ignorés).

Dans la version étudiée ici, on s'inspire de cette solution, mais, au lieu de prendre systématiquement l'arête la moins coûteuse du sous-graphe G′, on adopte la démarche du minimum local définie comme suit. On va calculer la somme des arêtes les moins coûteuses pour les n' sommets de G′, avant de retrancher le maximum de ces n' arêtes, afin d'éviter les cycles. De cette façon, la valeur obtenue est bien un minorant du coût des éventuelles chaînes hamiltoniennes de G′.

Le schéma suivant montre comment se calcule la valeur de $f([146******])$. La partie (a) montre un graphe non orienté valué, de huit nœuds. La partie (b) met en évidence, en gras, la chaîne déjà parcourue. Son coût $g^*([146******])$ s'élève à $3+4=7$. La partie (c) permet de connaître un minorant du coût de toute chaîne hamiltonienne du sous-graphe G′ défini par les nœuds $2, 3, 5, 7$ et 8. Le coût minimum de chaque sommet est celui de l'arête qui porte la flèche. Au total, on obtient un coût de $4+3+4+3+4-\max(\{4,3,4,3,4\}) = 14$. Enfin, la partie (d) montre comment se raboutent la chaîne déjà parcourue (schéma (b)) et le graphe G′, en prenant l'arête de coût minimal issue du sommet 1 (resp. 6) dont le coût est 1 (resp. 2). Au total $f([146******]) = 7+14+(1+2) = 24$.



70 - Q 1 **Question 1.** On considère le graphe (a) de la figure 6.7 dans lequel la partie en gras est la chaîne déjà parcourue. Fournir la valeur de f pour cette configuration. Faire de même pour le graphe (b).

70 - Q 2 **Question 2.** On considère à présent le graphe (c) de la figure 6.7. Construire l'arbre de recherche, tout d'abord avec la méthode du coût uniforme (voir page 187), puis avec la méthode du minimum local décrite ci-dessus.

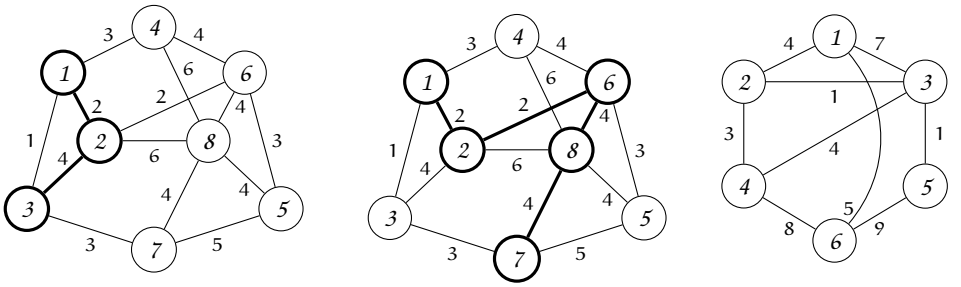


Fig. 6.7 – *Les trois cas de figure étudiés.*

Exercice 71. Le taquin

Le taquin est un jeu solitaire où il s'agit d'atteindre une situation finale donnée à partir d'une situation initiale, en un minimum de coups. En général, la résolution informatique se fait en utilisant l'algorithme A, variante de PSEP adaptée aux situations où l'ensemble des états considérés est organisé en graphe. Dans cet exercice, nous appliquons la méthode PSEP selon les principes exposés dans l'introduction. Un point mérite d'être souligné : le cardinal de l'ensemble des candidats est ici infini dénombrable.*

Le taquin est un jeu constitué d'une grille de taille $n \times n$ (typiquement $n = 4$) contenant $(n^2 - 1)$ tuiles, numérotées de 1 à $(n^2 - 1)$, qui peuvent glisser horizontalement ou verticalement en utilisant l'emplacement laissé libre. La figure 6.8 présente deux exemples de configuration pour $n = 4$:

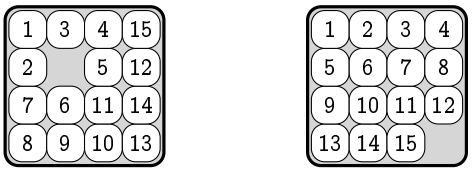
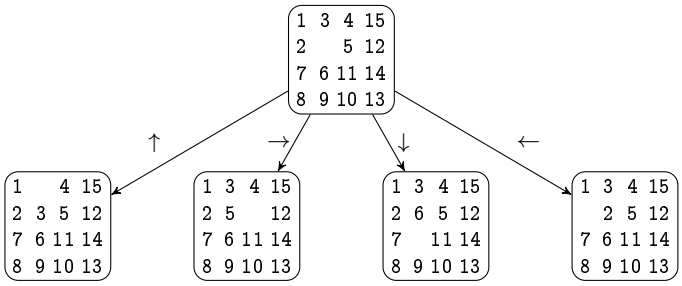


Fig. 6.8 – *Le taquin : deux exemples de configuration*

À partir de la configuration (a), on peut atteindre, en un seul coup, les quatre configurations apparaissant à la base du schéma ci-dessous (les flèches représentent le sens du déplacement du trou) :



Le but du jeu est, partant d'un configuration donnée (par exemple la configuration (a) de la figure 6.8), de déterminer la séquence de déplacements la plus courte possible permettant d'atteindre la configuration canonique (b). La question de savoir s'il existe une séquence finie de déplacements entre la configuration de départ et celle d'arrivée n'est pas anodine puisque seule une partie des configurations possibles permet de rejoindre la configuration finale (b). Pour décider si une configuration est ou non soluble, on considère tout d'abord la distance de Manhattan d (cf. question 1 ci-après) entre la position initiale et la position finale $((n, n))$ de la case vide (pour le taquin de la figure 6.8, $d = 4$). On considère ensuite la permutation des entiers $1 \,..\, n^2$ obtenue en plaçant bout à bout les lignes de la configuration initiale et en remplaçant la case vide par la valeur n^2 (16 dans le schéma (a) de la figure 6.8). Soit s le nombre (pair ou impair) d'inversions dans cette permutation (voir exercice 101, page 264). On montre [8] que la configuration est soluble si et seulement si d a la même parité que s. Dans le cas (a) de la figure 6.8, s est impair (on dénombre 37 inversions) et d est pair, la configuration est donc insoluble. Notons que le calcul de cette précondition que nous considérerons satisfaite dans la suite, s'effectue en évaluant $\Theta(n^4)$ conditions.

Face à un tel problème, la tentation est grande de considérer que l'espace d'états à prendre en compte est celui des configurations du taquin. Ce type d'approche se prête cependant mal à l'application de la démarche PSEP, une même configuration risquant de se retrouver sur deux branches différentes de l'arbre de recherche. Il est préférable de considérer que l'ensemble C des candidats est l'ensemble des *séquences* permettant de passer de la situation initiale du taquin à la situation canonique. Pour un exemple hypothétique, C pourrait être représenté par l'ensemble $\{\langle\leftarrow,\leftarrow,\uparrow\rangle, \langle\downarrow,\leftarrow,\downarrow,\uparrow,\rightarrow\rangle, \langle\downarrow,\rightarrow,\downarrow,\uparrow,\uparrow\rangle, \ldots\}$. Ici, compte tenu des boucles qu'il est possible de parcourir, cet ensemble est infini dénombrable, mais, comme on le verra, cette caractéristique ne présente pas de conséquences néfastes sous réserve que la précondition évoquée précédemment soit satisfaite. Une *solution* est un *candidat* minimisant le nombre de déplacements. Ainsi qu'il est préconisé dans l'introduction de ce chapitre, les deux premières étapes, dans la réalisation d'un algorithme PSEP, consistent à décider de la représentation d'un sous-ensemble de candidats et, pour un sous-ensemble donné, de proposer une stratégie de partitionnement.

71 - Q 1 **Question 1.** Comment peut-on déterminer si une séquence de déplacements est ou non un candidat [9] ? Que peut-on dire d'une séquence qui n'est pas un candidat [9] ? Comment peut-on partitionner un ensemble de candidats en plusieurs sous-ensembles non vides ?

8. Voir edouardlucas.free.fr/fr/liste_des_oeuvres.htm
9. Ou, pour être exact : un *ensemble* ne comprenant qu'un seul candidat.

Préoccupons-nous à présent de la fonction d'évaluation f. Pour un sous-ensemble non vide E de candidats, cette fonction peut se décomposer comme suit : $f(E) = g^\star(E) + h(E)$. $g^\star(E)$ est le coût réel (c'est-à-dire le nombre de déplacements ou encore la longueur de la séquence E). La fonction heuristique $h(E)$ est une estimation minorante du nombre de déplacements qu'il faut rajouter à E pour obtenir un candidat. Un choix naïf consiste à prendre $h = 0$. Le parcours de l'arbre se fait alors en largeur d'abord. Un choix plus judicieux pour h est celui de la distance de Manhattan. Pour une tuile w quelconque d'une configuration I du taquin, la distance de Manhattan est la somme des déplacements horizontaux et verticaux nécessaires à w pour atteindre sa position dans la configuration finale F, soit $(|w_{h_I} - w_{h_F}| + |w_{v_I} - w_{v_F}|)$ avec w_{h_X} et w_{v_X} les coordonnées horizontale et verticale de la tuile w en position X (initiale ou finale). La distance de Manhattan d'une configuration est somme de la distance de Manhattan de ses $(n^2 - 1)$ tuiles, soit : $\sum_{w=1}^{n^2-1} |w_{h_I} - w_{h_F}| + |w_{v_I} - w_{v_F}|$.

Question 2. Montrer que cette fonction heuristique répond à la condition du théorème de la page 181. Quelle est la distance de Manhattan de la configuration suivante ? 71 - Q 2

4	1	3
7	2	5
	8	6

La disponibilité de la fonction heuristique h permet maintenant de déterminer facilement si une séquence est ou non un candidat. De quelle façon ? Fournir l'arbre de recherche correspondant, ainsi que la solution trouvée.

Exercice 72. Le plus proche voisin ○ ⁝

Les caractéristiques de cet exercice nous obligent à une mise en garde. De par le caractère très restrictif de ses conditions d'utilisation et l'existence d'une solution naïve aux performances acceptables, cet exercice a pour seul objectif de mettre en pratique la démarche PSEP, sans aucune ambition applicative. Par ailleurs, cet exercice illustre deux caractéristiques peu courantes dans les mises en œuvre de l'approche PSEP : d'une part, l'ensemble des candidats est ici défini en extension (alors qu'en général il est défini en compréhension) et, d'autre part, la fonction d'évaluation f *ne se décompose pas en une somme des fonctions* $g^\star$ *et* h.

Soit C un ensemble non vide de points du plan $\mathbb{R}^2$ et soit a un point de ce plan ($a \notin C$) « éloigné » des points de C (dans un sens précisé ci-dessous). On cherche à identifier l'un quelconque des points de C le plus proche de a, au sens de la distance euclidienne d. Autrement dit, on recherche un point c_0 tel que :

$$d(c_0, a) = \min_{c \in C}(d(c, a)).$$

Une solution simple consiste à coder la formule ci-dessus comme une recherche séquentielle. Pour des raisons purement didactiques, notre choix est différent.

Soit R un rectangle aux côtés parallèles aux axes, qui englobe tous les points de C, et soit D le disque circonscrit à R. On impose en outre (précondition « d'éloignement ») que

n°	coord.	n°	coord.
1	$(5,7)$	5	$(7,3)$
2	$(1,6)$	6	$(1,2)$
3	$(3,5)$	7	$(5,2)$
4	$(6,5)$	8	$(2,1)$

Tab. 6.4 – *Tableau des coordonnées des huit points de l'exemple de la figure 6.9*

$a \notin D$. Soit m le centre du disque et soit r son rayon. La figure 6.9 montre un exemple d'une telle situation (R est ici un carré).

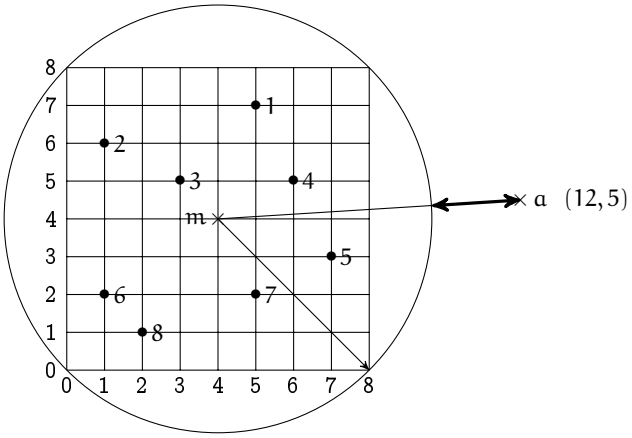


Fig. 6.9 – *Exemple avec un ensemble de huit points candidats. La distance entre* a *et le disque est la longueur de la double flèche en gras. Les points sont numérotés de* 1 *à* 8. *Les coordonnées précises apparaissent dans le tableau 6.4. La solution unique est le point numéro* 5.

Comme mentionné en page 189 de l'introduction, deux aspects doivent être abordés en priorité : la représentation de l'ensemble de candidats (de l'ensemble des points pour cet exercice) et la stratégie de séparation. Ci-dessus, nous suggérons de représenter un ensemble de candidats par un rectangle. Celui-ci sera ouvert à droite et en haut (les points situés sur ces frontières n'appartiennent donc pas au rectangle) et fermé en bas et à gauche. Ce choix permet de satisfaire la contrainte de partitionnement imposée par la méthode PSEP. Quant à la séparation, une solution consiste à éclater un rectangle en quatre rectangles de même dimension, en le coupant en deux par la longueur et par la largeur.

Soit S un tel rectangle. Une ébauche de la fonction d'évaluation f consiste à assimiler $f(S)$ à la distance entre le point de référence a et le disque circonscrit à S : $f(S) = (d(m,a) - r)$.

72 - Q 1 **Question** 1. La fonction f telle qu'ébauchée ci-dessus ne garantit pas l'admissibilité de l'algorithme PSEP. Pourquoi ? Raffiner cette fonction, ainsi que la stratégie de séparation ; puis démontrer que la version de f qui en résulte est correcte.

Question 2. Appliquer l'algorithme PSEP en utilisant la fonction d'évaluation de la question précédente et fournir l'arbre de recherche pour l'exemple de la figure 6.9 page 196 et du tableau 6.4. 72 - Q 2

Question 3. Proposer une seconde fonction d'évaluation. La discuter par rapport à la première. 72 - Q 3

CHAPITRE 7

Algorithmes gloutons

Un gourmet est un glouton qui se domine.

(Francis Blanche)

7.1 Introduction

7.1.1 Présentation

Comme les algorithmes à « essais successifs » (voir chapitre 5) ou PSEP (voir chapitre 6), les algorithmes gloutons (ou voraces, en anglais *greedy*) s'appliquent à des problèmes *a priori* combinatoires. Cependant, contrairement à ceux de ces deux familles, les algorithmes gloutons ont la particularité d'être déterministes : ils ne remettent jamais en cause les choix qu'ils effectuent.

Un algorithme glouton peut être vu comme une descente directe de la racine à une feuille dans l'arbre parcouru par la technique des « essais successifs ». Plus exactement, à chaque nœud de cet arbre, un algorithme glouton applique une règle de choix permettant de sélectionner une valeur du domaine de la variable considérée, avant de passer au nœud suivant, jusqu'à rencontrer une feuille.

On pourrait penser qu'un algorithme glouton a peu de chances de trouver la solution recherchée, puisqu'en réalité il n'en construit qu'une seule. Il existe cependant des problèmes non triviaux pour lesquels un algorithme glouton trouve toujours une (la) solution ; dans ce cas, on parle d'algorithme glouton *exact*, ou *optimal* s'il s'agit d'un problème d'optimisation. Notons que pour certains problèmes d'optimisation dont on ne connaît pas de solution en termes d'algorithme glouton optimal, on peut utiliser un algorithme glouton dit *approché*. Pour autant que le résultat délivré soit jugé suffisamment proche de l'optimal recherché, cette approche de faible coût est préférée à un algorithme non glouton atteignant l'optimal à un prix élevé, voire prohibitif. Par la suite, nous nous intéresserons essentiellement aux algorithmes gloutons exacts (exercices 82 à 85) ou optimaux (exemple de la section 7.1.3 et exercices 73 à 81).

L'avantage des algorithmes gloutons est évidemment leur faible complexité temporelle (polynomiale), puisqu'ils sont fondés sur une itération (voir page 64). De ce fait, l'attention sera plus portée sur le caractère exact ou optimal des algorithmes traités que sur leur complexité.

Une caractéristique notable est qu'ici plus qu'ailleurs la découverte d'une solution (c'est-à-dire de la règle de choix glouton) est d'abord une affaire d'intuition. Il est donc essentiel de compléter celle-ci par une preuve du caractère exact ou optimal de l'algorithme qui est construit (voir section suivante), ce qui peut se révéler parfois difficile, alors qu'exhiber

un contre-exemple suffit à montrer que la stratégie de choix glouton envisagée ne conduit pas à un algorithme exact ou optimal.

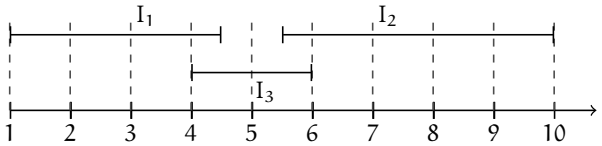
7.1.2 Comment démontrer qu'un algorithme glouton est exact ou optimal ?

Comme il a été dit auparavant, un algorithme glouton se construit comme une boucle et obéit donc à la méthodologie mise en évidence au chapitre 3. Le plus souvent, on va intégrer la preuve du caractère exact ou optimal à la construction de l'itération (en tant qu'élément de la spécification) et donc dans la postcondition et l'invariant. Pour diverses raisons (notamment pédagogiques, mais voir aussi l'exercice 85, page 237), il arrive qu'on soit amené à faire une preuve *a posteriori*, c'est-à-dire après la construction de la solution. Les problèmes sans critère d'optimalité ne posant pas de difficulté spécifique, nous nous attachons dans la suite au cas des problèmes d'optimisation. L'intégration de la preuve à la construction vise à établir que l'algorithme fait « la course en tête » du début à la fin au sens du critère d'optimalité. Dans la technique de la preuve a posteriori, la version la plus pratiquée est celle de « l'argument de l'échange ». Son point de départ est un algorithme construit à partir d'une spécification qui n'intègre pas la propriété d'optimalité. Il s'agit alors, à partir d'une version quelconque Q de la solution, de montrer qu'elle n'est jamais meilleure qu'une version gloutonne G. D'autres variantes existent, comme celle de la technique de transformation qui consiste à considérer, non pas une version quelconque Q, mais une version optimale O fictive, et à montrer que O peut se transformer en la solution gloutonne G, tout en restant optimale, ce qui prouve le caractère optimal de G.

Nous allons, sur un exemple commun, illustrer l'application de la technique de la course en tête et de celle de la preuve *a posteriori*.

7.1.3 Un exemple : la répartition des tâches sur un photocopieur

Soit n tâches de photocopie ($n \geqslant 0$), chacune pouvant être réalisée seulement dans l'intervalle (fermé, non vide) de temps $I_i = (d_i, f_i)$, pour $i \in 1\,..\,n$. Les tâches ne peuvent être fractionnées, ce qui signifie que la personne mandatée pour réaliser la tâche i n'est libre qu'entre les moments d_i et f_i et que le temps nécessaire consacré à la tâche s'élève à $(f_i - d_i)$. Le but visé est de planifier les passages sur la photocopieuse afin de réaliser le plus de tâches possible, sachant que la photocopieuse n'accepte qu'une seule tâche à un instant t donné. Considérons par exemple le cas de figure ci-dessous, où les trois tâches envisagées sont matérialisées par les trois intervalles suivants : $I_1 = (1, 4.5)$, $I_2 = (5.5, 10)$ et $I_3 = (4, 6)$. Le nombre maximal de tâches réalisables vaut 2. En effet, l'unique façon d'obtenir ce résultat est de réaliser la tâche I_1, puis la tâche I_2. Il est impossible de réaliser les trois, ni d'effectuer les tâches I_1 et I_3 (qui sont dites incompatibles puisque leurs intervalles se recouvrent), pas plus que les tâches I_3 et I_2.



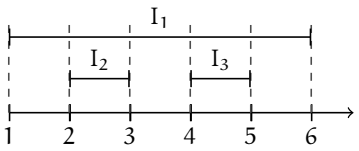
Dans la suite, si I_k est un intervalle, on note $d(I_k)$ (resp. $f(I_k)$) l'origine (resp. l'extrémité) de I_k. Un planning candidat G est constitué d'une liste d'intervalles (compatibles) $\langle g_1, \ldots, g_k \rangle$ telle que, pour tout i, $d(g_i) < f(g_i)$ et $f(g_i) \leqslant d(g_{i+1})$.

7.1.4 La méthode de la course en tête

Ainsi que nous l'avons dit ci-dessus, cette technique intègre, dès le départ, le critère d'optimalité dans la spécification de l'algorithme. Comment concevoir une stratégie gloutonne pour le problème ci-dessus ? Il faut choisir la première tâche selon un certain critère, puis, parmi celles qui restent à réaliser, la seconde selon le même critère, etc. La stratégie gloutonne interdit de remettre en cause les choix réalisés. Avant d'essayer des stratégies simples, rappelons qu'il s'agit de maximiser le nombre de tâches effectuées sur le photocopieur.

— La première idée est de choisir les tâches par durée décroissante. Dans le cas du schéma ci-dessus, cela conduit à choisir la tâche I_3 en premier. Ensuite, aucune tâche ne peut plus être planifiée et l'algorithme s'arrête sans avoir trouvé la meilleure solution. Nous avons donc prouvé, par un contre-exemple, que cet algorithme n'est pas un glouton optimal.

— On pourrait choisir les tâches par ordre croissant de leur heure de départ. Dans notre exemple, on planifierait la tâche I_1, puis la tâche I_2 (on écarte la tâche I_3, car elle recouvre partiellement la tâche I_1), et l'optimum serait atteint. Mais cela ne prouve pas qu'en général cet algorithme est un glouton optimal. Un contre-exemple est facile à trouver, comme le montre le schéma ci-dessous avec trois tâches définies par les intervalles $I_1 = (1,6)$, $I_2 = (2,3)$ et $I_3 = (4,5)$.



— On peut aussi constater que choisir la tâche qui va s'achever en premier laisse un maximum de temps pour placer d'autres tâches à sa suite, avec l'espoir que cela se traduise par la planification d'un maximum de tâches. Sur les deux exemples ci-dessus, cet algorithme fonctionne et nous butons sur la mise en évidence d'un contre-exemple. Notre objectif est maintenant de construire le programme correspondant, tout en démontrant l'optimalité de cette stratégie.

Les deux principales structures de données utilisées dans les algorithmes gloutons sont d'une part une file d'entrée contenant les données à traiter (souvent il s'agit d'une file de priorité, voir section 1.7, page 33), et d'autre part une file de sortie qui rassemble les résultats. Dans le cas qui nous intéresse, ces entités peuvent toutes deux être mises en œuvre par une file FIFO (First In, First Out : premier entré, premier sorti, voir section 1.8, page 34). Une procédure spécifique au problème traité ici est ajoutée au jeu d'opérations des files FIFO :

> **procédure** *InitFifo*(T, P : **modif**) : si T est un tableau d'intervalles, cette opération, qui est une surcharge de la version originale de *InitFifo*, place dans la file P les intervalles de T triés par ordre croissant sur les extrémités des intervalles.

Construction de l'algorithme Ci-dessous, F est la file d'entrée et R la file de sortie. Nous réalisons la construction en appliquant l'heuristique du travail réalisé en partie (voir chapitre 3). La solution est obtenue à l'issue de deux tentatives.

Première tentative

Invariant Imaginons que les i premiers intervalles ont été traités.

1. R_i est la configuration de la file de sortie R lorsque la stratégie gloutonne a été appliquée à la liste d'intervalles $\langle I_1, \ldots, I_i \rangle$. $R_i = \langle g_1, \ldots, g_k \rangle$ est un planning optimal pour la liste $\langle I_1, \ldots, I_i \rangle$.
2. F_i est la configuration de la file d'entrée F lorsque la stratégie gloutonne a été appliquée à la liste d'intervalles $\langle I_1 \ldots, I_i \rangle$, soit $F_i = \langle I_{i+1}, \ldots, I_n \rangle$.

Exemple La figure 7.1 illustre une telle situation lorsque $i = 5$. On a alors $R_5 = \langle g_1, g_2 \rangle = \langle I_1, I_3 \rangle$. Les intervalles en pointillés sont incompatibles avec ceux de R_5. On note que la liste $\langle I_1, I_4 \rangle$ est aussi une solution. Elle n'est cependant pas obtenue par la stratégie gloutonne définie ci-dessus.

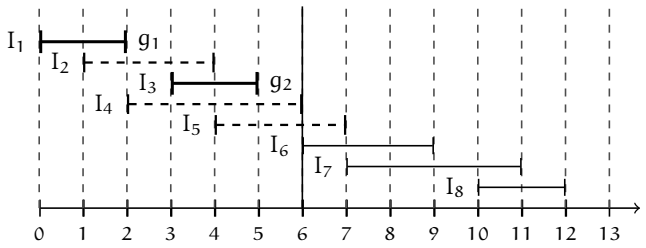
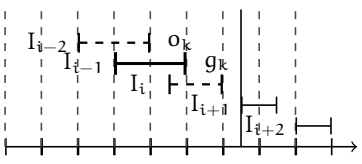


Fig. 7.1 – *Traitement « glouton » de la liste* $\langle I_1, \ldots, I_8 \rangle$. *Situation atteinte lorsque* $i = 5$. $F_5 = \langle I_6, I_7, I_8 \rangle$ *et* $R_5 = \langle g_1, g_2 \rangle = \langle I_1, I_3 \rangle$.

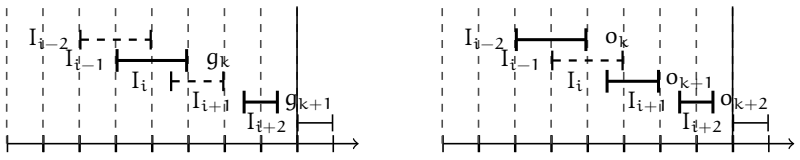
Condition d'arrêt La file F est vide. La conjonction de cette proposition et de l'invariant implique que R est une solution optimale.

Progression On déplace la tête t de la liste F en queue de la liste R, et on supprime de F tous les intervalles incompatibles avec l'intervalle t. Cette dernière action exige une boucle. Sa construction n'est pas développée ici. En l'appliquant à l'exemple de la figure 7.1, on obtient $R_6 = \langle I_1, I_3, I_6 \rangle$ et $F_6 = \langle I_8 \rangle$. L'intervalle I_7, incompatible avec I_6, est supprimé de la file F.

Toutes les propriétés de l'invariant sont trivialement rétablies, à l'exception de l'optimalité. Est-on assuré que la progression ne puisse faire perdre son caractère optimal à la solution gloutonne ? Autrement dit, peut-on exclure la situation suivante, où l'on dispose d'une part de la solution gloutonne $R = \langle g_1, \ldots, g_k \rangle$ et d'autre part d'une solution optimale $R' = \langle o_1, \ldots, o_k \rangle$, comme le montre le schéma ci-dessous :



pour atteindre, à l'étape suivante (voir schémas ci-dessous), soit la situation (a), où la solution gloutonne s'est allongée d'un intervalle, soit la situation (b) où la solution R' s'est, quant à elle, allongée de *deux* intervalles. Cela montrerait que la solution gloutonne n'est pas optimale.



(a) (b)

La transition vers la situation (b) n'est rendue possible que parce que $f(o_k) < f(g_k)$. Intuitivement, cela ne peut survenir puisque la solution gloutonne prend toujours l'intervalle qui s'achève en premier. Pour montrer cette propriété avec la rigueur nécessaire, nous allons renforcer l'invariant en y introduisant cette propriété. Cela nous conduit à une seconde tentative de construction.

Seconde tentative

Invariant On renforce la précédente version de l'invariant en modifiant le premier point comme suit.

1. R_i est la configuration de la file de sortie R lorsque la stratégie gloutonne a été appliquée à la liste d'intervalles $\langle I_1, \ldots, I_i \rangle$. $R_i = \langle g_1, \ldots, g_k \rangle$ est donc une solution optimale pour la liste $\langle I_1, \ldots, I_i \rangle$, et il n'existe aucune autre solution optimale s'achevant avant $f(g_k)$.

Condition d'arrêt La condition d'arrêt est inchangée. Sa conjonction avec l'invariant implique bien le but visé.

Progression Nous nous focalisons sur le maintien de l'optimalité de la solution gloutonne. La conjonction de l'invariant et de la négation de la condition d'arrêt entraîne qu'il existe au moins un intervalle I_{i+1} à traiter. Soit $R' = \langle o_1, \ldots, o_k \rangle$ une solution optimale quelconque pour la liste $\langle I_1, \ldots, I_i \rangle$. Le choix d'un nouvel intervalle compatible o_{k+1} entraîne que $f(o_k) \leqslant d(o_{k+1})$. Par ailleurs, d'après l'invariant, $f(g_k) \leqslant f(o_k)$. Au total, par transitivité, on obtient $f(g_k) \leqslant d(o_{k+1})$. L'intervalle o_{k+1} est donc compatible avec g_k. Mais, la stratégie gloutonne impose de choisir g_{k+1} de sorte que $f(g_{k+1})$ soit le plus petit possible – et en particulier inférieur ou égal à $f(o_{k+1})$ –, ce qui rétablit l'invariant.

Initialisation Pour instaurer l'invariant lorsque $i = 0$, il suffit que R soit vide et F soit constituée de la liste triée sur les extrémités croissantes des intervalles à considérer.

Terminaison Au moins une requête de F est éliminée à chaque pas de progression ; en conséquence $|F|$ (le nombre de requêtes de la file F) est une fonction de terminaison convenable.

On en déduit l'algorithme suivant :

1. **constantes**
2. $\quad n \in \mathbb{N}_1$ **et** $n = \ldots$ **et** $T \in 1..n \to (\mathbb{R}_+ \times \mathbb{R}_+)$ **et** $T = [\ldots]$
3. **variables**
4. $\quad F \in \text{FIFO}(\mathbb{R}_+ \times \mathbb{R}_+)$ **et** $R \in \text{FIFO}(\mathbb{R}_+ \times \mathbb{R}_+)$ **et** $t \in (\mathbb{R}_+ \times \mathbb{R}_+)$

```
5.  début
6.     InitFifo(T, F) ; /% Création de F à partir du tableau T des intervalles %/

7.     InitFifo(R) ;
8.     tant que non EstVideFifo(F) faire
9.        t ← TêteFifo(F) ; SupprimerFifo(F) ;
10.       AjouterFifo(R, t) ; /% Déplacement de t en queue de la file R %/
11.       tant que non(EstVideFifo(F) ou sinon d(TêteFifo(F)) ⩾ f(t)) faire
12.          SupprimerFifo(F)
13.       fin tant que
14.    fin tant que
15. fin
```

Dans cet algorithme, l'opération la plus coûteuse est le tri réalisé par la procédure *InitFifo*, qui est supposé en $\mathcal{O}(n \cdot \log_2(n))$. C'est aussi la complexité de cet algorithme.

La démonstration par la méthode de la « course en tête » est utilisée d'une manière plus générale dans la théorie des *matroïdes*. En deux mots, un matroïde est une collection de sous-ensembles d'un ensemble fini dont chaque élément est pondéré positivement ; cette collection doit par définition vérifier certains axiomes. Sous ces axiomes, on peut définir le meilleur sous-ensemble d'un matroïde : celui dont la somme des poids de ses éléments est maximale, parmi ceux, dits *indépendants*, qui respectent une certaine propriété. Les matroïdes ont des propriétés particulières, dues à leurs axiomes, telles que l'on peut démontrer qu'il existe un algorithme glouton pour découvrir ce meilleur sous-ensemble. La justesse de cet algorithme est prouvée par la méthode de la « course en tête ».

Quand on est capable d'identifier le problème d'optimisation que l'on a à résoudre avec le problème de recherche du meilleur sous-ensemble indépendant d'un matroïde, on a directement un algorithme glouton exact sous la main. Mais cette façon de faire se heurte à deux écueils :

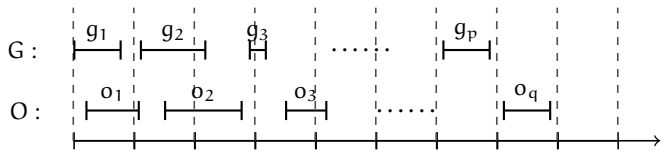
— Cette identification peut être impossible. Il existe des problèmes d'optimisation possédant une solution gloutonne dont la solution ne peut être obtenue en employant la méthode des matroïdes.

— Cette identification peut être tellement difficile à faire qu'il est plus simple (si l'on pense qu'il existe une solution gloutonne) de tenter de démontrer directement l'optimalité de l'algorithme glouton par la méthode de la « course en tête ».

Pour en savoir plus sur les matroïdes et la théorie des algorithmes gloutons, nous conseillons de consulter le livre [17].

7.1.5 Preuve *a posteriori* : technique de la transformation

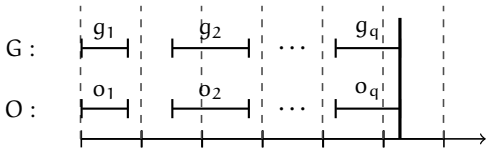
Reprenons l'exemple du photocopieur. Conformément aux principes à la base de cette technique, nous supposons ici que l'algorithme est conçu sans tenir compte de l'optimalité et nous réalisons la preuve de celle-ci *après coup*. Soit, pour un ensemble de tâches donné, $G = \langle g_1, \ldots, g_p \rangle$ les p requêtes sélectionnées par l'algorithme glouton ci-dessus, et $O = \langle o_1, \ldots, o_q \rangle$ les q requêtes d'une solution optimale quelconque ($p \leqslant q$, puisque O est optimal) :



Nous allons montrer, par induction sur q, qu'il est possible de transformer la liste O en la liste G sans lui faire perdre son statut de solution optimale (ce qui montrera du même coup que G peut s'identifier à O, et donc que la liste G est aussi optimale).

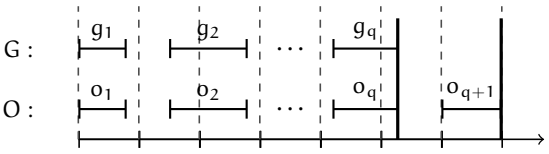
Base Pour $q = 0$, on a également $p = 0$ puisque $q \geqslant p$. On en déduit dans ce cas que $G = O$ et que G est optimal.

Hypothèse d'induction Pour $q \geqslant 0$, les requêtes G et O sont identiques, et $p = q$:



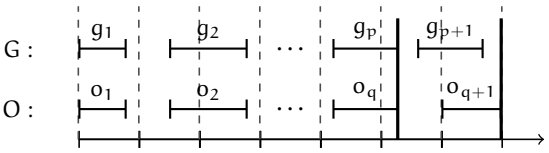
Induction Deux cas de figure sont à considérer. Le premier se caractérise par l'absence de l'élément g_{q+1} dans G (qui s'achève donc en g_q), le second par l'existence de g_{q+1}. Remarquons que la situation où g_{q+1} existe mais pas o_{q+1} est à écarter d'emblée puisqu'elle viole la condition $q \geqslant p$.

1. Le premier cas est illustré par le schéma ci-dessous :

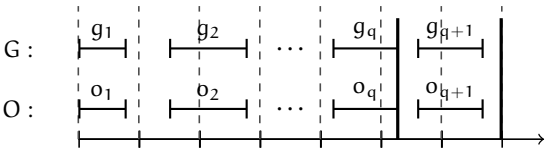


Cependant, ce cas doit être écarté car, conformément à sa stratégie, l'algorithme glouton aurait obligatoirement inclus la requête o_{q+1} dans sa liste G.

2. Le second cas, illustré ci-dessous,



est celui où les q premiers éléments de G et de O sont identiques (c'est l'hypothèse d'induction) et où la date de fin de g_{q+1} est antérieure ou égale à celle de o_{q+1} (en raison de la stratégie gloutonne utilisée). Pour retrouver l'hypothèse d'induction, il suffit de remplacer la requête o_{q+1} par g_{q+1}. C'est possible sans affecter l'optimalité de O :



Nous avons montré que l'on peut transformer O en G tout en préservant son caractère optimal. On en conclut que G, solution gloutonne, était déjà optimale.

7.2 Ce qu'il faut retenir des méthodes gloutonnes

Pour résumer, on peut considérer qu'un algorithme glouton se fonde sur les trois ingrédients suivants :

— deux structures de données : en général, une file d'entrée (souvent une file de priorité) et une file de sortie (le plus souvent une file FIFO), dans laquelle viennent se placer les constituants de la solution,

— une stratégie de traitement des éléments qui sont extraits de la file d'entrée, avant de rejoindre (ou pas !) la file de sortie,

— soit une preuve que l'algorithme fournit bien une solution exacte ou optimale, soit un contre-exemple.

On peut proposer le code générique suivant, dans lequel F est la file d'entrée, R la file de sortie, et t et t′ deux variables auxiliaires (les identificateurs des opérations ne véhiculent aucune hypothèse sur le type de file employé).

1. *InitFileEntrée*(F) ; /% *Création de la file d'entrée* %/
2. *InitFileSortie*(R) ; /% *Initialisation de la file de sortie* %/
3. **tant que non** *EstVideFileEntrée*(F) **faire**
4. t ← *TêteFileEntrée*(F) ;
5. *SupprimerFileEntrée*(F) ;
6. *TraitementÉlément*(t, t′) ;
7. *AjouterFileSortie*(R, t′)
8. **fin tant que**

Cela ne constitue qu'un cadre général. Ainsi que nous l'avons déjà dit, la file d'entrée est le plus souvent une file de priorité mais, dans la suite, on rencontre des variantes où les priorités sont figées dès le départ, ou encore où l'on se limite à prendre en compte des entiers successifs, ou à supprimer d'un seul coup plusieurs éléments de la file. La file de sortie est en général une file FIFO. Cependant, dans certains cas, il peut être nécessaire de modifier l'ordre ou la nature des éléments déjà introduits. Quant à la stratégie de traitement des éléments, elle est rarement aussi simple qu'un transfert de la file d'entrée vers la file de sortie : un calcul itératif, une ventilation ou une sélection des éléments est le plus souvent nécessaire. Le lecteur aura compris qu'une solution gloutonne à un problème particulier se présente comme une variation plus ou moins substantielle autour de cet algorithme.

7.3 Exercices

Exercice 73. À la recherche d'un algorithme glouton ⚇ •

Ce problème a déjà été étudié dans l'introduction du chapitre « essais successifs » (voir chapitre 5). Il s'agit ici de tester plusieurs stratégies gloutonnes.

On considère un tableau T[1 .. n] d'entiers positifs, avec n pair, trié par ordre croissant. On désire recopier T dans deux sacs S_1 et S_2, de même taille $n/2$ et de sommes respectives Som1 et Som2, de sorte que l'on ait $(\text{Som1} \leqslant \text{Som2})$ et $(\text{Som2} - \text{Som1})$ minimal. Plus

précisément, en supposant que T représente la file d'entrée, la postcondition de l'algorithme est constituée des quatre conjoints suivants :

1. Les sacs ont même cardinal : $|S_1| = |S_2|$.
2. Les sommes Som1 et Som2 sont telles que $(\text{Som1} \leqslant \text{Som2})$.
3. La différence $(\text{Som2} - \text{Som1})$ est minimale.
4. La file d'entrée T est vide.
5. $S_1 \sqcup S_2$ est le sac des valeurs initiales de T.

Question 1. Sur la base de cette postcondition, imaginer trois stratégies gloutonnes pour traiter ce problème et montrer qu'elles ne produisent pas de solutions optimales. 73 - Q 1

Question 2. Cela prouve-t-il qu'il n'y a pas d'algorithme glouton pour ce problème ? 73 - Q 2

Exercice 74. Arbres binaires de recherche ○ •

Le problème traité ci-dessous est repris selon une approche par programmation dynamique à l'exercice 133 page 360. Ici, on s'en tient à une démarche gloutonne.

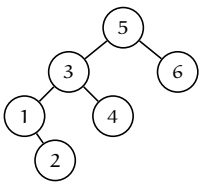
On dispose d'un ensemble de n valeurs entières $\{x_1, \ldots, x_n\}$. À chacune d'elles est attachée une probabilité $p(x_i)$. Afin de faciliter une recherche positive (recherche d'un élément dont on sait qu'il est présent dans l'ensemble), ces n valeurs sont enregistrées dans un arbre binaire de recherche (abr en abrégé). On définit le *coût* d'un tel abr A par :

$$\text{coût}(A) = \sum_{k=1}^{n} p(x_k) \cdot (d_k + 1), \tag{7.1}$$

où d_k est la profondeur du nœud x_k dans l'arbre A. La valeur $\text{coût}(A)$ est en fait l'espérance du nombre de comparaisons à effectuer pour trouver un élément présent dans l'arbre A. On cherche à construire, par une démarche gloutonne, l'abr de coût minimal.

Exemple La figure ci-dessous montre d'une part une liste de cinq valeurs x_i pondérées chacune par une probabilité $p(x_i)$, d'autre part un abr construit à partir de ces cinq valeurs.

x_i	1	2	3	4	5	6
$p(x_i)$	0.15	0.19	0.17	0.18	0.14	0.17



Selon la définition 7.1, page 207, le coût de cet arbre est de

$$1 \cdot p(5) + 2 \cdot p(3) + 2 \cdot p(6) + 3 \cdot p(1) + 3 \cdot p(4) + 4 \cdot p(2),$$

soit encore

$$1 \cdot 0.14 + 2 \cdot 0.17 + 2 \cdot 0.17 + 3 \cdot 0.15 + 3 \cdot 0.18 + 4 \cdot 0.19,$$

expression qui vaut 2.57.

74 - Q 1 **Question** 1. L'idée de placer les valeurs les plus probables le plus haut possible dans l'arbre semble favorable à une recherche optimale. Elle peut s'obtenir de manière gloutonne, soit en construisant l'arbre par insertion aux feuilles à partir d'une liste des valeurs triée sur les probabilités croissantes, soit au contraire en réalisant une insertion à la racine à partir d'une liste des valeurs triée sur les probabilités décroissantes. Une insertion aux feuilles d'une valeur v dans un abr se fait en insérant v dans le sous-arbre gauche ou droit selon la position relative de v par rapport à la racine, jusqu'à atteindre un arbre vide. Une insertion à la racine se fait en ventilant les valeurs de l'arbre initial dans deux sous-arbres, selon la valeur à insérer, puis en enracinant ces deux sous-arbres sur v. Donner l'arbre obtenu à partir du jeu d'essai ci-dessus, en appliquant la première de ces stratégies (l'insertion aux feuilles). Quel est son coût ?

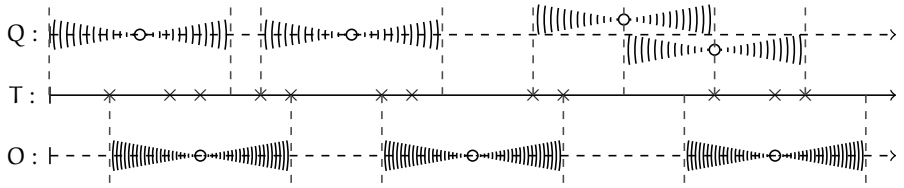
74 - Q 2 **Question** 2. En partant toujours du même jeu d'essai, montrer, par un contre-exemple, que cette stratégie n'est pas optimale.

Exercice 75. Les relais pour téléphones portables ⚇ •

Il s'agit d'un exercice voisin de l'exemple introductif (répartition des tâches sur un photocopieur). Il devrait donc être résolu sans difficulté par le lecteur.

On considère une longue route de campagne rectiligne, le long de laquelle sont dispersées des habitations. Chaque maison doit être reliée au réseau de téléphones portables par un opérateur. Une antenne-relais du réseau permet l'usage du téléphone dans une zone à distance fixe de $d/2$ autour du relais (toutes les antennes possèdent la même puissance). L'opérateur veut poser le moins d'antennes possibles pour « couvrir » toutes les maisons.

On peut formaliser le problème de la manière suivante. Un tableau T ($T \in 1..n \to \mathbb{R}_+$) représente la position de chaque maison sur le bord de la route. On cherche une liste S de valeurs réelles, ayant un nombre d'éléments p minimal, $S = \langle s_1, \ldots, s_p \rangle$, telle que pour toute valeur $T[i]$, il existe une valeur s_j vérifiant la contrainte $(|T[i] - s_j| \leqslant d)$. S est une liste optimale de positions d'antennes-relais. Dans le schéma ci-dessous, la ligne T représente la position des maisons, la ligne O matérialise une couverture optimale, avec trois relais, tandis que la ligne Q couvre bien toutes les maisons, mais avec quatre relais (et un recouvrement des deux relais de droite).



Question 1. Que peut-on dire des stratégies gloutonnes suivantes ? 75 - Q 1

a) On place un relais au niveau de chaque maison.

b) En progressant de la gauche vers la droite, on place un relais au niveau de chaque maison qui n'est pas encore couverte par les relais déjà posés.

Question 2. Proposer une troisième stratégie gloutonne dont on puisse espérer qu'elle soit optimale. 75 - Q 2

Question 3. En développant une démarche du type « course en tête » (voir section 7.1.2, page 200), construire, sur la base de la stratégie gloutonne de la question précédente, un algorithme glouton exact résolvant le problème. Quelle est sa complexité (on pourra se référer à celle d'un tri) ? 75 - Q 3

Question 4. On suppose maintenant que l'optimalité de la solution n'a pas été prouvée lors de la construction de l'algorithme. Montrer, par une méthode *a posteriori* (voir section 7.1.2, page 200), que la stratégie gloutonne précédente est optimale. 75 - Q 4

Exercice 76. Ordonner des achats dont le prix varie

Cet exercice est une illustration simple de la méthode de l'argument de l'échange. Le code de l'algorithme n'est pas demandé.

Le propriétaire d'un club de football veut acheter des joueurs auprès d'un centre de formation. La législation lui interdit d'en acheter plus d'un par mois. Le prix des joueurs est le même au départ – il est noté s – mais ce prix varie dans le temps, différemment selon les joueurs. C'est ainsi que, puisque le joueur j vaut s au départ, il vaudra $(s \cdot r_j^t)$ t mois plus tard. Le taux r_j dépend de la vitesse de progression du joueur, mais il est toujours strictement supérieur à 1. On se base sur un taux de progression estimé, connu au temps $t = 0$. Pour simplifier, on suppose aussi que ce taux est différent pour chaque joueur. L'objectif est de définir une stratégie gloutonne qui permet d'acheter un joueur par mois et qui assure d'acquérir les joueurs convoités en dépensant le moins possible. On suppose enfin que l'acheteur n'a pas de concurrent.

Question 1. Donner deux stratégies simples susceptibles de servir de base à un algorithme glouton. Laquelle des stratégies semble être la meilleure ? 76 - Q 1

Question 2. Montrer, en appliquant l'argument de l'échange (voir section 7.1.2, page 200), qu'elle est optimale. 76 - Q 2

Exercice 77. Diffusion d'information à moindre coût depuis une source : algorithmes de Prim et de Dijkstra

Les algorithmes de Prim et de Dijkstra présentent de nombreuses similitudes : même date de publication 1959 (même si le premier a été en fait « redécouvert » puisque initialement publié par V. Jarnik en 1930), algorithmes visant « des coûts minimaux » dans des graphes (non orientés et connexes pour le premier, orientés pour le second), et enfin (et surtout) algorithmes gloutons exacts. Ces deux algorithmes sont des cas d'école que l'on retrouve dans la littérature, aussi bien dans la rubrique « graphe » que dans la rubrique « glouton ». Le présent énoncé met l'accent sur la correction de la boucle qui les constitue. Une caractéristique de l'algorithme de Prim proposé réside dans le fait que son résultat ne fait pas appel à une file FIFO.

Deux problèmes voisins

On dispose d'un réseau de communication constitué de n sites. On envisage un premier problème (dont l'algorithme de Prim est une solution) où les sites sont connectés entre eux par des liens bidirectionnels de sorte que tout site peut atteindre tout autre (au sens de l'acheminement d'information) en empruntant une suite de liens. On distingue un site appelé source à partir duquel on souhaite diffuser une information vers tous les autres. Cependant, chaque lien a un coût d'utilisation propre (entier positif) et on recherche un acheminement tel que la somme des coûts associés aux liens empruntés pour atteindre l'ensemble des sites soit minimale. Dans le second problème (résolu par l'algorithme de Dijkstra), les sites sont connectés par des liens orientés et il se peut qu'il n'existe pas de cheminement entre un ou plusieurs couples de sites. L'objectif est le même que précédemment, mais on recherche un acheminement dont l'optimalité porte sur la somme des coûts associés aux liens empruntés pour atteindre chaque site. Hormis la nature du réseau, ces deux problèmes diffèrent donc essentiellement par la fonction à minimiser.

De nombreux autres problèmes conduisent à ces mêmes paradigmes de recherche de solution optimale : par exemple, la détermination du trajet optimal de câblage d'un ensemble de bâtiments ou le choix d'un système optimal d'irrigation de parcelles étagées depuis un point d'eau les dominant.

Avant de poursuivre, illustrons ces deux problèmes. Dans la figure 7.2, on donne dans la partie (b) le trajet optimal de diffusion de l'information pour un réseau ayant cinq sites, décrit dans la partie (a). Dans la partie (a) de la figure 7.3, on considère un réseau orienté calqué sur le précédent et on fournit pour chacun des sites 2 à 5, d'une part, la valeur du plus court chemin depuis la source (site 1) dans la partie (b), d'autre part, l'arbre associé au cheminement optimal dans la partie (c).

Remarque On observe que les résultats obtenus ne « coïncident » pas. En effet, selon la partie (b) du premier schéma, l'acheminement d'une information du site 1 au site 4 (resp. 5) coûte 7 (resp. 9) et n'est donc pas minimal, alors que le coût global d'acheminement (au sens du premier problème) pour le second schéma a pour valeur 12 (arcs $(1,2)$, $(1,3)$, $(1,4)$ et $(4,5)$) et n'est pas minimal lui non plus. Ceci justifie (s'il en était besoin) l'existence de deux problèmes bien distincts et donc d'algorithmes différents pour les résoudre.

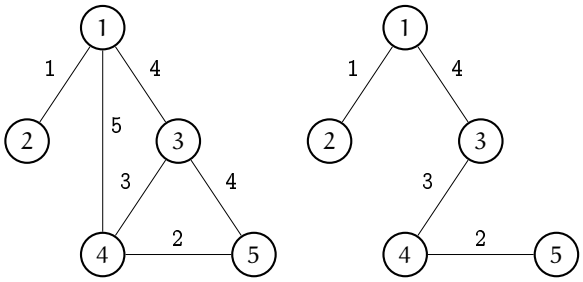


Fig. 7.2 – *Un réseau et le trajet globalement optimal associé*

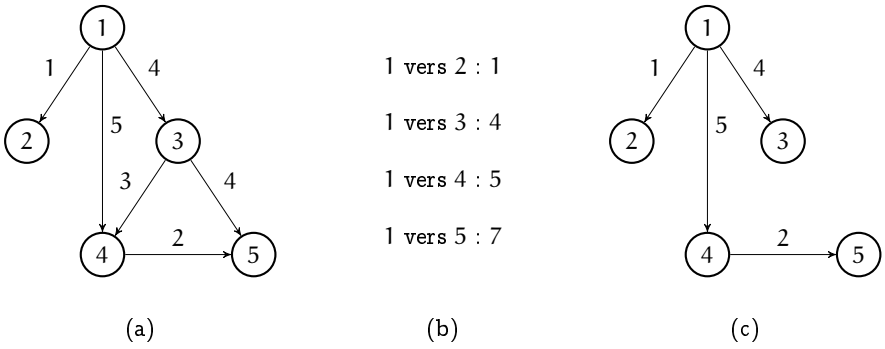


Fig. 7.3 – *Un réseau, les coûts minimaux de diffusion depuis le site* 1 *et l'arbre qui leur est associé*

Algorithme de Prim

Définitions et propriétés

Diverses notions et notations relatives aux graphes et aux arbres sont utilisées tout au long de cet exercice et nous renvoyons le lecteur aux sections 1.5, page 22 et 1.6, page 30. Dans la suite, $G = (N, A, P)$ désigne un graphe non orienté dont l'ensemble des sommets (resp. arêtes) est noté N (resp. A), valué sur les entiers positifs ($P \in A \to \mathbb{N}_1$) et connexe, c'est-à-dire tel qu'il existe une chaîne reliant tout couple de sommets distincts. De façon analogue, $T = (N, A, P)$ désigne un arbre non orienté composé de l'ensemble de sommets (resp. arêtes) N (resp. A), valué sur les entiers positifs. On appelle poids d'un arbre la somme des valeurs de ses arêtes (ou branches). On remarque d'une part qu'un arbre ayant k sommets possède $(k-1)$ arêtes, d'autre part qu'une chaîne élémentaire unique relie toute paire de sommets distincts d'un arbre. On appelle arbre de recouvrement du graphe $G = (N, A, P)$ un arbre (*a priori* non enraciné) incluant tous les sommets de G. Un arbre de recouvrement de G de plus faible poids est appelé arbre de recouvrement de poids minimal (arpm) ou parfois arbre sous-tendant de poids minimal (astm). Il est aisé de montrer que

tout graphe G (connexe) admet au moins un arpm et qu'un arbre est son propre arpm. En référence à un graphe $G = (N, A, P)$, on dit d'un arbre T qu'il est *prometteur* si c'est un sous-arbre (au sens large) d'un arpm.

Propriété 5 :
L'adjonction à un arbre d'une arête joignant deux de ses sommets provoque l'apparition d'un cycle élémentaire.

77 - Q 1 **Question 1.** Démontrer la validité de la propriété 5.

L'idée sous-jacente à l'algorithme de Prim consiste à fixer un sommet comme source, sommet qui constitue un arbre prometteur (le sommet de numéro 1 par la suite) et à « faire grossir » un arbre prometteur en lui ajoutant une nouvelle arête (et donc un nouveau sommet) du graphe G que l'on veut recouvrir. En vertu de la propriété 5, un tel ajout ne peut se faire qu'avec une arête *mixte* constituée d'un sommet pré-existant de l'arbre initial (sommet *interne*) et d'un sommet n'en faisant pas encore partie (sommet *externe*). La question est donc de déterminer quelle arête de $G = (N, A, P)$ peut/doit être insérée dans un arbre prometteur pour que le nouvel arbre soit lui aussi prometteur.

Propriété 6 :
Soit $G = (N, A, P)$ *un graphe et* $T = (N', A', P')$ *avec* $N' \subset N$ *un sous-graphe de* G *qui est un arbre prometteur. L'adjonction à* T *de l'une des arêtes* mixtes *de* G *de valeur minimale conduit à un nouvel arbre prometteur.*

77 - Q 2 **Question 2.** Démontrer la propriété 6.

Construction de l'algorithme

On va construire l'algorithme de Prim comme un glouton exact en adoptant le principe de « la course en tête » (voir section 7.1.4, page 201). La postcondition du programme à réaliser peut s'énoncer : « $T = (N', A', P')$ est un arbre prometteur de $G = (N, A, P)$ et $\text{card}(A') = \text{card}(N') - 1 = \text{card}(N) - 1 = n - 1$ ». Il est donc naturel de poser :

Invariant $T = (N', A', P')$ est un arbre prometteur de G avec $N' \subseteq N$.

Condition d'arrêt $\text{card}(A') = n - 1$.

Progression En vertu de la propriété 6, on intègre dans l'arbre prometteur $T = (N', A', P')$ avec $N' \subset N$ l'une des arêtes *mixtes* de G de valeur minimale, c'est-à-dire une arête $(s1, s2)$ avec $s1$ un sommet externe et $s2$ un sommet interne ($s2 \in N'$).

Terminaison $n - 1 - \text{card}(A')$.

Initialisation On prend comme arbre prometteur l'arbre réduit au seul sommet 1 (la source).

On observe que l'ensemble des arêtes mixtes peut être représenté par une *fonction* totale aM associant à chaque sommet externe le sommet interne le plus proche au sens de la valeur des arêtes reliant les sommets ($aM \in 2..n \to 1..n$). Il convient donc de mettre à jour cette fonction après l'insertion d'une arête mixte dans la progression. Malgré l'orientation liée à aM, on continuera de parler d'arêtes par la suite.

Parmi les nombreuses façons de représenter l'arbre résultant, en cohérence avec aM, on choisit un arbre inverse enraciné sur la source (le sommet 1). En conséquence, la fonction *totale* aI va représenter les arêtes qui font partie intégrante de l'arbre (en cours d'élaboration) en associant à tout sommet, source exclue, son père dans cet arbre ($aI \in 2..n \to 1..n$). L'ensemble des sommets externes et celui des sommets internes constituent une partition de l'ensemble N des sommets de G. C'est la signification qu'il faut accorder à la ligne 6 du code ci-dessous. Cette remarque sera pleinement exploitée plus tard, dans la mise en œuvre de l'algorithme.
On note P^T la matrice transposée de la matrice P et, puisque le graphe considéré est non orienté, on peut le représenter par une matrice P telle que $P = P^T$. Dans ces deux matrices, l'absence d'arête dans le graphe se traduit par la valeur conventionnelle $+\infty$.
Compte tenu de ces choix, la version abstraite de l'algorithme est présentée ci-dessous :

```
1.  constantes
2.    n ∈ ℕ1 et n = ... et P ∈ 1..n × 1..n → ℕ1 et P = P^T et P = [...] et
3.    EstConnexe(G)
4.  variables
5.    aI ∈ 2..n → 1..n et aM ∈ 2..n → 1..n et
6.    dom(aI) ∪ dom(aM) = 2..n et dom(aI) ∩ dom(aM) = ∅
7.  début
8.    aI ← ∅ ; aM ← 2..n × {1};
9.    pour k ∈ 2..n faire
10.     soit e, i tel que
11.       (e, i) ∈ aM et P(e, i) = min({P(l, j) | (l, j) ∈ aM})
12.     début
13.       aI ← aI ∪ {{e, i}}; aM ← aM − {(e, i)};
14.       /% mise à jour de aM pour le rétablissement complet de l'invariant,
          à savoir que, pour tout sommet externe e, aM(e) désigne le sommet
          interne le plus proche de e %/
15.       pour e' ∈ dom(aM) faire
16.         si P(e', e) < P(e', aM(e')) alors
17.           aM(e') ← e
18.         fin si
19.       fin pour
20.     fin
21.   fin pour ;
22.   écrire(aI)
23. fin
```

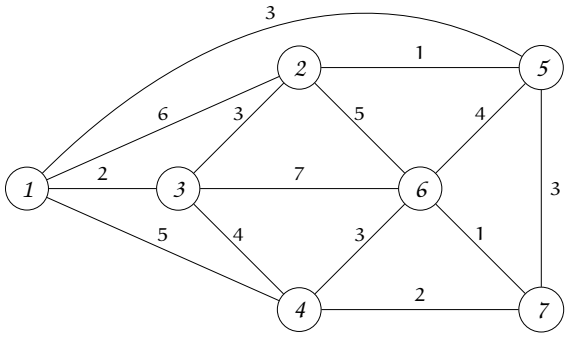
Mises en œuvre

Concernant la mise en œuvre effective, on agrège les deux ensembles aI et aM en un seul tableau noté Arc. Ce tableau est défini sur l'intervalle $2..n$ (le sommet 1 est exclu, car il n'est l'origine d'aucun arc, mais il fait toujours partie des sommets internes) et à valeur dans $1..n$. La partition qui distingue les sommets externes des sommets internes est représentée par le tableau de booléens Externe, également défini sur l'intervalle $2..n$.

Question 3. Écrire la fonction *SommetExterneCoutMin* qui raffine les lignes 10 et 11 de l'algorithme générique et effectue la recherche du sommet externe le plus proche d'un sommet interne. Quelle en est la complexité en termes de nombre de conditions évaluées ? 77 - Q 3

77 - Q 4 **Question** 4. Donner le code de l'algorithme de Prim dans le cadre de la mise en œuvre proposée et sa complexité (en nombre de conditions évaluées).

77 - Q 5 **Question** 5. Appliquer l'algorithme précédent au graphe ci-après :



77 - Q 6 **Question** 6. On constate que l'approche de mise en œuvre proposée n'utilise ni file d'entrée ni file de sortie. Préciser ce qui leur est substitué.

Avec une mise en œuvre reposant sur une file de priorité F comme file d'entrée, la trame de l'algorithme devient :

1. **constantes**
2. /% *déclaration de* n *et* P %/
3. **variables**
4. /% *déclaration des variables dont* aI *et* F %/
5. **début**
6. /% *initialisation de* aI *et* F %/ ;
7. **pour** $k \in 2..n$ **faire**
8. /% *retrait de la tête de file de* F *et mise de l'arête associée dans* aI %/ ;
9. /% *mise à jour de la file* F *pour le rétablissement complet de l'invariant ; on effectue une boucle parcourant* F *dans laquelle l'arête mixte relative à chaque sommet externe* e *est, le cas échéant, remplacée (suppression et insertion) par une de valeur moindre* %/
10. **fin pour** ;
11. écrire(aI)
12. **fin**

77 - Q 7 **Question** 7. En déduire une classe de complexité minimale (en termes de conditions évaluées) d'une telle solution et conclure quant à son intérêt.

Remarques sur l'algorithme de Prim

1. L'algorithme de Prim peut être facilement adapté à la recherche d'un arbre de recouvrement de poids *maximum*, ainsi qu'à des graphes de valuations de signe quelconque.
2. Le lecteur pourra vérifier que, appliqué à l'exemple de la figure 7.2, page 211, l'algorithme de Prim délivre (à l'inversion de l'orientation près) l'arbre de la partie c.

3. L'algorithme de Kruskal résout le même problème que celui de Prim. Sa complexité en $\Theta(m \cdot \log_2(m))$ avec $m = \text{card}(A)$ est due au tri préalable des arêtes du graphe $G = (N, A, P)$ selon leur valeur croissante. Il présente l'avantage de pouvoir traiter des graphes non connexes (en délivrant alors un arbre par composante connexe).

Algorithme de Dijkstra

Soit $G = (N, V, P)$ un graphe orienté, où $\text{card}(N) = n$ et P est une valuation des arcs sur $\mathbb{N}_1$. Soit d un sommet particulier appelé source ($d = 1$ comme précédemment). Le problème que l'on se pose est de déterminer, pour tout sommet f de N, le coût minimum (appelé *distance* ci-dessous) pour aller de d à f. En l'absence de chemin entre d et f, la distance est notée de façon conventionnelle $+\infty$.
L'algorithme que l'on cherche à construire est fondé sur une itération telle que, pour un sous-graphe G' de G, les distances sont déjà connues et qui, à chaque étape, ajoute un nouveau sommet dans G'. Le résultat final se présente sous la forme d'un tableau L, défini sur l'intervalle $1..n$, tel que $L[i]$ est la distance de d à i (en particulier $L[d] = 0$). Le graphe de la figure 7.4 sert d'illustration dans la suite.

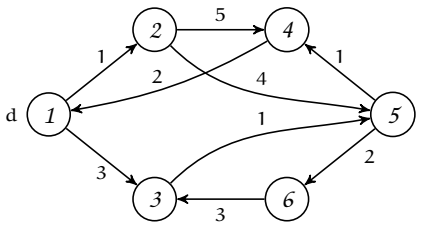


Fig. 7.4 – *Exemple de graphe orienté valué*

Notations

— Soit c un chemin dans G ; $\text{coût}(c)$ est la somme des valuations des arcs qui composent ce chemin (si le chemin comporte des circuits, certains arcs sont comptabilisés plusieurs fois).

— Soit f un sommet de G ($f \in N$) ; $\text{chem}(f)$ est l'ensemble (possiblement infini) des chemins de d à f ; $\text{dist}(f)$ est la distance de d à f, soit :

$$\text{dist}(f) \mathrel{\widehat{=}} \min_{c \in \text{chem}(f)} (\text{coût}(c)).$$

Pour la figure 7.4 page 215, avec $d = 1$, nous avons :
— $\text{coût}(\langle 1, 2, 4\rangle) = 6$,
— $\text{chem}(4) = \{\langle 1, 2, 4\rangle, \langle 1, 3, 5, 4\rangle, \langle 1, 2, 4, 1, 2, 4\rangle, \langle 1, 3, 5, 6, 3, 5, 4\rangle, \ldots\}$,
— $\text{dist}(5) = 4$, $\text{dist}(6) = 6$.

Construction de l'algorithme – première version

Comme pour l'algorithme de Prim, nous recherchons une solution du type « course en tête » (voir section 7.1.4, page 201) ; il s'agit donc de construire une itération.

Invariant Nous appliquons la stratégie du « travail réalisé en partie » (voir section 3.4, page 69). Formulons l'hypothèse que le tableau L contient les distances de d à tous les sommets de N′ ($N' \subseteq N$). Plus précisément :

$$I_1 \ \widehat{=} \ \forall f \cdot (f \in N' \Rightarrow L[f] = dist(f)).$$

En outre, si N′ n'est pas vide, $d \in N'$:

$$I_2 \ \widehat{=} \ (N' \neq \varnothing \Rightarrow d \in N').$$

Nous introduisons la variable N″ ($N'' \mathrel{\widehat{=}} N - N'$) et posons :

$$I_3 \ \widehat{=} \ (N = N' \cup N'') \text{ et } (N' \cap N'' = \varnothing).$$

Dans l'exemple de la figure 7.5, $d = 1$, $N' = \{1,2,3\}$ et les trois distances entre le sommet 1 et ces trois sommets sont connues. Elles sont notées dans la partie inférieure de chacun de ces sommets.

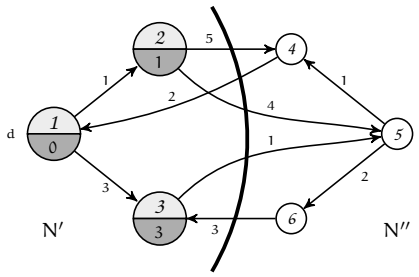


Fig. 7.5 – *Situation après insertion de trois sommets dans* N′ *(la valeur de* L *pour ces trois sommets apparaît dans la zone inférieure de ces sommets).*

Condition d'arrêt

$$N'' = \varnothing.$$

On peut vérifier que la conjonction de l'invariant et de la condition d'arrêt implique bien le but : les distances de d à tous les sommets sont connues.

Progression Il s'agit de sélectionner un sommet de N″ et de le déplacer dans N′ tout en s'assurant que l'invariant est bien préservé. Cependant, en l'absence d'informations complémentaires, il est difficile de choisir un sommet qui permettrait le rétablissement de l'invariant. Nous sommes conduits à proposer une seconde version qui est obtenue en renforçant l'invariant (I_1 **et** I_2 **et** I_3) par un quatrième conjoint I_4 destiné à faciliter la construction de la progression.

Construction de l'algorithme – seconde version

Invariant Une stratégie gloutonne consiste à attribuer à chaque sommet f de N″ une sur-estimation de la distance entre d et f et, à chaque pas de progression, à déplacer le meilleur sommet de N″ dans N′ (en espérant, pour avoir un glouton exact, que, pour ce sommet, l'estimation soit *exactement* la distance recherchée). Il faut s'attendre à ce que l'introduction de cette propriété de sur-estimation exige l'ajout d'un fragment

de code pour son maintien. Avant d'apporter plus de précisions, il est nécessaire de compléter les notations ci-dessus.

— On appelle $eChem(f)$ (pour e-chemin) l'ensemble de tous les chemins de la source d ($d \in N'$) à un sommet f de N'' dont tous les sommets sont dans N', à l'exception de f.

— On note $eDist(f)$ (pour e-distance) le coût le plus faible parmi ceux des e-chemins ($eChem$) de d à f :

$$eDist(f) \mathrel{\widehat{=}} \min_{c \in eChem(f)} (coût(c)).$$

Si $eChem(f) = \varnothing$ alors $eDist(f) = +\infty$.

Le prédicat I_4 que nous adjoignons à l'invariant précédent (I_1 et I_2 et I_3) précise que, pour tout élément f de N'', $L[f]$ est l'e-distance de d à f :

$$I_4 \mathrel{\widehat{=}} \forall f \cdot (f \in N'' \Rightarrow L[f] = eDist(f)).$$

L'exemple de la figure 7.5 se complète alors comme le montre la figure 7.6, où $L[4] = eDist(4) = 6$, $L[5] = eDist(5) = 4$. En revanche, il n'y a pas (encore) de e-chemin de d vers le sommet 6, et donc $L[6] = eDist(6) = +\infty$.

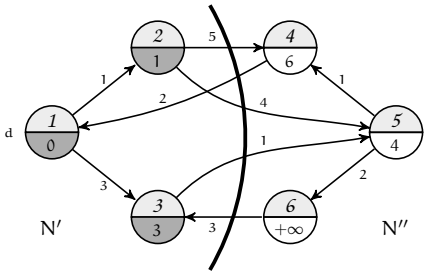
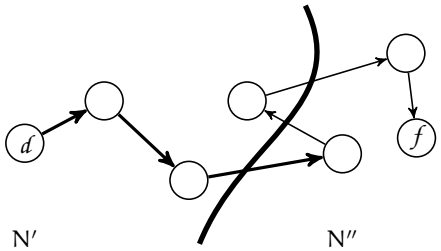


Fig. 7.6 – *Situation après trois pas de progression. Les valeurs de* $dist(f)$, *pour* $f \in N'$, *et de* $eDist(f)$, *pour* $f \in N''$, *sont notées dans la partie inférieure de chaque sommet. Elles correspondent à* $L[f]$.

Dès lors que $d \in N'$, si $f \in N''$, tout chemin de d à f possède un e-chemin comme préfixe (lui-même le cas échéant). Soit le chemin $\langle d, a_1, \ldots, a_p, f \rangle$. Son e-chemin est $\langle d, a_1, \ldots, a_i \rangle$ tel que, pour tout j de l'intervalle $1 \,..\, i$, $a_j \in N'$ et $a_{i+1} \in N''$. Ainsi, le schéma ci-dessous montre un chemin de d à f et, en gras, le e-chemin qui lui correspond :



Un tel chemin n'est donc jamais moins coûteux que le e-chemin qui lui correspond, d'où la propriété suivante (qui n'est pas formellement démontrée) :

Propriété 7 :
Soit $f \in N''$, $d \in N'$, c *un chemin de* d *à* f *et* c' *le e-chemin correspondant. On a* $\text{coût}(c') \leqslant \text{coût}(c)$.

Dans la figure 7.5 page 216, $c = \langle 1,2,5,6\rangle$ est un chemin de $d = 1$ à $f = 6$, qui a comme coût 7. Le e-chemin correspondant est $c' = \langle 1,2,5\rangle$, qui a comme coût 5.

Condition d'arrêt La condition d'arrêt est inchangée par rapport à la première version.

Progression Ainsi que nous l'avons mentionné ci-dessus, on choisit, parmi tous les éléments de N'', celui qui, en termes de e-distance, est le plus proche de d. Il en existe obligatoirement au moins un, puisque, selon la précondition de la progression, N'' est non vide. Le code de la progression s'obtient alors en introduisant une « constante » locale g, la partie de code (C) reste à instancier :

1. **soit** g **tel que**
2. $\quad g \in N''$ **et** $L[g] = \min\limits_{f \in N''} (L[f])$
3. **début**
4. $\quad N'' \leftarrow N'' - \{g\}$; $N' \leftarrow N' \cup \{g\}$;
5. $\quad \vdots$ (C)
6. **fin**

77 - Q 8 **Question 8.** Montrer que, si l'on parvient à achever sa construction, cet algorithme est un glouton exact (autrement dit, que pour le sommet g sélectionné $L[g] = \text{dist}(g)$). Sur l'exemple de la figure 7.6, quelle est la situation atteinte après l'exécution de la ligne 4 de la progression ?

77 - Q 9 **Question 9.** Compléter la construction de la progression (c'est-à-dire rédiger le fragment de code (C) qui rétablit le conjoint I_4 de l'invariant). Sur l'exemple de la figure 7.6, quelle est la situation atteinte après l'exécution de la progression ?

77 - Q 10 **Question 10.** Compléter la construction de l'algorithme et fournir son code.

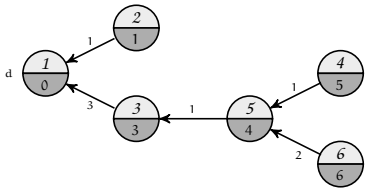
77 - Q 11 **Question 11.** Qu'en serait-il de la propriété 7 si la précondition qui exige que la valuation P ne soit jamais négative était abandonnée ?

77 - Q 12 **Question 12.** On étudie un raffinement à l'algorithme fourni en réponse à la question 10 basé sur les éléments suivants : i) les ensembles N'/N'' sont représentés par un vecteur caractéristique W ($W \in 1\,..\,n \to \mathbb{B}$), ii) afin d'éviter une évaluation complexe de sa condition d'arrêt, la boucle **tantque** est remplacée par une boucle **pour**, dont le corps est exécuté n fois. Faire une description informelle de l'algorithme qui en résulte. Quelle est sa complexité en termes de conditions évaluées ?

77 - Q 13 **Question 13.** Dans la perspective d'un second raffinement, on considère les files de priorité de type tas (voir section 1.7, page 33). Sur cette base, décrire de façon informelle les différentes structures de données, ainsi que les étapes de l'algorithme, et fournir son code. Analyser sa complexité temporelle (en termes de conditions évaluées) et la comparer à celle obtenue en réponse à la question précédente.

Question 14. Dans ce qui précède, il a été suggéré que la boucle **pour** principale de l'algorithme soit exécutée n fois. Expliquer comment on pourrait limiter le nombre de pas à $(n-2)$ dans l'algorithme fondé sur une file de priorité. 77 - Q 14

Question 15. Jusqu'à présent, nous ne nous sommes préoccupés que des distances. En général, on souhaite également connaître un *chemin* optimal. L'ensemble des chemins optimaux de d vers chaque sommet peut se représenter par un arbre inverse (un fils désigne son père) dont la racine est d, comme le montre le schéma suivant pour l'exemple de la figure 7.4, page 215 : 77 - Q 15



Indiquer quels changements sont à apporter à l'algorithme pour qu'il construise cet arbre.

Question 16. L'article original d'E.W. Dijkstra porte sur la recherche de la distance entre deux sommets donnés quelconques d et s de G. Comment peut-on aménager l'algorithme fourni en réponse à la question 10 pour résoudre cette variante du problème traité ici ? 77 - Q 16

En conclusion

Le lecteur pourra vérifier que, appliqué au graphe de la partie (a) de la figure 7.3, page 211, l'algorithme de Dijkstra délivre le tableau L ci-après :

1	2	3	4	5
0	1	4	7	9

correspondant aux valeurs données dans l'arbre de la partie (b) de la figure 7.3, page 211.

L'algorithme de Dijkstra s'adapte à la recherche des chemins de valeur maximale depuis une source avec un graphe orienté dont les arcs portent des valeurs négatives ou nulles.

D'autres algorithmes de calcul de chemins de valeur minimale sont présentés dans les exercices 129 page 350, 130 page 352 et 131 page 355 au chapitre 9.

Pour comparer des algorithmes de Prim et de Dijkstra, on notera que :

1. Les valeurs portées par les arêtes ou les arcs ont été choisies entières, mais pourraient tout aussi bien être réelles.
2. Si l'algorithme de Prim peut fonctionner avec des valuations de signe quelconque, on a vu à la question 11 qu'il n'en va pas de même pour l'algorithme de Dijkstra (ce qui peut en limiter l'utilisation pour certains problèmes).
3. Les deux algorithmes se ressemblent, ayant comme structure générale une boucle, ce qui n'étonne pas puisque ce sont deux algorithmes gloutons. Mais la similitude va au-delà, car, dans les deux cas, on est amené à : i) partitionner l'ensemble des sommets du graphe considéré, ii) choisir à chaque étape un sommet constituant un choix optimal et enfin iii) effectuer une mise à jour relative aux sommets qui n'ont pas encore été choisis.

Exercice 78. Compression de données : l'algorithme de Huffman

L'objectif de cet exercice est de construire un algorithme qui fournit un code permettant de compacter des données (c'est-à-dire de les compresser sans provoquer de perte d'information). Par de nombreux aspects, la solution étudiée ici occupe une place à part dans les exercices de cet ouvrage. Par son importance tout d'abord : malgré son âge (il a été publié en 1952), l'algorithme de Huffman occupe souvent l'un des étages des applications de compression de données. Par sa simplicité apparente d'autre part, qui se traduit par un algorithme d'une grande concision, qui contraste avec les efforts qu'il faut déployer pour prouver son optimalité. À son crédit on pourrait ajouter son efficacité, sa couverture en termes de structures de données, etc., bref un excellent exercice.

Introduction

Le codage binaire de symboles (typiquement des caractères typographiques) fait l'objet de normes internationales. Les codes ainsi définis sont le plus souvent de longueur fixe (huit bits pour le code Ascii, seize bits pour le code UTF-16, etc.). Cependant, de par leur vocation universelle, leur utilisation se révèle en général coûteuse (en termes de place pour le codage de fichiers, en temps pour leur transmission). Une amélioration substantielle peut être obtenue en utilisant à la place un code *ad hoc* (dépendant uniquement du texte considéré) de longueur *variable*, qui tient compte de la fréquence de chaque caractère dans le texte considéré. Considérons par exemple le texte t suivant, de 17 caractères :

$$\mathrm{t} = \mathit{elle}\sqcup\mathit{aime}\sqcup\mathit{le}\sqcup\mathit{miel}$$

exprimé sur le vocabulaire $V = \{a, e, i, l, m, \sqcup\}$ (le caractère $\sqcup$ représente l'espace). En utilisant un code de longueur fixe de huit bits, ce texte occupe $17 \cdot 8 = 136$ bits. Un code de longueur fixe de trois bits (c'est le mieux que l'on puisse faire ici, en utilisant un code de longueur fixe, pour un vocabulaire V de six symboles) exige $17 \cdot 3 = 51$ bits.

Un code de longueur variable peut permettre d'améliorer la situation. C'est ce que montre celui de la table 7.1 qui permet de coder le texte t ci-dessus par la chaîne de 49 (au lieu de 51) bits suivante (le point dénote l'opération de concaténation) :

10·1111·1111·10·110·00·1110·01·10·110·1111·10·110·01·1110·10·1111

Intuitivement, cela sera d'autant plus vrai que les mots de code les plus courts seront affectés aux caractères les plus fréquents.

symboles	a	i	m	$\sqcup$	l	e
fréquences	1	2	2	3	4	5
mots de code	00	1110	01	110	1111	10

Tab. 7.1 – *Exemple de code et de fréquences pour le vocabulaire* $V = \{a, e, i, l, m, \sqcup\}$

L'objectif de l'exercice est de construire un algorithme (dû à D.A. Huffman, 1952) qui, pour un texte t donné (et donc un vocabulaire et une fréquence donnés), fournit un code optimal, c'est-à-dire qui code t avec le moins de bits possibles.

Dans la suite de cette introduction, on présente les concepts de code et d'arbre préfixes avant de définir les notions de code et d'arbre de Huffman.

Code/arbre préfixes Un code préfixe est un code dans lequel il n'existe pas deux caractères dont le mot de code de l'un soit le préfixe de celui de l'autre. Ceci interdit par exemple de coder e par 1 et a par 1011. L'avantage d'un code préfixe réside dans la phase de décodage (passage de la chaîne de bits à la chaîne de caractères qui lui correspond), dans la mesure où cette étape peut s'effectuer de manière déterministe [1] : dès qu'un mot de code c est identifié au début de la chaîne à décoder b, il suffit de le traduire par le caractère correspondant, de le supprimer de b et de réappliquer le processus sur ce qui reste de b. Les codes de longueur fixe sont, par construction, préfixes ; le code de la table 7.1 l'est également.

Un code préfixe peut se représenter par un arbre binaire complet (c'est-à-dire sans point simple, voir section 1.6, page 30), dont les branches gauches sont étiquetées par des 0 et les branches droites par des 1, et dont les feuilles sont étiquetées par un caractère. Le code de la table 7.1 est représenté par l'arbre (a) de la figure 7.7.

Préfixe n'est cependant pas synonyme d'optimal : pour le texte t ci-dessus (et donc pour le vocabulaire V et les fréquences de la table 7.1), le code représenté par l'arbre préfixe (b) de la figure 7.7 est meilleur que celui représenté par l'arbre (a) puisqu'il code le texte t en 42 bits au lieu de 49. En revanche, on sait (affirmation admise dans la suite) qu'un code optimal peut toujours se représenter par un code préfixe.

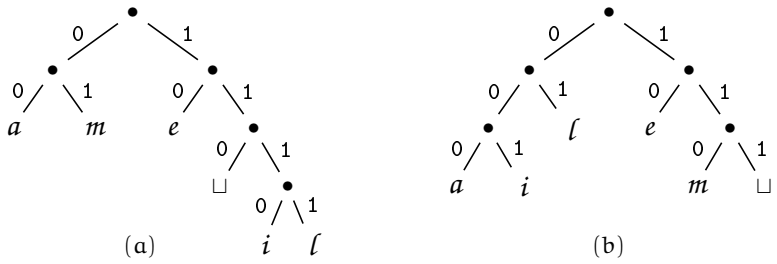


Fig. 7.7 – *Deux arbres préfixes pour le vocabulaire* $V = \{a, e, i, \ell, m, \sqcup\}$. *L'arbre* (a) *correspond au code du tableau 7.1, page 220, l'arbre* (b) *est un second arbre préfixe.*

Le coût $L(A)$ de l'arbre préfixe A se définit par la longueur de la chaîne de bits résultant du codage du texte t par A. Plus précisément, soit $V = \{v_1, \ldots, v_n\}$ $(n \geqslant 2)$ un vocabulaire, f $(f \in V \to \mathbb{N}_1)$ la fréquence des v_i dans le texte t (leur nombre d'occurrences), et A un arbre préfixe,

$$L(A) = \sum_{v \in V} f(v) \cdot l_A(v), \tag{7.2}$$

où $l_A(v)$ est la longueur du mot de code de v (ou la profondeur de la feuille v dans A) [2].

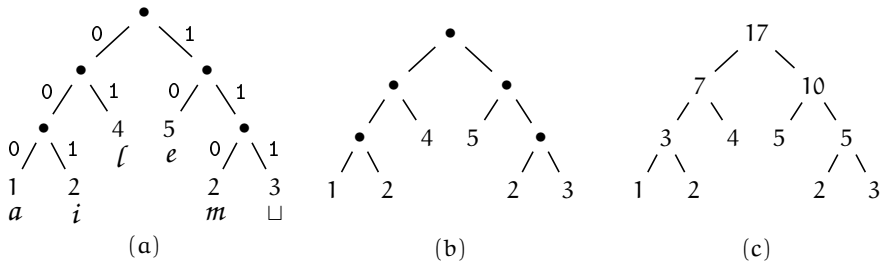
1. On ne s'intéresse ici qu'aux codes déterministes, c'est-à-dire aux codes pour lesquels le processus de codage n'exige pas de retour arrière.
2. $L(A)$ est aussi appelé « longueur de chemin pondéré » de l'arbre A.

Code/arbre de Huffman Un arbre préfixe A représente un code de Huffman s'il n'existe pas d'arbre (préfixe) A' tel que $L(A') < L(A)$. En général, un arbre A de Huffman n'est pas unique : il existe des arbres A' tels que $L(A') = L(A)$. Pour le couple (V, f) de la table 7.1 page 220, un arbre de Huffman A est tel que $L(A) = 42$.

78 - Q 1 **Question 1.** Vérifier qu'en utilisant le couple (V, f) cité précédemment, l'arbre (b) de la figure 7.7 est tel que $L(b) = 42$. Pour le même couple (V, f), proposer un second arbre de Huffman qui ne soit pas obtenu par de simples échanges de sous-arbres.

L'algorithme de Huffman

Les arbres produits par l'algorithme de Huffman ne sont pas parfaitement identiques aux arbres préfixes optimaux tels que définis ci-dessus. Ils sont enrichis (renforcés) par une information redondante, qui facilite leur construction : chaque nœud est complété par la somme des fréquences de toutes ses feuilles. En outre, pour ce qui nous concerne, nous renonçons à deux informations qui s'avèrent superflues lors de la construction de l'arbre : les caractères placés aux feuilles et les étiquettes apposées aux branches. Le schéma ci-dessous montre comment un arbre de Huffman (a) se transforme en un « arbre optimal » (c) en passant par un arbre « externe » (b) (c'est-à-dire un arbre où l'information non structurelle est portée uniquement par les feuilles).



Arbres de fréquences

Définition 24 (Arbre de fréquences) :
Un arbre (complet) de fréquences est un élément de l'ensemble P *des arbres binaires complets, tel que chaque nœud est étiqueté par la somme des fréquences de ses feuilles.* P *se définit par :*

$$P = \{(/, h, /) \mid h \sqsubset F\} \cup \{(g, h, d) \mid g \in P \text{ et } d \in P \text{ et } h = g.h + d.h\}.$$

L'opérande gauche de l'opérateur $\cup$ permet de n'obtenir que des arbres complets. F est le sac des valeurs prises par les feuilles (le sac des fréquences). L'arbre (c) ci-dessus est un arbre de fréquences défini sur le sac $[\![1, 2, 4, 5, 2, 3]\!]$.

Le coût $L(A)$ d'un arbre de fréquence A sur le sac des fréquences F se définit par :

$$L(A) = \sum_{k \sqsubset F} k \cdot l_A(k), \tag{7.3}$$

où $l_A(k)$ est la profondeur de la feuille k dans A. Cette définition est compatible avec celle de la formule 7.2 page 221.

Propriété 8 :
Soit G *(resp.* D*) un arbre de fréquences défini sur le sac de fréquences* F_G *(resp.* F_D*), soit* $A = (G, G.h + D.h, D)$ *l'arbre de fréquences défini sur le sac de fréquences* $F_G \sqcup F_D$*.* A *vérifie la propriété suivante :*

$$L(A) \;=\; L(G) + G.h + D.h + L(D). \tag{7.4}$$

Question 2. L'arbre (c) ci-dessus est un arbre de fréquences. Vérifier que les deux formules 7.3 et 7.4 fournissent bien le même coût pour cet arbre. Démontrer la propriété 8. 78 - Q 2

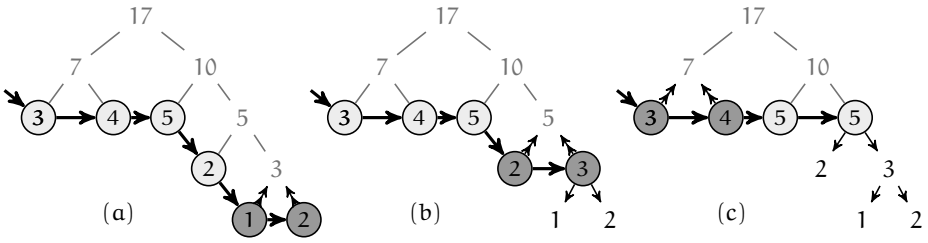
Arbres optimaux

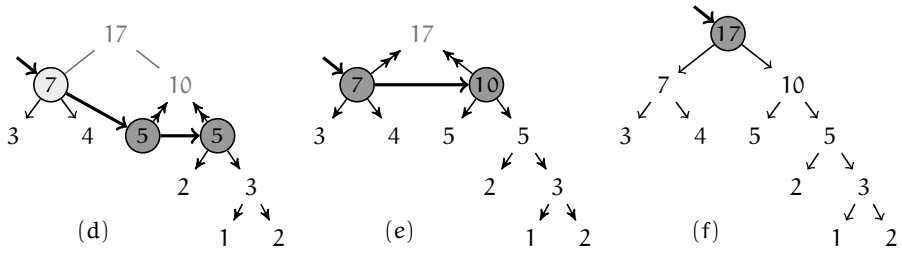
Définition 25 (Arbre optimal) :
Un arbre de fréquences A *(défini sur les fréquences* F*) est un arbre optimal, si et seulement s'il n'existe pas d'arbre (de fréquences sur* F*)* A′ *tel que* $L(A') < L(A)$*.*

Question 3. En supposant que P représente l'ensemble des arbres de fréquences, définir formellement H ($H \subseteq P$), sous-ensemble des arbres optimaux de fréquences de P. 78 - Q 3

Construction d'un arbre optimal Dans une première étape, on présente la construction d'un arbre optimal de manière intuitive, avant de se préoccuper de la construction de l'algorithme. La principale difficulté de cette construction réside dans la preuve de l'optimalité. Nous recherchons une solution du type « course en tête » (voir section 7.1.4, page 201).

Deux possibilités s'offrent à nous pour ce qui regarde la construction de l'arbre : descendante ou ascendante. Nous optons pour la seconde. Initialement, les différentes fréquences sont placées dans une liste B, puis, à chaque pas d'itération, deux fréquences sont enracinées en un arbre dont la racine porte la somme des deux fréquences. Quelles fréquences choisir dans la liste ? Nous sommes dans une logique gloutonne : nous retenons les deux fréquences les plus faibles. Le processus est réitéré. Chaque pas fait décroître d'une unité la longueur de la liste : l'algorithme s'achève quand la liste ne contient plus qu'un seul élément. C'est ce que montrent les six schémas ci-dessous.





Ces six schémas montrent l'évolution de la liste B (en gras) et la construction ascendante de l'arbre optimal. À chaque étape, la liste contient une forêt (voir par exemple le schéma (d)) d'arbres optimaux qui est incluse dans l'arbre finalement construit par l'algorithme (schéma (f)). Cet exemple présente comme particularité que, dans la liste B, deux fréquences minimales sont toujours voisines. Le cas échéant, le processus de construction fournit toujours un arbre différent, mais toujours optimal. Concrètement, ainsi qu'il est précisé ci-dessous, l'algorithme représente B par une file de priorité.

Liste d'arbres La section précédente nous conduit à définir les notions de liste d'arbres de fréquences et de liste optimale (d'arbres de fréquences).

Définition 26 (Liste de fréquences) :
Soit $A_1, \ldots, A_m$ m *arbres de fréquences sur respectivement* $F_1, \ldots, F_m$. $B = \langle A_1, \ldots, A_m \rangle$ *est une liste de fréquences sur* $F = F_1 \sqcup \ldots \sqcup F_m$.

Définition 27 (Coût d'une liste) :
Soit $B = \langle A_1, \ldots, A_m \rangle$ *une liste de fréquences. Son coût se définit par* $L(B) = \sum_{i=1}^{m} L(A_i)$.

Définition 28 (Liste optimale) :
B *est une liste optimale (de fréquences) sur* F *si, pour toute liste* B' *sur* F, $L(B) \leqslant L(B')$.

78 - Q 4 **Question 4.** Montrer que si $B = \langle A_1, \ldots, A_m \rangle$ est une liste optimale, alors chaque A_i est un arbre optimal.

Construction de l'algorithme de Huffman L'objectif est d'obtenir un arbre optimal sur le sac des fréquences $F = F_1 \sqcup \cdots \sqcup F_n$. Il s'agit d'un algorithme itératif basé sur une liste optimale triée (une file de priorité) d'arbres de fréquences. Nous proposons un invariant à partir duquel se construisent les autres constituants de la boucle.

Invariant $B = \langle A_1, \ldots, A_m \rangle$ $(m \in 1..n)$ est une liste optimale sur les fréquences respectives $F_1, \ldots, F_m$ et $F_1 \sqcup \cdots \sqcup F_m = F$.

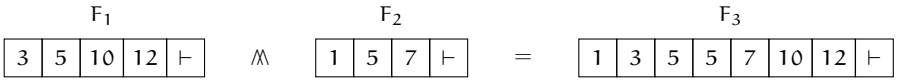
78 - Q 5 **Question 5.** Compléter la construction de la boucle. Fournir le texte de l'algorithme de Huffman. Calculer sa complexité.

Exercice 79. Fusion de fichiers

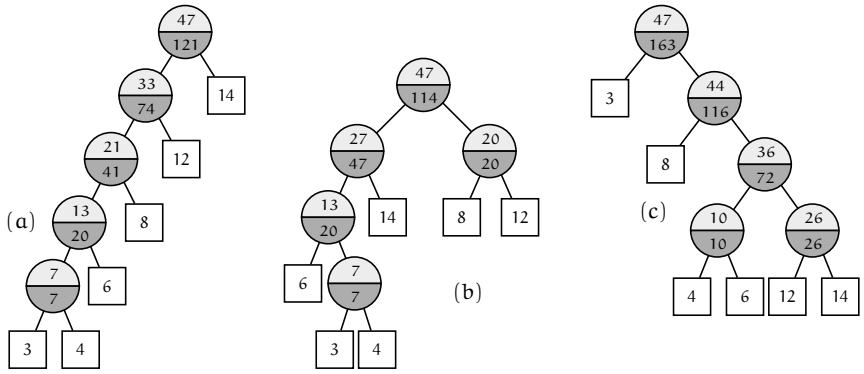
On considère ici des fichiers séquentiels triés sur une clé. Contrairement à ce que suggère l'intuition, le coût de la fusion de n fichiers, en nombre de comparaisons de clés, dépend de l'ordre dans lequel les fusions deux à deux sont réalisées. Il existe un algorithme glouton qui détermine cet ordre.

Il est conseillé de résoudre l'exercice 78 page 220 sur le codage d'Huffman, ainsi que l'exercice 86 page 248 sur le tri-fusion, avant d'aborder celui-ci.

La fusion est une opération qui permet par exemple, à partir de deux fichiers séquentiels triés F_1 et F_2, de produire un troisième F_3, trié lui aussi :



Dans la suite, on considère que fusionner deux fichiers de e_1 et e_2 enregistrements présente un coût de e_1+e_2 unités (en nombre de conditions évaluées par exemple). Fusionner n ($n > 2$) fichiers peut se faire en fusionnant successivement des couples de fichiers jusqu'à l'obtention d'un seul fichier. Cependant, le coût total de l'opération dépend de l'ordre dans lequel on choisit les couples à traiter. Considérons, par exemple, les six fichiers de $3, 4, 6, 8, 12$ et 14 enregistrements traités dans la figure ci-dessous. Dans le schéma (a), les fichiers sont traités selon l'ordre croissant de la taille des six fichiers de départ (encadrés dans le schéma). Le coût résultant est de 121. En effet, la fusion des deux fichiers de trois et quatre éléments donne un fichier de sept éléments (avec un coût de 7) ; ces sept éléments viennent se fusionner avec le fichier de six éléments pour donner 13 éléments (avec un coût de 20), etc. Le coût total est la somme des valeurs encerclées, soit $7+13+21+33+47 = 121$.



Pour le schéma (b), le coût s'élève à 114, et le traitement se caractérise par le fait que la fusion se fait systématiquement sur les deux fichiers les plus petits, indépendamment de leur origine. Quant au schéma (c), qui opère de manière aléatoire, son coût revient à 163.

L'objectif de l'exercice est de construire un algorithme glouton qui détermine un arbre de fusion optimal pour un ensemble de n fichiers quelconques.

Question 1. Sachant que pour un jeu de six fichiers dotés respectivement de $5, 6, 7, 8, 9$ et 10 enregistrements le coût optimal s'élève à 116 unités, fournir l'arbre optimal. 79 - Q 1

79 - Q 2 **Question** 2. Construire l'algorithme glouton qui détermine un arbre optimal pour tout jeu de n fichiers et démontrer son optimalité.

Exercice 80. Encore le photocopieur ○ ⁝

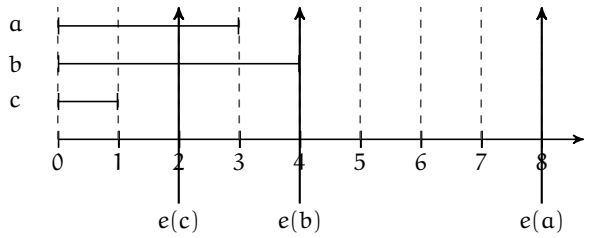
Cet exercice est une variante du problème traité en introduction de ce chapitre. Il peut se décliner de nombreuses façons et être la source d'une variété d'exercices voisins, plus ou moins difficiles. On peut le vérifier en recherchant une permutation qui maximise le bénéfice d'une tâche, ou en ne considérant que les fins de tâches qui sont pénalisantes, etc. C'est son principal intérêt.

Comme dans l'exemple introductif de ce chapitre, on doit réaliser un certain nombre de tâches de photocopies, mais les demandes des clients sont différentes. Il y a n tâches à réaliser, elles doivent toutes êtres accomplies et ne peuvent être fractionnées. Une tâche t_i exige une certaine durée $d(t_i)$. Chaque tâche t_i peut être placée n'importe où dans le planning de la photocopieuse à condition qu'elle n'entre pas en concurrence avec une autre tâche. L'heure de fin est notée $f(t_i)$. Chaque tâche s'accompagne d'une heure d'échéance $e(t_i)$ qui est telle que si t_i se termine avant l'heure $e(t_i)$, le bénéfice est positif et s'élève à $(e(t_i) - f(t_i))$ si elle se termine après, le « bénéfice » est négatif et vaut (toujours) $(e(t_i) - f(t_i))$; enfin, si t_i se termine exactement à l'heure $e(t_i)$, le bénéfice de l'opération est nul.

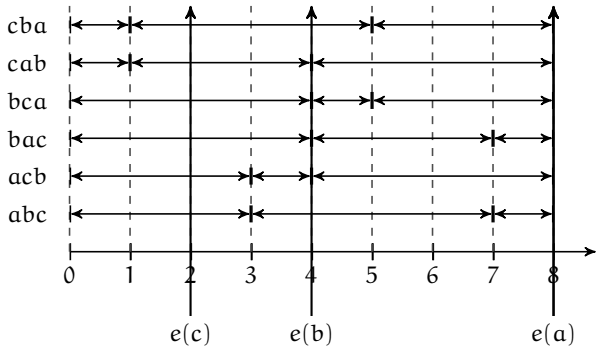
L'objectif de l'exercice est d'obtenir un algorithme glouton exact, qui détermine un ordre d'exécution des n tâches qui optimise (maximise) le bénéfice. Plus précisément, le cœur de l'exercice consiste à montrer que la stratégie gloutonne proposée est optimale.

Dans la suite, on admet (la démonstration est aisée) qu'il existe une solution optimale qui occupe la photocopieuse sans temps mort. On recherche une solution de ce type.

Exemple On considère les trois tâches a, b et c suivantes et leurs échéances :



Les six permutations possibles sont représentées dans le schéma ci-après.



Le bénéfice obtenu par exemple pour la permutation abc se calcule de la manière suivante :

$$
\begin{aligned}
& (e(a) - f(a)) + (e(b) - f(b)) + (e(c) - f(c)) \\
= & \qquad \text{arithmétique} \\
& (e(a) + e(b) + e(c)) - (f(a) + f(b) + f(c)) \\
= & \qquad \text{définition de } f(t_i) \\
& (e(a) + e(b) + e(c)) - (d(t(a)) + (d(t(a)) + d(t(b))) + (d(t(a)) + d(t(b)) + d(t(c)))) \\
= & \qquad \text{arithmétique} \\
& (e(a) + e(b) + e(c)) - (3 \cdot d(t(a)) + 2 \cdot d(t(b)) + 1 \cdot d(t(c))) \\
= & \qquad \text{application numérique} \\
& (8 + 4 + 2) - (3 \cdot 3 + 2 \cdot 4 + 1 \cdot 1) \\
= & \qquad \text{arithmétique} \\
& -4.
\end{aligned}
$$

Question 1. Compléter le calcul pour les cinq autres permutations. En déduire que la stratégie gloutonne consistant à ordonner les tâches selon les échéances croissantes n'est pas optimale. 80 - Q 1

Question 2. Montrer, en utilisant une démonstration du type « argument de l'échange », que la stratégie gloutonne consistant à ordonner les tâches selon leur durée croissante est optimale. Quelle est la complexité de l'algorithme résultant ? 80 - Q 2

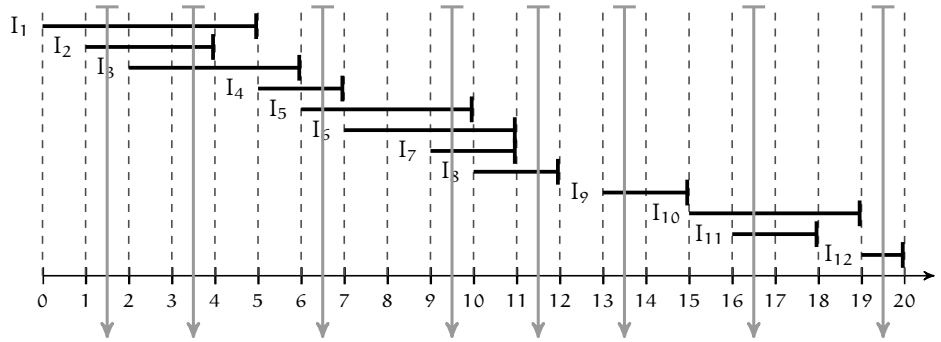
Exercice 81. Un problème d'épinglage

La difficulté principale de cet exercice réside dans la recherche d'une stratégie gloutonne et dans la preuve de son optimalité. L'énoncé guide le lecteur dans la recherche d'une solution.

On considère la demi-droite des réels positifs $\mathbb{R}_+^*$. Étant donné un ensemble fini I de n ($n \geqslant 0$) intervalles ouverts à gauche et fermés à droite, on dit qu'un ensemble de points T *épingle* I si chaque intervalle de I contient au moins une fois un point de T. L'objectif

de l'exercice est de construire un programme glouton qui calcule un ensemble T de taille minimale [3].

Exemple Dans l'exemple illustré ci-dessous, les douze intervalles sont épinglés par un ensemble $T = \{1.5, 3.5, 6.5, 9.5, 11.5, 13.5, 16.5, 19.5\}$ de huit points.



On s'aperçoit facilement que T n'est pas minimal. Par exemple, le point d'abscisse 1.5 épingle les intervalles I_1 et I_2, qui sont également épinglés par le point d'abscisse 3.5. Puisqu'il n'épingle pas d'autres intervalles, le premier point de T est donc inutile.

81 - Q 1 **Question** 1. Peut-on enlever d'autres points de T tout en préservant son statut ?

Dans la perspective d'une démarche gloutonne, on envisage deux stratégies de placement des points de T. Ces deux stratégies ont en commun qu'elles parcourent la file d'entrée « de gauche à droite » et épinglent les intervalles sur l'extrémité fermée (sur la droite de l'intervalle). Dans le premier cas, on considère que la file d'entrée F est triée sur les *origines* croisantes, dans le second, elle est triée sur les *extrémités* croissantes.

81 - Q 2 **Question** 2. Quel est le résultat de l'application de ces deux stratégies sur l'exemple ci-dessus ? Que peut-on en conclure ?

81 - Q 3 **Question** 3. Construire un programme glouton fondé sur la stratégie de votre choix parmi les deux stratégies étudiées dans la seconde question. Montrer qu'elle est optimale. La technique de la course en tête est préconisée. Quelle est la complexité de cette solution ?

81 - Q 4 **Question** 4. Planter une épingle peut se faire en n'importe quel point d'un intervalle s'achevant à la position déterminée dans les questions précédentes sans altérer l'optimalité de la solution. Comment cet intervalle se définit-il ?

3. Ce problème est aussi connu sous le nom anglais de *stabbing intervals*.

Exercice 82. Coloriage d'un graphe avec deux couleurs

Dans cet exercice, comme dans les suivants, on s'intéresse à la résolution d'un problème dans lequel il n'est pas question d'optimalité. Nous étudions ici un algorithme de coloriage d'un graphe avec deux couleurs. Une version plus générale (coloriage avec un nombre quelconque de couleurs) est étudiée dans l'exercice 58, page 159. Cependant, la présente version se révèle beaucoup plus efficace. En outre, elle est en relation étroite avec une catégorie de graphes qui possède de nombreuses applications : les graphes bipartites.

Étant donné un graphe non orienté connexe $G = (N, V)$ ($\text{card}(N) > 0$), on cherche, *quand c'est possible,* à le colorier en noir et blanc de manière à ce que deux sommets adjacents ne soient jamais d'une même couleur. Un tel graphe est alors dit *bicolorié.* L'algorithme glouton que nous allons construire à cette fin est apparenté à l'algorithme de parcours d'un graphe « en largeur d'abord » qui est tout d'abord étudié.

Parcours de graphe en largeur d'abord : rappels

Introduction Tout d'abord, nous définissons les notions de « distance entre deux sommets » et de « parcours en largeur d'abord » d'un graphe non orienté connexe.

Définition 29 (Distance entre deux sommets) :
Soit $G = (N, V)$ un graphe non orienté connexe, s et s' deux sommets de G. On appelle distance entre s et s' la longueur du plus court chemin entre s et s'.

Définition 30 (Parcours en largeur d'abord) :
Soit G un graphe non orienté connexe, s un sommet de G. On appelle « parcours en largeur d'abord » de G depuis s tout procédé qui rencontre les sommets de G selon les distances croissantes par rapport à s.

Du schéma (b) de la figure 7.8 page 230, on peut conclure que la liste $\langle a, b, c, d, e, f, g, h\rangle$ correspond à un « parcours en largeur d'abord » depuis le sommet a. Il en est de même de la liste $\langle a, c, b, d, e, h, f, g\rangle$.

L'invariant de boucle Nous souhaitons construire un algorithme itératif, glouton de surcroît et nous nous limitons ici à rechercher un invariant de boucle, le reste de la construction étant laissé à la charge du lecteur. Imaginons qu'une partie du travail a été réalisée (voir section 3, page 63), et donc que pour un graphe partiel $G' = (N', V')$ (sous-graphe de G induit par N', contenant le sommet de départ s), on dispose d'une liste associée au « parcours en largeur d'abord » de G', depuis s. Traditionnellement, cette liste est appelée CLOSE. Progresser consiste à allonger cette liste en y ajoutant un sommet, absent de CLOSE, le plus proche possible de s.

En l'absence d'autres hypothèses, la progression est possible, mais difficile à développer autant que coûteuse, puisque tout sommet absent de CLOSE est un candidat possible au

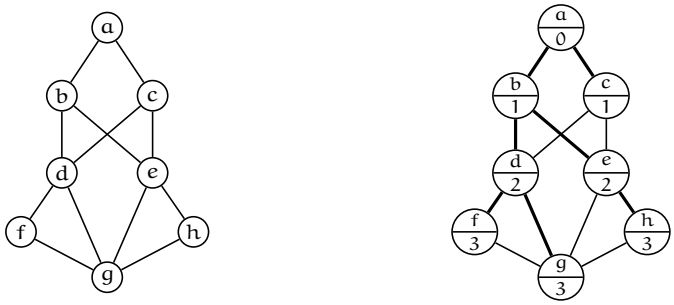


Fig. 7.8 – *Exemple de graphe. Le schéma* (a) *présente le graphe qui illustre les exemples de l'énoncé. Le schéma* (b) *montre, en traits gras, pour le graphe* (a)*, et pour chaque sommet, un plus court chemin depuis le sommet* a *vers tous les autres sommets. Dans le schéma* (b)*, l'entier qui accompagne chaque sommet est la distance par rapport au sommet* a*.*

transfert dans CLOSE. Nous proposons d'enrichir cette première version de l'invariant en lui ajoutant une structure de données (appelons-là OPEN), contenant tous les sommets absents de CLOSE à la condition qu'ils soient voisins d'au moins l'un des sommets de CLOSE. *A priori,* OPEN se présente comme une file de priorité gérée sur les distances de ses éléments par rapport à s, puisque l'élément à déplacer dans CLOSE est celui qui est le plus proche de s. Nous verrons ci-dessous qu'une version simplifiée d'une file de priorité est possible. Il suffit, pour maintenir cette nouvelle version de l'invariant, de déplacer la tête de la file OPEN en queue de la file CLOSE et – c'est la contrepartie du renforcement de l'invariant – d'introduire les « nouveaux » voisins de l'élément déplacé dans la file OPEN, ceux qui ne sont ni dans OPEN ni dans CLOSE (il s'agit d'un choix glouton).

Cependant, étant donné un élément e de OPEN, s'enquérir directement de la présence ou non de l'un de ses voisins dans OPEN ou dans CLOSE peut se révéler coûteux. Une meilleure solution consiste a effectuer un (nouveau) renforcement par la proposition suivante : dans la perspective du coloriage à venir, une « couleur » est attribuée à chaque sommet du graphe, blanc si le sommet est soit dans OPEN, soit dans CLOSE, et gris sinon (en fait, ici, les deux couleurs jouent le rôle de valeurs booléennes). De cette façon, à condition qu'un accès direct aux sommets soit possible, la mise à jour de OPEN est facilitée. Dans la progression, la préservation de ce complément de l'invariant s'obtient en peignant en blanc tout sommet qui rejoint OPEN.

Revenons sur la stratégie de gestion de la file OPEN. Est-il possible d'utiliser, au lieu d'une file de priorité, une simple file FIFO (voir section 1.8, page 34)? Si c'est le cas, la gestion de OPEN s'en trouvera grandement simplifiée. Pour ce faire, lorsque le sommet e quitte OPEN pour rejoindre CLOSE, il faudrait que les voisins de e candidats à l'introduction dans OPEN soient à une distance supérieure ou égale à tous les éléments présents dans OPEN, ce qui permettrait de retrouver une file triée. Ceci revient à dire que, si e est à une distance k de s, tous les autres éléments de OPEN sont à une distance de k ou de $(k+1)$ de s, puisque les voisins « gris » de e sont à la distance $(k+1)$ de s. Nous ajoutons cette hypothèse à notre invariant. Le lecteur vérifiera qu'elle est bien instaurée par

l'initialisation de la boucle. Restera à démontrer qu'elle est préservée par la progression.

Au final, nous proposons l'invariant suivant, constitué de quatre conjoints.

1. CLOSE est une file FIFO dont le contenu représente un « parcours en largeur d'abord » du sous-graphe de G induit par les sommets présents dans CLOSE.
2. OPEN est une file FIFO des sommets voisins des sommets présents dans CLOSE. L'intersection ensembliste de OPEN et de CLOSE est vide.
3. Si la tête de la file OPEN contient un sommet dont la distance à s est k, alors tous les autres éléments de OPEN sont à une distance de k ou de $(k+1)$ de s.
4. Dans le graphe G, les sommets présents, soit dans CLOSE soit dans OPEN, sont coloriés en blanc, les autres sont en gris.

La figure 7.9 page 232, montre les différentes étapes du « parcours en largeur d'abord » du graphe de la figure 7.8 page 230. Dans chaque graphe de la figure, les sommets présents dans CLOSE apparaissent en traits gras, ceux de OPEN sont en traits doubles. Les distances ne sont mentionnées que pour mémoire, l'algorithme de les exploite pas. Commentons par exemple l'étape qui fait passer du schéma (e) au schéma (f). Dans le schéma (e), CLOSE contient la liste de « parcours en largeur d'abord » du sous-graphe induit par les sommets a, b, c et d. Le sommet e, tête de la file OPEN, va se déplacer en queue de la file CLOSE. Quels sont les voisins de e destinés à rejoindre la liste OPEN ? c et b sont déjà dans CLOSE, ils ne sont pas concernés. g est déjà dans OPEN, il n'est pas affecté. Reste le sommet h, qui va venir rejoindre la liste OPEN et se colorier en blanc.

Les structures de données Deux types de structures de données sont utilisées dans cet algorithme. Le premier, les files FIFO, est décrit à la page 34. Le second concerne une variante « coloriée » des graphes.

La structure de données « graphe non orienté colorié » Nous avons besoin de colorier les sommets d'un graphe, de consulter leur couleur et de parcourir la liste des voisins, d'où les définitions suivantes (l'ensemble Couleurs est supposé défini).

— **procédure** $ColorierGr(G, s, coul)$: opération qui colorie le sommet s de G en utilisant la couleur coul.

— **fonction** $CouleurGr(G, s)$ **résultat** Couleurs : fonction qui délivre la couleur du sommet s de G.

— **procédure** $OuvrirVoisinsGr(G, s)$: opération qui initialise le parcours de la liste des voisins du sommet s du graphe G.

— **fonction** $FinListeVoisinsGr(G, s)$ **résultat** $\mathbb{B}$: fonction qui délivre **vrai**, si et seulement si le parcours dans le graphe G de la liste des voisins de s est terminé.

— **procédure** $LireVoisinsGr(G, s, s')$: opération qui délivre dans s' l'identité du sommet « sous la tête de lecture » de la liste des voisins de s, puis qui avance d'une position cette tête de lecture.

Pour cette application, le meilleur raffinement, en termes d'expression de l'algorithme et d'efficacité, est la représentation par liste d'adjacence (voir le schéma (d) de la figure 1.3 page 23 pour une représentation similaire dans le cas des graphes orientés). Le graphe est donc défini comme un *triplet* $(G = (N, V, R))$ où la composante R correspond aux couleurs (pour le moment « blanc » et « gris »).

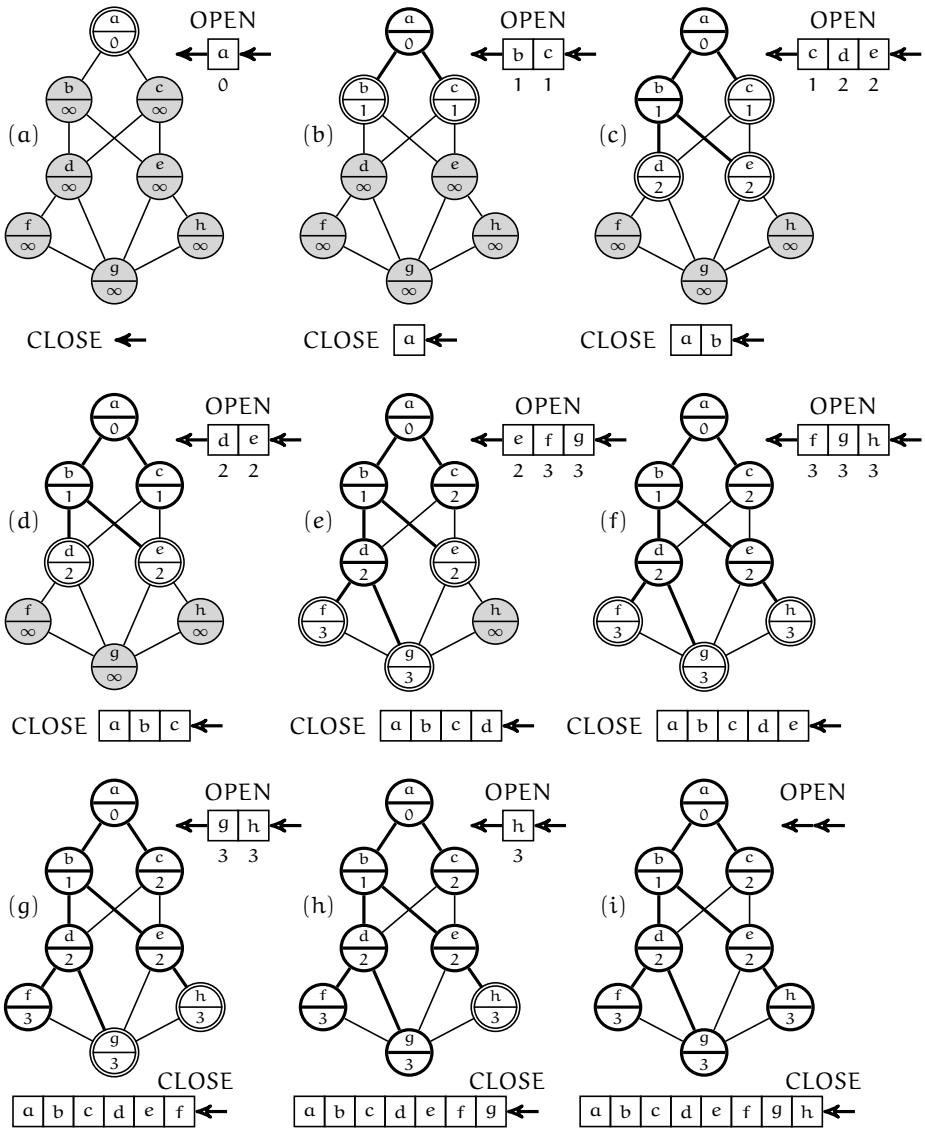


Fig. 7.9 – *Les différentes étapes du « parcours en largeur d'abord » du graphe du schéma* (a) *de la figure 7.8 page 230. Les sommets encerclés en traits gras sont les sommets de* CLOSE, *ceux doublement encerclés sont les sommets de* OPEN. *La valeur entière qui accompagne chaque sommet est la distance connue par rapport au sommet* a. *Les deux files* OPEN *et* CLOSE *sont représentées respectivement au nord-est et au sud des graphes.*

L'algorithme Outre le graphe G, cet algorithme utilise les variables sc (le sommet courant) et v pour parcourir la liste des voisins.

```
1.  constantes
2.    n ∈ ℕ1 et n = ... et N = 1 .. n et Couleurs = {gris, blanc} et
3.    V ∈ N × N et V = {...}
4.  variables
5.    R ∈ N → Couleurs et G = (N, V, R) et
6.    s ∈ N et sc ∈ N et v ∈ N et CLOSE ∈ FIFO(N) et OPEN ∈ FIFO(N)
7.  début
8.    /% coloriage en gris de tous les sommets : %/
9.    pour w ∈ N faire
10.     ColorierGr(G, w, gris)
11.   fin pour ;
12.   InitFifo(CLOSE) ; InitFifo(OPEN) ;
13.   s ← ... ; /% choix du sommet initial : %/
14.   ColorierGr(G, s, blanc) ;
15.   AjouterFifo(OPEN, s) ;
16.   tant que non EstVideFifo(OPEN) faire
17.     sc ← TêteFifo(OPEN) ; SupprimerFifo(OPEN) ;
18.     AjouterFifo(CLOSE, sc) ;
19.     OuvrirGr(G, sc) ;
20.     tant que non FinListeGr(G, sc) faire
21.       LireVoisinsGr(G, sc, v) ;
22.       si CouleurGr(G, v) = gris alors
23.         ColorierGr(G, v, blanc) ;
24.         AjouterFifo(OPEN, v)
25.       fin si
26.     fin tant que
27.   fin tant que ;
28.   écrire(CLOSE)
29. fin
```

Question 1. Quelle est la complexité asymptotique de cet algorithme en termes de conditions évaluées ? 82 - Q 1

Question 2. Expliciter le principe d'un algorithme glouton de coloriage s'appuyant sur cet algorithme. 82 - Q 2

L'algorithme de coloriage de graphe avec deux couleurs

Nous sommes à présent armés pour aborder le problème qui fait l'objet de l'exercice : le coloriage d'un graphe avec les deux couleurs noir et blanc. Il s'agit d'aménager la construction de l'algorithme ci-dessus de façon à colorier alternativement en noir et blanc, selon la profondeur par rapport au sommet de départ, soit jusqu'à épuisement des sommets, soit jusqu'à la découverte d'une impossibilité.

Question 3. Construire l'algorithme de coloriage. 82 - Q 3

Question 4. Montrer, sur le graphe de la figure 7.8 page 230, les différentes étapes du coloriage à partir du sommet a. 82 - Q 4

82 - Q 5 **Question** 5. Fournir le code de l'algorithme, ainsi que sa complexité.

82 - Q 6 **Question** 6. L'exercice 58 page 159 aborde le problème plus général de coloriage avec m ($m \geqslant 2$) couleurs. Discuter de la possibilité de généraliser l'algorithme fourni en réponse à la question 5, au cas $m > 2$.

Remarque Il existe une propriété caractéristique intéressante : un graphe est bicoloriable si et seulement s'il ne contient aucun cycle de longueur impaire. Cependant, cette propriété n'est pas constructive : l'établir ne fournit pas le coloriage.

Exercice 83. D'un ordre partiel à un ordre total : le tri topologique

Deux versions de l'algorithme du tri topologique sont étudiées. La première, naïve mais peu efficace, s'obtient sans difficulté. La seconde exige un renforcement d'invariant ; implantée en termes de pointeurs, elle constitue un excellent exercice de raffinement et de manipulation de structures dynamiques. L'algorithme construit ici s'apparente à l'algorithme de Marimont permettant la mise en niveau d'un graphe sans circuit.

Pour aborder cet exercice, il faut au préalable avoir résolu l'exercice 2, page 35.

Soit $(E, \prec)$ un couple tel que E est un ensemble fini de n éléments et $\prec$ une relation d'ordre partiel sur E. On cherche à construire sur E une relation d'ordre total $\leqslant$ compatible avec $\prec$, c'est-à-dire telle que pour tout couple (a, b) d'éléments de E : $(a \prec b) \Rightarrow (a \leqslant b)$. Un élément sans prédécesseur dans $(E, \prec)$ est appelé *minimum*.

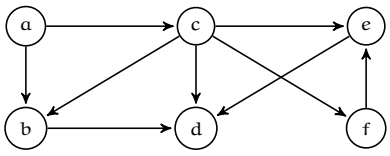
Exemple Dans le cadre d'un cursus informatique, on note $c_1 \prec c_2$ le fait que le module d'enseignement c_1 doit précéder le module c_2 afin de respecter les prérequis nécessaires à la compréhension de ce dernier. Considérons les six modules suivants :

a	Logique du premier ordre	b	Spécification et programmation impérative
c	Théorie des ensembles	d	Conception de systèmes d'informations
e	Bases de données	f	Structures de données

La relation $\prec$ est (par exemple) définie par :

$$a \prec b, a \prec c, b \prec d, c \prec b, c \prec d, c \prec e, c \prec f, e \prec d, f \prec e.$$

Elle peut se représenter par le graphe suivant :



Ce type de graphe se caractérise par le fait qu'il est orienté et qu'il ne contient pas de circuit (en anglais "directed acyclic graph" ou DAG). Dans un tel graphe, un élément sans prédécesseur (un minimum de l'ordre partiel) est appelé *point d'entrée*.

L'exercice a pour objectif de construire un algorithme glouton qui propose un ordre total compatible avec l'ordre partiel fourni en entrée. Pour l'exemple ci-dessus, une solution consiste à proposer l'ordre total a, c, b, f, e, d.

Question 1. Montrer qu'un DAG non vide, privé de l'un quelconque de ses points d'entrée, est encore un DAG. 83 - Q 1

Question 2. On note $G = (N, V)$ un graphe quelconque, et $n = \text{card}(N)$. Montrer qu'il existe des DAG tels que $\text{card}(V) \in \Theta(n^2)$. 83 - Q 2

Nous ébauchons à présent la construction de la boucle « gloutonne » de l'algorithme, avant de l'appliquer à l'exemple ci-dessus. La méthode de construction utilisée se fonde sur la technique de « la course en tête ». La suite des questions porte sur l'algorithme, son raffinement et sa complexité.

Première tentative de construction

Soit $G = (E, V)$ le DAG de n sommets fourni en entrée.

Invariant Soit S la file de sortie contenant l'ensemble E_S ($E_S \subseteq E$) des sommets triés selon un ordre total compatible avec l'ordre $\prec$, et tel que tout sommet v de E n'appartenant pas à E_S ($v \in (E - E_S)$) est supérieur, selon l'ordre partiel, à tout sommet de E_S.

Condition d'arrêt Tous les sommets sont dans la file S, soit : $|S| = n$. La conjonction de l'invariant et de la condition d'arrêt implique bien que S est une liste triée selon un ordre total compatible avec l'ordre partiel.

Progression La progression consiste à insérer dans S l'un des sommets de $(E - E_S)$. Il en résulte que ce sommet est dans le sous-graphe $\mathit{Induit}(G, E - E_S)$. On va renforcer l'invariant dans ce sens.

Afin de choisir le sommet à déplacer en toute connaissance de cause, il faut introduire une structure de données apte à exploiter le sous-graphe $\mathit{Induit}(G, E - E_S)$.

Seconde tentative de construction

Le graphe G devient une variable.

Invariant On adjoint à la version précédente de l'invariant le prédicat : G est un DAG.

Condition d'arrêt Elle est inchangée.

Progression On recherche l'un des points d'entrée de G afin de déplacer ce sommet de $(E - E_S)$ vers la file S. On vérifie facilement que S satisfait alors la première version de l'invariant et que G, le nouveau sous-graphe induit, est bien un DAG (en vertu de la propriété établie en réponse à la question 1).

Remarquons que G joue le rôle de la file d'entrée des algorithmes gloutons et que la conjonction de l'invariant et de la condition d'arrêt implique bien l'objectif visé.

Initialisation L'invariant est instauré en partant d'une file S vide et d'un graphe G qui s'identifie au graphe initial.

Terminaison $n - |S|$ est une fonction de terminaison convenable puisqu'à chaque pas de progression, un élément est déplacé de G vers S.

Notons que cet algorithme fournit par construction un résultat correct. Il fait naturellement « la course en tête ».

83 - Q 3 **Question** 3. Appliquer l'algorithme ci-dessus à l'exemple introductif.

83 - Q 4 **Question** 4. En supposant disponibles la fonction $d_G^-(s)$ qui délivre le demi-degré intérieur d'un sommet s dans un graphe G et la fonction *Induit* (voir section 1.5, page 22, pour les notions liées aux graphes), fournir le code de cet algorithme. Dans l'hypothèse d'une représentation de graphes par listes de successeurs (voir figure 1.3, page 23), montrer que la complexité de cet algorithme est en $\mathcal{O}(n^2)$ en termes de nombres de sommets visités.

83 - Q 5 **Question** 5. Dans la version obtenue en réponse à la question 4, le facteur pénalisant du point de vue de la complexité est la recherche d'un minimum parmi tous les sommets restant à considérer. Proposer, sur la base d'un renforcement de l'invariant ci-dessus, une solution plus efficace pour des graphes peu denses. Que peut-on en conclure quant à l'efficacité de cette solution ?

Exercice 84. Tournois et chemins hamiltoniens

Voici un bien étrange exercice glouton : une file d'entrée sans file et une file de sortie en perpétuelle modification. De surcroît, on constate une surprenante similitude avec le tri par insertion simple. Tous les ingrédients sont réunis pour piquer la curiosité du lecteur.

Soit $G = (N, V)$ un graphe orienté sans boucle (on pose $\text{card}(N) = n$, et $n \geqslant 1$). G est appelé graphe de tournoi (ou plus simplement tournoi) si, pour tout couple de sommets u et v ($u \neq v$), l'un des deux arcs (u, v) ou (v, u) existe (soit $(u, v) \in V$, soit $(v, u) \in V$). L'objectif de l'exercice est de construire un algorithme glouton qui recherche un chemin hamiltonien dans un tournoi.

On rappelle (voir chapitre 1) qu'un chemin élémentaire dans un graphe orienté est un chemin qui ne passe pas deux fois par le même sommet, et qu'un chemin hamiltonien est un chemin élémentaire qui passe par tous les sommets. Le graphe du schéma (a) de la figure 7.10 est un tournoi, tandis que le schéma (b) montre un chemin hamiltonien.

Nous allons tout d'abord montrer de façon constructive que tout tournoi possède au moins un chemin hamiltonien, puis exploiter cette preuve dans un algorithme glouton délivrant un tel chemin. Nous admettons la propriété suivante : « soit $N' \subset N$; le sous-graphe G' induit du tournoi G par N' est aussi un tournoi (la preuve se fait aisément par l'absurde) ».

84 - Q 1 **Question** 1. Cette première question concerne un lemme qui est utilisé pour démontrer l'existence d'un chemin hamiltonien dans un tournoi. Soit une chaîne binaire de longueur $n \geqslant 2$, qui commence par un *0* et finit par un *1*. Montrer qu'elle comporte au moins une fois la sous-chaîne *01*.

84 - Q 2 **Question** 2. Montrer, par récurrence, que tout tournoi possède un chemin hamiltonien.

84 - Q 3 **Question** 3. Construire un algorithme glouton qui produit un chemin hamiltonien quelconque dans un tournoi. Quelle est sa complexité ?

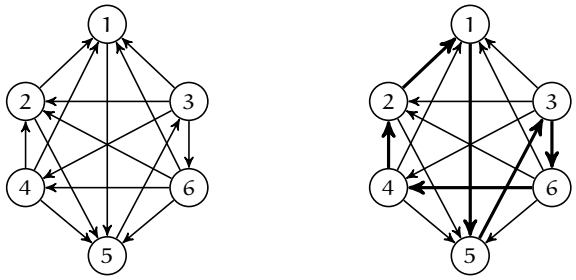


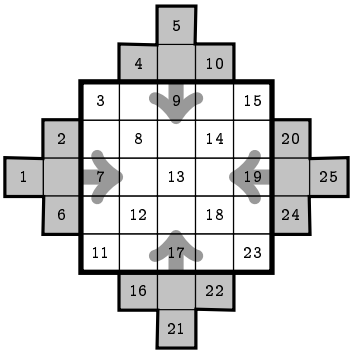
Fig. 7.10 – *Le schéma* (a) *montre un tournoi de six sommets. Le schéma* (b) *met en évidence, dans le graphe* (a)*, le chemin hamiltonien* $\langle 3, 6, 4, 2, 1, 5 \rangle$.

Exercice 85. Carrés magiques d'ordre impair

Cet exercice présente de nombreuses singularités. Tout d'abord, il déroge à la démarche constructive prônée tout au long de cet ouvrage : on se borne à prouver a posteriori *que l'algorithme de Bachet (17ê siècle !), dont on ignore tout de la génèse, délivre un résultat correct. De plus, ici, pas de file de priorité ni de file FIFO en entrée ou en sortie et donc pas de déplacements : la solution de ce problème à propos des carrés magiques appartient-elle bien à la catégorie des algorithmes gloutons ? Le lecteur jugera. Restent une démonstration et un exercice de programmation non triviaux.*

On s'intéresse à la construction de carrés magiques d'ordre impair. Un carré magique d'ordre n est une matrice $n \times n$ dans laquelle apparaît une fois et une seule chacun des entiers de l'intervalle $1 \,..\, n^2$ et tel que la somme des valeurs des lignes, des colonnes et des diagonales principales est la même. Cette somme, notée M_n, est appelée « nombre magique d'ordre n ».

On étudie plus particulièrement la méthode dite de Bachet (d'après Claude-Gaspard Bachet dit de Méziriac, 1612). Cette méthode a comme point de départ un damier crénelé de $(n^2 + (n-1)^2)$ cellules (41 cellules pour $n = 5$), comme le montre le schéma de gauche ci-dessous pour $n = 5$. Dans un tel carré crénelé, il existe des diagonales sud-ouest/nord-est alternativement de n et $(n-1)$ cellules, appelées par la suite respectivement « grandes » diagonales et « petites » diagonales. L'étape suivante consiste à remplir successivement chacune des n « grandes diagonales » avec les n^2 valeurs, en remontant de leur coin inférieur gauche à leur coin supérieur droit. Une fois cette phase achevée, les « petites » diagonales sont vides, mais certaines valeurs sont déjà placées dans le carré central. Elles ne seront pas déplacées.



3	16	9	22	15
20	8	21	14	2
7	25	13	1	19
24	12	5	18	6
11	4	17	10	23

Au regard des autres valeurs (en gris sur le schéma de gauche), laissons la parole à C.-G. Bachet : « tu les mettras dans les places vides qui restent, usant seulement de transpositions, c'est à savoir que ceux d'en haut tu les mettras en bas, et ceux d'en bas tu les porteras en haut ; ceux du côté gauche passeront au côté droit, et ceux du côté droit iront au côté gauche. » Le résultat de cette dernière phase apparaît sur le schéma de droite (en gris les valeurs déplacées). C'est un carré magique.

85 - Q 1 **Question** 1. Calculer la formule qui fournit M_n, le nombre magique d'ordre n. Que valent M_5 et M_7 ?

85 - Q 2 **Question** 2. Construire, selon la méthode de Bachet, le carré magique d'ordre 7.

85 - Q 3 **Question** 3. Montrer que, pour tout n impair, cette méthode construit bien des carrés magiques. On pourra se limiter à montrer que la somme des valeurs situées sur les deux diagonales principales du carré, ainsi que celle d'une ligne quelconque est égale à M_n.

85 - Q 4 **Question** 4. Construire l'algorithme qui produit un carré magique d'ordre n selon la méthode de Bachet.

CHAPITRE 8

Diviser pour Régner

À force de ruminer des choses...
voilà ce que nous découvrîmes :
fallait diviser pour résoudre !...
C'était l'essentiel !...
Tous les emmerdeurs en deux classes !...

(L. F. Céline.)

8.1 Introduction

8.1.1 Présentation et démarche/principe

La méthode Diviser pour Régner (*Divide and Conquer*), DpR en abrégé, a pour principe de casser un problème de taille n en plusieurs sous-problèmes identiques au (de même nature que le) problème initial, ayant des tailles strictement inférieures à celle du problème initial, de résoudre chaque sous-problème, puis de rassembler les résultats pour transformer les solutions des sous-problèmes en une solution globale. Il faut de plus connaître une ou des tailles pour lesquelles le problème n'a pas à être divisé, car il peut être résolu directement. Ces tailles correspondent à des problèmes dits *élémentaires*.

Cette démarche s'apparente à celle utilisée dans le chapitre 4 (« Diminuer pour résoudre, récursivité »), qu'elle généralise quant à la taille des sous-problèmes engendrés. Afin de s'assurer de la correction de la solution proposée, on va le plus souvent recourir à une construction inductive (récurrence forte sur $\mathbb{N}$, induction de partition ou raisonnement par induction à proprement parler) comportant quatre éléments : 1) la *base* dans laquelle on explicite le(s) cas élémentaire(s) et la(les) solution(s) associée(s), 2) l'*hypothèse d'induction* où il est supposé que l'on sait résoudre tout problème de taille $k < n$, 3) l'*induction* à proprement parler où l'on démontre comment résoudre le problème $\mathrm{Pb}(n)$ de taille n en utilisant l'hypothèse d'induction, et 4) la *terminaison* dans laquelle on vérifie que pour toute taille de départ on atteint le(s) cas traité(s) dans la base.

À partir d'une telle construction, on peut préciser le modèle (ou schéma) de résolution par division du problème considéré, en dériver une équation dont la solution donne la complexité temporelle de la solution, et aussi procéder à son codage. Cette démarche méthodologique permet donc de procéder par étapes et d'aboutir à un algorithme « correct par construction ».

Dans la suite de cette introduction, on considèrera un problème générique Pb unidimensionnel, ce qui correspond à de nombreuses situations concrètes et permet une présentation simple et lisible. Cependant, certains exercices illustreront des problèmes bi-dimensionnels

pour lesquels la démarche proposée reste applicable en adaptant les raisonnements, les calculs, le schéma de division et le codage décrits dans le cas unidimensionnel.

Comment cette technique peut-elle s'appliquer en pratique ? Illustrons-la tout d'abord à travers un exemple.

8.1.2 Un exemple : le tri par fusion

Pour présenter la méthode DpR, prenons le problème classique (et fondamental) du tri. Une de ses versions s'inspirant de l'approche DpR est appelée « tri par fusion », ou en raccourci le tri-fusion. Nous décrivons son principe ici ; quelques compléments feront l'objet de l'exercice 86, page 248.

Il s'agit donc de trier un tableau d'entiers de taille n ($n \geqslant 1$) par ordre croissant. Remarquons d'abord qu'il est facile de construire une méthode de tri dont la complexité est en $\Theta(n^2)$ ou en $\mathcal{O}(n^2)$ (l'opération élémentaire est la comparaison). Par exemple, on parcourt tout le tableau pour trouver son plus petit élément et on l'échange avec l'élément de rang 1. Puis on recommence sur les $(n-1)$ éléments non triés (de rang 2 à n), et ainsi de suite jusqu'à la fin. Cet algorithme exige exactement $(n-1)+(n-2)+\ldots+2 = (n\cdot(n-1)/2)-1$ comparaisons. Appelons *TriNaïf*$(1,n)$ cet algorithme qui trie les éléments de 1 à n d'un tableau. Essayons la méthode Diviser pour Régner pour trier ce tableau. Pour simplifier, nous supposerons que $n = 2^p$, même si cette hypothèse n'est pas indispensable comme on le verra dans l'exercice 86, page 248.

Coupons d'abord le tableau en deux moitiés et résolvons les deux sous-problèmes. On applique donc *TriNaïf*$(1, n/2)$ et *TriNaïf*$(n/2+1, n)$, ce qui demande exactement $2\cdot(n/2\cdot(n/2-1)/2-1) = n^2/4 - n/2 - 2$ comparaisons. On a maintenant deux demi-tableaux triés qu'il faut rassembler.

Il est intéressant de constater que ce problème de rassemblement (ici appelé *fusion*) des deux demi-tableaux triés en un seul tableau trié est assez simple. Sans entrer dans les détails, il est aisé de voir qu'il suffit d'avancer pas à pas dans un demi-tableau ou dans l'autre (en comparant l'élément auquel on est arrivé dans chaque demi-tableau) afin d'aboutir à un tableau trié de taille n. L'exercice 86, page 248, permettra de voir sur un exemple comment fonctionne cet algorithme de fusion et de l'écrire précisément.

Cette phase de fusion prend certainement au plus $2 \cdot n/2 = n$ comparaisons. Pour simplifier, on peut dire qu'elle en prend exactement n, une borne supérieure légèrement pessimiste (voir l'exercice 86, page 248).

Revenons à notre problème de départ : on sait que les deux tris prennent $n^2/4-n/2-2$ comparaisons et que le rassemblement en prend n. Au total, on a trié le tableau original avec $(n^2/4+n/2-2)$ comparaisons, ce qui est mieux que les $(n^2/2-n/2-1)$ de départ. Par exemple, si $n = 128$, on a réalisé 4158 comparaisons au lieu de 8127. Ceci constitue un bon début.

Pourquoi s'arrêter en si bon chemin ? Calculons combien de comparaisons prendrait la technique suivante :

- diviser le tableau en quatre sous-tableaux de taille $n/4$,
- trier chaque sous-tableau,
- fusionner deux fois deux sous-tableaux triés de taille $n/4$,
- fusionner deux tableaux triés de taille $n/2$.

Le calcul donne : $(n^2/8+n/2-4+(2n/2+n))$, soit $(n^2/8+3n/2-4)$, c'est-à-dire 2236 pour $n = 128$. Encore mieux !

Il ne reste plus qu'à pousser le raisonnement jusqu'au bout et à diviser jusqu'à ce qu'aucun tri ne soit nécessaire (au sens de l'utilisation de *TriNaïf*) et qu'il ne subsiste que les phases de rassemblement, puisqu'un tableau de taille 1 est par définition trié. En procédant de la sorte, on a construit un algorithme DpR de tri de tout tableau de taille $n = 2^p$ ($p \geqslant 0$) selon le schéma inductif suivant :

Base Un tableau de taille 1 est trié.

Hypothèse d'induction On admet que l'on sait trier tout tableau dont la taille m est puissance de 2 et $m < n$.

Induction Pour trier un tableau dont la taille n est une puissance de 2 ($n > 1$), on le partage en deux moitiés de taille $n/2$ que l'on trie tout d'abord (ce que l'on sait faire d'après l'hypothèse d'induction), puis qui sont fusionnées grâce à la procédure *Fusion*.

Terminaison La taille du tableau diminue strictement à chaque étape pour finalement atteindre 1 la valeur de la base.

Ce raisonnement peut se décrire à travers le *modèle de division* suivant :

$$\begin{array}{ll} \text{TriFusion}(1) \text{ élémentaire} & \\ \text{TriFusion}(n) \to 2 \cdot \text{TriFusion}\left(\frac{n}{2}\right) + \text{Fusion}\left(\frac{n}{2}\right) & n > 1 \end{array}$$

Il est important de remarquer qu'au fond construire un algorithme DpR suppose de ne pas chercher à comprendre ce qui se passe au-delà de la division, mais à raisonner par récurrence, en supposant les problèmes plus petits résolus. En pratique, il ne faut donc pas procéder comme dans l'exemple ci-dessus, mais chercher d'emblée un modèle de division du type de celui que l'on vient de donner.

Pour ce qui est du codage, l'algorithme qui se dessine se dérive de façon canonique du modèle de division. Plus précisément, il s'agit d'une procédure récursive dont la partie principale se compose de deux appels récursifs et d'un appel à la procédure *Fusion* (de profil « **procédure** *Fusion*(p, q, r) » qui fusionne les deux tableaux $T[p..q]$ et $T[q+1..r]$) qui sera étudiée plus en détail à l'exercice 86, page 248. Ici, le cas terminal est vide puisque le problème élémentaire consiste à ne rien faire. Notons que le fait d'avoir choisi $n = 2^p$ assure que les indices calculés lors des divisions par 2 sont toujours des valeurs entières. Si le tableau T à trier est une variable globale, on a donc la procédure ci-après et son code d'appel :

```
1. procédure TriFusion(i, j) pré
2.     i ∈ 1..n et j ∈ i..n
3. début
4.     si i ≠ j alors
5.         TriFusion(i, (i+j)/2) ;
6.         TriFusion((i+j)/2 + 1, j) ;
7.         Fusion(i, (i+j)/2, j)
8.     fin si
9. fin
```

```
1. constantes
2.     n ∈ ℕ1 et ∃p · (p ∈ ℕ et n = 2^p)
       et n = ...
3. variables
4.     T ∈ 1..n → ℕ
5. début
6.     T ← [...] ;
7.     TriFusion(1, n)
8. fin
```

Le calcul de la complexité de cet algorithme (l'opération élémentaire étant la comparaison) pour un tableau de taille $n = 2^p$ s'appuie sur l'équation récurrente suivante :

$$\begin{array}{ll} C(1) = 0 & \\ C(n) = 2 \cdot C\left(\frac{n}{2}\right) + n & n > 1. \end{array}$$

En effet, il est fait deux fois appel à la procédure *TriFusion* sur des données de taille $n/2$ et une fois à la procédure *Fusion* pour deux tableaux de taille $n/2$.

Pour calculer explicitement $C(n)$, on peut utiliser la méthode dite des « facteurs sommants » qui consiste à écrire la récurrence pour les valeurs n, $n/2$, $n/4$, ..., 1, ce qui est possible grâce à l'hypothèse selon laquelle $n = 2^p$:

$$\begin{array}{l} C(n) = 2 \cdot C\left(\frac{n}{2}\right) + n \\ C(\frac{n}{2}) = 2 \cdot C\left(\frac{n}{4}\right) + \frac{n}{2} \\ \cdots \\ C(1) = 0 \end{array}$$

Il y a $p = \log_2(n)$ lignes. En multipliant la première par 1, la seconde par $2, \ldots,$ la dernière par 2^p et en additionnant toutes ces égalités terme à terme, on trouve :

$$C(n) = n + 2 \cdot \frac{n}{2} + 4 \cdot \frac{n}{4} + \cdots + 2^{p-1} \cdot \frac{n}{2^{p-1}} + 0 = n + n + \cdots + n + 0 = (p-1) \cdot n \leqslant n \cdot \log_2(n).$$

Par la méthode DpR, on aboutit donc à un algorithme de complexité $\mathcal{O}(n \cdot \log_2(n))$ alors que l'algorithme de tri naïf n'est pas en $\mathcal{O}(n \cdot \log_2(n))$, mais en $\Theta(n^2)$.

8.1.3 Schéma général de Diviser pour Régner

On s'intéresse maintenant à l'algorithme DpR générique de résolution d'un problème Pb de taille n, noté $Pb(n)$. Comme il a été dit auparavant, le mécanisme de résolution s'appuie le plus souvent sur une étape de construction inductive aboutissant à un modèle de division ayant deux constituants :

Cas terminal Il répertorie les tailles pour lesquelles on connaît la solution directe du problème considéré (problèmes élémentaires).

Cas général La solution de $Pb(n)$ s'exprime comme la composition des solutions de a (sous-)problèmes de même nature que Pb, de tailles strictement inférieures à n.

On a donc le modèle de division générique :

$$\begin{array}{ll} Pb(m_1) \text{ élémentaire}, \cdots, Pb(m_k) \text{ élémentaire} & \\ Pb(n) \to a \cdot Pb(n_i) + Rassembler(n) & n \cdots \end{array}$$

où $n_i < n$ et $a \geqslant 1$, et $Rassembler(n)$ est la fonction qui rassemble les résultats des a sous-problèmes de taille $n_1, \ldots, n_a$ pour construire celui du problème Pb.

On en déduit ensuite (le plus souvent très aisément) un algorithme proprement dit dont la version générique s'écrit :

```
1.  procédure DpR(m; ResG : modif) pré
2.     m ∈ ℕ et ResG ... et ResP_1 ... et ... et ResP_a ... et
3.     n_1 ∈ ℕ et ... et n_a ∈ ℕ
4.  début
5.     si m ∈ {m_1, ..., m_k} alors
6.        si m = m_1 alors
7.           ResG ← solution du problème élémentaire
8.           ...
9.        sinonsi m = m_k alors
10.          ResG ← solution du problème élémentaire
11.       fin si
12.    sinon
13.       DpR(n_1, ResP_1) ;
14.       ... ;
15.       DpR(n_a, ResP_a) ;
16.       ResG ← Rassembler(ResP_1, ..., ResP_a)
17.    fin si
18. fin
```

dont l'appel se présente sous la forme :

```
1. constantes
2.    n ∈ ℕ_1 et n = ...
3. variables
4.    R...
5. début
6.    DpR(n, R)
7. fin
```

Ainsi, pour résoudre un problème de taille m, on le divise en a problèmes qui fournissent les résultats partiels $ResP_1, \ldots, ResP_a$. Ceux-ci sont ensuite utilisés par la procédure *Rassembler* pour la construction du résultat global ResG. Une étape non triviale d'explicitation et de construction des sous-problèmes peut être nécessaire dans certains problèmes, comme l'illustrent notamment les exercices 93 page 254, et 94 page 256. On l'a omise dans la procédure générique précédente afin de ne pas l'alourdir.

Notons enfin qu'une formule de récurrence (au sens usuel) constitue en elle-même un modèle de division pour des problèmes particuliers où on calcule une grandeur numérique (voir exercice 103, page 268). Dans ce genre de situation, Rassembler est constituée simplement de l'ensemble des calculs agrégeant les termes de la récurrence.

8.1.4 Une typologie des algorithmes Diviser pour Régner

Dans bon nombre d'algorithmes du type Diviser pour Régner, la division des données est faite « à la moitié », c'est-à-dire par exemple que dans le cas d'un tableau $T[1..n]$, la division se fait en *deux* sous-tableaux *contigus* $T[1..\lfloor n/2 \rfloor]$ et $T[\lfloor n/2 \rfloor + 1..n]$. Le procédé de division se répète alors régulièrement jusqu'à atteindre la taille 1 pour laquelle on connaît une solution directe ($Pb(1)$ est alors le seul problème élémentaire). En admettant que $n/2$ désigne indifféremment $\lfloor n/2 \rfloor$ ou $\lceil n/2 \rceil$, on aboutit alors à l'un des deux schémas suivants :

$$\text{Pb}(1) \text{ élémentaire}$$
$$\begin{cases} \text{Pb}(n) \to 2 \cdot \text{Pb}\left(\frac{n}{2}\right) + \text{Rassembler}(n) \\ \quad \text{ou} \\ \text{Pb}(n) \to \text{Pb}\left(\frac{n}{2}\right) \end{cases} \qquad n > 1$$

selon que la résolution nécessite la solution des deux sous-problèmes ou d'un seul d'entre eux.

Certains algorithmes relatifs à des tableaux ne divisent pas en 2, mais en un nombre *supérieur* de parties. Dans certains problèmes, il peut arriver que la division ne se fasse pas en sous-tableaux contigus, mais *entrelacés*. Dans ce cas, les données d'un tableau peuvent par exemple être divisées en données d'indice pair et en données d'indice impair (voir exercice 109, page 291). La division est parfois faite en deux tableaux contigus, mais pas à la moitié : l'indice de division est alors choisi de façon *non déterministe*, ou calculé par une méthode plus raffinée. Une autre possibilité est que la division se fasse à un endroit (pour la division contiguë) ou selon un procédé (pour la division entrelacée) qui *dépend des données* sur lesquelles on travaille, c'est-à-dire que l'algorithme ne choisira pas la même division selon qu'il s'applique à un tableau ou à un autre (voir exercice 114, page 306). Enfin, il existe un [1] algorithme DpR dont le nombre de divisions en sous-tableaux dépend pour partie de la *taille* du tableau. On le présente sous le nom de « lâchers d'œufs par la fenêtre » (exercice 112, page 299).

Pour illustrer cette grande variété, on a choisi de présenter les types d'algorithmes DpR, à la figure 8.1 page 245, sous la forme d'un arbre de décision (voir chapitre 1) dans lequel les critères précédents sont mis en évidence. Le résultat est que les feuilles de l'arbre, c'est-à-dire les familles distinctes selon notre typologie, sont de taille très variable. Il existe une variété foisonnante d'algorithmes du type 4, c'est-à-dire pour lesquels on opère une division en deux parties contiguës de tailles égales (ou différentes de 1). En revanche, on n'a rencontré que peu d'algorithmes où la division se fait aussi en deux parties contigües, mais qui dépendent des données ; de même, les algorithmes où la division est entrelacée sont exceptionnels [2].

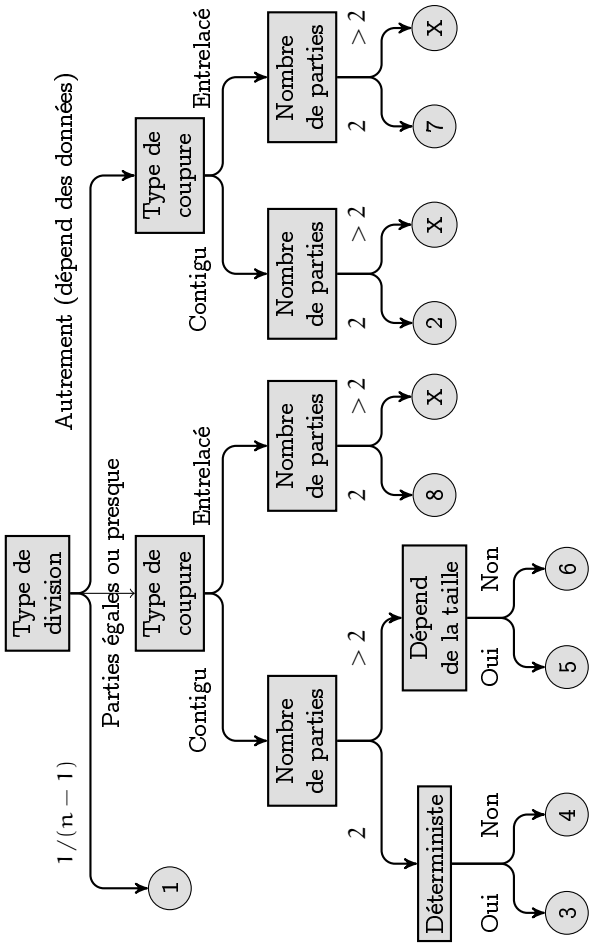
D'autres algorithmes sont les seuls représentants (ou presque) de leur catégorie. Ce qui peut être dû soit à l'extrême singularité de l'énoncé (types 5 et 6), soit à l'art des inventeurs (types 2 et 7).

8.1.5 Complexité de Diviser pour Régner

Introduction au théorème maître

La complexité temporelle d'un algorithme DpR s'exprime en fonction d'(au moins) une opération élémentaire apparaissant dans l'instance du modèle de division. Le plus souvent, celle-ci est caractéristique du problème à traiter (condition évaluée pour une recherche dans une structure tabulaire, tracé élémentaire de segments dans la réalisation d'un dessin ou encore pesée pour la recherche d'une fausse pièce). Cette complexité dépend fortement du modèle de division utilisé pour résoudre le problème considéré. Cependant, vu sa fréquence élevée, le modèle de division suivant pour une donnée de taille n :

1. Les auteurs ne connaissent pas d'autres exemples que celui-là.
2. Pourtant, la *Fast Fourier Transform*, ou FFT, qui est de ce type, est l'un des quelques algorithmes les plus utilisés en pratique et se trouve même comme fonction intégrée au tableur Excel.



1. Algorithmes *Diminuer pour résoudre* (récursion classique : voir chapitre 4)
2. Algorithme de Hirschberg (exercice 106, page 278)
3. *Quicksort* de base. Le problème de la sélection (exercice ??, page ??)
4. Le tri fusion (exercice 86, page 248). Et beaucoup d'autres ...
5. Lâchers d'œufs par la fenêtre (exercice 112, page 299)
6. La pièce fausse (exercice 95, page 257)
7. La sous-séquence bègue (exercice 108, page 289)
8. La FFT (exercice 109, page 291)

Fig. 8.1 – *Une typologie des algorithmes Diviser pour Régner. Les feuilles marquées X n'ont pas de représentant dans les exercices.*

$$\begin{array}{ll} \text{Problème}(1) \text{ élémentaire} & \\ \text{Problème}(n) \rightarrow a \cdot \text{Problème}\left(\frac{n}{b}\right) + \text{Rassembler}(n) & n > 1 \end{array}$$

(n/b représente indifféremment $\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$) mérite une attention toute particulière.

Ce modèle est appelé *réduction logarithmique* puisque la taille du problème diminue d'un facteur b à chaque pas. Pour ce qui concerne l'évaluation de la complexité temporelle, ce modèle générique se traduit par une équation récurrente $C(n)$ du type :

$$\begin{array}{ll} C(1) = c & \\ C(n) = a \cdot C\left(\frac{n}{b}\right) + f(n) & n > 1 \end{array}$$

où :

- c ($c \in \mathbb{N}$) représente le coût du traitement d'une donnée de taille 1,
- a ($a \in \mathbb{N}_1$) est le nombre de sous-problèmes traités récursivement,
- n/b ($n/b \in \mathbb{N}$) est l'entier représentant la taille de chaque sous-problème ($b > 1$),
- La fonction f étant positive ou nulle, $f(n)$ est le coût du traitement *Rassembler* mentionné ci-dessus et concerne le coût du rassemblement des solutions partielles pour obtenir la solution globale.

Remarques

1. La première équation, $C(1) = c$ ne joue aucun rôle particulier dans le théorème maître présenté ci-dessous. Celui-ci reste valable dès lors que cette équation se présente sous la forme $C(d) = c$, d étant une constante, pour autant que la récurrence s'exprime sur $n > d$. Cette équation est donc ignorée dans la suite.
2. Dans le cas où l'unité de mesure de la complexité est la condition (c'est-à-dire une expression booléenne contrôlant les boucles ou des alternatives), il conviendrait d'ajouter 1 à chaque partie droite de l'équation récurrente pour tenir compte de l'évaluation de la condition de l'alternative typique des algorithmes de type DpR. Cependant, asymptotiquement cette constante ne joue aucun rôle. Elle est donc souvent ignorée.
3. Dans le cas d'un passage de paramètres par valeur et selon l'opération élémentaire choisie, le coût de la copie peut devoir être pris en compte dans l'équation.

Théorème (Théorème maître) :
Soit $C(n) = a \cdot C(n/b) + f(n)$, *avec* $a \geqslant 1$, $b > 1$, $f(n)$ *une fonction positive ou nulle et* n/b *représentant indifféremment* $\lfloor n/b \rfloor$ *ou* $\lceil n/b \rceil$.

1. *Si* $f(n) \in \mathcal{O}(n^{\log_b(a)-\varepsilon})$ *pour une constante* $\varepsilon > 0$, *alors* $C(n) \in \Theta(n^{\log_b(a)})$.
2. *Si* $f(n) \in \Theta(n^{\log_b(a)})$, *alors* $C(n) \in \Theta(n^{\log_b(a)} \cdot \log_b(n))$.
3. *Si* $f(n) \in \Omega(n^{\log_b(a)+\varepsilon})$ *pour une constante* $\varepsilon > 0$ *et si* $a \cdot f(n/b) \leqslant k \cdot f(n)$ *pour une constante* $k < 1$ *et pour* n *suffisamment grand, alors* $C(n) \in \Theta(f(n))$.

Une démonstration de ce théorème (le *master theorem*) est présentée dans [17]. Il a pour corollaire le théorème suivant.

Corollaire du théorème maître Soit $C(n)$ la récurrence $C(n) = a \cdot C(n/b) + p(n)$, où a et b sont des constantes telles que $a \in \mathbb{N}_1$, $b \in \mathbb{N}_1$ et $b > 1$ et $p(n)$ un polynôme positif ou nul de degré k.

1. Si $a > b^k$, alors $C(n) \in \Theta(n^{\log_b(a)})$.
2. Si $a = b^k$, alors $C(n) \in \Theta(n^k \cdot \log_b(n))$.
3. Si $a < b^k$, alors $C(n) \in \Theta(n^k)$.

Ces théorèmes permettent de démontrer les cas particuliers suivants que l'on retrouve fréquemment dans la suite de ce chapitre :

$$C(n) = C\left(\frac{n}{2}\right) + p(n) \quad \begin{cases} k = 0, \text{ cas 2 du corollaire : } C(n) \in \Theta(\log_2(n)) & (8.1) \\ k \geqslant 1, \text{ cas 3 du corollaire : } C(n) \in \Theta(n^k) & (8.2) \end{cases}$$

$$C(n) = 2 \cdot C\left(\frac{n}{2}\right) + p(n) \quad \begin{cases} k = 0, \text{ cas 1 du corollaire : } C(n) \in \Theta(n) & (8.3) \\ k = 1, \text{ cas 2 du corollaire : } C(n) \in \Theta(n \cdot \log_2(n)) & (8.4) \\ k \geqslant 2, \text{ cas 3 du corollaire : } C(n) \in \Theta(n^k) & (8.5) \end{cases}$$

$$C(n) = 4 \cdot C\left(\frac{n}{2}\right) + p(n) \quad \begin{cases} k \leqslant 1, \text{ cas 1 du corollaire : } C(n) \in \Theta(n^2) & (8.6) \\ k = 2, \text{ cas 2 du corollaire : } C(n) \in \Theta(n^2 \cdot \log_2(n)) & (8.7) \\ k \geqslant 3, \text{ cas 3 du corollaire : } C(n) \in \Theta(n^k) & (8.8) \end{cases}$$

$$C(n) = 3 \cdot C\left(\frac{n}{2}\right) + p(n) \quad k \leqslant 1, \text{ cas 1 du corollaire : } C(n) \in \Theta(n^{\log_2(3)}) \quad (8.9)$$

$$C(n) = 3 \cdot C\left(\frac{n}{4}\right) + p(n) \quad k = 1, \text{ cas 3 du corollaire : } C(n) \in \Theta(n) \quad (8.10)$$

Autres types d'équations de récurrence

En revanche, les théorèmes précédents ne permettent pas de conclure en ce qui concerne par exemple la récurrence suivante (où $c \in \mathbb{N}_1$) :

$$C(n) = 2 \cdot C\left(\frac{n}{2}\right) + c \cdot n \cdot \log_2(n).$$

Un exemple de ce type se retrouve dans l'exercice 105, page 274, où il fait l'objet de la question 3. Dans cet exercice, la démonstration est faite i) que le théorème maître ne permet pas de conclure, ii) que $C(n) \in \Theta(n \cdot (\log_2(n))^2)$ (pour n puissance de 2).

Il n'est pas rare de rencontrer des équations qui se présentent sous la forme :

$$C(n) = a \cdot C\left(\frac{n}{b} + k\right) + c \cdot n + g$$

avec $a \in \mathbb{N}_1$, $b \in \mathbb{N}_1$, $k \in \mathbb{N}_1$, $c \in \mathbb{N}_1$ et $g \in \mathbb{N}$. Strictement parlant, le théorème maître ne s'applique pas (ni son corollaire par conséquent). Une démonstration au cas par cas est possible. Ainsi, une instance du théorème ci-dessous (pour $k = 1$, $e = 2$ et $g = 2$), est utilisée et démontrée dans la solution de l'exercice 113, page 303.

Théorème :
Soit $C(n)$ *la récurrence* $C(n) = C(\lfloor n/2 \rfloor + k) + c \cdot n + g$ *avec* $k > 0$, $c \in \mathbb{N}_1$ *et* $g \in \mathbb{N}$. *Alors on a :*

$$C(n) \in \mathcal{O}(n).$$

On observe une grande variété de classes de complexité associées aux algorithmes DpR. Ceci nous conduit à étudier de façon systématique leur complexité dans les exercices qui suivent.

8.1.6 À PROPOS DE LA TAILLE DU PROBLÈME

De façon générale, l'objectif visé dans les exercices de ce chapitre consiste à élaborer au moins une solution « efficace » (au sens de la complexité temporelle) et valide pour toute taille n de problème. Il pourra arriver que l'on s'intéresse tout d'abord à une solution valable seulement pour des tailles particulières. C'est le cas, par exemple, pour le tri-fusion dans la version étudiée auparavant. Ceci peut permettre de simplifier à la fois la présentation de la solution et le calcul de complexité associé, quitte à généraliser ensuite (voir l'exercice 86, page 248, pour ce qui concerne le tri-fusion). La complexité temporelle dépend elle aussi de la taille du problème et fera systématiquement l'objet de questions. Elle concernera donc soit des tailles particulières, soit le cas général selon la spécification de la solution recherchée. De façon exceptionnelle, dans le cadre d'une solution valide pour toute taille n, on se contentera de calculer la (classe de) complexité pour des tailles particulières, soit parce que cela suffira à caractériser la complexité de la solution considérée, soit pour éviter des développements trop longs.

8.2 Ce qu'il faut retenir de la démarche DpR

La démarche DpR est une brique essentielle de l'arsenal des méthodes à la disposition de l'informaticien. Elle apparaît souvent comme « miraculeuse » aux yeux du débutant. C'est que sa parfaite maîtrise passe par la compréhension des liens intimes qu'elle entretient avec l'induction mathématique. Cette relation étant acquise, son application à un problème particulier peut alors se voir comme un contournement de ce problème dans la mesure où le résoudre revient i) à composer deux ou plusieurs solutions, ii) à trouver une solution pour un problème de petite taille. Cependant, l'optimalité de la solution en termes de complexité n'étant pas systématique, il est en général nécessaire d'évaluer son efficacité. Là aussi les mathématiques (le théorème maître de la page 246) peuvent venir au secours du développeur. En synergie avec d'autres méthodes, comme la programmation dynamique (voir par exemple l'exercice 106, page 278) ou la transformation de domaine (voir par exemple l'exercice 109, page 291), la démarche DpR peut être à la base de joyaux algorithmiques aussi esthétiques qu'efficaces.

8.3 Exercices

Exercice 86. Le tri-fusion

L'aspect purement DpR de cet algorithme et l'intérêt de cette approche ont été abordés à la section 8.1.2, page 240. Le présent exercice est simplement destiné à approfondir quelques points laissés en suspens dans cette section introductive. C'est en particulier le cas de l'algorithme de fusion. Celui-ci est traité ici de manière purement itérative.

Le lecteur est invité à reprendre les éléments de l'énoncé présentés à partir de la page 240.

Question 1. On considère que les sous-tableaux T[p..q] et T[q+1..r] sont triés. Le tableau T étant supposé global, écrire l'algorithme *Fusion*(p, q, r) qui accepte les indices p, q et r en entrée et fusionne les tranches T[p .. q] et T[q + 1 .. r] en un tableau trié T[p .. r]. 86 - Q 1

Question 2. Combien de comparaisons entre éléments de T nécessite la fusion des tableaux [3, 6, 7, 9] et [4, 5, 10, 12] ? des tableaux [3, 4, 5, 6] et [7, 9, 10, 12] ? des tableaux [3, 5, 7, 10] et [4, 6, 9, 12] ? Combien de conditions doivent être évaluées au pire pour la fusion des tableaux T[p .. q] et T[q + 1 .. r] ? 86 - Q 2

Question 3. Quelles adaptations faut-il faire dans la procédure *TriFusion* si la taille de T n'est pas une puissance de 2 ? 86 - Q 3

Exercice 87. Recherches dichotomique, trichotomique et par interpolation

Cet exercice sur la recherche dichotomique est un classique de l'application du principe DpR. Cependant, une extension à la trichotomie (c'est-à-dire à la division récursive par 3) est proposée. Se pose alors le problème de la comparaison des complexités de ces deux solutions. Une solution complémentaire au problème de la recherche de l'existence d'une valeur dans un tableau fait l'objet de la dernière question : la recherche par interpolation. Bien que simple dans son principe, cet algorithme exige une grande rigueur pour obtenir une solution correcte.

On considère un tableau T[1 .. n] ($n \in \mathbb{N}_1$), d'entiers tous différents, trié par ordre croissant. On cherche à savoir si l'entier v s'y trouve. Les deux premières questions portent sur les algorithmes et leur construction, les questions 3, 4 et 5 sont consacrées à la comparaison des complexités exactes, la dernière question concerne la recherche par interpolation.

Question 1. On recherche une solution DpR fondée sur une division en deux sous-tableaux de tailles approximativement égales. Parmi les nombreuses versions possibles, on se focalise sur la version dite de Bottenbruch qui ne teste *l'égalité* entre v et un élément du tableau que si le tableau en question ne possède *qu'un seul* élément. Construire cette solution et en déduire le modèle de division puis le code. 87 - Q 1

Question 2. On s'intéresse maintenant à une solution trichotomique. Il existe plusieurs façons de diviser un tableau en trois sous-tableaux de tailles approximativement égales. 87 - Q 2

a) On peut par exemple diviser le tableau (de longueur n) en trois sous-tableaux de tailles respectives $\lfloor n/3 \rfloor$, $\lfloor n/3 \rfloor$ et $(\lfloor n/3 \rfloor + (n \bmod 3))$. Montrer que la courbe qui dénombre les comparaisons pour la recherche de $v \geqslant T[n]$ n'est pas monotone. Conclusion ?

b) Il existe également une solution dite « par nécessité », qui commence par considérer le premier sous-tableau de longueur $\lfloor n/3 \rfloor$ puis, si nécessaire, divise par 2 le résidu pour fournir un second sous-tableau de taille $\lfloor (n - \lfloor n/3 \rfloor)/2 \rfloor$ et un troisième de taille $\lceil (n - \lfloor n/3 \rfloor)/2 \rceil$. On s'impose à nouveau la contrainte de Bottenbruch : le test sur *l'égalité* entre v et un élément du tableau n'intervient que si le tableau ne possède pas plus de deux éléments. Montrer que les trois sous-tableaux ont des tailles respectives de $\lceil (n-2)/3 \rceil$, $\lceil (n-1)/3 \rceil$ et $\lceil n/3 \rceil$. Construire cette solution et fournir le modèle de division puis le code.

Dans les trois questions qui suivent, on s'intéresse à la complexité exacte au pire des deux algorithmes développés ci-dessus, complexité exprimée en nombre de comparaisons entre v et un élément du tableau. L'objectif annoncé est de montrer que, dans le pire des cas, la solution trichotomique n'est *jamais* meilleure que la solution dichotomique. Pour ce faire, on procède de la manière suivante. On cherche à déterminer $C_2(n)$, complexité au pire de la recherche dichotomique. On fait de même pour $C_3(n)$ et la recherche trichotomique, avant de comparer les fonctions $C_2(n)$ et $C_3(n)$.

87 - Q 3 **Question** 3. La solution la pire pour la recherche dichotomique[3] est atteinte quand $v \geqslant T[n]$. En conséquence, montrer que l'équation récurrente qui définit $C_2(n)$ est :

$$C_2(n) = \lceil \log_2(n) \rceil + 1. \tag{8.11}$$

87 - Q 4 **Question** 4. Pour la recherche trichotomique, on va mettre en évidence l'équation récurrente $C_3(n)$, avant d'en rechercher une solution.

a) Pour une taille donnée n du tableau, l'ensemble des exécutions possibles de l'algorithme de recherche par trichotomie peut être représenté par un arbre de décision (voir chapitre 1 et exercice 94, page 256, pour un autre exemple utilisant les arbres de décision). Dans notre cas, l'arbre de décision est un arbre binaire dans lequel chaque nœud matérialise une comparaison (de type $\leqslant$, $>$, $\neq$ ou $=$) entre v et un élément $T[i]$. Fournir les arbres de décision A_n de la trichotomie pour $n \in 1..7$. Par quelle relation la hauteur $h(A_n)$ de l'arbre est-elle liée à la complexité au pire $C_3(n)$ de l'algorithme ?

b) En adoptant la notation $\langle A_g, T[i], A_d \rangle$ pour représenter l'arbre de décision $\begin{array}{ccc} & T[i] & \\ \swarrow & & \searrow \\ A_g & & A_d \end{array}$, fournir une définition inductive de A_n. En déduire une définition inductive de sa hauteur h. Montrer que la fonction h est monotone (au sens large). Que conclure ?

c) Fournir l'équation récurrente définissant $C_3(n)$. Soit l'ensemble E ($E \subset \mathbb{N}_1$) défini par :

$$E = \{3^0\} \cup \bigcup_{p \geqslant 0} (2 \cdot 3^p + 1 \,..\, 3^{p+1})$$

et soit 1_E sa fonction caractéristique. Montrer que :

$$C_3(n) = 2 \cdot \lceil \log_3(n) \rceil + 1_E(n). \tag{8.12}$$

87 - Q 5 **Question** 5. Montrer, en utilisant pour C_2 et C_3 les représentations de votre choix, que pour tout $n \in \mathbb{N}_1$, $C_2(n) \leqslant C_3(n)$.

87 - Q 6 **Question** 6. Cette dernière question est consacrée à la recherche par interpolation. En général, la recherche d'une entrée dans un dictionnaire ne s'effectue pas par dichotomie, mais exploite l'estimation de la position de la valeur recherchée pour ouvrir le dictionnaire sur une page susceptible de contenir l'entrée en question. C'est le principe de la recherche par interpolation. Construire l'opération correspondante, fournir le modèle de division ainsi que le code de l'opération. Comparer avec la recherche dichotomique.

3. Le lecteur insatisfait par cette affirmation pourra s'inspirer de la question 4 pour la démontrer.

Exercice 88. Recherche d'un point fixe

Au premier abord, cet exercice n'est qu'un exemple de plus sur la recherche dichotomique (voir par exemple les exercices 87, page 249, et 89, page 251). On s'attend donc à obtenir un algorithme dont la complexité est en $\Theta(\log_2(n))$. Cependant, et c'est l'originalité de cet exercice, l'exploitation fine de sa spécification conduit à distinguer différents cas de figure. La seconde question se caractérise par une évaluation extrêmement simple de la complexité moyenne.

Soit T[1..n] ($n \in \mathbb{N}_1$) un tableau, trié par ordre croissant, d'entiers *relatifs* tous distincts. On désire construire l'opération « **fonction** *PointFixe* **résultat** $\mathbb{B}$ » qui permet de savoir s'il existe au moins un point fixe dans T, c'est-à-dire s'il existe un indice p tel que T[p] = p (on ne recherche pas la valeur de p).

Question 1. Construire une solution à ce problème. Fournir le code de l'opération *PointFixe* (T est un tableau global). Que peut-on dire de la complexité de cette opération ? 88 - Q 1

Question 2. On considère à présent que T est un tableau, trié par ordre croissant, d'entiers *naturels positifs* tous distincts. Construire la nouvelle version de l'opération « **fonction** *PointFixe* **résultat** $\mathbb{B}$ », fournir son code et sa complexité moyenne. 88 - Q 2

Exercice 89. Le pic

Cet exercice est l'un des nombreux exemples d'application du principe de la recherche dichotomique, qui se décline de manière itérative ou récursive. On se limite ici à la solution récursive.

Par définition, un tableau d'entiers T[deb .. fin] (deb .. fin $\neq \varnothing$) présente un pic en position p si et seulement si : i) T est injectif (toutes les valeurs de T sont différentes), ii) T[deb .. p] est trié par ordre croissant, iii) T[p .. fin] est trié par ordre décroissant.

Question 1. Construire une solution DpR au problème de la recherche du pic T[p] dans un tel tableau. On ne demande pas le code de l'opération. 89 - Q 1

Question 2. Donner le modèle de division de la construction précédente. Quel est l'ordre de grandeur de complexité de cette solution (l'opération élémentaire retenue est la comparaison) ? 89 - Q 2

Exercice 90. Tableau trié cyclique ∘ ⁝

Le principal atout de l'exercice est de montrer que l'on peut effectuer une recherche dans un tableau « presque trié » (dans le sens de « tableau cyclique » défini ci-dessous) avec (asymptotiquement parlant) une efficacité comparable à celle de la recherche dichotomique dans un tableau trié.

Un tableau cyclique trié $T[1\,..\,n]$ ($n \in \mathbb{N}_1$) est un tableau d'entiers naturels (sans doublons), dans lequel il existe une frontière f ($f \in 1\,..\,n$) telle que les deux sous-tableaux $T[1\,..\,f]$ et $T[f+1\,..\,n]$ sont triés par ordre croissant et telle que tous les éléments de $T[1\,..\,f]$ sont supérieurs à ceux de $T[f+1\,..\,n]$.

Par exemple, le tableau T suivant :

i	1	2	3	4	5	6	7
T[i]	9	11	12	13	2	5	8

est trié cyclique, sa frontière f est en position 4. Remarquons que, puisque les doublons sont interdits, dans un tableau trié cyclique la frontière est unique et qu'un tableau trié non vide est un tableau trié cyclique.

Plus formellement, en généralisant aux tableaux sans doublons définis sur un intervalle $i\,..\,s$, on définit le prédicat $\mathit{EstTriéCycl}(T[i\,..\,s])$ par :

$$\mathit{EstTriéCycl}(T[i\,..\,s]) \mathrel{\widehat{=}} i \leqslant s \text{ et } \exists f \cdot \left(f \in i\,..\,s \text{ et } \left(\begin{array}{l} \mathit{EstTrié}(T[i\,..\,f]) \text{ et} \\ \mathit{EstTrié}(T[f+1\,..\,s]) \text{ et} \\ \forall (j,k) \cdot (j \in i\,..\,f \text{ et} \\ k \in f+1\,..\,s \Rightarrow T[j] > T[k]) \end{array} \right) \right)$$

90 - Q 1 **Question 1.** Démontrer les propriétés suivantes :

Propriété 9 :
Si EstTriéCycl(T[i..s]), *alors l'élément suivant (circulairement) la frontière* f *dans* T[i..s] *est le plus petit élément de* T[i..s]. *Plus formellement, si* f *est la frontière,* $T[((f-i+1) \bmod (s-i+1)) + i] = \min(\mathit{codom}(T[i\,..\,s]))$.

Propriété 10 :
Soit $T[i\,..\,s]$ $(i < s)$ *un tableau cyclique trié et* $m \in i\,..\,s-1$. *Les sous-tableaux* $T[i\,..\,m]$ *et* $T[m+1\,..\,s]$ *sont des tableaux cycliques triés et au moins l'un des deux est trié.*

Propriété 11 :
Soit $T[i\,..\,s]$ $(i < s)$ *un tableau cyclique trié et* $m \in i\,..\,s-1$. *Si* $T[m\,..\,s]$ *est trié, alors le plus petit élément de* $T[i\,..\,s]$ *appartient au sous-tableau* $T[i\,..\,m]$, *sinon il appartient au sous-tableau* $T[m+1\,..\,s]$.

Propriété 12 :
Soit $T[i..s]$ $(i \leqslant s)$ *un tableau cyclique trié ;* $T[i..s]$ *est trié si et seulement si* $T[i] \leqslant T[s]$.

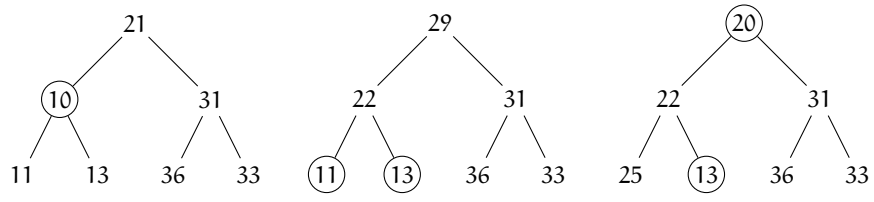
Question 2. Construire, selon une approche DpR, l'opération « **fonction** *PlusPetit*(i, s) **résultat** $\mathbb{N}$ » qui détermine le plus petit élément du tableau trié cyclique T[i .. s] (T est supposé être un tableau global). On montrera dans la partie inductive du raisonnement comment on peut déduire des propriétés précédentes le critère sur lequel on peut décider du demi-tableau dans lequel rechercher le plus petit élément d'un tableau cyclique trié T. Quel est le modèle de division qui s'applique ? Fournir le code de l'opération. Quelle est, en nombre de conditions évaluées, la complexité de cette fonction ? 90 - Q 2

Question 3. On cherche à obtenir selon une approche DpR l'opération « **fonction** *Appartient*(i, s, v) **résultat** $\mathbb{B}$ », qui décide si la valeur v est présente ou non dans T[i .. s]. Construire cette opération. Pour le cas inductif, décrire avec précision les conditions qui orientent la recherche dans l'un ou l'autre demi-tableau. On vise un ordre de grandeur de complexité en $\Theta(\log_2(n))$. Cet objectif est-il atteint ? 90 - Q 3

Exercice 91. Minimum local dans un arbre binaire ○ •

Il s'agit de l'un des seuls exercices de l'ouvrage qui traite d'une structure de données inductive (les arbres binaires). Comme souvent dans cette situation, la forme des raisonnements qui y sont développés s'inspire de celle de la structure de données. C'est le principal enseignement à retenir ici.

On dispose d'un arbre binaire *plein* (voir chapitre 1) de poids n ($n > 0$) : tous les nœuds ont zéro ou deux fils, jamais un seul fils, et toutes les feuilles sont à la même profondeur. n est de la forme $2^p - 1$ et $p - 1$ est la hauteur de l'arbre. À chaque nœud est affectée une valeur entière différente. Un nœud est un *minimum local* s'il est (du point de vue de sa valeur) plus petit que son père et que ses deux fils (s'il en a). Dans les exemples ci-dessous, les minima locaux sont encerclés.



Question 1. Montrer qu'il y a toujours (au moins) un minimum local dans un tel arbre. 91 - Q 1

Question 2. Construire l'opération « **fonction** MinLoc(a) **résultat** $\mathbb{Z}$ » qui délivre l'un des minima locaux (on vise une complexité en $\mathcal{O}(\log_2(n))$ conditions évaluées). Donner le modèle de division. Fournir le code l'opération ainsi qu'une séquence d'appel. L'opération a-t-elle bien la complexité souhaitée ? 91 - Q 2

Question 3. La méthode proposée est-elle applicable à un arbre binaire quelconque ? 91 - Q 3

Exercice 92. Diamètre d'un arbre binaire ○ ⁝

Trois enseignements peuvent être tirés de cet exercice qui traite d'arbres binaires. Le premier concerne l'amélioration de la complexité qui résulte d'un renforcement d'une hypothèse d'induction. On y trouve ici un nouvel exemple. Le second porte sur la forme des raisonnements qui y sont développés. Elle peut avec profit s'inspirer de celle de la structure de données. Le troisième enseignement se rapporte à l'usage qui est fait de la structure d'arbre pour évaluer la complexité. Lorsque le traitement à réaliser sur chaque nœud est en $\Theta(1)$ (c'est le cas ici), il suffit – quand cela est possible – de dénombrer les nœuds rencontrés lors du calcul.

Les principales définitions ainsi que les éléments de vocabulaire les plus fréquents portant sur les arbres binaires sont regroupés à la section 1.6 page 30. Dans la suite, on considère que le type ab pour les arbres binaires finis non étiquetés se définit par :

$$ab = \{/\} \cup \{(l, r) \mid l \in ab \text{ et } r \in ab\}$$

Un élément de ab est donc soit $/$ (qui *représente* l'arbre vide) soit un couple (l, r) dont chaque constituant est un arbre binaire (ab).

92 - Q 1 **Question 1.** Montrer par un exemple que, dans un arbre non vide, un chemin dont la longueur est le diamètre de l'arbre ne passe pas nécessairement par la racine de l'arbre.

92 - Q 2 **Question 2.** La hauteur h d'un arbre binaire n'est pas définie pour un arbre vide. Cependant, dans un souci de concision, on accepte la définition inductive suivante :

$$\begin{cases} h(/) & = & -1 \\ h((l, r)) & = & \max(\{h(l), h(r)\}) + 1 \end{cases}$$

En prenant comme base le nombre de nœuds rencontrés lors du calcul, quelle est la complexité C_h de la fonction associée pour un arbre de poids n (c'est-à-dire de n nœuds) ? Définir de façon analogue le diamètre d d'un arbre binaire. Fournir le code de l'opération « **fonction** $d(a)$ **résultat** $\mathbb{N} \cup \{-1\}$ » qui calcule le diamètre de l'arbre binaire a. Pour ce qui concerne la complexité, les arbres filiformes constituent les cas les plus défavorables. En effet, chaque arc de l'arbre fait alors partie du diamètre. Montrer que dans le cas d'un arbre filiforme de poids n la complexité C_d, évaluée en nombre de nœuds rencontrés, est en $\Theta(n^2)$.

92 - Q 3 **Question 3.** La version précédente du calcul du diamètre peut être améliorée sur le plan de la complexité en utilisant une heuristique classique. Celle-ci consiste, plutôt que de *calculer* une valeur nécessaire à un instant donné, à *supposer cette valeur disponible*. On renforce ainsi l'hypothèse d'induction à la base de la construction. Appliquer cette technique pour la construction d'une nouvelle version du calcul du diamètre et montrer (en supposant la hauteur disponible) qu'en dénombrant les nœuds rencontrés on obtient une solution en $\Theta(n)$.

Exercice 93. Le problème de la sélection et de la recherche de l'élément médian

L'originalité de cet exercice réside dans l'utilisation d'un raisonnement du type « induction de partition » (voir section 1.1.4, page 7) non standard. En outre, cet exercice montre (une nouvelle fois) que pour résoudre un certain problème (celui de l'élément médian), il est souvent intéressant de résoudre un problème plus général (celui de la sélection) et donc a priori *plus difficile. Cet exercice est posé (et résolu) dans un formalisme ensembliste. On diffère ainsi les aspects techniques liés à l'implantation.*

On se donne un sous-ensemble fini non vide E de $\mathbb{N}$, de cardinal n. On cherche à résoudre le problème de la *sélection du* k^e *plus petit élément* $(1 \leqslant k \leqslant n)$: v est le k^e plus petit élément de E s'il existe $(k-1)$ éléments inférieurs à v dans E. Quand k vaut 1 ou n, ce problème se résout facilement en $\Theta(n)$, avec la comparaison pour opération élémentaire. Dans quelle mesure peut-on atteindre la même performance pour k quelconque ?

Question 1. Cette première question concerne un algorithme auxiliaire itératif utilisé pour résoudre le problème de la sélection du k^e plus petit élément par une approche DpR. La procédure correspondante est nommée *Éclater*(E, E^-, E^+, a). On choisit aléatoirement un élément de l'ensemble E (tous les éléments sont supposés équiprobables). Cet élément, noté a, est appelé le *pivot*. *Éclater*(E, E^-, E^+, a) sépare les éléments de l'ensemble $E-\{a\}$ en deux autres ensembles E^- et E^+, le premier contenant tous les éléments de E strictement inférieurs au pivot, le second contenant tous les éléments de E strictement supérieurs au pivot. 93 - Q 1

a) Fournir les constituants d'une construction itérative de la procédure *Éclater*.

b) Dans l'optique d'un raffinement de cette procédure, montrer comment les ensembles peuvent être implantés sous forme de tableaux. En déduire que la complexité d'*Éclater* est en $\Theta(n)$.

Question 2. Soit « **fonction** *Sélection*(E, k) **résultat** $\mathbb{N}$ » l'opération qui délivre le k^e plus petit élément de l'ensemble E. Cette opération est préconditionnée par $k \in 1..\,\mathrm{card}(E)$. 93 - Q 2

a) Soit a un élément quelconque de E. Quelle relation liant E, k et a permet-elle d'affirmer que a est l'élément recherché ?

b) En supposant disponible la procédure *Éclater*, construire une solution DpR au problème de la sélection. Quel modèle de division s'applique-t-il ?

c) En déduire le code de la fonction *Sélection*.

d) Quelles sont les spécificités de cette solution par rapport au modèle standard DpR ?

e) Fournir les éléments d'un raffinement algorithmique.

f) Quelle est la complexité au mieux de la fonction *Sélection* ? Quel ordre de grandeur de complexité obtient-on si la procédure *Éclater* coupe à chaque étape l'ensemble E en deux sous-ensembles E^- et E^+ de tailles égales ou différant de 1 ? Quelle est la complexité au pire de la fonction *Sélection* ? Quelle en est à votre avis la complexité moyenne ?

93 - Q 3 **Question** 3. Comment peut-on adapter la solution au problème de la sélection pour obtenir une solution au problème de l'élément médian, c'est-à-dire celui de la recherche du $\lfloor \text{card}(E)/2 \rfloor^e$ élément de E ?

Exercice 94. Écrous et boulons ∘ ⁝

Cet exercice s'apparente à un tri dans la mesure où l'on recherche une bijection dotée de certaines propriétés (un tri est une permutation, donc une bijection). Il n'est donc pas surprenant que l'on retrouve ici les procédés algorithmiques et les techniques de calcul de complexité proches de ceux utilisés dans les problèmes de tri.

Dans une boîte à outils, il y a en vrac un ensemble E de n ($n > 0$) écrous de diamètres tous différents et l'ensemble B des n boulons correspondants. La différence de diamètre est si faible qu'il est impossible de comparer visuellement la taille de deux écrous ou de deux boulons entre eux. Pour savoir si un boulon correspond à un écrou, la seule façon est d'essayer de les assembler. La tentative d'appariement boulon-écrou est l'opération élémentaire choisie pour l'évaluation de la complexité de ce problème. Elle fournit l'une des trois réponses suivantes : soit l'écrou est plus large que le boulon, soit il est moins large, soit ils ont exactement le même diamètre. Le problème consiste à établir la bijection qui, à chaque écrou, associe son boulon.

94 - Q 1 **Question** 1. Fournir le principe d'un algorithme naïf. Quelle est sa complexité la meilleure ? La pire ?

94 - Q 2 **Question** 2. Construire un algorithme DpR qui assemble chaque boulon avec son écrou. Quel est le modèle de division qui s'applique ?

94 - Q 3 **Question** 3. Fournir une version ensembliste de l'opération « **procédure** *Apparier*(E, B) » qui assemble les n écrous de l'ensemble E aux n boulons de l'ensemble B. Montrer que la complexité au pire est en $\Theta(n^2)$.

94 - Q 4 **Question** 4. On cherche à présent à déterminer, en utilisant la méthode des arbres de décision (voir chapitre 1), une borne inférieure à la complexité du problème des écrous et des boulons dans le cas le plus défavorable. Un arbre de décision pour un algorithme résolvant le problème considéré est un arbre ternaire complet qui représente les comparaisons effectuées entre écrous et boulons (voir exercice 87 page 249, pour un autre exemple utilisant les arbres de décision). Dans un tel arbre, un sous-arbre gauche (resp. central, droit) prend en compte la réponse $>$ (resp. $=$, $<$) à la comparaison. À chaque feuille de l'arbre est associée l'une des bijections possibles entre E et B. Fournir l'arbre de décision pour la méthode naïve dans le cas où $E = \{1, 2, 3\}$ et $B = \{a, b, c\}$. Sachant que $\log_3(n!) \in \Omega(n \cdot \log_3(n))$ (d'après la formule de Stirling), montrer que tout algorithme qui résout le problème des écrous et des boulons a une complexité au pire qui est en $\Omega(n \cdot \log_3(n))$.

Exercice 95. La fausse pièce – division en trois et quatre tas

Nous abordons ici un problème classique de pesée, pour lequel de nombreuses variantes existent. Strictement parlant, il ne s'agit pas d'un problème informatique (d'ailleurs, aucun algorithme n'est demandé). En dépit de cela, il s'agit bien d'un problème DpR. L'une de ses caractéristiques est le contraste qui se manifeste entre la simplicité de l'énoncé et la difficulté d'une solution rigoureuse et exhaustive.

Considérons un ensemble de $n \geqslant 1$ pièces de même apparence, dont $(n-1)$ sont en or et une en métal léger plaqué or. Nous disposons d'une balance à deux plateaux qui indique, à chaque pesée, si le poids placé sur le plateau de gauche est inférieur, supérieur ou égal au poids mis sur le plateau de droite. Notons qu'il n'est possible d'exploiter le résultat d'une pesée que si le nombre de pièces est identique sur chaque plateau.

Le but de l'exercice est de trouver la pièce fausse avec le moins de pesées possible dans le cas le plus défavorable (au pire). Deux stratégies sont étudiées.

Stratégie de la division en trois tas

Considérons la stratégie suivante (dite stratégie à trois tas), pour laquelle on sépare les n pièces en deux tas de même cardinal k et un troisième (éventuellement vide) qui contient le reliquat de pièces (donc k peut varier de 0 à $\lfloor n/2 \rfloor$). La fonction $C_3(n)$ fournit le nombre de pesées nécessaires (dans le cas le plus défavorable).

Base Si $n = 1$ (k vaut alors 0), nous sommes face à une seule pièce, c'est la fausse pièce. Aucune pesée n'est nécessaire : $C_3(1) = 0$.

Hypothèse d'induction Pour tout m tel que $1 \leqslant m < n$, on sait déterminer $C_3(m)$.

Induction Soit $k \in 1\,..\,\lfloor n/2 \rfloor$. Le principe de cette stratégie consiste à séparer les n pièces en trois tas, deux tas de k pièces et un tas de $n-2k$ pièces. Une pesée est réalisée en plaçant un tas de k pièces dans chaque plateau de la balance. Deux cas peuvent alors se présenter.

Premier cas La balance est déséquilibrée. La fausse pièce se trouve dans le tas le plus léger. Le nombre de pesées restant à réaliser est donc $C_3(k)$.

Second cas La balance est équilibrée. La fausse pièce se trouve donc dans le troisième tas. Le nombre de pesées restant à réaliser est donc $C_3(n-2k)$ [4].

Dans les deux cas, par l'hypothèse d'induction, on sait trouver la valeur cherchée.

Terminaison Le nombre de pièces pesées à chaque étape diminue. Ceci assure la terminaison de ce procédé.

Question 1. Fournir l'équation récurrente de $C_3(n)$, pour $n > 0$. 95 - Q 1

Question 2. Nous allons à présent tenter de simplifier cette équation. Montrer tout d'abord (par induction sur n) que $C_3(n)$ est croissante. En déduire que l'équation obtenue dans la première question est équivalente à l'équation récurrente suivante : 95 - Q 2

4. Notons que si n est pair et $k = \lfloor n/2 \rfloor$, $C_3(n-2k) = C_3(0)$, mais la pièce est alors dans l'un des deux tas de k pièces. Nous retombons alors dans le premier cas. Du point de vue des calculs, pour contourner cet écueil, nous admettons dans la suite que $C_3(0) = 0$.

$$\begin{cases} C_3(1) = 0 \\ C_3(n) = 1 + C_3\left(\left\lceil \frac{n}{3} \right\rceil\right) & n > 1. \end{cases}$$

95 - Q 3 **Question** 3. Résoudre cette équation. En déduire que la détection de la fausse pièce se fait au pire en $\lceil \log_3(n) \rceil$ pesées.

Stratégie de la division en quatre tas

La stratégie de la division en trois tas peut s'étendre de différentes manières au cas des quatre tas. Nous nous intéressons à l'une d'entre elles pour laquelle, si $n \geqslant 3$, le tas initial est séparé en trois tas *de même cardinal* k, plus un tas de $(n - 3k)$ pièces.

Question 4. Refaire le développement précédent pour cette stratégie.

Exercice 96. La valeur manquante ⚇ •

Cet exercice porte certes sur l'application du principe DpR, mais le lecteur est invité à traiter préalablement le problème selon des méthodes itératives. C'est l'occasion de rappeler que les meilleurs algorithmes de tri ne sont pas toujours au pire en $n \cdot \log_2(n)$: ils peuvent par exemple être linéaires ! Cela dépend de la précondition.

On dispose d'un tableau constant $T[1\,..\,n]$ ($n \in \mathbb{N}_1$) contenant tous les entiers de l'intervalle $1\,..\,n+1$, sauf un. On veut déterminer quel est l'entier absent de T. Les calculs de complexité se feront sur la base de l'évaluation de conditions.

96 - Q 1 **Question** 1. Construire un algorithme qui résout le problème en temps linéaire, sans utiliser de tableau auxiliaire.

96 - Q 2 **Question** 2. Le tableau est maintenant supposé variable. On cherche *simultanément* à déterminer la valeur manquante et à trier le tableau. On peut supposer qu'une cellule supplémentaire de T est disponible à la position $n+1$. Construire une solution itérative qui résout le problème en temps linéaire.

96 - Q 3 **Question** 3. Le tableau $T[1\,..\,n]$ est à présent supposé trié. Construire une solution DpR basée sur la recherche dichotomique. Fournir le modèle de division ainsi que l'algorithme. Quelle est sa complexité au pire ?

96 - Q 4 **Question** 4. Reprendre les trois questions précédentes quand le tableau contient tous les entiers de l'intervalle $1\,..\,n+2$, sauf deux.

Exercice 97. Le meilleur intervalle ○ ⁝

Cet énoncé propose d'étudier une version DpR d'un problème également traité par programmation dynamique à l'exercice 116, page 331. Cette version constitue un exemple simple d'amélioration de l'efficacité des solutions trouvées à un problème donné, améliorations obtenues tout d'abord par une application du principe DpR puis par un renforcement approprié de la postcondition. Sur le plan de la complexité temporelle, le résultat est comparable à la programmation dynamique.

On dispose d'un tableau constant $T[1\,..\,n]$ ($n \geqslant 1$) de valeurs réelles non négatives ($T \in 1\,..\,n \to \mathbb{R}_+$). Il existe au moins deux indices, i et j, définissant l'intervalle $i\,..\,j$, avec $1 \leqslant i \leqslant j \leqslant n$, tels que la valeur de l'expression $T[j] - T[i]$ soit maximale. On cherche cette valeur maximale (la valeur du *meilleur* intervalle). Le cas particulier où $i = j$ caractérise un tableau monotone strictement décroissant : la valeur cherchée est alors nulle.

Un exemple d'application de cet algorithme : le tableau T comporte les valeurs quotidiennes de l'action de la société Machin le mois dernier. On se demande aujourd'hui quel aurait été le gain optimal en achetant et revendant une action le mois dernier.

Question 1. Soit l'opération « **procédure** *MeilleurIntervalle1*(deb, fin; mi : **modif**) » qui délivre la valeur du meilleur intervalle mi pour le tableau T[deb .. fin]. 97 - Q 1

a) En appliquant le principe de l'induction de partition (voir section 1.1.4, page 7) construire la procédure *MeilleurIntervalle1*.

b) Sur quel modèle de division cette solution s'appuie-t-elle ? En déduire sa complexité en nombre de conditions évaluées.

Question 2. La version DpR ci-dessus conduit à des calculs superflus. 97 - Q 2

a) Lesquels ? La mise en œuvre d'une amélioration passe par un renforcement de la postcondition, qui se traduit par l'ajout de paramètres dans la procédure. Spécifier informellement cette nouvelle procédure *MeilleurIntervalle2*.

b) Construire cette procédure. Fournir le modèle de division correspondant, ainsi que le code de la procédure. En déduire la complexité de cette solution. Que peut-on en dire par rapport à celle de *MeilleurIntervalle1 ?*

Exercice 98. Le sous-tableau de somme maximale ○○ ⁝

L'intérêt principal de cet exercice réside dans la succession de renforcements de l'hypothèse d'induction exigée par la recherche d'une solution efficace et dans leur caractère constructif (montrant que le développement peut le plus souvent se faire de manière rationnelle et que le travail déjà réalisé guide[a] celui à venir).

a. Il existe des exceptions à cette règle, comme celle de l'exercice 104 page 271.

Soit un tableau T[1 .. n] ($n \geqslant 1$) de nombres réels. On appelle *somme* d'un sous-tableau de T la somme des éléments de ce sous-tableau. On cherche la valeur maximale prise par la somme lorsque l'on considère tous les sous-tableaux de T (y compris les sous-tableaux vides et le tableau T au complet). Par exemple, dans le tableau suivant :

i	1	2	3	4	5	6	7	8
T[i]	3.	1.	−4.	3.	−1.	3.	−0.5	−1.

la valeur maximale prise par la somme est atteinte pour le sous-tableau T[4 .. 6] et vaut 5. En revanche, dans le tableau suivant :

i	1	2	3	4	5	6	7	8
T[i]	−5.	−4.	−4.	−3.	−1.	−8.	−0.5	−15.

n'importe quel sous-tableau de longueur nulle (comme par exemple T[4 .. 3]) fournit la solution. La somme maximale vaut alors 0.0.

98 - Q 1 **Question 1.** Une solution naïve, dans laquelle on considère explicitement *tous* les sous-tableaux, est possible. Elle est cependant coûteuse en termes de complexité temporelle (avec l'addition comme opération élémentaire). On recherche une solution de type DpR qui se présente sous la forme *SousTabMax1*(deb, fin; sm : **modif**) (où deb et fin – paramètres d'entrée – sont les bornes du tableau considéré et sm – paramètre de sortie – la somme maximale atteinte si l'on considère tous les sous-tableaux de T[deb .. fin]).

a) Caractériser le cas élémentaire.

b) Formuler l'hypothèse d'induction à la base du cas général. Décrire le traitement à réaliser pour rassembler les deux solutions partielles obtenues. En déduire le modèle de division puis le code de la procédure *SousTabMax1*.

c) Quelle est la complexité de cette solution ?

98 - Q 2 **Question 2.** Dans la solution précédente, l'existence de calculs répétitifs présents dans le rassemblement laisse présager une solution de type DpR plus efficace que celle construite.

a) Quel renforcement de l'hypothèse d'induction est-il raisonnable de formuler ? Spécifier l'en-tête de la nouvelle version *SousTabMax2* de cette procédure.

b) Décrire le traitement à réaliser pour le rassemblement des résultats obtenus dans le cas inductif. Quel est le modèle de division associé ?

c) En déduire l'ordre de grandeur de complexité qui en résulte. Conclusion ?

98 - Q 3 **Question 3.** Les réponses à la question 2 fournissent une suggestion quant à un nouveau renforcement qu'il est possible de réaliser.

a) Quel renforcement de l'hypothèse d'induction est-il raisonnable de formuler ? Spécifier l'en-tête de la nouvelle version *SousTabMax3* de cette procédure.

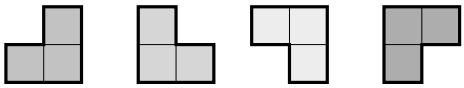
b) Comment le rassemblement décrit dans la question 2.b) peut-il être aménagé pour prendre en compte la nouvelle hypothèse d'induction ? Quel est le modèle de division associé ? En déduire le code la procédure *SousTabMax3*.

c) Quelle est l'ordre de grandeur de complexité qui en résulte ?

Exercice 99. Pavage d'un échiquier par des triminos ○ •

Il s'agit de l'un des seuls exercices pour lesquels le problème initial s'éclate en quatre sous-problèmes. De plus, l'étape de base est vide, ainsi que l'étape de rassemblement. L'équation fournissant la complexité fait partie du répertoire qui accompagne le théorème maître (voir page 246). Cependant, l'énoncé exige ici de résoudre exactement *cette équation.*

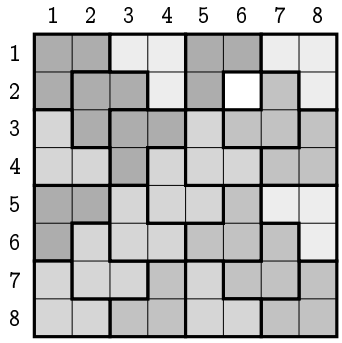
On considère un échiquier $n \times n$ tel que $n = 2^m$ et $m \geqslant 0$, ainsi que les quatre types de triminos ci-après composés chacun d'un carré 2×2 auquel une cellule a été enlevée :



(a) (b) (c) (d)

On se pose le problème de recouvrir intégralement l'échiquier, à l'exception d'une cellule particulière donnée (le « trou ») de coordonnées (lt, ct) (*l*igne du *t*rou et *c*olonne du *t*rou), à l'aide des motifs ci-dessus, de sorte que chaque cellule soit recouverte par un seul motif.

Exemple On dispose d'un échiquier de huit lignes et de huit colonnes, le trou est en $(2, 6)$: deuxième ligne, sixième colonne.



Le but de l'exercice est de concevoir une solution de type DpR au problème posé.

Question 1. Démontrer par récurrence que pour m entier (avec $m \geqslant 0$), l'expression $(2^{2m} - 1)$ est divisible par 3. Conclusion ? 99 - Q 1

Question 2. Construire par induction l'opération « **procédure** $Pavage(l, c, n, lt, ct)$ » qui pave un échiquier de $n \times n$ cellules (n est une puissance de 2), dont le coin nord-ouest est situé à la ligne l et à la colonne c et pour lequel le trou est situé à la ligne lt et à la colonne ct. 99 - Q 2

Question 3. Fournir le modèle de division qui s'applique. 99 - Q 3

Question 4. Donner le code de la procédure *Pavage*. L'opération « **procédure** $Poser(l, c, td)$ » (qui place un trimino de type td ($td \in (a) .. (d)$) de sorte que sa cellule manquante soit située en (l, c)) est supposée disponible. Établir que la complexité exacte 99 - Q 4

de cette opération est égale à $((n^2-1)/3)$ si on prend comme opération élémentaire la pose d'un trimino.

Exercice 100. La bâtière

Outre son intérêt intrinsèque lié à l'exploitation de l'ordre sur les lignes et les colonnes d'une matrice, cet exercice met en évidence l'incidence du caractère strict ou non de cet ordre sur le problème à résoudre.

On considère un tableau à deux dimensions de valeurs entières positives telles que les valeurs d'une même ligne et celles d'une même colonne sont ordonnées, *non forcément strictement*. Un tel tableau est appelé *bâtière*.

Exemple Le tableau ci-dessous est une bâtière à quatre lignes et cinq colonnes.

2	14	25	30	69
3	15	28	30	81
7	15	32	43	100
20	28	36	58	101

Il est à noter que tout sous-tableau d'une bâtière est lui-même une bâtière, et cette propriété est utilisée implicitement dans la suite.

On étudie la recherche d'une valeur v fixée dans une bâtière B. Plus précisément, si v est présent, on souhaite connaître les coordonnées d'une de ses occurrences, sinon on délivre $(0,0)$.

Diviser pour Régner $(1, n-1)$ = Diminuer pour résoudre

Une première solution consiste en un balayage séquentiel de B par ligne (ou par colonne) jusqu'à trouver v.

100 - Q 1 **Question** 1. Décrire le principe de cette première stratégie en termes de méthode DpR.

100 - Q 2 **Question** 2. Quelle est sa classe de complexité au pire (en termes de nombre de comparaisons), si m est le nombre de lignes et n le nombre de colonnes de B ? Peut-on l'améliorer en utilisant le fait que les lignes et les colonnes sont triées ?

Diviser pour Régner $(n/2, n/2)$

Dans l'approche précédente, on n'a pas exploité (double recherche séquentielle), ou alors pas totalement (recherche séquentielle sur les lignes et dichotomique dans une ligne) le fait que B est une bâtière. Pour tirer parti de cette propriété, on envisage maintenant une solution dans le cas particulier où la bâtière est un carré de côté $n = 2^k$ $(k \geqslant 1)$, et on distingue les deux valeurs : $x = B[n/2, n/2]$, $y = B[n/2+1, n/2+1]$.

Question 3. Montrer que si la valeur recherchée v est telle que $v > x$, on peut éliminer une partie (à préciser) de la bâtière pour poursuivre la recherche. Préciser ce qu'il est possible de faire quand $v < y$. 100 - Q 3

Question 4. En déduire un modèle de résolution de type DpR en réduction logarithmique et donner la procédure associée. 100 - Q 4

Question 5. Établir l'ordre de complexité au pire de cette méthode (en termes de nombre de comparaisons). 100 - Q 5

Question 6. Comment adapter cette stratégie au cas d'une bâtière quelconque ? 100 - Q 6

De plus en plus fort

On considère maintenant la recherche d'une valeur v dans une bâtière B de dimension quelconque (m lignes, n colonnes) en distinguant la valeur $z = B[1, n]$.

Question 7. Que convient-il de faire selon que z est égal, supérieur, ou inférieur à v ? 100 - Q 7

Question 8. En déduire un modèle de résolution de type DpR de la forme : 100 - Q 8

$Pb(m, n) \rightarrow$ test relatif à la valeur z + $Pb(m', n')$

en précisant les valeurs de m' et n'.

Question 9. Écrire la procédure récursive correspondante, qui a pour en-tête **procédure** *Bâtière3*$(lDeb, cFin; lig, col$: **modif**$)$ et est appelée du programme principal par *Bâtière3*$(1, n, l, c)$, où les paramètres de sortie l (resp. c) reçoivent l'indice de ligne (resp. de colonne) d'une case de la bâtière $B[1 .. m, 1 .. n]$ contenant la valeur v recherchée, ou $(0, 0)$ si celle-ci n'apparaît pas dans la bâtière. 100 - Q 9

Question 10. Établir la classe de complexité au pire de cette dernière méthode (en termes de nombre de comparaisons) et conclure sur la démarche à adopter pour résoudre le problème de la recherche d'une valeur donnée dans une bâtière. 100 - Q 10

Un problème voisin : le dénombrement des zéros

Question 11. On suppose maintenant que la bâtière $B[1 .. m, 1 .. n]$ contient des entiers relatifs, et l'on veut compter le nombre de zéros. L'approche développée précédemment pour la recherche de la présence d'une valeur fondée sur l'élimination de lignes ou colonnes peut-elle servir de base à la résolution de ce problème ? Qu'en serait-il si l'ordre était strict dans les lignes et colonnes ? 100 - Q 11

Exercice 101. Nombre d'inversions dans une liste de nombres ◦ ⁝

Cet exercice est un nouvel exemple de problème qui montre que l'application du principe DpR n'est pas systématiquement un gage d'amélioration de la complexité d'un algorithme. Il confirme également que renforcer la postcondition de la spécification (c'est-à-dire chercher à en faire « plus » que demandé initialement) peut parfois être la clé de l'amélioration recherchée.

Le but de cet exercice est de construire un algorithme rapide pour compter le nombre d'*inversions* présentes dans une liste sans doublon. Pour fixer les idées, on travaille sur des *permutations* des n ($n \geqslant 1$) premiers entiers positifs, donc sur l'intervalle $1\,..\,n$. Une liste de valeurs sans doublons est rangée dans le tableau $T[1\,..\,n]$, et l'on dit que les nombres i et j, tels que $1 \leqslant i < j \leqslant n$, forment une inversion si $T[i] > T[j]$.

Par exemple, pour $n = 8$, le nombre d'inversions de la liste suivante vaut 13 :

i	1	2	3	4	5	6	7	8
T[i]	3	5	2	8	6	4	1	7

Écrites en tant que liste des couples d'indices (i, j) tels que $i < j$ et $T[i] > T[j]$, les inversions sont les suivantes :

$$\langle (1,3), (1,7), (2,3), (2,6), (2,7), (3,7), (4,5), (4,6), (4,7), (4,8), (5,6), (5,7), (6,7) \rangle$$

101 - Q 1 **Question** 1. Construire un algorithme itératif naïf qui calcule le nombre d'inversions dans une liste sans doublons. Donner l'ordre de grandeur de complexité de cet algorithme en prenant l'évaluation de conditions comme opération élémentaire.

On va supposer maintenant que $n = 2^k$, pour k entier supérieur ou égal à 1. Pour $k > 1$, on peut partitionner les inversions en trois catégories :

— celles dont les deux termes sont dans la première moitié de T, par exemple : $(2,3)$ dans l'exemple ci-dessus,

— celles dont les deux termes sont dans la seconde moitié de T, par exemple : $(6,7)$,

— celles qui ont le premier terme dans la première moitié de T et le second dans la seconde moitié, par exemple : $(2,6)$.

101 - Q 2 **Question** 2. Construire l'opération « **fonction** *NbInv1*(i, s) **résultat** $0\,..\,(n \cdot (n+1))/2$ », qui est telle que l'appel *NbInv1*$(1, n)$ calcule, selon une approche DpR, le nombre d'inversions présentes dans le tableau $T[1\,..\,n]$ (T est une variable globale). Quel est le modèle de division qui s'applique ? Fournir le code de cette fonction. Donner l'ordre de grandeur de complexité de cette solution. Que peut-on en conclure par rapport à la première question ?

101 - Q 3 **Question** 3. Pour améliorer l'efficacité de la solution, il faut trouver un moyen de ne pas comparer tous les couples possibles. On propose de renforcer la postcondition, non seulement en recherchant le nombre d'inversions, mais également en triant le tableau T. Étudions tout d'abord le cœur de cette solution en considérant un tableau dont chaque moitié est triée comme dans l'exemple suivant :

i	1	2	3	4	5	6	7	8
T[i]	2	3	5	8	1	4	6	7

a) Montrer que dans une configuration telle que :

où $T[1 \,..\, mil]$ et $T[mil + 1 \,..\, n]$ sont triés par ordre croissant, si $T[j] > T[k]$, alors, pour tout $l \in j \,..\, mil$, $T[l] > T[k]$. Que peut-on en conclure ?

b) En déduire un algorithme itératif qui, dans ce cas, compte les inversions en un temps linéaire.

Question 4. Construire (en s'inspirant de la question précédente ainsi que de l'exercice 86, page 248, qui porte sur le tri-fusion) l'opération de type DpR « **procédure** *NbInv2*$(i, s; nb$: **modif**$)$ » qui est telle que l'appel *NbInv2*$(1, n, nbi)$ à la fois trie le tableau $T[1 \,..\, n]$ (T est une variable globale) et comptabilise, dans nbi, les inversions présentes dans la configuration initiale de $T[1 \,..\, n]$. On cherche à améliorer l'ordre de grandeur de complexité par rapport à la solution de la question 2. Fournir le modèle de division ainsi que le code. 101 - Q 4

Question 5. Peut-on adapter les algorithmes de la question précédente dans le cas où n n'est pas une puissance de 2 ? 101 - Q 5

Exercice 102. Le dessin du skyline

L'une des stratégies classiques pour améliorer l'efficacité des algorithmes consiste à changer la structure de données sous-jacente. Dans cet exercice, on débute le développement sur la base d'une structure de données « naïve » avant d'optimiser celle-ci en éliminant une forme de redondance. Cet exercice montre que l'amélioration attendue par le recours à une démarche DpR n'est pas toujours au rendez-vous.

On s'intéresse au dessin du *skyline* d'un centre-ville construit avec des immeubles rectangulaires, au bord de la mer. Le *skyline* (en pointillés dans les schémas ci-dessous) est la ligne qui sépare le ciel des immeubles, quand on se place assez loin en mer et que l'on regarde en direction de la ville. La partie (a) de la figure 8.2 montre la projection à deux dimensions d'un centre-ville composé de trois immeubles, et la partie (b) donne le *skyline* correspondant.

On suppose que toutes les dimensions sont entières et que les immeubles sont construits entre les abscisses 0 et n ($n \geqslant 1$). L'ajout d'un quatrième immeuble se concrétise comme le montrent les parties (a) et (b) de la figure 8.3, page 266.

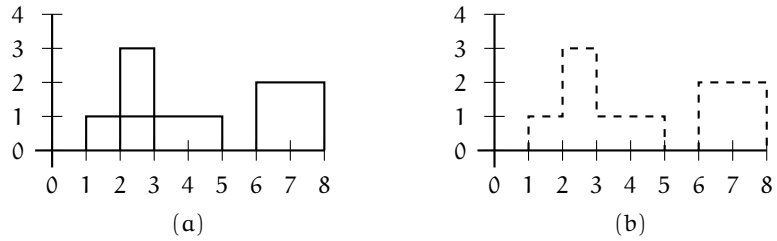


Fig. 8.2 – *Trois immeubles :* (a) *par projection* – (b) *le* skyline *correspondant*

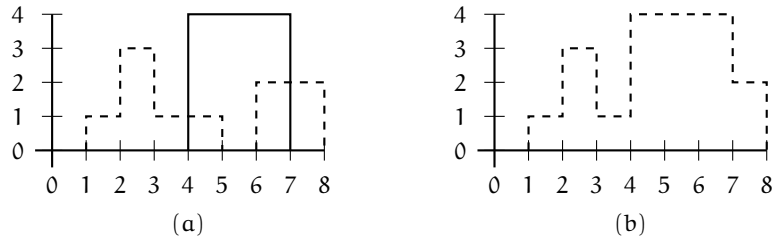


Fig. 8.3 – *Ajout d'un immeuble :* (a) *le nouvel immeuble* – (b) *le nouveau* skyline

Une première approche

Un *skyline* est représenté dans cette première partie par un tableau $S[1\,..\,n]$, dont la composante i indique la hauteur du *skyline* entre les abscisses $i-1$ et i. Pour l'exemple de la partie (a) de la figure 8.2, page 266, le *skyline* se représente donc par :

i	1	2	3	4	5	6	7	8
S[i]	0	1	3	1	1	0	2	2

On choisit de représenter un immeuble par le *skyline* qu'il aurait s'il était tout seul. La représentation du troisième immeuble de la figure 8.2, page 266, (l'immeuble le plus à droite) est donc :

i	1	2	3	4	5	6	7	8
$I_3[i]$	0	0	0	0	0	0	2	2

Un algorithme itératif de construction On cherche à obtenir le *skyline* d'un ensemble[5] I de m ($m \geqslant 0$) immeubles ($I = \{I_1, \ldots, I_m\}$).

102 - Q 1 **Question 1.** Construire l'algorithme en supposant que l'ensemble des immeubles I est défini par $I \in 1\,..\,m \to (1\,..\,n \to \mathbb{N})$ (I est un tableau de m immeubles ; chaque immeuble est un tableau de n hauteurs).

102 - Q 2 **Question 2.** Quelle est la complexité exacte en nombre de conditions du calcul du *skyline* de I, en fonction de n et m ?

5. Ou d'un multiensemble, s'il existe des immeubles identiques.

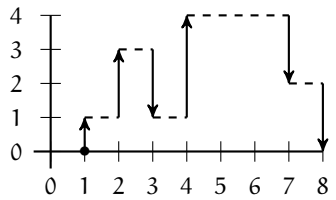
Un algorithme DpR Dans la perspective d'une amélioration de l'efficacité de l'algorithme, on souhaite appliquer une démarche DpR pour calculer le *skyline* de l'ensemble d'immeubles $I[1\,..\,m]$. Le profil de la procédure correspondante est $SkyLine1(deb, fin; S : \textbf{modif})$, où l'ensemble d'immeubles considéré est celui représenté par $I[deb\,..\,fin]$ et où S est le *skyline* qui lui correspond.

Question 3. Construire la procédure *SkyLine1*. En déduire le modèle de division qui s'applique. Fournir le code de la procédure. 102 - Q 3

Question 4. Poser l'équation récurrente qui caractérise la complexité de cette procédure, sur la base du nombre de conditions évaluées. En déduire la classe de complexité. Conclusion ? 102 - Q 4

Une seconde approche

Une voie alternative pour tenter d'améliorer l'efficacité d'un algorithme consiste à choisir une meilleure représentation pour les entités manipulées (ici les immeubles et les *skylines*). On peut constater que la représentation précédente est redondante, dans la mesure où l'échantillonnage concerne *la totalité* des n points. En se basant sur le *skyline* de la figure 8.3, page 266, redessiné de la manière suivante :



il est possible de redéfinir une représentation dans laquelle seules les coordonnées de l'extrémité des vecteurs verticaux sont conservées. On obtient une liste de couples à laquelle on ajoute le couple $(n+1, 0)$ en guise de sentinelle et complétée si nécessaire par des couples $(0, 0)$ jusqu'à la position $n+1$. Le tout est enregistré dans un tableau défini sur l'intervalle $0\,..\,n+1$. Ainsi, pour l'exemple ci-dessus, la nouvelle représentation est :

$$T[0\,..\,n+1] = [(0,0),(1,1),(2,3),(3,1),(4,4),(7,2),(8,0),(9,0),(0,0),(0,0)]$$

La liste de couples est triée sur la première coordonnée, qui constitue un identifiant. Cette nouvelle représentation préserve toute l'information tout en garantissant l'absence de redondance. Dans la suite, cette représentation est caractérisée par le prédicat EstSkyLine.

Question 5. Fournir la représentation du *skyline* de la figure 8.2, page 266. 102 - Q 5

Un algorithme itératif

Question 6. Une solution itérative prenant en compte cette nouvelle représentation peut s'inspirer de la réponse à la question 1. Les boucles externes sont identiques dans les deux cas. En revanche, le traitement qui réalise la fusion de deux *skylines* est inédit. Deux solutions peuvent être envisagées. La première opère une fusion grossière présentant des redondances qui doivent être éliminées dans une phase ultérieure. Cette solution est assez facile à construire, mais elle est assez coûteuse et peu élégante. La seconde solution consiste à obtenir le résultat en un seul passage sur les deux *skylines*. C'est celle qui fait l'objet 102 - Q 6

de la question et dont on demande la construction sous la forme d'une procédure au profil suivant : « *FusionSkyLines*(S1, S2; F : **modif**) » (S1 et S2 sont les deux *skylines* dont on veut obtenir la fusion, et F le *skyline* résultant).

Suggestion La principale difficulté de l'algorithme réside dans l'identification de la cascade de cas et de sous-cas qui se présente dans la progression de la boucle. On conseille au lecteur de bien séparer deux phases, celle de la mise en évidence des différents cas et celle de l'optimisation (factorisation des cas identiques).

102 - Q 7 **Question** 7. En se fondant sur le nombre de conditions évaluées, quelle est, en fonction de m, la complexité au pire de la procédure *FusionSkyLines* ?

102 - Q 8 **Question** 8. En déduire une solution itérative utilisant la procédure *FusionSkyLines*. Quelle est l'ordre de grandeur de complexité au pire (en nombre de conditions évaluées) de ce calcul itératif en fonction de m ?

Un algorithme DpR Toujours avec la même représentation, on recherche à présent une solution DpR. Le profil de la procédure est *SkyLine2*(deb, fin; S : **modif**).

102 - Q 9 **Question** 9. Construire la procédure *SkyLine2*. En déduire le modèle de division qui s'applique. Fournir le code de la procédure.

102 - Q 10 **Question** 10. Donner l'équation récurrente qui caractérise la complexité de cette procédure, sur la base du nombre de conditions évaluées. Conclusion ?

Exercice 103. La suite de Fibonacci

Cet exercice illustre de façon spectaculaire comment l'application du principe DpR peut permettre de réduire la complexité des solutions à un problème donné. Quatre versions sont étudiées depuis une version naïve, dont la complexité est exponentielle, jusqu'à deux versions DpR, de complexités logarithmiques.

Soit la suite de Fibonacci :

$$\begin{cases} \mathcal{F}_0 = \mathcal{F}_1 = 1 \\ \mathcal{F}_n = \mathcal{F}_{n-1} + \mathcal{F}_{n-2} & n \geqslant 2. \end{cases}$$

Le problème que l'on étudie dans cet exercice est celui du calcul de $\mathcal{F}_n$ pour n quelconque. La solution récursive triviale, celle dont la structure reflète exactement la définition ci-dessus, est très inefficace. Ainsi par exemple, le calcul de $\mathcal{F}_6$ conduit à calculer $\mathcal{F}_5$ et $\mathcal{F}_4$, celui de $\mathcal{F}_5$ exige le calcul de $\mathcal{F}_3$ et à nouveau celui de $\mathcal{F}_4$. Chacun des calculs de $\mathcal{F}_4$ conduit au calcul de $\mathcal{F}_3$, etc. On peut remarquer que, en nombre d'additions, ce calcul dépasse très vite n, n^2, n^3, etc. On montre que la complexité de cet algorithme (en termes d'additions ou de conditions évaluées) est en fait exponentielle. Une solution plus efficace consiste à enregistrer dans un tableau les résultats déjà connus de façon à éviter de les recalculer. Ce principe, appelé « mémoïsation », fait partie des techniques appliquées en programmation dynamique (voir chapitre 9). Si M, initialisé à 0, est le tableau en question, ce principe se décline de la manière suivante pour ce qui concerne le calcul d'un élément de la suite de Fibonacci :

```
1. fonction Fibo1(n) résultat ℕ₁ pré
2.    n ∈ 0 .. Maxi
3. début
4.    si n = 0 ou n = 1 alors
5.       résultat 1
6.    sinon
7.       si M[n] = 0 alors
8.          M[n] ← Fibo1(n − 1) + Fibo1(n − 2)
9.       fin si ;
10.      résultat M[n]
11.   fin si
12. fin
```

Ainsi que le montre le contexte d'appel suivant, le tableau M doit être initialisé à 0 avant le premier appel à *Fibo1* :

```
1. constantes
2.    Maxi ∈ ℕ₁ et Maxi = ... et n ∈ 0 .. Maxi et n = ...
3. variables
4.    M ∈ (2 .. Maxi) → ℕ
5. début
6.    M ← (2 .. Maxi) × {0} ;
7.    écrire(Fibo1(n))
8. fin
```

Cette solution est au pire (resp. au mieux) en $\Theta(n)$ (resp. $\Theta(1)$) additions ou conditions évaluées. En outre, elle s'accompagne d'une précondition restrictive sur la valeur de n. Peut-on améliorer la complexité au pire ? C'est ce que l'on va tenter de faire à travers deux méthodes de type DpR.

Face à une suite récurrente linéaire d'ordre 2 telle que $\mathcal{F}_n$, il est souvent intéressant de se ramener à une suite récurrente linéaire d'ordre 1. Le développement y gagne en simplicité. En revanche, si la suite initiale est une suite scalaire, la transformation conduit à une suite vectorielle. C'est ce principe que l'on va appliquer ci-dessous. On développe tout d'abord un algorithme pour la version vectorielle, avant de l'utiliser pour obtenir la version scalaire de $\mathcal{F}_n$.

Une première solution de type DpR

Question 1. On va montrer que la suite de Fibonacci peut se transformer en une suite vectorielle d'ordre 1. Pour ce faire, on note $\mathcal{V}_n = \begin{bmatrix} \mathcal{F}_{n-1} \\ \mathcal{F}_n \end{bmatrix}$ un vecteur colonne à deux éléments et F une matrice carrée (2×2). On demande de calculer la matrice F telle que : 103 - Q 1

$$\begin{aligned} &\mathcal{V}_1 = \begin{bmatrix} \mathcal{F}_0 \\ \mathcal{F}_1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\ &\mathcal{V}_n = F \times \mathcal{V}_{n-1} \qquad\qquad n > 1. \end{aligned}$$

103 - Q 2 **Question** 2. Montrer que la solution de l'équation de récurrence correspondante peut s'écrire $\mathcal{V}_n = F^{n-1} \times \mathcal{V}_1$.

103 - Q 3 **Question** 3. En supposant disponible la fonction de profil *ProduitMatrice*(A, B) (A et B sont des matrices carrées (2×2)) qui délivre la matrice $A \times B$, construire, selon une démarche DpR, une fonction de profil *PuissanceMatrice*(M, n), qui délivre la matrice M^n pour tout $n \in \mathbb{N}_1$. En déduire la procédure *FiboV*$(n; u, v :$ **modif**$)$ qui, pour $n \in \mathbb{N}_1$ donné, délivre les valeurs u et v telles que $\mathcal{V}_n = \begin{bmatrix} u \\ v \end{bmatrix}$.

103 - Q 4 **Question** 4. Montrer comment la procédure *FiboV* peut être utilisée pour définir la fonction *Fibo2*(n) qui délivre $\mathcal{F}_n$, pour $n \in \mathbb{N}$. En prenant la multiplication de matrices (2×2) comme opération élémentaire, fournir l'équation de récurrence de la complexité de cette fonction lorsque n est de la forme 2^k. Que dire de la complexité en nombre d'additions ? Conclure.

103 - Q 5 **Question** 5. Donner un minorant et un majorant (en explicitant les cas où ils sont atteints) de la complexité exacte de la fonction *PuissanceMatrice* en termes de nombre de multiplications de matrices (2×2). Comment ces valeurs se traduisent-elles en nombre d'additions et de multiplications ?

103 - Q 6 **Question** 6. Montrer à travers un contre-exemple que l'algorithme *PuissanceMatrice* d'élévation à la puissance n n'est pas systématiquement optimal. Pour ce faire, développer le calcul de *Fibo2*(15).

Une seconde solution de type DpR

Cette solution se fonde également sur une transformation de la suite $\mathcal{F}_n$ en une suite vectorielle. Cependant, cette fois, cette suite n'est pas linéaire.

103 - Q 7 **Question** 7. Montrer par récurrence sur p que :

$$\forall p \cdot (p \in \mathbb{N}_1 \Rightarrow \ \forall n \cdot (n \in \mathbb{N}_1 \Rightarrow \mathcal{F}_{n+p} = \mathcal{F}_n \cdot \mathcal{F}_p + \mathcal{F}_{n-1} \cdot \mathcal{F}_{p-1})).$$

103 - Q 8 **Question** 8. Appliquer la formule précédente pour $p = n$, $p = n - 1$ et $p = n + 1$, afin d'en déduire $\mathcal{F}_{2n}$, $\mathcal{F}_{2n-1}$ et $\mathcal{F}_{2n+1}$ en fonction de $\mathcal{F}_n$ et de $\mathcal{F}_{n-1}$.

103 - Q 9 **Question** 9. En notant $\mathcal{W}_n$ le vecteur $\begin{bmatrix} \mathcal{F}_{n-1} \\ \mathcal{F}_n \end{bmatrix}$, en déduire que $\mathcal{W}_{2n}$ et $\mathcal{W}_{2n+1}$ se calculent en fonction de $\mathcal{F}_n$ et de $\mathcal{F}_{n-1}$.

103 - Q 10 **Question** 10. On recherche une procédure *FiboW*$(n; u, v :$ **modif**$)$ de type DpR qui, pour n donné, délivre $\mathcal{W}_n = \begin{bmatrix} u \\ v \end{bmatrix}$. Construire cette procédure par application du principe DpR. En déduire le modèle de division qui s'applique. Fournir le code de la procédure *FiboW*. Montrer comment la procédure *FiboW* peut être utilisée pour définir la fonction *Fibo3*(n) qui délivre $\mathcal{F}_n$, pour $n \in \mathbb{N}$.

103 - Q 11 **Question** 11. Donner et résoudre l'équation de récurrence de la complexité de la procédure *FiboW* sur la base du nombre d'additions effectuées. Comparer les deux solutions DpR.

Exercice 104. Élément majoritaire (le retour)

Déjà abordé à l'exercice 40 page 89, cet exercice passe en revue trois solutions DpR au problème de la recherche d'un élément majoritaire dans un sac. Si la première solution est classique, les deux autres font appel à une technique plus originale. En effet, dans cet ouvrage et dans ce chapitre en particulier, on a fréquemment exploité l'heuristique éprouvée qui consiste à renforcer *la postcondition afin d'obtenir une bonne efficacité temporelle ; ici au contraire, on* affaiblit *la postcondition. D'un côté, cette dernière heuristique permet d'obtenir un algorithme simple (mais ne répondant que partiellement à la spécification), de l'autre elle oblige à adjoindre un algorithme complémentaire dont le but est de s'assurer que le résultat obtenu dans la première étape est (ou non) conforme à la spécification initiale. Cette technique peut avec profit enrichir le bagage de tout développeur. La dernière solution s'apparente à celle développée à l'exercice 40 page 89.*

On considère un sac S de n éléments ($n \geqslant 1$), d'entiers strictement positifs. S est dit *majoritaire* s'il existe un entier x tel que :

$$\text{mult}(x, S) \geqslant \left\lfloor \frac{n}{2} \right\rfloor + 1$$

$\text{mult}(x, S)$ désignant la fonction délivrant le nombre d'occurrences de x dans S ; x est alors appelé *élément majoritaire* de S et est unique. Le problème posé est celui de la recherche d'un élément majoritaire dans S. La détermination d'un candidat ayant obtenu la majorité absolue lors d'un scrutin ou la conception d'algorithmes tolérants aux fautes sont des applications possibles de ce problème.

Par la suite, la complexité sera exprimée en nombre de conditions évaluées.

Dans les trois solutions ci-dessous, nous raffinons le sac S par un tableau d'entiers positifs $T[1 \,..\, n]$ ($n \geqslant 1$). Il est difficile d'imaginer que l'on puisse identifier, s'il existe, l'élément majoritaire, sans connaître sa multiplicité. C'est pourquoi on renforce d'emblée la postcondition, en transformant la recherche de l'élément majoritaire en la recherche du couple (x, nbx) (x étant l'élément majoritaire et nbx sa multiplicité).

Un algorithme DpR en $\Theta(n \cdot \log_2(n))$

Pour $T[1 \,..\, n]$, on va calculer le couple (x, nbx) tel que :

— si $T[1 \,..\, n]$ n'est pas majoritaire, on renvoie le couple $(0, 0)$,

— si $T[1 \,..\, n]$ est majoritaire, on renvoie (x, nbx), où x est l'élément majoritaire et nbx sa multiplicité.

On applique une technique DpR travaillant sur deux « moitiés » de $T[1 \,..\, n]$ de tailles égales ou différant de 1. Le couple (x, nbx) est alors calculé à partir de $(xg, nbxg)$ (issu de la moitié gauche de T) et de $(xd, nbxd)$ (issu de la moitié droite de T), en dénombrant si nécessaire la valeur majoritaire d'une certaine moitié dans la moitié opposée.

Exemple Dans le tableau ci-dessous, le couple $(1, 8)$ calculé pour l'intervalle $1..13$ apparaît en bas de la figure, entre les indices 6 et 7. Il signifie que 1 est élément majoritaire et que sa multiplicité est 8. Ce résultat est obtenu en « rassemblant » (d'une manière qui reste

à préciser) le couple $(0,0)$ obtenu pour l'intervalle $1\,..\,6$ et le couple $(1,5)$ obtenu pour l'intervalle $7\,..\,13$.

1	2	3	4	5	6	7	8	9	10	11	12	13
2	1	1	3	1	2	3	1	1	1	1	2	1
2,1	1,1	1,1	3,1	1,1	2,1	3,1	1,1	1,1	1,1	1,1	2,1	1,1
	(1,2)			(0,0)			(1,2)		(1,2)		(0,0)	
(1,2)			(0,0)			(1,2)				(1,3)		
		(0,0)						(1,5)				
						(1,8)						

Dans le cas général d'un tableau $T[d\,..\,f]$ de longueur t ($t = f - d + 1$), on va considérer deux demi-tableaux contigus, $T[d\,..\,d + \lfloor t/2 \rfloor - 1]$ pour lequel on calcule $(xg, nbxg)$, et $T[d + \lfloor t/2 \rfloor\,..\,d + t - 1]$ pour lequel on calcule $(xd, nbxd)$.

104 - Q 1 **Question 1.** Construire, par un raisonnement de type induction de partition (voir section 1.1.4, page 7), l'opération « **procédure** *Majoritaire1*$(d, t; x, nbx$: **modif**$)$ » où d et t identifient le sous-tableau $T[d\,..\,d + t - 1]$ et (x, nbx) est le couple recherché. L'appel *Majoritaire1*$(1, n, em, nbem)$ calcule donc le couple $(em, nbem)$ pour le tableau $T[1\,..\,n]$. Donner le modèle de division utilisé, puis le code la la procédure.

104 - Q 2 **Question 2.** Quelle est la classe de complexité au pire de cette solution ? Comparer avec la solution itérative de l'exercice 40, page 89.

Un algorithme DpR plus efficace (linéaire)

La théorie (voir section 3.3.1, page 66) nous apprend que si X et Y sont des programmes corrects pour les spécifications respectives (P, R) et (R, Q) alors X;Y est un programme correct pour la spécification (P, Q). Une utilisation particulièrement intéressante de cette propriété de la séquentialité apparaît lorsque R est un prédicat *plus faible* que Q ($Q \Rightarrow R$). Appliqué à notre situation, plutôt que de rechercher directement l'éventuel élément majoritaire, cette propriété permet de rechercher simplement un *candidat* (à être élément majoritaire). C'est le rôle de X. Si le résultat s'avère satisfaisant pour ce qui concerne la complexité, il reste alors à construire le fragment Y qui vérifie si ce candidat est (ou non) l'élément majoritaire du tableau T.

Définition On dira que le couple (x, mx) est candidat majoritaire (CM) dans le tableau T (constitué de t éléments positifs) si et seulement si :

1. mx est un majorant du nombre d'occurrences de x dans T : $\text{mult}(x, T) \leqslant mx$,
2. mx est strictement supérieur à la demi-longueur par défaut de T : $mx > \lfloor t/2 \rfloor$,
3. pour tout y différent de x, le nombre d'occurrences de y dans T est inférieur ou égal à $t - mx$: $\forall y \cdot (y \in \mathbb{N}_1$ **et** $y \neq x \;\Rightarrow\; \text{mult}(y, T) \leqslant t - mx)$.

Par abus de langage, on dira aussi que x est CM dans T s'il existe (au moins) un mx tel que le couple (x, mx) est CM dans T.

Exemples

a) Dans $T = [1, 1, 1, 6, 6]$, le couple $(1, 3)$ est CM. En effet :

1. 3 est un majorant du nombre d'occurrences de 1 dans T,

2. $\lfloor \frac{5}{2} \rfloor < 3 \leqslant 5$,
3. 6 n'est présent que deux fois et $2 \leqslant 5 - 3$.

b) Dans le tableau $T = [1, 1, 5, 5]$, le couple $(1, 3)$ n'est pas CM. En effet :
   a) 3 est un majorant du nombre d'occurrences de 1 dans T,
   b) $\lfloor \frac{4}{2} \rfloor < 3 \leqslant 4$,
   c) mais 5 est présent deux fois et $2 \not\leqslant 4 - 3$.

c) En revanche, le couple $(1, 3)$ est CM dans $T = [1, 1, 4, 5]$ mais T *n'est pas* majoritaire.

d) Les couples $(1, 3)$ et $(1, 4)$ sont tous deux CM dans le tableau $T = [1, 1, 1, 6, 5]$.

Question 3. On cherche à construire la procédure *CandMaj1*(d, t; x, mx : **modif**) qui calcule un couple (x, mx) pour la tranche $T[d \,..\, d + t - 1]$. Ce couple (x, mx) possède la signification suivante : 104 - Q 3

a) si l'on est *certain*, au vu de la situation courante, que le sous-tableau considéré *n'est pas* majoritaire, alors la procédure délivre $(0, 0)$,

b) sinon (on a obligatoirement un CM) la procédure délivre le couple (x, mx) comme CM.

Construire cette procédure par un raisonnement de type induction de partition (voir section 1.1.4, page 7). Donner le modèle de division utilisé ainsi que le code de la procédure *CandMaj1*. Quelle est sa complexité ?

Question 4. Quel traitement complémentaire doit-on faire pour obtenir le résultat attendu (la valeur de l'élément majoritaire s'il en existe un) ? Quelle est alors la complexité du traitement global ? 104 - Q 4

Question 5. Appliquer la solution proposée aux tableaux suivants : 104 - Q 5
$T1 = [1, 2, 1, 3, 2, 1, 1, 3, 3, 2, 3, 1, 1, 1]$, $T2 = [2, 1, 1, 3, 1, 2, 3, 1, 1, 1, 1, 2, 1]$ et
$T3 = [1, 1, 2, 1, 3, 1, 3, 2, 2]$.

Une seconde solution DpR linéaire, simple et originale

On va maintenant mettre en évidence un algorithme de complexité linéaire applicable à un tableau T de taille quelconque, fondé sur un autre type de division de T. Cet algorithme vise lui aussi à déterminer un CM pour le tableau T ; il est construit à partir de la seule propriété 13 suivante :

Propriété 13 :
Si x *est élément majoritaire de* $T[1 \,..\, n]$ *et que* $T[1 \,..\, i]$ *n'est pas majoritaire, alors* x *est élément majoritaire de* $T[i + 1 \,..\, n]$.

Question 6. Démontrer la propriété 13. 104 - Q 6

Question 7. Donner le principe d'une solution de type DpR fondée sur la propriété ci-dessus. Donner le modèle de division utilisé, le(s) problème(s) élémentaire(s) et la complexité globale. 104 - Q 7

Question 8. Écrire la fonction *CandMaj2* correspondante de profil *CandMaj2*(d, t) **résultat** $\mathbb{N}$ rendant 0, si $T[d \,..\, d + t - 1]$ n'a à coup sûr pas d'élément majoritaire et x $(x > 0)$ si x est CM de $T[d \,..\, d + t - 1]$. Traiter les exemples : 104 - Q 8

$T1 = [1,2,1,3,2,1,1,3,3,2,3,1,1,1]$, $T3 = [1,1,2,1,3,1,3,2,2]$,
$T2 = [2,1,1,3,1,2,3,1,1,1,1,2,1]$, $T4 = [1,2,1,1,2,3]$.

Exercice 105. Les deux points les plus proches dans un plan

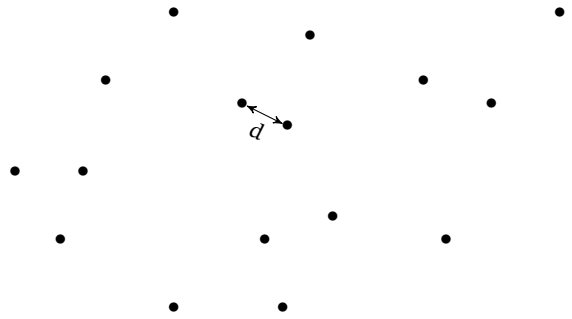
Cet exercice illustre deux points essentiels du développement d'algorithmes : le raffinement et le renforcement. Partant d'une spécification ensembliste, on effectue tout d'abord un choix pour la représentation des ensembles. Une solution naïve pour ce raffinement ne donnant pas satisfaction du point de vue de la complexité, on renforce la postcondition d'une étape de l'algorithme afin d'obtenir une solution en $\Theta(n \cdot \log_2(n))$.

On cherche un algorithme du type DpR pour résoudre le problème suivant : on dispose d'un ensemble fini E de n ($n \geqslant 2$) points dans un plan. Quelle est la distance qui sépare les deux points les plus proches[6] ?

Plus formellement : soit l'espace $\mathbb{R}^2$, muni de la métrique euclidienne notée Δ. Soit E un ensemble fini de n points dans $\mathbb{R}^2$. Soit $p = (p_x, p_y)$ et $q = (q_x, q_y)$ deux points de $\mathbb{R}^2$. On note $\Delta(p,q)$ la distance euclidienne entre p et q, soit $\Delta(p,q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$. On recherche le réel positif d défini par

$$d = \min(\{a \in E \text{ et } b \in E \text{ et } a \neq b \mid \Delta(a,b)\})$$

Exemple Pour l'ensemble de points ci-dessous :



d est la valeur recherchée.

Dans la suite, la complexité sera exprimée en nombre de conditions évaluées.

105 - Q 1 **Question 1.** Quel est l'ordre de complexité de l'algorithme naïf de calcul de d ?

6. Il n'y a jamais unicité de ce couple de points puisque si (a,b) est un tel couple, c'est aussi le cas de (b,a). La question de l'identification de *l'ensemble* de tels couples conduit à un aménagement trivial des programmes développés ci-dessous. Cette question n'est pas abordée.

Dans la suite, n est une puissance de 2. L'opération « **fonction** *PlusProches1*(S) **résultat** $\mathbb{R}_+$ » délivre la distance entre les voisins les plus proches dans l'ensemble des points S. Dans le contexte d'appel suivant :

```
1. constantes
2.    Coord = {x, y | x ∈ ℝ et y ∈ ℝ} et E ⊂ Coord et E = {...}
3. début
4.    écrire(PlusProches1(E))
5. fin
```

la fonction *PlusProches1* ci-dessous fournit une première ébauche :

```
1. fonction PlusProches1(S) résultat ℝ+ pré
2.    S ⊂ Coord et ∃k · (k ∈ ℕ1 et card(S) = 2^k) et S1 ⊂ S et S2 ⊂ S et
3.    (S1 ∩ S2) = ∅ et (S1 ∪ S2) = S et card(S1) = card(S2) et
4.    d ∈ ℝ+* et d1 ∈ ℝ+* et d2 ∈ ℝ+*
5. début
6.    si card(S) = 2 alors
7.       soit a, b tel que
8.          a ∈ S et b ∈ S et a ≠ b
9.       début
10.         résultat Δ(a, b)
11.      fin
12.   sinon
13.      d1 ← PlusProches1(S1) ;
14.      d2 ← PlusProches1(S2) ;
15.      d ← min({d1, d2}) ;
16.      Rassembler1(...) ;
17.      résultat ...
18.   fin si
19. fin
```

La quantification existentielle précise que le nombre d'éléments de S est une puissance de 2. La valeur d_1 (resp. d_2) est la plus petite distance trouvée dans le sous-ensemble S_1 (resp. S_2).

Question 2. Donner les quatre points clés de la construction inductive de la fonction *PlusProches1* (il s'agit ici de faire du « reverse engineering »). 105 - Q 2

Question 3. Quel est l'ordre de complexité de la fonction *PlusProches1* si la procédure *Rassembler1* est en $\Theta(n^2)$? en $\Theta(n \cdot \log_2(n))$? en $\Theta(n)$? 105 - Q 3

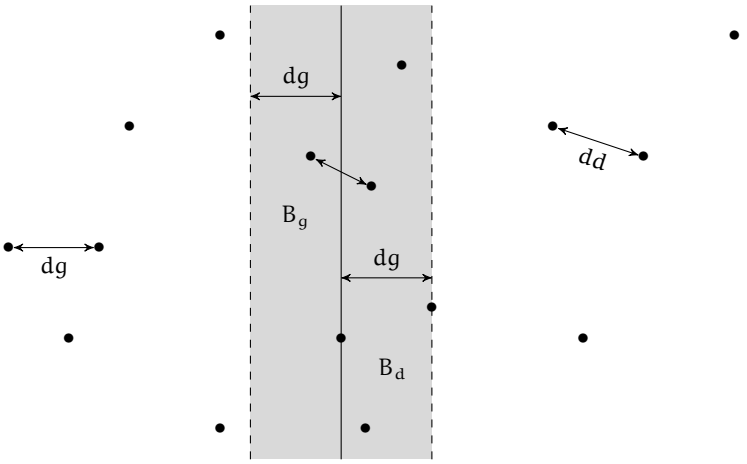
On cherche à présent à raffiner la représentation des ensembles S, S_1 et S_2. Il existe en général de nombreuses façons de partitionner un ensemble S de taille paire en deux sous-ensembles de même taille. Il est intéressant, pour des raisons de calcul de distances, de réaliser cette partition par une *droite* séparatrice. Le choix d'une droite qui soit verticale ou horizontale est raisonnable dans la mesure où la distance entre un point et la droite en question se limite alors à un calcul sur *une seule* des coordonnées du point. Dans la suite, on considère arbitrairement que cette droite se présente verticalement. Le raffinement de l'ensemble S peut alors se faire par un tableau de couples T[i .. s] (T[1 .. n] pour l'ensemble initial E, de sorte que, si Coord est l'ensemble des couples (x, y) de réels, alors $T[1 .. n] \in 1 .. n \rightarrow \text{Coord}$), trié sur les abscisses croissantes (ceci assure que la proximité des points

avec la droite séparatrice se traduit par une proximité dans le tableau). Si $mil = \lfloor (i+s)/2 \rfloor$, les ensembles S_1 et S_2 se raffinent par les sous-tableaux $T[i\,..\,mil]$ et $T[mil+1\,..\,s]$. d_1 et d_2 sont raffinés par dg et dd (distances gauche et droite). Le point $T[mil]$ appartient à la droite séparatrice. C'est aussi le cas de tous les points qui ont la même abscisse que $T[mil]$, qu'ils appartiennent à S_1 ou à S_2. la fonction *PlusProches1* se raffine alors de la manière suivante (T est ici une structure globale, seules les bornes i et s sont passées en paramètres) :

```
1.  fonction PlusProches2(i, s) résultat ℝ+ pré
2.      i ∈ 1..n et s ∈ i+1..n et ∃k·(k ∈ ℕ1 et s−i+1 = 2^k) et
3.      EstTriéX(T[i..s]) et mil ∈ i..s−1 et d ∈ ℝ+* et dg ∈ ℝ+* et dg ∈ ℝ+*
4.  début
5.      si s−i+1 = 2 alors
6.          résultat Δ(T[i], T[s])
7.      sinon
8.          mil ← ⌊(i+s)/2⌋ ;
9.          dg ← PlusProches2(i, mil) ;
10.         dd ← PlusProches2(mil+1, s) ;
11.         d ← min({dg, dd}) ;
12.         Rassembler2(...) ;
13.         résultat ...
14.     fin si
15. fin
```

Le conjoint *EstTriéX*(T[i..s]) exprime que le (sous-)tableau T[i..s] est trié sur les abscisses croissantes.

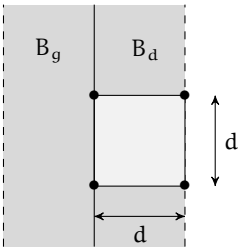
Exemple Le schéma ci-dessous reprend l'exemple précédent. On remarque que le point T[mil] est situé sur la droite séparatrice et que sept points sont à sa gauche et huit à sa droite.



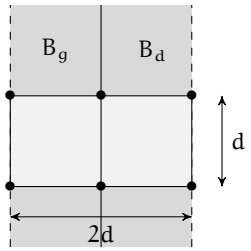
La valeur dg (resp. dd) est la meilleure distance trouvée dans la moitié de gauche (resp. de droite). La valeur dg est la plus petite des deux valeurs dg et dd (c'est le d de la ligne 11 du code de *PlusProches2*).

Posons $T[mil] = (m_x, m_y)$. On appelle B_g (resp. B_d) l'ensemble des points situés dans la bande verticale délimitée par les droites d'équation $x = m_x - d$ et $x = m_x$ (resp. $x = m_x$ et $x = m_x + d$). Par conséquent, si deux points p_1 et p_2 sont tels que $p_1 \in T[i\,..\,mil]$ et $p_2 \in T[mil + 1\,..\,s]$ et $\Delta(p_1, p_2) \leqslant d$ alors $p_1 \in B_g$ et $p_2 \in B_d$.

Question 4. Montrer que toute fenêtre carrée ouverte sur l'une des deux bandes B_g ou B_d contient au plus quatre points : 105 - Q 4



Il est facile d'en conclure (puisque S est un *ensemble* de points) qu'il existe au plus six points sur un rectangle de longueur 2d et de hauteur d s'étalant sur toute la largeur de la bande :



Question 5. Dans l'hypothèse où les points appartenant aux bandes B_g et B_d sont enregistrés dans un tableau Y, que faut-il imposer à ce tableau pour que la recherche d'un meilleur voisin pour Y[j] se fasse dans un sous-tableau de Y, *de taille bornée*, débutant en $Y[j + 1]$? Donner un majorant à cette borne. 105 - Q 5

Question 6. Construire une version itérative de l'opération « **procédure** *Rassembler2*(deb, fin; md : **modif**) », où deb et fin sont tels que T[deb..fin] représente l'ensemble de points S, et où md est, en entrée, la meilleure valeur trouvée dans S_1 ou dans S_2 et, en sortie, la meilleure valeur trouvée dans S. Cette procédure commence par construire le tableau Y de la question précédente avant de l'exploiter pour rechercher un éventuel meilleur couple. Quelle est l'ordre de grandeur de complexité de cette procédure ? Expliciter le modèle de division de la solution utilisant *PlusProches2*. Quelle est sa complexité ? 105 - Q 6

Question 7. La solution précédente n'est pas entièrement satisfaisante sur le plan de la complexité : on aurait espéré une complexité en $\Theta(n \cdot \log_2(n))$. Cet objectif n'est pas atteint. Peut-on identifier l'origine du surcoût observé ? Dans un souci de meilleure efficacité, on propose de renforcer l'hypothèse d'induction de la manière suivante : l'opération *PlusProches3* délivre (non seulement) la distance entre les voisins les plus proches dans 105 - Q 7

T[i .. s] et (mais aussi) une version R[1 .. s − i + 1] de T[i .. s] triée sur les ordonnées croissantes. Construire l'opération « **procédure** *PlusProches3*(i, s; d, R : **modif**) ». En donner la complexité.

105 - Q 8 **Question** 8. Comment peut-on traiter le cas n quelconque ($n \geqslant 2$) ?

Exercice 106. Distance entre séquences : l'algorithme de Hirschberg

La compréhension de l'énoncé de cet exercice exige d'avoir assimilé au préalable l'introduction au chapitre portant sur la programmation dynamique (voir chapitre 9). Celle-ci porte sur un algorithme de recherche de l'une des plus longues sous-séquences communes (en général il n'y a pas unicité), dénommé WFlg *et sur un algorithme de recherche de* la longueur *des plus longues sous-séquences communes (*WFLgAvant*).*

Dans sa version la plus simple, l'algorithme d'Hirschberg recherche l'une des plus longues sous-séquences communes (plssc) à deux chaînes. Il est fondé sur une technique DpR, mais l'étape de division utilise le principe de programmation dynamique. L'avantage proclamé de cette méthode est le gain de place par rapport aux méthodes fondées uniquement sur le principe de la programmation dynamique. D.S. Hirschberg l'ayant imaginé dans les années 70, cet argument avait alors plus de force que de nos jours (encore que si l'on traite des séquences biologiques de milliers de lettres, si on l'exploite pour faire la correction de dictées ou pour des recherches de similarité sur le Web, il présente encore une certaine utilité). Mais c'est surtout un exemple extraordinaire, à notre connaissance unique, de l'association de deux techniques si différentes. Enfin, cet algorithme s'appuie sur un théorème d'optimalité qui montre, s'il en était besoin, qu'il est souvent nécessaire de disposer d'un bagage minimal en mathématiques discrètes pour être à même de construire méthodiquement des applications informatiques. Compte tenu de son objectif visant à optimiser la ressource mémoire, cet exercice est l'un des seuls où le souci d'une bonne complexité spatiale conditionne le développement.

Soit x et y deux chaînes sur un alphabet Σ. Soit m et n les longueurs respectives de x et de y ($m = |x|$ et $n = |y|$). On recherche une sous-séquence commune à x et y de longueur maximale. L'ensemble des plus longues sous-séquences communes de x et y est noté $PLSSC(x, y)$. Cet ensemble n'est jamais vide, puisque la chaîne vide ε est une sous-séquence de toute chaîne. L'ensemble des sous-séquences communes à x et à y est noté $SSC(x, y)$. Dans la suite de cet exercice, Σ est implicitement l'alphabet latin de 26 lettres : $\Sigma = \{a, b, \ldots, z\}$.

Exemple Considérons l'alphabet Σ et les deux chaînes $u = attentat$ et $v = tante$. La chaîne *teta* est une sous-séquence de u. La chaîne *tnte* est une sous-séquence de v mais *n'est pas* une sous-séquence de u (les symboles ne s'y rencontrent pas dans le même ordre). Les chaînes ε, *tt*, *at* et *tnt* sont des sous-séquences communes à u et à v ; elles appartiennent donc à $SSC(u, v)$. On a ici $PLSSC(u, v) = \{ant, ate, tat, tnt, tte\}$.

Notations

— Si c (resp. $c[i\,..\,s]$) est une chaîne, on note $\overline{c}$ (resp. $\overline{c[i\,..\,s]}$) la chaîne miroir. Si C est un ensemble de chaînes, on note $\overline{C}$ l'ensemble des chaînes miroirs de C. On remarque que $\overline{\overline{c}} = c$ et que $\overline{PLSSC}(\overline{x},\overline{y}) = PLSSC(x,y)$.

— Pour un ensemble $PLSSC(x,y)$ donné, tous ses éléments sont, par définition, de même longueur, que l'on note $lg(PLSSC(x,y))$.

Le principe général à la base de l'algorithme d'Hirschberg pour la recherche d'un élément de l'ensemble $PLSSC(x,y)$ consiste à diviser x en deux sous-chaînes et, pour chaque partie x_1 et x_2, à rechercher un préfixe y_1 et un suffixe y_2 de y (avec $y_1 \cdot y_2 = y$), tels que si $c_1 \in PLSSC(x_1,y_1)$ et $c_2 \in PLSSC(x_2,y_2)$ alors

$$c_1 \cdot c_2 \in PLSSC(x,y).$$

On peut ensuite appliquer le même principe sur les couples de chaînes (x_1,y_1) et (x_2,y_2). La principale difficulté de l'algorithme réside dans la découverte de sous-chaînes y_1 et y_2 appropriées.

Exemple Pour $u = attentat$, $u_1 = atte$, $u_2 = ntat$ et $v = tante$, la table 8.1 répertorie les ensembles $PLSSC(u_1,v_1)$ et $PLSSC(u_2,v_2)$ pour tous les couples (v_1,v_2) possibles ($v_1 \cdot v_2 = v$). L'avant-dernière colonne fournit toutes les concaténations possibles entre les éléments de $PLSSC(u_1,v_1)$ et ceux de $PLSSC(u_2,v_2)$. Les plus longues sous-séquences sont obtenues pour les couples $(v_1,v_2) = (tante, \varepsilon)$, $(v_1,v_2) = (ta, nte)$, $(v_1,v_2) = (t, ante)$ et $(v_1,v_2) = (\varepsilon, tante)$. On note que l'ensemble des plus longues sous-séquences apparaissant dans la colonne « Concat. » de la table 8.1 (celles de longueur 3) est égal à l'ensemble $PLSSC(u,v)$. Faisons temporairement l'hypothèse qu'il en est toujours ainsi : le résultat ne dépend pas de la position à laquelle on coupe x. La confirmation est une conséquence du théorème de la page 281.

$u_1 = atte$			$u_2 = ntat$				
v_1	(1)	$lg_1()$	v_2	(2)	$lg_2()$	Concat.	(3)
tante	{*ate*, *tte*}	3	ε	{ε}	0	{*ate*, *tte*}	3
tant	{*at*, *tt*}	2	*e*	{ε}	0	{*at*, *tt*}	2
tan	{*a*, *t*}	1	*te*	{*t*}	1	{*at*, *tt*}	2
ta	{*a*, *t*}	1	*nte*	{*nt*}	2	{*ant*, *tnt*}	3
t	{*t*}	1	*ante*	{*at*, *nt*}	2	{*tat*, *tnt*}	3
ε	{ε}	0	*tante*	{*tat*}	3	{*tat*}	3

Tab. 8.1 – *Sous-séquences communes et plus longues sous-séquences communes. La colonne* (1) *(resp.* (2) *et* (3)*) représente* PLSSC(u_1,v_1) *(resp.* PLSSC(u_2,v_2) *et* $lg_1() + lg_2()$*).*

Détaillons à présent cette partie de l'algorithme d'Hirschberg. Rappelons tout d'abord que l'algorithme de programmation dynamique *WFlg* mentionné dans l'introduction fournit, dans un tableau de m colonnes et de n lignes et pour chaque préfixe x′ de x et y′ de y, la longueur des plssc à x′ et à y′ (voir schéma (a) de la figure 8.4, page 280). En particulier, la dernière colonne fournit la longueur de la plssc de x et de *tous* les préfixes

y' de y ; le coin nord-est (grisé sur le schéma) est la longueur de la plssc de x et de y. On peut en déduire que chaque ligne et chaque colonne du tableau sont triées (au sens large) par ordre croissant (le tableau est une bâtière, voir exercice 100, page 262). Dans la suite, la colonne de la position j est notée P_j et se définit par :

$$P_j[i] = lg(PLSSC(x[1\,..\,j], y[1\,..\,i])) \text{ pour } i \in 0\,..\,n.$$

De manière symétrique, l'algorithme dual qui traite les chaînes miroirs fournit, pour chaque préfixe x' de $\overline{x}$ et y' de $\overline{y}$, la longueur de la plssc à x' et à y' (voir schéma (b) de la figure 8.4, page 280). Dans ce cas de figure, la colonne de gauche, lue de haut en bas, fournit la longueur de la plssc entre $\overline{x}$ et tous les préfixes de $\overline{y}$. Le coin sud-ouest du tableau (grisé sur le schéma) est la longueur de la plssc de $\overline{x}$ et de $\overline{y}$ (cette longueur est bien sûr la même que celle trouvée entre x et y). La colonne de la position j est notée P^*_j, elle se définit par :

$$P^*_j[i] = lg(PLSSC(\overline{x[j\,..\,m]}, \overline{y[n-i+1\,..\,n]})) \text{ pour } i \in 0\,..\,n.$$

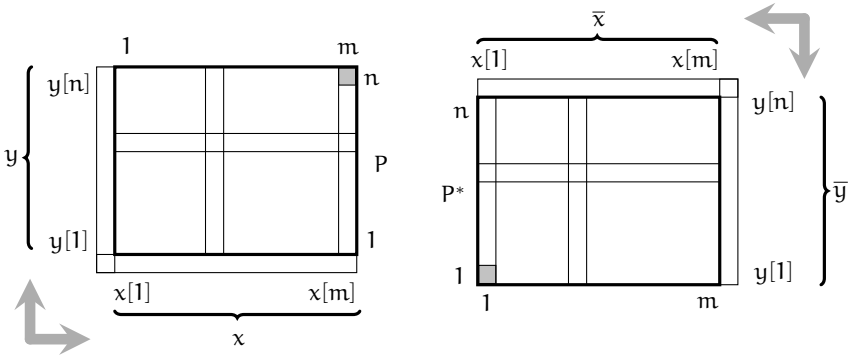


Fig. 8.4 – *Recherche de la longueur de la plus longue sous-séquence commune par programmation dynamique. Schéma* (a) *: tableau* P *pour les chaînes* x *et* y. *Schéma* (b) *: tableau* P^* *pour les chaînes* $\overline{x}$ *et* $\overline{y}$.

On suppose à présent que l'on coupe arbitrairement x en deux parties x_1 ($x_1 = x[1\,..\,j]$) et x_2 ($x_2 = x[j+1\,..\,m]$) ($x = x_1 \cdot x_2$) et y en y_1 ($y_1 = y[1\,..\,i]$) et y_2 ($y_2 = y[i+1\,..\,n]$) ($y = y_1 \cdot y_2$), avant de calculer le tableau « avant » pour le couple (x_1, y_1) et le tableau « arrière » pour le couple $(\overline{x_2}, \overline{y_2})$. C'est ce que montre le schéma (a) de la figure 8.5. Soit $c_1 \in \text{PLSSC}(x_1, y_1)$ et $c_2 \in \overline{\text{PLSSC}}(\overline{x_2}, \overline{y_2})$. On a donc $|c_1| = P_j[i]$ et $|c_2| = P^*_{j+1}[n-i]$. Si on pose $c = c_1 \cdot c_2$ on a $|c| = P_j[i] + P^*_{j+1}[n-i]$. Rechercher la plssc de x et de y est équivalent à rechercher la plus grande valeur prise par l'expression $P_j[i] + P^*_{j+1}[n-i]$ lorsque i varie entre 0 et n. C'est ce que suggère la partie (b) de la figure 8.5.

Un algorithme DpR fondé sur ces seules considérations aurait une complexité spatiale en $\Omega(m \cdot n)$, ce qui n'est pas compatible avec nos ambitions. L'introduction du chapitre consacré à la programmation dynamique, page 319, fournit une solution puisqu'elle montre qu'il est possible de « linéariser » l'algorithme *WFlg* de façon à obtenir une version (dénommée *WFLgAvant*) dont la complexité spatiale est en $\Theta(n)$. La partie (c) de la figure 8.5 montre comment la linéarisation permet de réduire l'espace utilisé.

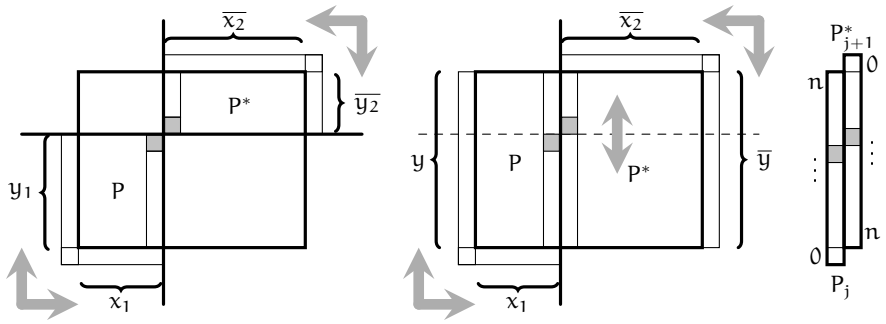


Fig. 8.5 – *Principe de l'algorithme d'Hirschberg*

Exemple Dans l'exemple de la figure 8.6, page 282, les colonnes grisées fournissent les vecteurs P_4 et P_5^* pour les appels $WFLgAvant(atte, tante, P_4)$ et de $WFLgArrière(ntat, tante, P_5^*)$. Les autres colonnes ne sont présentes que pour la clarté de l'exposé.

Pour le cas de la figure 8.6, page 282, appelons M_4 (en référence au théorème qui suit) la plus grande valeur trouvée parmi les six sommes $P_4[0] + P_5^*[5], P_4[1] + P_5^*[4], P_4[2] + P_5^*[3], P_4[3] + P_5^*[2], P_4[4] + P_5^*[1], P_4[5] + P_5^*[0]$. M_4 vaut 3 et c'est aussi la valeur de $lg(PLSSC(x, y))$. Cette valeur est atteinte avec quatre des six sommes : la première, la seconde, la troisième et la sixième.

Question 1. On a vu ci-dessus que l'on a besoin de l'opération « **procédure** *WFLgArrière* $(x, y; Q$: **modif**) », qui fournit, dans le vecteur Q, la longueur des plus longues sous-séquences communes à $\overline{x[1\,..\,m]}$ et à $\overline{y[i+1\,..\,n]}$, pour $i \in 0\,..\,n$. Aménager le code de la procédure *WFLgAvant* (voir section 9.2 page 323) de façon à obtenir celui de la procédure *WFLgArrière*. Que peut-on dire des complexités temporelle (en nombre de conditions évaluées liées aux procédures *WFLgAvant* et *WFLgArrière*) et spatiale de cet algorithme ? 106 - Q 1

Le développement réalisé ci-dessus se formalise par le théorème suivant :

Théorème (d'optimalité d'Hirschberg) :
Si

$$M_j = \max_{i \in 0..n} \left(P_j[i] + P^*_{j+1}[n-i]\right)$$

alors

$$M_j = P_m[n]$$

pour $j \in 0\,..\,m$.

Autrement dit, en se reportant à l'exemple de la figure 8.6, page 282, si pour une certaine colonne j donnée, M_j est la plus grande valeur obtenue en ajoutant les valeurs $P_j[i]$ et $P^*_{j+1}[n-i]$ ($i \in 0\,..\,n$), alors j est un candidat possible pour trouver un bon découpage de y. Ce théorème est la clé de voûte de l'algorithme d'Hirschberg.

Question 2. Calculer la valeur de M_5 pour $u =$ *esclandre* et $v =$ *scandale*. Pour quel unique indice k de P_5 cette valeur est-elle atteinte ? Rechercher « à la main » les ensembles $PLSSC(u[1\,..\,5], v[1\,..\,k])$ et $\overline{PLSSC}(\overline{u[6\,..\,9]}, \overline{v[k+1\,..\,8]})$. En déduire $PLSSC(u, v)$. 106 - Q 2

106 - Q 3 **Question** 3. Que penser de la suggestion suivante : on peut calculer le vecteur P^*_{j+1} en utilisant l'algorithme *WFLgAvant* appliqué aux suffixes x_2 et y_2 ?

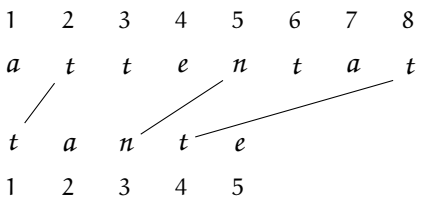
		0	1	2	3	4	4	3	2	1	0		
		P_0	P_1	P_2	P_3	P_4	P^*_5	P^*_6	P^*_7	P^*_8	P^*_9	v	
			a	*t*	*t*	*e*	*n*	*t*	*a*	*t*			u
							0	0	0	0	0		0
5	*e*	0	1	2	2	3	0	0	0	0	0	*e*	1
4	*t*	0	1	2	2	2	1	1	1	1	0	*t*	2
3	*n*	0	1	1	1	1	2	1	1	1	0	*n*	3
2	*a*	0	1	1	1	1	2	2	2	1	0	*a*	4
1	*t*	0	0	1	1	1	3	3	2	1	0	*t*	5
0		0	0	0	0	0							
v			*a*	*t*	*t*	*e*	*n*	*t*	*a*	*t*			
	u	P_0	P_1	P_2	P_3	P_4	P^*_5	P^*_6	P^*_7	P^*_8	P^*_9		
		0	1	2	3	4	5	6	7	8			

Fig. 8.6 – *Appels de* WFLgAvant *et* WFLgArrière. *Les deux vecteurs* P_4 *et* P^*_5 *se lisent en sens inverse, de bas en haut pour* P_4*, de haut en bas pour* P^*_5.

106 - Q 4 **Question** 4. Démontrer le théorème d'optimalité d'Hirschberg. Suggestion : démontrer d'une part que $M_j \leqslant P_m[n]$ et d'autre part que $M_j \geqslant P_m[n]$.

106 - Q 5 **Question** 5. Décrire le raisonnement DpR qui permet de construire l'opération « **procédure** *HirschPLSSC*$(x, y; c$: **modif**$)$ » qui, à condition que le paramètre effectif d'entrée-sortie correspondant à c soit préalablement initialisé à la chaîne vide, délivre dans ce paramètre l'une quelconque des chaînes de l'ensemble $PLSSC(x, y)$. Quel est le modèle de division qui s'applique ? Sachant que l'on vise une complexité spatiale en $\mathcal{O}(\min(\{m, n\}))$, fournir le code cette opération. Que peut-on dire de la complexité temporelle de cet algorithme ? Pour simplifier les calculs, on peut se limiter au cas où m est une puissance de 2.

On s'intéresse à présent à l'aménagement de l'algorithme d'Hirschberg afin d'obtenir non plus une chaîne mais une trace. Cette notion est définie dans l'exercice 137, page 366. On se limite ici à la présentation d'un exemple. Considérons à nouveau les chaînes $u = \textit{attentat}$ et $v = \textit{tante}$. Une trace possible entre u et v est fournie par :



Cette structure peut se matérialiser par la liste triée des couples de la relation trace : $\langle(2,1),(5,3),(8,4)\rangle$. La recherche d'une trace oblige à disposer des indices « absolus » des

symboles dans les chaînes. Pour ce faire, on décide d'identifier les chaînes x et y et leurs sous-chaînes par l'indice de leurs extrémités (respectivement ix, sx, iy et sy ci-dessous).

Disposer d'une trace entre deux chaînes permet d'obtenir facilement la plus longue sous-séquence commune, ainsi que l'alignement correspondant, la trace d'édition (c'est-à-dire la liste optimale des opérations d'édition permettant de transformer la première chaîne en la seconde), le coût de la transformation ou encore la distance entre les deux chaînes (voir exercice 137, page 366).

L'objectif de cette question est d'adapter l'algorithme d'Hirschberg de façon à calculer une trace entre deux chaînes. On se focalise tout d'abord sur la partie inductive de l'algorithme et plus particulièrement sur l'étape de rassemblement. Cette fois, il ne s'agit plus simplement de concaténer les séquences optimales de gauche et de droite, mais – si la situation l'exige – de prendre en considération les coordonnées d'un symbole commun à x et à y. Contrairement à la version développée à la question 5, il faut se donner les moyens de comparer le symbole qui se trouve au milieu de x au symbole de y situé sur la ligne qui sépare y en deux parties. L'indice q séparateur de y étant supposé disponible, trois situations sont à distinguer (voir figure 8.7, page 283).

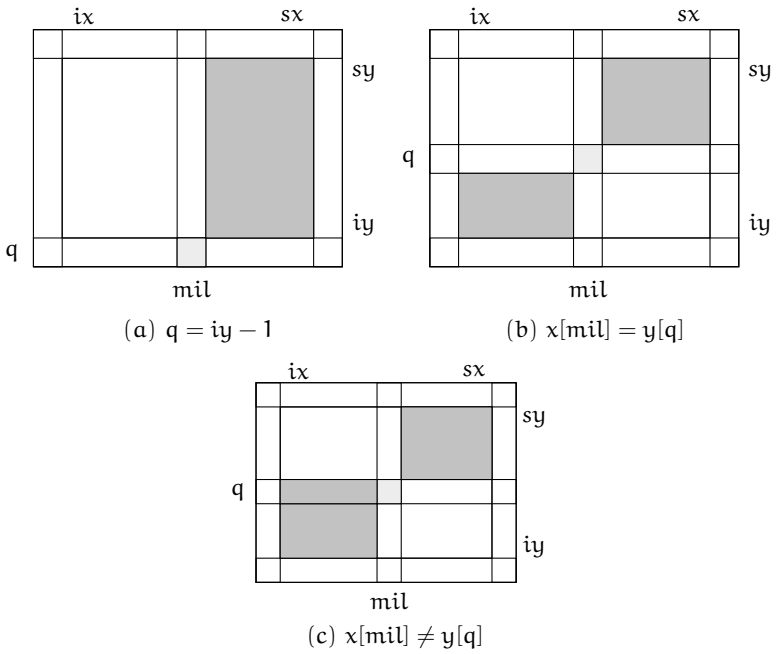


Fig. 8.7 – *Calcul de la trace – les trois cas à considérer*

Le premier cas (voir partie (a) de la figure 8.7) est celui où l'indice séparateur q est égal à $iy - 1$. Dans ce cas, la trace est à rechercher uniquement sur le rectangle allant du coin $(mil + 1, iy)$ au coin (sx, sy) (le rectangle en gris foncé de la figure). Ce cas se retrouve à la figure 8.6 si l'on choisit comme indice séparateur $q = 0$. Le second cas (voir partie (b) de la figure 8.7) est celui où $x[mil] = y[q]$. Le couple (mil, q) doit être inséré dans la trace, et la recherche doit se poursuivre sur les deux rectangles grisés. Ce cas se retrouve

à la figure 8.6, page 282, si l'on choisit $q = 5$. Enfin, le troisième cas (voir partie (c) de la figure 8.7) est celui où $x[mil] \neq y[q]$. Le symbole $x[mil]$ n'est aligné avec aucun élément de la chaîne $y[ix \,..\, q]$: on peut éliminer la colonne mil pour poursuivre la recherche sur les deux rectangles grisés. Ce cas correspond, sur la figure 8.6, soit à $q = 1$, soit à $q = 2$.

106 - Q 6

Question 6. Construire la procédure *HirschTrace* (pour simplifier, on pourra s'affranchir des contraintes liées à la complexité spatiale). Fournir le modèle de division qui s'applique, ainsi que le code de la procédure. Pour ce faire, on suppose disponible le type $Trace$, liste de couples de naturels, doté de l'opérateur de concaténation $\cdot$ et du constructeur « $ct(a, b)$ » (resp. *tv*) qui crée une trace constituée du couple (a, b) (resp. une trace vide).

Exercice 107. L'enveloppe convexe

L'intérêt de cet exercice réside principalement dans les trois points suivants : i) le choix de la structure de données pour représenter une enveloppe convexe, qui conditionne largement l'efficacité du résultat, ii) la phase de rassemblement qui se met en œuvre par une itération non triviale, et iii) le renforcement de l'hypothèse d'induction (objet de la quatrième question), qui simplifie la première solution et la rend plus efficace.

L'objectif de l'exercice est de rechercher l'enveloppe convexe d'un ensemble E fini non vide de n points du plan. L'enveloppe convexe de E est un polygone convexe P tel que tous les sommets de P appartiennent à E et tous les autres points de E sont situés « à l'intérieur » du polygone. Dans la suite, on considère que trois points de E ne sont jamais alignés [7].

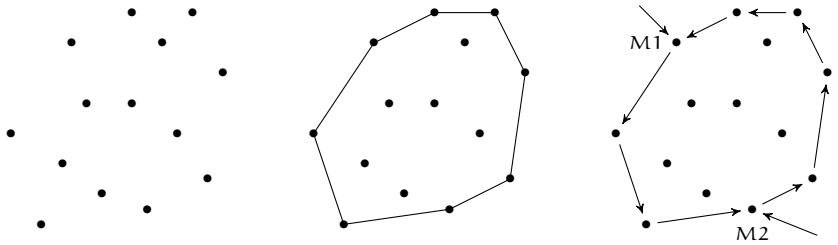


Fig. 8.8 – (a) *L'ensemble de points* E – (b) *L'enveloppe convexe de* E – (c) *L'enveloppe orientée de* E

De nombreux algorithmes existent pour résoudre ce problème. On s'intéresse à un algorithme DpR particulier connu sous le nom de « fusion des enveloppes ». Dans cet algorithme, la principale difficulté réside dans la partie « rassemblement » de l'algorithme

7. Dans le cas contraire, il suffirait (si l'on peut dire) de ne conserver que les deux points les plus éloignés pour obtenir l'enveloppe recherchée.

DpR. Obtenir un algorithme de rassemblement efficace (en $\mathcal{O}(n)$ conditions évaluées si on souhaite une solution en $\mathcal{O}(n \cdot \log_2(n))$) exige de disposer d'une structure de données bien adaptée.

Principe de la solution

On suppose (cas de base) que l'on sait trouver directement l'enveloppe convexe d'un ensemble d'un et de deux points. Le cas inductif consiste à diviser l'ensemble des points en deux sous-ensembles gauche et droit ayant approximativement le même cardinal. Tous les points du sous-ensemble gauche (resp. droit) ont des abscisses strictement inférieures (resp. strictement supérieures) à tous les points du sous-ensemble droit (resp. gauche). Pour parvenir à une solution efficace, cette contrainte exige que l'ensemble E soit raffiné par un tableau T trié sur les abscisses croissantes. Elle a aussi une incidence sur l'algorithme de séparation, puisque deux points de même abscisse doivent appartenir au même sous-ensemble [8].

Si, selon l'hypothèse d'induction, on sait calculer l'enveloppe convexe des sous-ensembles gauche et droit, il reste – phase de rassemblement – à rechercher les deux segments tangents à chacune des deux enveloppes afin de fusionner le tout en une seule enveloppe (voir figure 8.9, page 286).

Définitions – Notations – Structure de données – Propriétés

On présente ici plusieurs notions nécessaires à la compréhension et à la mise en œuvre de la solution. Plutôt que de considérer une enveloppe convexe P comme un ensemble de segments, on décide de représenter P comme une *succession* de *vecteurs*, ce qui dote P d'une orientation. Dans la suite, on choisit l'orientation directe (c'est-à-dire selon le sens trigonométrique), comme dans la partie (c) de la figure 8.8 page 284.

Une enveloppe convexe en tant que telle ne présente que peu d'intérêt si l'on ne dispose pas (d'au moins) un sommet par lequel y accéder.

Définition et notation 1 (Enveloppe convexe à clé) :
Soit P *une enveloppe convexe orientée. Si* M *est un sommet de cette enveloppe, on note* $\widehat{\mathrm{M}}$ *le couple* (P, M). M *est la clé (d'entrée) de l'enveloppe* P.

Cette définition présuppose qu'un sommet n'appartient qu'à une seule enveloppe, ce qui se trouve être toujours le cas par la suite. La partie (c) de la figure 8.8, page 284, montre $\widehat{\mathrm{M1}}$ et $\widehat{\mathrm{M2}}$, deux enveloppes à clé de la même enveloppe convexe. Dans la suite, le contexte permet de déterminer s'il est question d'une enveloppe simple ou d'une enveloppe à clé.

La structure de données Le raffinement de la structure de données « enveloppe » peut par exemple se faire en utilisant une liste doublement chaînée. L'identificateur EnvConv dénote l'ensemble des enveloppes convexes à clé possibles. Les opérations suivantes sont supposées définies sur cette structure de données :

— **fonction** *CréerEnvConv1*(M) **résultat** EnvConv : fonction qui crée l'enveloppe $\widehat{\mathrm{M}}$ à partir d'un ensemble constitué du seul point M.

8. C'est pour cette raison que le cas de base doit prendre en considération le cas d'un ensemble de *deux* points. En effet, dans le cas contraire, on ne pourrait exclure que couper un ensemble de deux points donne d'une part l'ensemble vide et d'autre part l'ensemble en question.

— **fonction** *CréerEnvConv2*(M1, M2) **résultat** EnvConv : fonction qui crée l'enveloppe $\widehat{M1}$ à partir d'un ensemble constitué des deux seuls points M1 et M2.

— **fonction** *Succ*($\widehat{M}$) **résultat** point : fonction qui délivre le point qui suit M dans l'enveloppe $\widehat{M}$.

— **fonction** *Pred*($\widehat{M}$) **résultat** point : fonction qui délivre le point qui précède M dans l'enveloppe $\widehat{M}$.

— **fonction** *Fusion*(GD_N, GD_S) **résultat** EnvConv : si $GD_N = (\widehat{G_N}, \widehat{D_N})$ et $GD_S = (\widehat{G_S}, \widehat{D_S})$, cette fonction fusionne les deux enveloppes afin d'en former une troisième, selon le principe illustré par la figure 8.9, page 286.

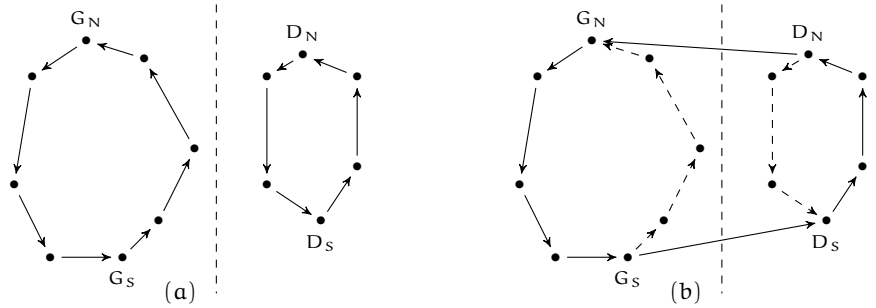


Fig. 8.9 – (a) *Avant la fusion* – (b) *Après la fusion*

Cette opération exige comme préconditions i) que les deux enveloppes soient situées de part et d'autre d'une ligne verticale, ii) que la ligne support du vecteur $\overrightarrow{D_N G_N}$ (resp. $\overrightarrow{G_S D_S}$) soit une tangente inférieure (resp. supérieure)[9] commune aux deux enveloppes.

Par ailleurs, l'ensemble point des points du plan est supposé défini par l'identificateur point qui est tel que $\text{point} = \{x, y \mid x \in \mathbb{R} \text{ et } y \in \mathbb{R}\}$.

Définition 31 (Déterminant de deux vecteurs) :
Soit deux vecteurs $\vec{v} = \begin{pmatrix} x \\ y \end{pmatrix}$ *et* $\vec{v'} = \begin{pmatrix} x' \\ y' \end{pmatrix}$ *du plan orienté. Le déterminant de* $\vec{v}$ *et de* $\vec{v'}$, *noté* $\det(\vec{v}, \vec{v'})$ *se définit par le scalaire* $x \cdot y' - x' \cdot y$. *Si ce déterminant est positif, l'angle que font les deux vecteurs a la même orientation que celle du plan, s'il est nul, les deux vecteurs sont colinéaires, s'il est négatif l'angle a une orientation opposée à celle du plan.*

La figure 8.10 fournit une illustration de cette définition.

La notion de déterminant permet de savoir si un point est ou non situé à l'extérieur d'un polygone convexe. Plus intéressant pour nous, tout en évitant des calculs explicites d'angles (coûteux et sujets aux erreurs d'arrondis), elle permet également de décider si d'un point extérieur on « voit » ou non un côté donné d'une enveloppe convexe. Le côté PQ d'une

9. La tangente d'un polygone convexe est une droite qui a un et un seul point commun avec la surface délimitée par le polygone.

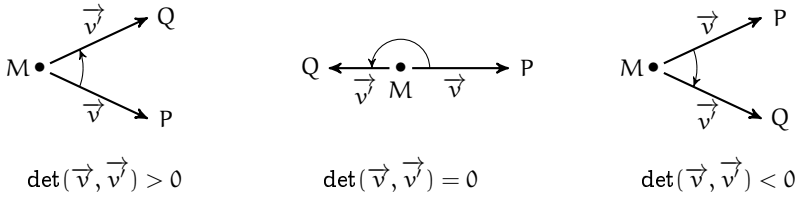


Fig. 8.10 – *Signe du déterminant de deux vecteurs*

enveloppe est visible du point M si $\det(\overrightarrow{MP},\overrightarrow{MQ}) < 0$, et non visible si $\det(\overrightarrow{MP},\overrightarrow{MQ}) > 0$. C'est ce qu'illustre la figure 8.11, page 287. Le cas $\det(\overrightarrow{MP},\overrightarrow{MQ}) = 0$ ne peut se présenter que si les trois points sont alignés, ce qui par hypothèse est exclu ici, ou si au moins deux d'entre eux sont confondus. Ce dernier cas est à prendre en compte dans la fonction *TangenteSup* de la question 2.

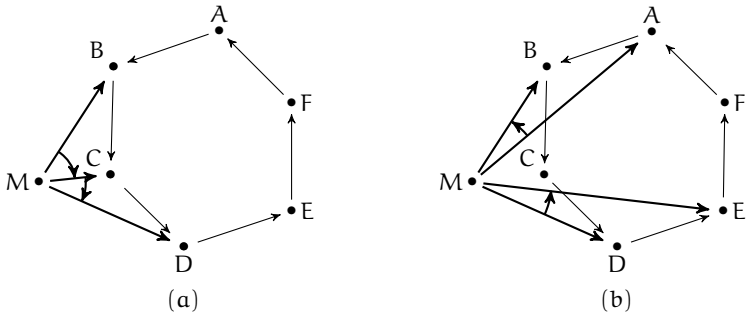


Fig. 8.11 – *Visibilité. Schéma* (a) *:* $\overrightarrow{BC}$ *et* $\overrightarrow{CD}$ *sont visibles depuis* M. $\det(\overrightarrow{MB},\overrightarrow{MC}) < 0$ et $\det(\overrightarrow{MC},\overrightarrow{MD}) < 0$ – *Schéma* (b) *:* $\overrightarrow{AB}$ *et* $\overrightarrow{DE}$ *ne sont pas visibles depuis* M. $\det(\overrightarrow{MA},\overrightarrow{MB}) > 0$ et $\det(\overrightarrow{MD},\overrightarrow{ME}) > 0$

Définition 32 (Pont entre deux enveloppes) :
Soit deux enveloppes G *et* D *situées de part et d'autre d'une droite verticale. Un pont entre* G *et* D *est un segment joignant un sommet de* G *et un sommet de* D*, dont tous les points (à l'exception des extrémités) sont extérieurs à* G *et à* D*.*

La partie (a) de la figure 8.12, page 288, répertorie les dix ponts existant entre les deux enveloppes. Un segment tel que BI n'est pas un pont : il est en partie à l'intérieur de l'enveloppe gauche. Trois ponts (en gras) peuvent en général être distingués : DI, qui rejoint le point le plus à droite de l'enveloppe gauche et le point le plus à gauche de l'enveloppe droite. C'est le seul pont qui puisse être identifié directement à partir de la connaissance des enveloppes. AG (resp. CJ) est un pont particulier – la tangente supérieure (resp. la tangente inférieure) commune aux deux enveloppes – qui se caractérise par le fait qu'aucun point de E n'est *au-dessus* (resp. *au-dessous*) de la droite support du segment.

Ces tangentes revêtent une grande importance : ce sont elles qu'il faut prendre en compte pour, lors de la fusion, obtenir une véritable enveloppe (voir figure 8.9, page 286).

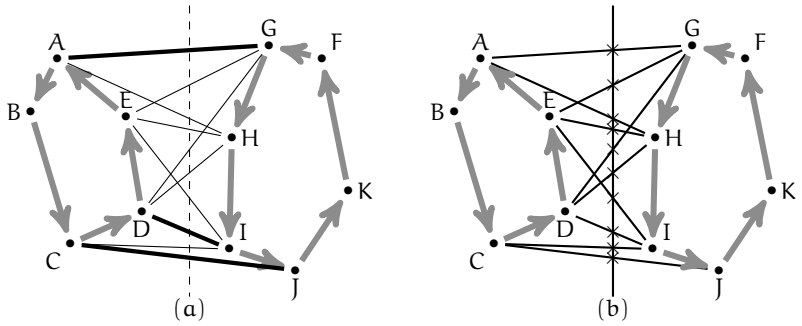


Fig. 8.12 – (a) *Ensemble des ponts entre* $\widehat{D}$ *et* $\widehat{I}$ – (b) *Intersection entre une verticale et les ponts*

La partie (b) de la figure 8.12, page 288, met l'accent sur les points d'intersection entre une verticale séparatrice et les différents ponts. Ces points – en nombre fini – sont utiles dans la suite pour démontrer la terminaison de l'algorithme de fusion.

107 - Q 1 **Question 1.** La recherche des tangentes supérieure et inférieure se fait à partir du seul pont identifiable directement, celui qui lie le sommet le plus à droite de l'enveloppe de gauche et le sommet le plus à gauche de l'enveloppe de droite, en progressant de pont en pont. Les coordonnées de ces deux sommets sont connues suite à la coupure. Mais pour progresser, il faut aussi avoir accès aux nœuds successeurs et prédécesseurs. Il faut donc connaître leurs situations *au sein des enveloppes*. L'opération « **fonction** *Recherche*$(\widehat{e}, v)$ **résultat** EnvConv » prend en compte l'enveloppe $\widehat{e}$ (enveloppe désignée par le point e), le point d'abscisse v, et délivre la même enveloppe mais cette fois désignée par v. Construire cette fonction sur la base d'une recherche séquentielle. Que peut-on dire de sa complexité en nombre de conditions évaluées ?

107 - Q 2 **Question 2.** On cherche à présent à construire une version itérative de l'opération « **fonction** *TangenteSup*$(\widehat{g}, \widehat{d}, i)$ **résultat** EnvConv × EnvConv » qui, à partir de deux enveloppes $\widehat{g}$ et $\widehat{d}$ (séparées verticalement) et de l'indice i désignant dans T le point le plus à gauche de $\widehat{d}$, délivre le couple $(\widehat{L}, \widehat{R})$ tel que le segment LR est la tangente supérieure de $\widehat{g}$ et de $\widehat{d}$. Que peut-on dire de la complexité de cette fonction en nombre de conditions évaluées ?

107 - Q 3 **Question 3.** On suppose disponible l'opération « **fonction** *Coupure*(bi, bs) **résultat** $\mathbb{N}_1$ » telle que si m est le résultat délivré, $T[bi\,..\,m-1]$ et $T[m\,..\,bs]$ sont les deux ensembles de points à partir desquels se font les recherches d'enveloppes gauche et droite. Quel est le modèle de division qui s'applique ? Fournir le code de l'opération « **fonction** *EnveloppeConvexe*(bi, bs) **résultat** EnvConv », qui délivre l'enveloppe convexe des points de $T[bi\,..\,bs]$ en employant la méthode DpR décrite ci-dessus. Que peut-on dire de la complexité de cette solution, en termes de conditions évaluées, par rapport à n ?

107 - Q 4 **Question 4.** Bien que n'intervenant pas sur la complexité, les appels à la fonction *Recherche* peuvent se révéler pénalisants. Fournir le principe d'une solution dans laquelle cette fonction devient inutile.

Exercice 108. La sous-séquence bègue

Deux solutions de type DpR sont développées ici. La principale originalité de cet exercice réside dans la seconde solution, où l'étape d'éclatement est plus ingénieuse que la simple division par 2 pratiquée en général. Le bénéfice en retour est une meilleure efficacité.

Soit un alphabet Σ de cardinal s ($s \geqslant 1$). Soit x une séquence sur Σ de longueur n ($n \geqslant s$) et $y = a_1 \dots a_m$ une séquence sur Σ de longueur m ($m \geqslant s$). Si a^i représente la chaîne $\underbrace{a \dots a}_{i \text{ fois}}$, on note $\overset{i}{y}$ la séquence $a_1^i \dots a_m^i$. Dans la suite, on suppose, sans perte de généralité, que x et y utilisent *tous* les symboles de Σ.

On cherche la valeur la plus grande de i, notée $\mathrm{Maxi}(x, y)$ pour laquelle $\overset{i}{y}$ est une sous-séquence de x. i est aussi appelé *degré de bégaiement* de y dans x. Rappelons que les symboles d'une sous-séquence ne sont pas forcément contigus dans la séquence.

Par exemple, pour $\Sigma = \{a,b,c\}$, $y = abc$ et $x = cbbabaacbbabbcbbacccbac$, on trouve un degré de bégaiement $\mathrm{Maxi}(x, y)$ de 4 ($\overset{4}{y} = aaaabbbbcccc$), puisque $x = cbb\mathbf{a}b\mathbf{aa}cbb\mathbf{abb}c\mathbf{bb}a\mathbf{ccc}bac$ et que $\overset{5}{y}$ n'est pas une sous-séquence de x.

Question 1. Construire une version itérative de l'opération « **fonction** $Scan(x, y, i)$ **résultat** $\mathbb{B}$ » qui retourne **vrai** si et seulement si $\overset{i}{y}$ est une sous-séquence de x. En choisissant l'évaluation de conditions comme opération élémentaire, montrer que sa complexité asymptotique est en $\mathcal{O}(n + m)$. 108 - Q 1

Question 2. Montrer que $\mathrm{Maxi}(x, y) \in 0 \,..\, \lfloor n/m \rfloor$. 108 - Q 2

Question 3. On dispose à présent d'un intervalle fini sur lequel $\mathrm{Maxi}(x, y)$ prend sa valeur. Différentes techniques peuvent être appliquées afin de déterminer cette valeur. La recherche séquentielle en est une. La recherche dichotomique se prête également bien à la résolution de ce problème. C'est la solution à laquelle on s'intéresse dans cette question. Décrire le raisonnement DpR permettant de construire la fonction $Maxi0(x, y, bi, bs)$ qui délivre la valeur de $\mathrm{Maxi}(x, y)$ sur l'intervalle $bi \,..\, bs$ ($bi \,..\, bs \subseteq 0 \,..\, \lfloor n/m \rfloor$). Quel est le modèle de division qui s'applique ? Quel est, en nombre de conditions évaluées, l'ordre de grandeur de la complexité de cette solution ? On pourra limiter les calculs au cas où $\lfloor n/m \rfloor$ est une puissance de 2. 108 - Q 3

Le paradigme DpR, appliqué différemment, permettrait-il d'améliorer le résultat précédent ? Ci-dessus, DpR a été appliqué sur un intervalle d'entiers. Existe-t-il une alternative à ce choix ? On peut penser à appliquer DpR sur la séquence x. Cependant – le lecteur pourra le vérifier – couper x par le milieu est une tentative vaine s'agissant de la recherche d'une meilleure complexité.

Il existe un autre problème, traité ci-après : le calcul de la transformée de Fourier discrète rapide (voir exercice 109, page 291), dans lequel l'éclatement se fait non pas en coupant par le milieu mais en ventilant les éléments selon la parité de leurs indices. Le principe

que l'on applique ici s'apparente à celui-ci, les éléments de x étant ventilés selon la parité des indices *de chaque élément de l'alphabet* Σ. L'exemple suivant illustre ce principe (les symboles sont indicés pour faciliter la lecture) :

$x = c_1\,b_1\,b_2\,a_1\,b_3\,a_2\,a_3\,c_2\,b_4\,b_5\,a_4\,b_6\,b_7\,c_3\,b_8\,b_9\,a_5\,c_4\,c_5\,c_6\,b_{10}\,a_6\,c_7$

$\mathrm{Impair}(x) = c_1\,b_1\,a_1\,b_3\,a_3\,b_5\,b_7\,c_3\,b_9\,a_5\,c_5\,c_7$

$\mathrm{Pair}(x) = b_2\,a_2\,c_2\,b_4\,a_4\,b_6\,b_8\,c_4\,c_6\,b_{10}\,a_6$

Il serait alors possible, en démontrant au préalable que :

$$\mathrm{Maxi}(x,y) \in \left(\begin{array}{l} \mathrm{Maxi}(\mathrm{Impair}(x),y) + \mathrm{Maxi}(\mathrm{Pair}(x),y) - 1 \\ .. \\ \mathrm{Maxi}(\mathrm{Impair}(x),y) + \mathrm{Maxi}(\mathrm{Pair}(x),y) + 1 \end{array}\right)$$

de développer une solution DpR. Celle-ci présenterait cependant l'inconvénient de nécessiter un double appel récursif (sur $\mathrm{Impair}(x)$ et sur $\mathrm{Pair}(x)$) qui, d'après le corollaire du théorème maître et son cas particulier 8.4, page 247, conduirait à une solution en $n{\cdot}\log_2(n)$, comparable à la solution dichotomique précédente du point de vue de la complexité. Dans la suite, on cherche à éviter ce double appel récursif.

108 - Q 4 **Question 4.** Fournir le principe de l'algorithme de la fonction *Impair*(x) et montrer que sa complexité est en $\Theta(n)$ conditions évaluées.

108 - Q 5 **Question 5.** On veut montrer que $\mathrm{Maxi}(x,y)$ varie sur un certain intervalle et que toutes les valeurs de cet intervalle peuvent être atteintes.

On présente au préalable, à travers un exemple, la notion $S(x,y)$ de segmentation de x par rapport à y et de segmentation induite (par $S(x,y)$) de $\mathrm{Impair}(x)$ par rapport à y. On pose $X = \mathrm{Maxi}(x,y)$ et $I = \mathrm{Maxi}(\mathrm{Impair}(x),y)$. Pour $x = a_1b_1a_2a_3a_4a_5a_6b_2c_1c_2a_7c_3c_4c_5c_6c_7$ et $y = bac$, $S(x,y)$ est constitué de trois segments puisqu'il y a trois symboles dans y. On a par exemple :

$$S(x,y) = \sigma_1,\sigma_2,\sigma_3$$
$$x = \| \overbrace{a_1\,\mathbf{b_1}\,a_2\,a_3\,a_4\,a_5\,a_6\,\mathbf{b_2}}^{\sigma_1}{}_{2b} \| \overbrace{c_1\,c_2\,\mathbf{a_7}}^{\sigma_2}{}_{1a} \| \overbrace{\mathbf{c_3\,c_4\,c_5\,c_6\,c_7}}^{\sigma_3}{}_{5c} \|$$
$$y' = \mathit{bbaccccc}, \qquad X = \min(\{2,1,5\})$$

y' est la sous-séquence de x apparaissant en gras. Il n'y a pas unicité de la segmentation. En effet, en reprenant l'exemple ci-dessus, on a également :

$$S(x,y) = \sigma_1,\sigma_2,\sigma_3$$
$$x = \| \overbrace{a_1\,\mathbf{b_1}}^{\sigma_1}{}_{1b} \| \overbrace{\mathbf{a_2\,a_3\,a_4\,a_5\,a_6}\,b_2\,c_1\,c_2\,\mathbf{a_7}}^{\sigma_2}{}_{6a} \| \overbrace{\mathbf{c_3\,c_4\,c_5\,c_6\,c_7}}^{\sigma_3}{}_{5c} \|$$
$$y' = \mathit{baaaaaaccccc}, \qquad X = \min(\{1,6,5\})$$

En revanche, dans les deux cas $X = 1$. Concernant la segmentation induite (par $S(x,y)$)

de Impair(x) par rapport à y, le premier exemple donne :

$$x = \| \overbrace{a_1\, \mathbf{b_1}\, a_2\, a_3\, a_4\, a_5\, a_6\, \mathbf{b_2}}^{\sigma_1} \| \overbrace{c_1\, c_2\, \mathbf{a_7}}^{\sigma_2} \| \overbrace{\mathbf{c_3}\, \mathbf{c_4}\, \mathbf{c_5}\, \mathbf{c_6}\, \mathbf{c_7}}^{\sigma_3} \|$$

$$S(\mathrm{Impair}(x), y) = \sigma'_1, \sigma'_2, \sigma'_3$$

$$\mathrm{Impair}(x) = \| \overbrace{a_1\, \mathbf{b_1}\, a_3\, a_5}^{\sigma'_1} \| \overbrace{c_1\, \mathbf{a_7}}^{\sigma'_2} \| \overbrace{\mathbf{c_3}\, \mathbf{c_5}\, \mathbf{c_7}}^{\sigma'_3} \|$$

La relation qui lie X et I ne dépend pas bien sûr des segmentations choisies. Montrer que, dans le cas où X est pair, $2I = X$, et que dans le cas où X est impair on a soit $2I + 1 = X$, soit $2I - 1 = X$. En déduire que

$$\mathrm{Maxi}(x, y) \in (2 \cdot \mathrm{Maxi}(\mathrm{Impair}(x), y) - 1) \,..\, (2 \cdot \mathrm{Maxi}(\mathrm{Impair}(x), y) + 1)$$

et que les trois valeurs de l'intervalle peuvent être atteintes par Maxi(x, y).

Question 6. Sur la base du principe DpR, construire l'opération « **fonction** *Maxi1*(x, y) **résultat** ℕ » fondée sur les deux questions précédentes. Quel est le modèle de division qui s'applique ? Fournir le code de l'opération *Maxi1*. Démontrer la terminaison de l'algorithme. Que peut-on dire de la complexité de l'algorithme ? Comparer les deux solutions DpR du point de vue de la complexité. 108 - Q 6

Exercice 109. La transformée de Fourier rapide (FFT)

À cet algorithme de transformée de Fourier rapide sont associés bien des superlatifs. Utile, il l'est sans aucun doute au plus haut point. Il suffit pour s'en convaincre d'énumérer quelques-unes de ses applications : reconnaissance de parole, filtrage, analyse de spectre, produit de polynômes, compression de données, etc. Efficace, l'algorithme l'est assurément, à telle enseigne qu'historiquement il faut y rechercher l'une des clés de la prééminence des ordinateurs sur les calculateurs analogiques et indirectement, de la révolution numérique que l'on connaît. Intelligent, fondé certes sur le principe DpR, mais de manière très ingénieuse. Bref, un algorithme remarquable, un excellent exercice. Terminons par une citation de C. Villani (Médaille Fields 2010, dans [60], p. 35) : « l'influence de Joseph Fourier est maintenant bien plus importante que celle de Hugo lui-même ; son "grand poème mathématique" (comme disait lord Kelvin), enseigné dans tous les pays du monde, est utilisé chaque jour par des milliards *d'humains qui ne s'en rendent même pas compte. »*

Définition de la transformée de Fourier discrète

Une *transformée de Fourier discrète* est une transformation linéaire de $\mathbb{C}^n$ dans $\mathbb{C}^n$ définie de la manière suivante. Soit x un vecteur de nombres complexes défini sur l'intervalle

$0 \,..\, n-1$. Pour tout entier k, $k \in 0 \,..\, n-1$, la valeur $X[k]$ de la transformée de Fourier discrète du vecteur x se définit par :

$$X[k] \quad = \quad \sum_{j=0}^{n-1} x[j] \cdot e^{-\frac{2\pi \cdot i}{n} \cdot j \cdot k} \tag{8.13}$$

où e est la base du logarithme naturel et i le nombre complexe tel que $i^2 = -1$.

109 - Q 1 **Question 1.** À partir de cette définition, construire un algorithme qui calcule la transformée de Fourier discrète X d'un vecteur x de n nombres complexes. Quelle est sa complexité en nombre d'exponentiations et en nombre de multiplications ?

Propriétés des racines n^e de l'unité – Rappels

On cherche à présent à améliorer l'efficacité de la solution précédente par une approche DpR. Pour ce faire, on va exploiter les propriétés des racines n^e complexes de l'unité. Dans la suite, par hypothèse, n est toujours une puissance de 2. Une racine n^e complexe de l'unité est un nombre complexe w_n tel que $w_n^n = 1$. Il existe n racines n^e de l'unité qui sont, pour $k \in 0 \,..\, n-1$: $e^{-\frac{2\pi \cdot i}{n} \cdot k}$. La racine particulière obtenue pour $k = 1$, $e^{-\frac{2\pi \cdot i}{n}}$ est appelée racine principale (ou n-racine principale s'il est nécessaire de préciser). Elle est notée W_n. Les n racines de l'unité sont des puissances de W_n : $W_n^0, W_n^1, W_n^2, \ldots, W_n^{n-1}$.

La figure 8.13 représente, dans le plan complexe, les racines n^e de l'unité pour $n = 8$. Ces racines sont situées à des positions équidistantes sur le cercle complexe unité.

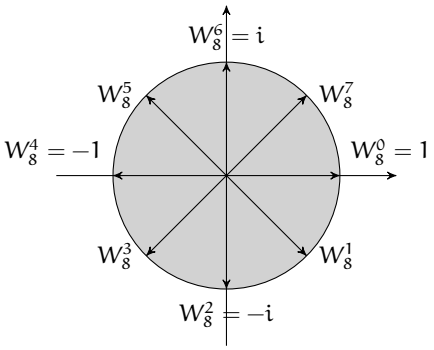


Fig. 8.13 – *Racines n^e de l'unité dans le plan complexe, pour $n = 8$*

Propriétés (non démontrées)

1. Pour tout $n > 0$ pair, la puissance $(n/2)^e$ de toute racine n^e de l'unité est égale à -1 :
$$w_n^{\frac{n}{2}} = -1 \quad (n > 0 \text{ pair}). \tag{8.14}$$
2. Pour tout $n > 0$ pair et pour tout $k \geqslant 0$, le carré de la n-racine principale élevée à la puissance k est égal à la $(n/2)$-racine principale élevée à la puissance k :
$$(W_n^k)^2 = W_{\frac{n}{2}}^k \quad (\text{pour } n > 0 \text{ pair}). \tag{8.15}$$

Transformée de Fourier discrète rapide (DFT)

Le principe du calcul rapide de la Transformée de Fourier Discrète par l'algorithme *DFT* (Discrete Fourier Transform) s'appuie sur les propriétés des carrés des racines de l'unité. L'application la plus directe du principe DpR consisterait à couper le vecteur x en son milieu :

$$\begin{aligned} & X[k] \\ = & \qquad \text{définition 8.13 et définition de } W_n \\ & \sum_{j=0}^{n-1} x[j] \cdot W_n^{j \cdot k} \\ = & \qquad \text{coupure de } x \text{ par le milieu} \\ & \sum_{j=0}^{\frac{n}{2}-1} x[j] \cdot W_n^{j \cdot k} + \sum_{j=\frac{n}{2}}^{n-1} x[j] \cdot W_n^{j \cdot k}. \end{aligned}$$

Cependant (le lecteur est invité à le vérifier), cette démarche conduit à une impasse pour ce qui concerne l'amélioration de la complexité. Il faut rechercher une autre stratégie de coupure. Une solution consiste à placer d'un côté les éléments d'indices pairs de x et de l'autre ceux d'indices impairs (utilisé en indice, i signifie *impair* et ne doit pas être confondu avec l'unité imaginaire également notée i). Pour ce faire, on pose :

$$x_p = \left[x[0], \quad x[2], \quad \cdots, \quad x[2p], \quad \cdots, \quad x[n-2]\right], \tag{8.16}$$

$$x_i = \left[x[1], \quad x[3], \quad \cdots, \quad x[2p+1], \quad \cdots, \quad x[n-1]\right]. \tag{8.17}$$

Pour $k \in 0..n/2-1$, nous avons donc $x_p[k] = x[2k]$ et $x_i[k] = x[2k+1]$. La transformée de Fourier X_p de x_p (pour $k \in 0..n/2-1$) donne lieu au calcul suivant :

$$\begin{aligned} & X_p[k] \\ = & \qquad \text{définition 8.13} \\ & \sum_{j=0}^{\frac{n}{2}-1} x_p[j] \cdot e^{-\frac{2\pi i}{\frac{n}{2}} \cdot j \cdot k} \\ = & \qquad \text{définition de } W_n \ (W_n = e^{-\frac{2\pi i}{n}}) \text{ et substitution} \\ & \sum_{j=0}^{\frac{n}{2}-1} x_p[j] \cdot W_{\frac{n}{2}}^{j \cdot k}. \end{aligned} \tag{8.18}$$

Pour les indices impairs et toujours pour $k \in 0..n/2-1$, nous avons une formule similaire :

$$X_i[k] = \sum_{j=0}^{\frac{n}{2}-1} x_i[j] \cdot W_{\frac{n}{2}}^{j \cdot k}. \tag{8.19}$$

La définition de la transformée de Fourier pour $k \in 0..n-1$ donne lieu au développement suivant :

$$\begin{aligned} & X[k] \\ = & \qquad \text{définition 8.13} \\ & \sum_{j=0}^{n-1} x[j] \cdot e^{-\frac{2\pi i}{n} \cdot j \cdot k} \end{aligned}$$

$$
\begin{aligned}
&= && \text{définition de } W_n \ (W_n = e^{-\frac{2\pi i}{n}}) \\
&\sum_{j=0}^{n-1} x[j] \cdot W_n^{j\cdot k} \\
&= && \text{séparation entre indices pairs et impairs} \\
&\sum_{j=0}^{\frac{n}{2}-1} x[2j] \cdot W_n^{2j\cdot k} + \sum_{j=0}^{\frac{n}{2}-1} x[2j+1] \cdot W_n^{(2j+1)\cdot k} \\
&= && \text{définitions 8.16 et 8.17} \\
&\sum_{j=0}^{\frac{n}{2}-1} x_p[j] \cdot W_n^{2j\cdot k} + \sum_{j=0}^{\frac{n}{2}-1} x_i[j] \cdot W_n^{(2j+1)\cdot k} \\
&= && \text{calculs sur les indices de } W_n \text{ et factorisation} \\
&\sum_{j=0}^{\frac{n}{2}-1} x_p[j] \cdot (W_n^{j\cdot k})^2 + W_n^k \cdot \sum_{j=0}^{\frac{n}{2}-1} x_i[j] \cdot (W_n^{j\cdot k})^2 \\
&= && \text{propriété 8.15} \\
&\sum_{j=0}^{\frac{n}{2}-1} x_p[j] \cdot W_{\frac{n}{2}}^{j\cdot k} + W_n^k \cdot \sum_{j=0}^{\frac{n}{2}-1} x_i[j] \cdot W_{\frac{n}{2}}^{j\cdot k}. && (8.20)
\end{aligned}
$$

La formule 8.20 est définie pour $k \in 0..n-1$. Elle est *a fortiori* valable pour $k \in 0..n/2-1$. Nous pouvons alors poursuivre le développement en nous limitant à ce dernier intervalle et en faisant intervenir les formules 8.18 et 8.19, page 293 :

$$
\begin{aligned}
&\sum_{j=0}^{\frac{n}{2}-1} x_p[j] \cdot W_{\frac{n}{2}}^{j\cdot k} + W_n^k \cdot \sum_{j=0}^{\frac{n}{2}-1} x_i[j] \cdot W_{\frac{n}{2}}^{j\cdot k} \\
&= && \text{formules 8.18 et 8.19, page 293, pour } k \in 0 \,..\, \frac{n}{2} - 1 \\
&X_p[k] + W_n^k \cdot X_i[k],
\end{aligned}
$$

ce qui définit par DpR les $n/2$ premiers éléments du vecteur X. Il faut à présent compléter le calcul sur l'intervalle $n/2\,..\,n-1$ afin d'obtenir les $n/2$ derniers éléments de X sous une forme analogue En repartant de la formule 8.20, pour $k \in 0\,..\,n/2-1$, nous avons :

$$
\begin{aligned}
&X\left[k+\frac{n}{2}\right] \\
&= && \text{expression 8.20 pour la substitution } k \leftarrow k+\frac{n}{2} \\
&\sum_{j=0}^{\frac{n}{2}-1} x_p[j] \cdot W_n^{2(k+\frac{n}{2})\cdot j} + W_n^{k+\frac{n}{2}} \cdot \sum_{j=0}^{\frac{n}{2}-1} x_i[j] \cdot W_n^{2(k+\frac{n}{2})\cdot j} \\
&= && \text{propriété 8.14 : } W_n^{k+\frac{n}{2}} = W_n^k \cdot W_n^{\frac{n}{2}} = -W_n^k \\
&\sum_{j=0}^{\frac{n}{2}-1} x_p[j] \cdot W_n^{2(k+\frac{n}{2})\cdot j} - W_n^k \cdot \sum_{j=0}^{\frac{n}{2}-1} x_i[j] \cdot W_n^{2(k+\frac{n}{2})\cdot j} \\
&= && \text{définition de } W_n^n : W_n^{2(k+\frac{n}{2})\cdot j} = (W_n^n)^j \cdot W_n^{2k\cdot j} = 1 \cdot W_n^{2k\cdot j} = (W_n^{k\cdot j})^2 \\
&\sum_{j=0}^{\frac{n}{2}-1} x_p[j] \cdot (W_n^{k\cdot j})^2 - W_n^k \cdot \sum_{j=0}^{\frac{n}{2}-1} x_i[j] \cdot (W_n^{k\cdot j})^2 \\
&= && \text{propriété 8.15}
\end{aligned}
$$

$$= \begin{array}{l} \sum_{j=0}^{\frac{n}{2}-1} x_p[j] \cdot W_{\frac{n}{2}}^{k\cdot j} - W_n^k \cdot \sum_{j=0}^{\frac{n}{2}-1} x_i[j] \cdot W_{\frac{n}{2}}^{k\cdot j} \\ \qquad\qquad \text{formules 8.18 et 8.19, page 293, pour } k \in 0\,..\,\frac{n}{2}-1 \\ X_p[k] - W_n^k \cdot X_i[k]. \end{array}$$

Pour résumer, on a montré que, dans l'hypothèse (d'induction) où l'on sait calculer $X_p[k]$ et $X_i[k]$ à partir des deux demi-vecteurs x_p et x_i, on sait composer les deux résultats partiels pour obtenir la transformée de Fourier X de x, en appliquant les deux formules :

$$\begin{aligned} X[k] &= X_p[k] + W_n^k \cdot X_i[k] \\ X\left[k+\frac{n}{2}\right] &= X_p[k] - W_n^k \cdot X_i[k] \end{aligned}$$

et ceci pour $k \in 0\,..\,n/2-1$.

Question 2. On suppose disponible la fonction $Pair(m, A)$ (resp. $Impair(m, A)$) qui, pour un vecteur A de longueur m paire, délivre le sous-vecteur de A contenant les éléments d'indices pairs (resp. d'indices impairs). Fournir le modèle de division qui s'applique pour le calcul de la DFT par DpR. En déduire la fonction $DFT(n, x)$ permettant de calculer la DFT X d'un vecteur x, $x \in (0\,..\,n-1) \to \mathbb{C}$, pour n puissance de 2. En supposant que les opérations élémentaires sont la multiplication (de nombres) et l'exponentiation, quelle est la complexité de cette opération ? Conclusion ? 109 - Q 2

Exercice 110. Le produit de polynômes

Cet exercice montre que l'application du principe DpR ne conduit pas systématiquement à une solution plus efficace qu'une solution itérative. Mais son principal intérêt réside dans une application remarquable de la transformée de Fourier rapide (voir exercice 109, page 291).

On cherche à multiplier deux polynômes sur $\mathbb{Z}$: $A(x) = a_{n-1} \cdot x^{n-1} + a_{n-2} \cdot x^{n-1} + \cdots + a_1 \cdot x + a_0$ et $B(x) = b_{m-1} \cdot x^{m-1} + b_{m-2} \cdot x^{m-2} + \cdots + b_1 \cdot x + b_0$. Ainsi par exemple, en s'inspirant de la présentation traditionnelle pour la multiplication d'entiers, on a :

$$\begin{array}{rrrrrrrrrrrrrrr} & & & & & & & 2x^4 & - & 3x^3 & + & 4x & + & 5 \\ & & & & & & \times & & & & & 2x^2 & + & 3 \\ \hline & & & & & 6x^4 & - & 9x^3 & + & 0x^2 & + & 12x & + & 15 \\ + & & & & & & & & & & & & & \\ & 4x^6 & - & 6x^5 & + & 0x^4 & + & 8x^3 & + & 10x^2 & & & & \\ \hline & 4x^6 & - & 6x^5 & + & 6x^4 & - & x^3 & + & 10x^2 & + & 12x & + & 15 \end{array}$$

Plus généralement, si $A(x)$ se définit par $\sum_{k=0}^{n-1} a_k \cdot x^k$ et $B(x)$ par $\sum_{k=0}^{m-1} b_k \cdot x^k$, le polynôme $C(x)$, produit de $A(x)$ et de $B(x)$, se définit par :

$$C(x) = \sum_{k=0}^{m+n-2} c_k \cdot x^k \quad \text{avec} \quad c_k = \sum_{\substack{i+j=k \\ i\in 0..n-1 \\ j\in 0..m-1}} a_i \cdot b_j. \tag{8.21}$$

Une solution naïve pour calculer les coefficients c_k de $C(x)$ consiste à utiliser les formules quantifiées ci-dessus. Si l'opération élémentaire pour mesurer la complexité est la multiplication des coefficients a_i et b_j, le coût de cette méthode est en $\Theta(n \cdot m)$. Peut-on améliorer cette situation ? C'est l'objectif que l'on se fixe, en utilisant pour ce faire la technique DpR. Dans la suite, on appelle *taille* d'un polynôme le nombre de monômes qu'il contient.

110 - Q 1 **Question 1.** Dans cette question, on suppose d'une part que tous les coefficients sont différents de 0 et d'autre part que $n = m = 2^k$ ($k \geqslant 0$). Comment le principe DpR se décline-t-il pour aboutir au calcul de $C(x)$? Quels sont les critères à retenir pour obtenir une « bonne » représentation d'un polynôme ? En déduire le modèle de division qui s'applique. Fournir le code sous la forme de l'opération « **fonction** *MultPolyn4*(A, B) **résultat** polynôme ». Quelle est alors la complexité de cette solution en termes de multiplications (de nombres) ? Conclusion ?

110 - Q 2 **Question 2.** À partir de la solution précédente, en conservant les mêmes hypothèses et en utilisant l'identité :

$$a \cdot d \ + \ b \cdot c \ \ = \ \ (a+b)(c+d) - (a \cdot c) - (b \cdot d),$$

trouver une variante qui réduise la complexité.

110 - Q 3 **Question 3.** Que faire quand les coefficients peuvent être nuls et quand n et m sont quelconques ? L'efficacité de la variante précédente est-elle conservée ?

110 - Q 4 **Question 4.** (Pour aborder cette question, il est nécessaire d'avoir au préalable répondu à l'exercice 109, page 291, sur la transformée de Fourier.) Une solution plus efficace existe. Elle se présente comme une application directe de la transformée de Fourier. Elle est fondée sur un paradigme scientifique classique consistant à effectuer un changement d'espace de représentation afin de simplifier les traitements[10]. Son principe se base sur l'existence de deux types de représentation pour les polynômes – la représentation traditionnelle par coefficients (utilisée ci-dessus) et la représentation par échantillons – et à effectuer le produit dans la représentation la plus efficace (la représentation par échantillons), précédé et suivi des conversions nécessaires. La représentation par coefficients considère le vecteur des coefficients. La représentation par échantillons consiste quant à elle, pour un polynôme $P(x)$ de degré inférieur ou égal à $(n-1)$, à évaluer $P(x)$ sur (au moins) n abscisses x_i différentes.

Exemple Considérons les deux polynômes $D(x) = 2x + 1$ et $E(x) = x + 2$, leurs représentations par coefficients sont $[2, 1]$ pour $D(x)$ et $[1, 2]$ pour $E(x)$. La représentation par échantillons passe tout d'abord par le choix des n abscisses différentes. Prenons $x_0 = 0$ et $x_1 = 1$. $D(x)$ est alors représenté par $[(0, D(0)), (1, D(1))]$, soit encore $[(0, 1), (1, 3)]$, tandis que $E(x)$ est représenté par $[(0, 2), (1, 3)]$.

10. Un exemple classique est celui de la multiplication de nombres effectuée par l'addition de leurs logarithmes.

Cette représentation possède l'avantage de faciliter certaines opérations, c'est notamment le cas du produit, puisqu'il suffit alors de multiplier les différentes valeurs prises par les polynômes sur tous les échantillons. Le calcul du produit exige cependant que, dans le cas de la représentation par échantillons, les abscisses d'échantillonnage soient identiques pour les deux polynômes à multiplier. Une difficulté ne doit pas nous échapper. Le produit de deux polynômes de degré $(n-1)$ est un polynôme de degré $(2n-2)$. Il est donc nécessaire de disposer de $(2n-1)$ échantillons pour chacun des deux polynômes à multiplier.
Pour l'exemple ci-dessus, on convient de compléter l'échantillonnage de $D(x)$ et de $E(x)$ sur l'abscisse 2. On a alors :

$$D(x) = [(0,1),(1,3),(2,5)] \quad \text{et} \quad E(x) = [(0,2),(1,3),(2,4)]$$

Le résultat $R(x) = D(x) \cdot E(x)$ est obtenu en multipliant les ordonnées respectives, soit :

$$R(x) = [(0,1\cdot 2),(1,3\cdot 3),(2,5\cdot 4)]$$

On est donc face au problème suivant. Étant donnés deux polynômes représentés par leurs coefficients, la meilleure solution que l'on connaît pour obtenir leur produit est en $\Theta(n^{\log_2(3)})$ (voir par exemple [53]). En revanche, avec une représentation par échantillons, la solution est en $\Theta(n)$. Est-il possible de faire mieux que $\Theta(n^{\log_2(3)})$, en réalisant un changement de représentation ? Si c'est le cas, le schéma du traitement se présente comme le montre la figure 8.14.

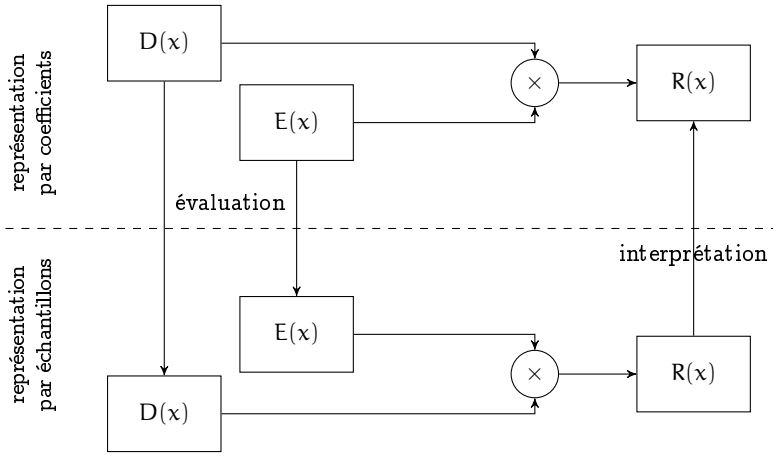


Fig. 8.14 – *Produit de polynômes par changement de représentation. La partie supérieure du schéma illustre le produit classique, totalement effectué dans la représentation par coefficients. La partie inférieure est son homologue pour la représentation par échantillons. La partie gauche montre le changement de représentation des deux arguments (l'évaluation), et la partie droite schématise le changement de représentation du résultat (l'interprétation).*

La conversion entre la représentation par coefficients et la représentation par échantillons est, du point de vue algorithmique, aisée (en utilisant par exemple le schéma de Horner). En revanche, son coût est élevé (de l'ordre de $\Theta(n^2)$). La conversion inverse, entre la représentation par échantillons et la représentation par coefficients, revient à résoudre un

système d'équations linéaire.
Ainsi, pour l'exemple ci-dessus, pour retrouver les trois coefficients a, b et c de $R(x) = a \cdot x^2 + b \cdot x + c$, il faut résoudre le système linéaire suivant :

$$\begin{bmatrix} 0^2 & 0 & 1 \\ 1^2 & 1 & 1 \\ 2^2 & 2 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 2 \\ 9 \\ 20 \end{bmatrix}$$

dont la solution est $a = 2, b = 5$ et $c = 2$. Cependant, là aussi la complexité est élevée (de l'ordre de $\Theta(n^2)$). En apparence, on se trouve face à une voie sans issue. Pourtant ...

Une porte de sortie : la transformée de Fourier On a vu que le choix des abscisses d'échantillonnage est arbitraire. Il est en particulier possible d'évaluer chacun des deux polynômes à multiplier sur les racines $2n^e$ complexes de l'unité. C'est justement ce que réalise la transformée de Fourier, de manière efficace si l'on utilise l'algorithme DFT traité à l'exercice 109, page 291, (complexité de l'orde de $\Theta(n \cdot \log_2(n))$). Pour ce qui est de la transformation réciproque, l'interpolation, il faut alors utiliser la DFT inverse, qui calcule x à partir de X. Celle-ci se définit, pour n points, par :

$$x[k] \quad = \quad \frac{1}{n} \cdot \sum_{j=0}^{n-1} X[j] \cdot e^{\frac{2\pi \cdot i}{n} \cdot j \cdot k}$$

où e la base du logarithme naturel et i le nombre complexe tel que $i^2 = -1$. L'algorithme correspondant est aussi en $\Theta(n \cdot \log_2(n))$.
En résumé, on a décrit une solution au problème du produit de polynômes qui est en $\Theta(n \cdot \log_2(n))$ multiplications (de nombres). Elle se présente en trois principaux points :

a) phase d'évaluation : conversion des deux polynômes de la représentation par coefficients à la représentation par échantillons (complexité en $\Theta(n \cdot \log_2(n))$),

b) phase de multiplication (complexité en $\Theta(n)$),

c) phase d'interpolation : conversion du résultat en sa représentation par coefficients (complexité en $\Theta(n \cdot \log_2(n))$).

C'est ce que présente le schéma de la figure 8.14, page 297. Le travail demandé dans cette question est de mettre en œuvre cette solution sous la forme de la fonction *ProdPolynDFT*(n, P, Q), sachant que les deux polynômes P et Q sont représentés par leurs coefficients et qu'ils ont la même taille n, n étant une puissance de 2.

Exercice 111. Loi de Coulomb ◦ ⁝

Strictement parlant, cet exercice ne comporte aucune question de type DpR. Cependant, le résoudre complètement exige d'exploiter les réponses fournies aux questions de l'exercice 110, page 295, (sur la multiplication de polynômes) et indirectement celles de l'exercice sur le calcul de la transformée de Fourier (exer-

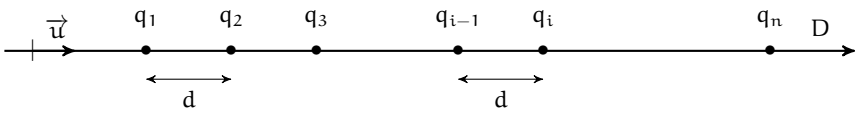
cice 109, page 291). Ces deux exercices doivent donc être traités avant d'aborder celui-ci.

En électrostatique, la loi de Coulomb exprime la force électrique $F_{1 \to 2}$ exercée par une charge électrique q_1 placée en un point M_1 sur une charge q_2 placée en un point M_2. Cette loi s'exprime sous forme vectorielle par la formule suivante :

$$\overrightarrow{F}_{1 \to 2} = \frac{q_1 \cdot q_2}{4\pi\epsilon_0 \|\overrightarrow{r}_{12}\|^2} \cdot \overrightarrow{u},$$

où $\overrightarrow{u}$ est le vecteur unité de la droite D, support des deux charges, et $\overrightarrow{r}_{12} = \overrightarrow{M_1M_2}$ est le vecteur qui relie le premier corps au deuxième. ϵ_0 est une constante.

Considérons à présent un ensemble de n charges $\{q_1, q_2, \ldots, q_n\}$ disposées à intervalles réguliers (de longueur d) sur la droite D :



Notons $\overrightarrow{F}_{\bullet i}$ la somme des forces exercées par les $(n-1)$ autres charges sur la charge q_i.

Question 1. Montrer que l'on a la relation suivante : 111 - Q 1

$$\|\overrightarrow{F}_{\bullet i}\| = |C \cdot q_i| \cdot \left| \sum_{j=1}^{i-1} \frac{q_j}{(i-j)^2} - \sum_{j=i+1}^{n} \frac{q_j}{(i-j)^2} \right|$$

où C est une constante.

Question 2. Montrer que l'on peut calculer $\|\overrightarrow{F}_{\bullet i}\|$ (pour $i \in 1\,..\,n$) en $\Theta(n \cdot \log_2(n))$ multiplications. *Suggestion :* s'inspirer de la question 4 de l'exercice 110 page 295, sur le produit de polynômes. 111 - Q 2

Exercice 112. Lâchers d'œufs par la fenêtre

Dans la plupart des algorithmes DpR, la division se fait en s sous-problèmes (approximativement) de même taille, où s est un nombre fixé à l'avance (typiquement 2). C'est là que réside l'intérêt de cet exercice puisque dans la deuxième partie (la radixchotomie), s dépend de la taille n du problème. Dans la troisième partie (la méthode triangulaire), s dépend toujours de n et de plus les sous-problèmes sont de tailles variables. L'optimisation de tests destructeurs d'échantillons constitue une application possible des algorithmes développés ici. L'exercice 128, page 347, envisage une variante de ce problème sous l'angle de la « programmation dynamique ».

Quand on laisse tomber un œuf par la fenêtre d'un immeuble, il peut se casser ou non : cela dépend de la hauteur de la chute. On cherche à connaître la résistance des œufs, c'est-à-dire la hauteur, exprimée en nombre f d'étages, à partir de laquelle un œuf se casse si on le laisse tomber par la fenêtre. Il est entendu que tous les œufs sont identiques et qu'un œuf se casse toujours s'il tombe d'un étage de rang supérieur ou égal à f et jamais s'il tombe d'un étage de rang inférieur à f.

Quand un œuf tombe sans se casser, on peut le ramasser et le réutiliser. S'il est cassé, on ne peut plus s'en servir. Les étages sont numérotés à partir de 1. Étant donné un immeuble de n ($n \geqslant 1$) étages et un certain nombre k ($k \geqslant 1$) d'œufs, on cherche à trouver f. Si le dernier étage n'est pas assez haut pour briser cette sorte d'œuf, le résultat attendu est $(n+1)$. L'objectif est de minimiser le nombre de lâchers pour n et k donnés.

On suppose disponible la fonction $Casse(h)$ qui laisse tomber un œuf du h^e étage et délivre la valeur **vrai** si l'œuf se casse et **faux** sinon. Si l'œuf se casse, le quota d'œufs disponibles décroît de 1, sinon il reste inchangé. Cette fonction a comme pré-condition qu'il reste encore au moins un œuf disponible et que $h \in 1\,..\,n$.

Une première technique : la recherche séquentielle

112 - Q 1

Question 1. On ne dispose que d'un œuf ($k = 1$). Donner le principe de l'opération « **fonction** *Œuf1*(bi, bs) **résultat** $\mathbb{N}_1$ » qui délivre le résultat pour la section de l'immeuble comprise entre les étages bi et bs et tel que l'appel **fonction** *Œuf1*$(1, n)$ fournit le résultat attendu pour tout l'immeuble. En déduire que la complexité au pire $S_1(n)$ (pour séquentiel avec un œuf), exprimée en nombre de lâchers, est égale à n.

Une deuxième technique : la radixchotomie

112 - Q 2

Question 2. On prend maintenant $k = 2$. Ce choix vise à améliorer la complexité. Si on veut être sûr de conclure, il faut conserver un œuf pour (en général) terminer par une recherche séquentielle. Avec le premier œuf, une stratégie possible consiste à diviser le nombre d'étages en s segments de longueur e de sorte que le nombre de lâchers associé à cette division ajouté à celui de la recherche séquentielle soit minimal dans le pire des cas. Il existe en général un segment résiduel de r étages. Ces trois valeurs entières sont liées par la relation :

$$s \cdot e + r \;=\; n \text{ et } r \in 0\,..\,e-1$$

qui n'est autre que la définition de la division euclidienne de n par e. Pour e donné, on a donc $s = \lfloor n/e \rfloor$ et $r = n - e \cdot \lfloor n/e \rfloor$. Du point de vue algorithmique, la première phase de la recherche est une recherche séquentielle qui s'effectue avec un pas de e, tandis que la seconde phase est une recherche séquentielle (avec un pas de 1) similaire à celle effectuée dans la première question, ainsi que le montre la figure 8.15.

La pire des situations (en nombre de lâchers) est atteinte pour $f = s \cdot e$ ou $f = s \cdot e - 1$: la première phase s'arrête à la position $s \cdot e$, tandis que la seconde phase explore le dernier segment complet jusqu'à la position $s \cdot e - 1$. Le cas de la recherche dans le segment résiduel n'est jamais pire que la recherche pour $f = s \cdot e - 1$ ou $f = s \cdot e$; on peut donc, dans le calcul qui suit, considérer que n est un multiple de e. Dans ce cas, on effectue au pire $R_2(n)$ (pour radixchotomie avec deux œufs) lâchers. $R_2(n)$ est tel que :

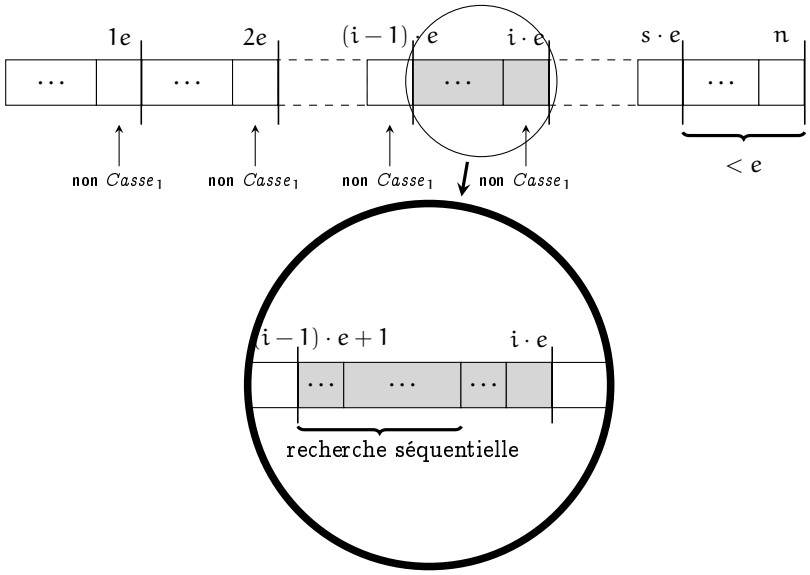


Fig. 8.15 – *La radixchotomie : les deux étapes de la recherche dans le cas ou l'on dispose de deux œufs*

$$R_2(n) = \frac{n}{e} + (e-1).$$

(s lâchers pour la première étape (soit n/e lâchers) et $e-1$ lâchers pour la seconde)

Il faut à présent déterminer une valeur acceptable pour e. Transformons temporairement le problème de la manière suivante : soit $g(e) = (e-1) + n/e$ une fonction réelle d'une variable réelle. On recherche une solution qui minimise $g(e)$. On s'impose en outre la contrainte que e est de la forme n^x. Par conséquent, les segments ont la même longueur. On a donc $g(e) = f(x) = (n^x - 1) + n/n^x$. La dérivée $f'(x) = n^x \cdot \ln(n) - n^{1-x} \cdot \ln(n)$ s'annule pour $x = 0.5$, qui constitue le résultat recherché. Puisque, dans la réalité, on recherche une solution *entière* pour e, on peut prendre $e = \lfloor \sqrt{n} \rfloor$. On a donc [11]

$$R_2(n) = \lfloor \sqrt{n} \rfloor - 1 + \frac{n}{\lfloor \sqrt{n} \rfloor}.$$

Montrons que $R_2(n) \in \Theta(\sqrt{(n)})$.

$$\sqrt{n} \in \Theta(\sqrt{n}) \text{ et pour } n \geqslant 10 \quad \frac{1}{2} \cdot \frac{n}{\sqrt{n}} \leqslant \left\lfloor \frac{n}{\lfloor \sqrt{n} \rfloor} \right\rfloor \leqslant 2 \cdot \frac{n}{\sqrt{n}}$$

$\Rightarrow$ $\lfloor x \rfloor \in \Theta(x)$ et définition de Θ

$$\lfloor \sqrt{n} \rfloor \in \Theta(\sqrt{n}) \text{ et } \lfloor \frac{n}{\lfloor \sqrt{n} \rfloor} \rfloor \in \Theta\left(\frac{n}{\sqrt{n}}\right)$$

$\Rightarrow$ calcul

11. D'où le terme (est-ce un néologisme ?) de *radixchotomie* : la division d'une donnée de taille n se fait en $\sqrt{n}$ parties, de tailles $\sqrt{n}$ ou presque.

$$\left(\lfloor\sqrt{n}\rfloor - 1 \in \Theta(\sqrt{n})\right) \text{ et } \left(\frac{n}{\lfloor\sqrt{n}\rfloor} \in \Theta\left(\sqrt{n}\right)\right)$$

$\Rightarrow$ règle de l'addition

$$\lfloor\sqrt{n}\rfloor - 1 + \left\lfloor\frac{n}{\lfloor\sqrt{n}\rfloor}\right\rfloor \in \Theta\left(\max\left(\{\sqrt{n}, \sqrt{n}\}\right)\right)$$

$\Rightarrow$ définition de R_2 et de max

$R_2(n) \in \Theta\left(\sqrt{n}\right)$.

Construire, sur la base de la démarche ci-dessus et en utilisant *Œuf1*, la fonction *Œuf2Radix* qui implante un algorithme itératif en $\mathcal{O}(\sqrt{n})$ lâchers pour calculer f. Donner son déroulement pour le jeu d'essai suivant : $n = 34$, $f = 29$.

112 - Q 3 **Question** 3. On généralise la question précédente au cas où l'on dispose initialement de k œufs. Construire l'opération « **fonction** *ŒufkRadix*(bi, bs) **résultat** $\mathbb{N}_1$ » qui est telle que *ŒufkRadix*$(1, n)$ calcule f avec une complexité en $\mathcal{O}(\sqrt[k]{n})$ pour un immeuble de n étages. Montrer que sa complexité est bien celle attendue. Suggestion : prendre un pas e de $\lfloor\sqrt[k]{n^{k-1}}\rfloor$.

Une troisième technique : la méthode triangulaire

On se place pour l'instant dans l'hypothèse ou l'on dispose de deux œufs ($k = 2$). Dans la méthode précédente, on s'est appuyé sur l'hypothèse d'un découpage de l'immeuble en segments de longueur uniforme. Il est parfois possible d'obtenir une meilleure solution dans les cas les pires en réalisant un découpage en segments de longueurs décroissantes. Prenons l'exemple d'un immeuble de 36 étages. Si l'on utilise la radixchotomie et que l'étage f recherché est le 36^e, on réalise 11 lâchers. Avec un découpage en segments consécutifs de $8, 7, 6, 5, 4, 3, 2$ et 1 étages on effectuera des lâchers aux étages $8, 15, 21, 26, 30, 33, 35$ et enfin 36, soit un total de huit lâchers. On note également que, quel que soit le segment retenu, si l'étage recherché est le dernier du segment, il faut exactement huit lâchers à chaque fois. Cette propriété est due au fait que 36 est le 8^e nombre triangulaire. Un nombre triangulaire t_i est un nombre de la forme $\sum_{k=1}^{i} k$, où i est appelé le *germe* du nombre t_i.

112 - Q 4 **Question** 4. Comment peut-on procéder si le nombre d'étages n'est pas un nombre triangulaire ? Développer l'exemple avec $n = 29$ et $f = 29$.

112 - Q 5 **Question** 5. Soit $T_2(n)$ (pour triangulaire avec deux œufs) la complexité au pire en nombre de lâchers de cette méthode. Montrer par récurrence sur n et i que $t_{i-1} < n \leqslant t_i \Rightarrow T_2(n) = i$. Autrement dit, si t_i est le nombre triangulaire le plus proche de n supérieurement, alors la stratégie décrite ci-dessus exige au pire exactement i lâchers (dans la suite, par abus de langage, i est aussi appelé le *germe* de n). En déduire que cette méthode est en $\mathcal{O}(\sqrt{n})$ lâchers.

112 - Q 6 **Question** 6. Dans la suite, on généralise la méthode à k quelconque ($k \geqslant 2$). La mise en œuvre de l'algorithme DpR correspondant requiert la disponibilité de l'opération « **fonction** *Germe*(ν) **résultat** $\mathbb{N}_1$ » qui, pour un entier naturel positif ν, délivre le germe de ν. Fournir le principe d'une solution pour cette opération en $\Theta(1)$ lâchers.

112 - Q 7 **Question** 7. De même qu'avec la radixchotomie, dès qu'un segment est identifié, on peut entamer une nouvelle phase de recherche sur le même principe, ou, s'il ne reste qu'un seul œuf, par une recherche séquentielle. L'opération « **fonction** *ŒufkTriangle*(bi, bs) **résultat** $\mathbb{N}_1$ » est telle que l'expression *ŒufkTriangle*$(1, n)$ délivre

le résultat recherché. Préciser le développement inductif sur lequel se fonde cette opération. En déduire le modèle de division qui s'applique ainsi que le code de la fonction *ŒufkTriangle*. Fournir son équation de complexité au pire, en nombre de lâchers.

Question 8. En utilisant le langage de programmation de votre choix, comparez expérimentalement les temps d'exécution des deux méthodes pour $n \in 1..500$, pour $f \in 1..n+1$ et pour $k < \lceil \log_2(n) \rceil$ (si $k \geqslant \lceil \log_2(n) \rceil$, une recherche dichotomique est possible et l'emporte sur les méthodes étudiées ci-dessus). 112 - Q 8

Remarque Lorsque $k < \lceil \log_2(n) \rceil$, une solution alternative consiste à débuter par une recherche dichotomique pour terminer, lorsqu'il ne reste plus qu'un seul œuf, par une recherche séquentielle. On montre que cette solution est en $\mathcal{O}(k + n/2^{k-1})$ lâchers.

Exercice 113. Recherche d'un doublon dans un sac

La principale difficulté, mais aussi le principal intérêt de l'exercice, résident dans la compréhension et la mise en œuvre de l'étape de séparation en sous-problèmes. Celle-ci est fondée sur une heuristique qui contribue au bon comportement de l'algorithme en termes de complexité. La boucle qui lui correspond doit être construite avec beaucoup de rigueur. Le raisonnement inductif qui permet l'application du principe DpR est quant à lui très simple.

Soit (c'est la précondition) un sac S de n éléments, $n \geqslant 2$, prenant ses valeurs sur l'intervalle $bi..bs$ ($bi \leqslant bs$) et tel que $n > \text{card}(bi..bs)$. En outre, les valeurs extrêmes bi et bs appartiennent au sac. Selon le principe dit des « cases de courrier » ou des « nids de pigeon » (voir exercice 2 page 35), un tel sac contient au moins un doublon. L'objectif de l'exercice est de construire un algorithme qui délivre l'un quelconque des doublons présents dans S.

Ainsi, pour l'intervalle $bi..bs = 12..19$ et le sac $[\![14, 17, 12, 19, 14, 16, 12, 14, 15]\!]$ composé de neuf éléments, il existe deux doublons (12 et 14). Le résultat fourni pourra être indifféremment l'un ou l'autre. Une solution triviale existe. Elle consiste, pour chaque valeur v du sac S, à vérifier si elle existe déjà dans $S \dot{-} [\![v]\!]$. Elle conduit à un algorithme en $\mathcal{O}(n^2)$.

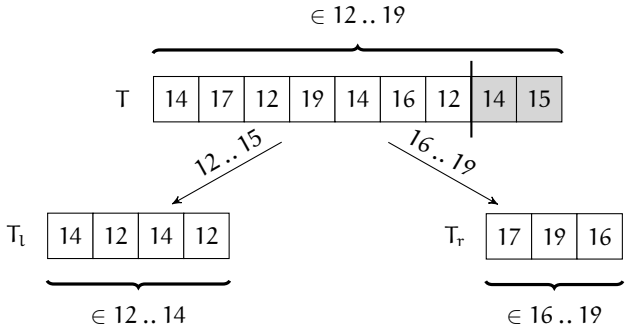
On se focalise sur une solution de type DpR visant une meilleure efficacité. Insistons sur le fait que la précondition ci-dessus ne correspond pas à une caractérisation générale de la présence d'un doublon dans un sac. Par exemple, le sac $[\![14, 16, 12, 12, 14]\!]$ contient des doublons sans que sa taille (5) n'excède celui de l'intervalle des valeurs y apparaissant (5 également).

Dans la suite, S est raffiné par un tableau $T[1..n]$ à valeurs dans l'intervalle $bi..bs$. Le tableau T est une variable globale dont les sous-tableaux sont identifiés par leurs bornes (bg borne gauche et bd borne droite).

Le principe DpR s'applique ici sur l'intervalle $bi..bs$. La phase de séparation mentionnée ci-dessus consiste à ventiler les valeurs de T dans deux tableaux T_l et T_r par rapport à la valeur mil, milieu de l'intervalle $bi..bs$, tout en tenant à jour l'intervalle $bil..bsl$ des valeurs de T_l et l'intervalle $bir..bsr$ des valeurs de T_r jusqu'à ce que l'on soit sûr que l'un

de ces deux tableaux contienne un doublon, auquel cas, si sa longueur excède 2, on peut alors lui ré-appliquer le même procédé.

Exemple Reprenons l'exemple ci-dessus. mil vaut $\lfloor(12+19)/2\rfloor = 15$. T_l reçoit donc les valeurs de T appartenant à l'intervalle $12\,..\,15$, tandis que T_r reçoit celles de l'intervalle $16\,..\,19$ (voir la figure ci-après).



Après avoir ventilé les sept premiers éléments de T dans T_l et dans T_r, on constate que T_l contient quatre éléments, qui appartiennent à l'intervalle $12\,..\,14$ de longueur 3. On en conclut que T_l contient (au moins) un doublon, et il devient inutile de poursuivre la ventilation ; on peut alors se limiter à poursuivre la recherche dans T_l.

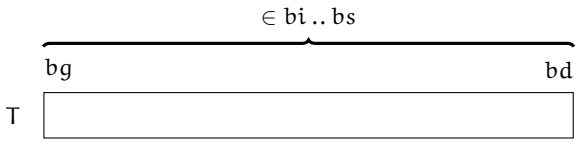
Plutôt que d'utiliser deux tableaux auxiliaires T_l et T_r, on va ventiler « sur place » (c'est-à-dire déplacer les valeurs destinées à T_l et T_r aux extrémités de T) en utilisant l'opération « **procédure** *Échange*(j, k) » qui échange les valeurs de T situées aux positions j et k.

L'opération *Ventiler* a comme profil :

procédure *Ventiler*(bi, bs, bg, bd; nbi, nbs, nbg, nbd : **modif**)

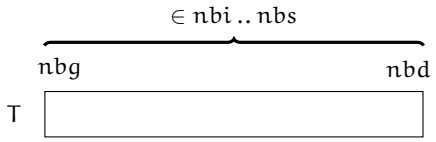
Schématiquement, cette procédure part d'un sous-tableau T[bg..bd] contenant un doublon et fournit un sous-tableau T[nbg .. nbs] strictement plus petit contenant également un doublon. Afin d'être opérationnelle, cette spécification doit être renforcée conformément à la précondition et à la postcondition suivante (voir chapitre 3, page 68 et suivantes) :

Précondition P :



1. Le tableau T[bg .. bd] prend ses valeurs dans l'intervalle bi .. bs.
2. card(bg .. bd) > 2 (ce tableau possède au moins trois éléments, le cas d'un tableau ayant deux éléments et un doublon étant trivial).
3. bi ∈ T[bg .. bd] et bs ∈ T[bg .. bd] (les valeurs extrêmes possibles appartiennent bien au tableau).
4. card(bg..bd) > card(bi..bs) (il y a plus de places dans T[bg..bd] (soit card(bg..bd)) que de valeurs possibles dans l'ensemble bi .. bs (soit card(bi .. bs)) : il existe donc au moins un doublon).

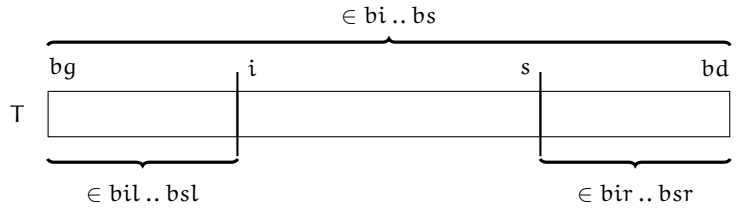
Postcondition Q :



1. nbg..nbd ⊂ bg..bd (le tableau T[nbg..nbd] est strictement plus petit que le tableau T[bg .. bd]).
2. Le tableau T[nbg .. nbd] prend ses valeurs dans l'intervalle nbi .. nbs.
3. card(nbg .. nbd) ⩾ 2 (le tableau possède au moins deux éléments).
4. card(nbg .. nbd) = card(bi .. bs) + 1 (il y a exactement une place de plus que de valeurs possibles : il existe donc au moins un doublon).
5. nbi ∈ T[nbg .. nbd] et nbs ∈ T[nbg .. nbd] (les valeurs extrêmes possibles appartiennent bien au tableau).
6. T[nbg..nbd] ⊑ T[bg..bd] (le sac des valeurs du tableau T[nbg..nbd] est inclus dans le sac des valeurs du tableau T[bg .. bd]).

Cependant, la postcondition Q ne se prête pas directement à la construction d'une boucle. Nous devons insérer entre les situations P et Q une situation intermédiaire R, qui va constituer la postcondition de la boucle. Le programme A spécifié par {R}A{Q} sera quant à lui constitué d'une alternative.

Situation intermédiaire R :



Nous allons distinguer quatre sortes de conjoints : les conjoints généraux (portant sur le tableau complet T[bg .. bd]), les conjoints spécifiques au sous-tableau T[bg .. i − 1], ceux spécifiques au sous-tableau T[s + 1 .. bd], et enfin le conjoint commun aux deux sous-tableaux.

Conjoints généraux.

1. mil = ⌊(bi + bs)/2⌋.
2. T[bg .. bd] est une permutation multiensembliste des valeurs initiales. Il existe donc toujours au moins un doublon dans le tableau. Ce tableau est modifié uniquement par des appels à la procédure *Échange*, le présent conjoint peut donc être oublié.

Conjoints spécifiques au sous-tableau T[bg .. i − 1].

1. i ∈ bg .. bd − 1 (le sous-tableau n'est jamais le tableau complet).
2. bil .. bsl ⊆ bi .. mil.
3. Le sous-tableau prend ses valeurs dans l'intervalle bil .. bsl.

4. $bil \,..\, bsl \neq \varnothing \Rightarrow bil \in T[bg \,..\, i-1]$ **et** $bsl \in T[bg \,..\, i-1]$ (si l'intervalle des valeurs du sous-tableau n'est pas vide, les valeurs extrêmes de cet intervalle sont présentes dans le tableau).
5. $card(bg \,..\, i-1) \in 0 \,..\, card(bil \,..\, bsl)+1$ (le nombre de places dans le sous-tableau est compris entre 0 et le nombre de valeurs possibles plus 1. Ce dernier cas implique l'existence d'au moins un doublon).

Conjoints spécifiques au sous-tableau $T[s+1 \,..\, bd]$. Ils sont similaires à ceux du sous-tableau $T[bg \,..\, i-1]$.

Conjoint commun aux deux sous-tableaux.
$card(bg \,..\, i-1) = card(bil \,..\, bsl)+1$ **ou** $card(s+1 \,..\, bs) = card(bir \,..\, bsr)+1$ (dans l'un des deux sous-tableaux, il y a plus de places que de valeurs possibles : deux emplacements contiennent la même valeur. Ce sous-tableau contient au moins un doublon).

113 - Q 1 **Question** 1.

a) Construire, pour la procédure *Ventiler*, la boucle B répondant à la spécification $\{P\}\,B\,\{R\}$.

b) Compléter le code de la procédure *Ventiler*.

c) Fournir une trace d'exécution des principales variables pour l'exemple introductif.

d) Fournir un majorant du nombre de conditions évaluées. En déduire la complexité au pire de la procédure *Ventiler*.

e) Montrer que la taille du sous-tableau de sortie $T[nbg \,..\, nbd]$ ne dépasse pas la moitié de la taille du tableau d'entrée $T[bg \,..\, bd]$ plus 1, autrement dit que :

$$card(nbg \,..\, nbd) \leqslant \left\lfloor \frac{card(bg \,..\, bd)}{2} \right\rfloor + 1. \tag{8.22}$$

Cette formule est à utiliser pour démontrer la terminaison de la procédure *Ventiler*.

113 - Q 2 **Question** 2. En déduire le raisonnement par induction qui permet de construire l'opération « **fonction** $CherchDoubl(bg, bd)$ **résultat** $bi \,..\, bs$ » et d'en prouver la terminaison. Cette opération délivre l'un quelconque des doublons du tableau $T[bg \,..\, bd]$. Quel est le modèle de division qui s'applique ? Fournir le code de cette opération. Montrer qu'elle est au pire en $\mathcal{O}(n)$ conditions évaluées.

Exercice 114. Le plus grand carré et le plus grand rectangle sous un histogramme

Il s'avère que cet exercice est remarquable à de nombreux points de vue.

- *Les deux problèmes abordés (carré et rectangle) présentent des spécifications proches. Cependant, les techniques utilisées sont différentes et difficilement transposables de l'un vers l'autre.*
- *Le second problème (rectangle) est abordé selon trois approches (DpR, méthode purement itérative et méthode mariant récursivité et itération).*

— *La solution DpR (dans sa version la plus élaborée) conduit à l'étude d'une structure de données originale et efficace, basée sur des arbres.*

— *La solution itérative utilise explicitement une pile qui exige, pour atteindre une solution efficace et élégante, un raffinement ingénieux.*

— *Cerise sur le gâteau, la dernière solution étudiée est un antidote à l'empirisme et à la « bidouille ». Le résultat, d'une concision et d'une pureté rare, ne peut être atteint qu'en appliquant scrupuleusement les préceptes défendus dans cet ouvrage.*

En outre, il ne s'agit pas d'un exercice purement « gratuit ». De simples extensions trouvent des applications au traitement d'images. De par la grande variété des solutions abordées, le placement de cet exercice dans ce chapitre est quelque peu arbitraire.

Le problème

Définition 33 (Histogramme d'un tableau de naturels) :
Soit t *un tableau défini sur l'intervalle* $a..b$ *et à valeurs dans l'intervalle* $i..s$. *Le tableau* h *défini sur l'intervalle* $i..s$ *et à valeurs dans l'intervalle* $0..b-a+1$ *est l'histogramme de* t *si pour chaque valeur* v *de l'intervalle* $i..s$, $h[v]$ *comptabilise le nombre de fois où* v *apparaît dans* t.

Cette définition se formalise comme suit. Soit

$$t \in a..b \to i..s \qquad \text{et} \qquad h \in i..s \to 0..b-a+1.$$

Le tableau h est l'histogramme de t si :

$$\forall k \cdot (k \in i..s \;\Rightarrow\; h[k] = \#j \cdot (j \in a..b \mid t[j] = k)).$$

Exemple Soit le tableau t suivant, défini sur l'intervalle $1..25$ et à valeurs dans l'intervalle $1..7$:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
1	3	2	1	4	5	4	3	2	1	6	1	5	3	7	6	3	4	5	6	1	4	6	1	3

Son histogramme h, défini sur l'intervalle $1..7$ et à valeurs dans l'intervalle $0..25$, se présente comme suit :

	1	2	3	4	5	6	7
h	6	2	5	4	3	4	1

Un tel tableau est souvent représenté comme le montre la figure 8.16 ci-dessous.

On peut noter (remarque exploitée ci-après) qu'il est possible de définir l'histogramme d'un histogramme. Ainsi, l'histogramme h' de h est défini sur l'intervalle $0..25$ et à valeurs dans l'intervalle $0..7$. Il se présente de la manière suivante :

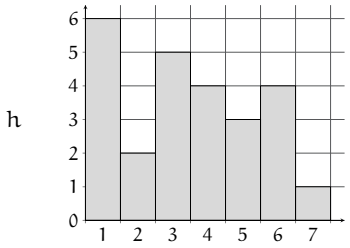
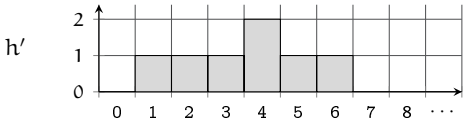


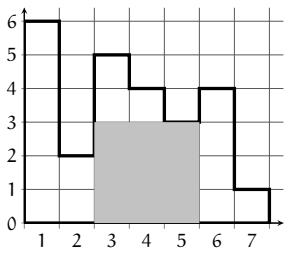
Fig. 8.16 – *Un exemple d'histogramme*

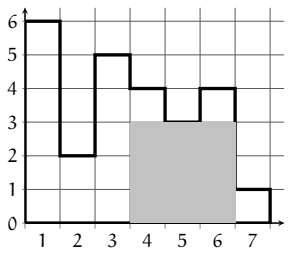


L'objectif de l'exercice est double. Il s'agit tout d'abord de rechercher le côté du plus grand *carré* sous l'histogramme, puis, dans une seconde étape, de rechercher l'aire du plus grand *rectangle* sous un histogramme. Dans chacun des cas, plusieurs solutions sont étudiées. L'exercice s'achève par une application aux images noir et blanc. La complexité des différentes solutions se mesure en nombre de conditions évaluées.

Recherche du côté du plus grand carré sous un histogramme

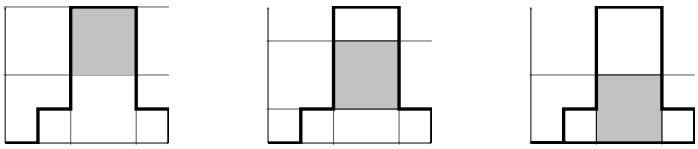
Pour l'exemple de la figure 8.16, la valeur recherchée est 3. Elle est atteinte par deux carrés différents :





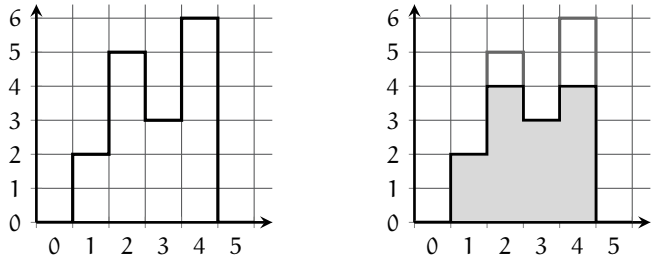
Deux solutions sont étudiées. La première est une version itérative, en $\mathcal{O}(n^2)$, la seconde, en $\Theta(n)$, est une optimisation de la précédente.

114 - Q 1 **Question 1.** Nous avons vu qu'il n'y a pas en général unicité de la solution. Une situation pour laquelle plusieurs solutions sont alignées verticalement peut exister, comme le montre le schéma ci-dessous :



Montrer qu'il est toujours possible de ne considérer, dans la résolution de ce problème, que les carrés qui touchent l'axe des abscisses. Cette propriété est appliquée systématiquement dans la suite.

Remarque Pour ce qui concerne plus particulièrement la recherche du plus grand carré sous un histogramme h défini sur l'intervalle $i \,..\, s$ et à valeurs dans l'intervalle $0 \,..\, b-a+1$, il est facile de constater qu'il est impossible d'y placer un carré dont le côté serait supérieur à $s-i+1$. Dans la suite, nous supposons – sans perte de généralité – que les histogrammes concernés par la recherche du plus grand carré sont écrêtés au-delà de $(s-i+1)$, comme le montre le schéma suivant :

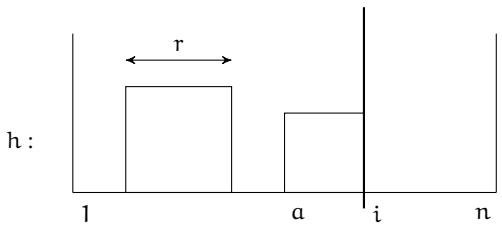


Histogramme original Histogramme écrêté (en gris)

Le prétraitement nécessaire à la satisfaction de cette contrainte n'est pas réalisé ici.

Plus grand carré sous un histogramme, version itérative naïve

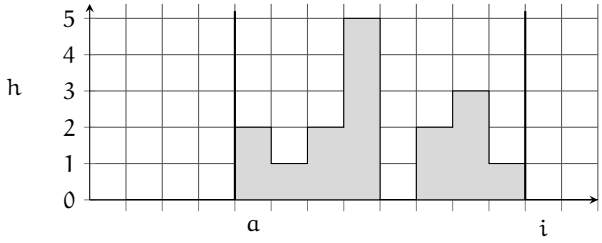
Question 2. Construire une solution itérative fondée sur l'invariant suivant : r est le côté du plus grand carré contenu dans l'histogramme $h[1 \,..\, i-1]$; le plus grand carré jouxtant la position i – le seul qui puisse encore s'agrandir – a comme côté $(i-a)$, avec $1 \leqslant a \leqslant i \leqslant n+1$. 114 - Q 2



Vérifier que la complexité est en $\mathcal{O}(n^2)$.

Plus grand carré sous un histogramme, version itérative optimale Du point de vue complexité, le problème que pose la solution naïve précédente réside dans le calcul d'une expression quantifiée, calcul pour lequel la solution la plus simple se fonde sur une boucle. On sait déjà que dans l'intervalle $a \,..\, i-1$, aucune valeur de h n'est inférieure à $a-i$ (dans le cas contraire le carré considéré n'existerait pas). Si on disposait de f, histogramme de $h[a \,..\, i-1]$, le raffinement de l'expression conditionnelle $\min(h[a \,..\, i-1]) \geqslant i-a+1$ se réduirait à $f[i-a]=0$ (en effet $f[i-a]=0$ signifie qu'aucune valeur de $h[a \,..\, i-1]$ n'est égale à $i-a$; on savait déjà qu'aucune d'entre elles n'est inférieure à $i-a$).

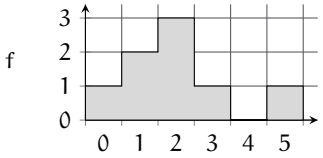
Exemple Considérons l'histogramme $h[a\,..\,i-1]$ suivant :



L'histogramme f de $h[a\,..\,i-1]$ est alors :

	0	1	2	3	4	5
f	1	2	3	1	0	1

Soit encore :

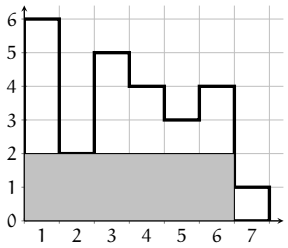


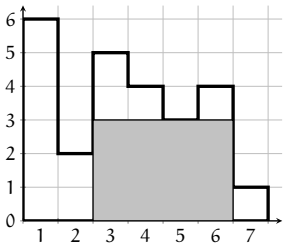
Ainsi, dans $h[a\,..\,i-1]$, il existe trois positions qui valent 2 (d'où $f[2]=3$).

114 - Q 3 **Question** 3. Construire une nouvelle solution basée sur l'observation ci-dessus. Fournir le code. Vérifier que cette solution est bien en $\Theta(n)$.

Recherche de l'aire du plus grand rectangle sous un histogramme

L'objectif de cette partie de l'exercice est de construire un programme qui détermine l'aire du plus grand rectangle sous l'histogramme. Pour l'exemple de la figure 8.16, page 308, cette valeur vaut 12. Elle est atteinte par deux rectangles différents :

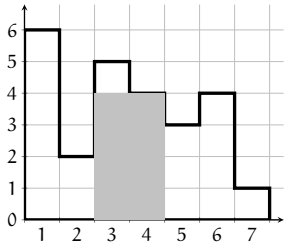




Trois solutions sont étudiées. La première est du type DpR (deux variantes sont proposées), la seconde est une solution itérative, enfin la troisième, de facture originale, panache efficacement itération et récursivité.

Définition 34 (am : aire maximale) :
Pour $1 \leqslant i \leqslant j \leqslant n+1$, $am(i,j)$ *est l'aire du plus grand rectangle sous la portion de l'histogramme dont les abscisses appartiennent à l'intervalle* $i\,..\,j-1$.

Pour l'exemple de la figure 8.16, page 308, $am(2,5)$ vaut 8 :



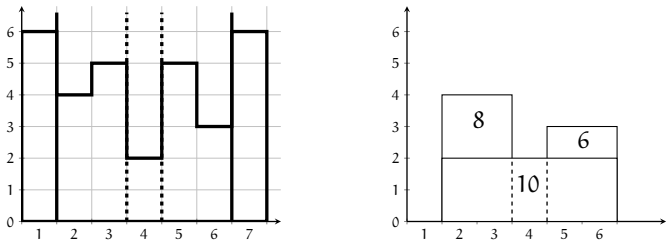
Plus grand rectangle sous un histogramme, version DpR naïve

Propriété 14 (de am. Non démontrée) :

- $am(i,i) = 0$ *pour* $1 \leqslant i \leqslant n+1$
- *si* $i \leqslant k < j$ *et* $h[k] = \min(h[i\,..\,j-1])$ *alors* $am(i,j) = \max(\{am(i,k), (j-i)\cdot h[k], am(k+1,j)\})$.

Cette propriété exprime qu'il est possible de déterminer l'aire du plus grand rectangle sous un histogramme, à condition de connaître d'une part la position d'une occurrence du minimum de l'histogramme, d'autre part l'aire des plus grands rectangles situés de part et d'autre du minimum en question.

Exemple Pour l'histogramme suivant :



$k = 4$ et la valeur de $am(2,7)$ est $\max(\{8,6,10\})$, soit 10.

Question 4. On suppose disponible la fonction $PosMin(i,s)$ qui délivre l'une quelconque des positions du minimum de $h[i\,..\,s-1]$. En appliquant directement la propriété 14 ci-dessus, fournir la version DpR de l'opération **fonction** $AmHDpR(i,s)$ **résultat** $\mathbb{N}$ qui délivre l'aire du plus grand rectangle présent sous l'histogramme $h[i..s-1]$. Quelle est la complexité de cette opération dans l'hypothèse où la fonction $PosMin$ est mise en œuvre par une recherche séquentielle ? 114 - Q 4

Plus grand rectangle sous un histogramme, version DpR et arbres de segments minimaux La solution précédente est basée sur une recherche linéaire standard (en $\Theta(n)$) du minimum d'un histogramme. Il existe cependant une solution plus efficace pour ce problème : celle

qui utilise les arbres de segments minimaux. Les principales définitions, ainsi que les éléments de vocabulaire les plus fréquents portant sur les arbres binaires, sont regroupés au chapitre 1.

Les arbres de segments minimaux (arbres de segments pour la recherche du minimum) Soit $h \in i..s \to \mathbb{N}$ un tableau. Un arbre de segments minimal pour h est un arbre binaire tel que chaque nœud est constitué de :

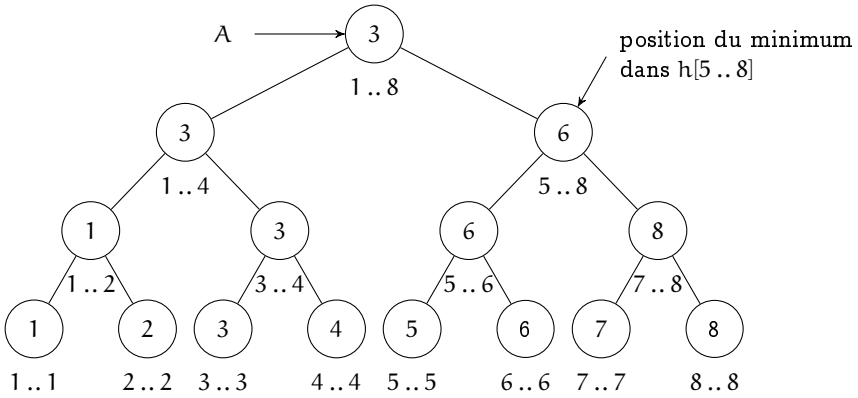
- la position p du (d'un) minimum de $h[i..s]$,
- l'intervalle $i..s$ en question,
- le sous-arbre de segments minimal gauche correspondant à la première moitié de l'intervalle $i..s$,
- le sous-arbre de segments minimal droit correspondant à la seconde moitié de l'intervalle $i..s$.

Dans la suite, pour des raisons de lisibilité, on se limite à des tableaux dont la taille est une puissance de 2 ($n = 2^k$). Les résultats obtenus se transposent à des valeurs n quelconques.

Exemple Soit h défini par :

1	2	3	4	5	6	7	8
2	6	1	5	9	3	8	4

L'arbre de segments minimal A correspondant est le suivant :



Un tel arbre est représenté par la structure

$$asm = \{/\} \cup \{(g, (m, i, s), d) \mid g \in asm \text{ et } d \in asm \text{ et } m \in \mathbb{N}_1 \text{ et } i \in \mathbb{N}_1 \text{ et } s \in \mathbb{N}_1\}$$

où $i..s$ est l'intervalle d'entiers du nœud considéré et m la position du minimum pour le tableau ainsi représenté. Soit « **fonction** $PosMinAux(a, p, q)$ **résultat** $\mathbb{N}$ » l'opération qui délivre la (l'une des) position(s) du minimum de $h[p..q]$ dans l'arbre de segments minimal a.

Question 5. Construire cette opération, fournir son code et calculer sa complexité en nombre de nœuds visités d'abord, de conditions évaluées ensuite. En déduire la complexité de la nouvelle version de l'opération *AmHDpR*. 114 - Q 5

Plus grand rectangle sous un histogramme, version itérative Le principe de cette version s'apparente à celui appliqué dans les versions itératives de la recherche du plus grand carré. Cependant, il s'en démarque par le fait qu'il existe ici en général *plusieurs* rectangles candidats à l'élargissement vers la droite. Cet ensemble de candidats est dénommé « ensemble des rectangles ouverts », il est noté O et $k = \text{card}(O)$. Dans la suite, on suppose que h est étendu en 0 et en $n+1$ par 0. Nous décidons de représenter un rectangle ouvert par le couple (g, ht), où g est l'abscisse la plus à gauche du rectangle et ht sa hauteur. Formellement, pour i fixé, un rectangle ouvert (g, ht) se définit par :

$$g \in 0\,..\,i-1 \text{ et } ht = \min(h[g\,..\,i-1]) \text{ et } h[g-1] < ht.$$

À la figure 8.17, pour $i = 10$, nous avons $O = \{(0,0),(1,1),(4,3),(6,4)\}$. L'extension de h en 0 permet de disposer en permanence, dans la pile O, du rectangle « neutre » de coordonnées $(0,0)$ et d'aire 0 (zéro).

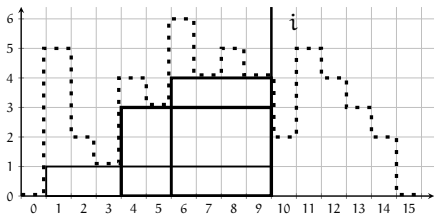


Fig. 8.17 – *Un histogramme et les quatre rectangles ouverts pour* $i = 10$

Question 6. Démontrer la propriété suivante : 114 - Q 6

Lemme 1 :
Soit $P = <(g_1, ht_1), \ldots, (g_k, ht_k)>$ *la* liste *des rectangles ouverts triée sur les* g_j *croissants. La liste* $< ht_1, \ldots, ht_k >$ *est également strictement croissante.*

Sans entrer dans le détail de ce qui constitue la question suivante, lors de la progression, soit on supprime de la liste P le rectangle ayant la hauteur la plus grande (c'est-à-dire le dernier élément de la liste), soit on allonge la largeur des rectangles ouverts, en créant éventuellement, à la queue de la liste P, un nouveau rectangle dont la hauteur est supérieure à celles de tous les rectangles présents dans P. La liste P se comporte donc comme une *pile*. L'extension de h en $(n+1)$ permet, lorsque i atteint cette valeur, de dépiler tous les rectangles ouverts à l'exception de celui de coordonnées $(0,0)$, puis de faire progresser i jusqu'à $n+2$.

Les informations présentes dans P sont redondantes. En effet, les hauteurs ht_j peuvent être retrouvées à partir des abscisses g_j. Afin de supprimer cette redondance, on décide de

raffiner la pile P par la pile P′ qui se présente comme suit : $P' = < g_1 - 1, \ldots, g_k - 1, s >$, où le sommet s de la pile P′ est l'abscisse la plus à droite telle que $s \in g_k + 1 \,..\, i - 1$ et $(g_k, h[s])$ est le sommet de la pile P (P′ contient un élément de plus que P). Ainsi, si $P = < (0,0), (1,1), (4,3), (6,4) >$ (voir figure 8.17), $P' = < -1, 0, 3, 5, 9 >$.

114 - Q 7 **Question 7.** Montrer que cette représentation P′ permet de retrouver toutes les informations présentes dans P.

114 - Q 8 **Question 8.** En supposant disponibles les opérations suivantes sur la pile P′ :

initPile procédure qui vide la pile,

sommetPile fonction qui délivre le sommet de la pile sans modifier celle-ci (précondition : la pile est supposée non vide),

empiler(v) procédure qui empile l'entier v,

dépiler procédure qui supprime le sommet de pile (précondition : la pile est supposée non vide),

construire la boucle sur laquelle est fondée cet algorithme. Quelle est sa complexité ?

Plus grand rectangle sous un histogramme, version Morgan Face à la solution précédente utilisant explicitement une pile, il est légitime de se poser la question de savoir s'il n'est pas possible d'utiliser implicitement (à la place) la pile d'exécution. La solution étudiée ici répond à cette interrogation, même s'il ne s'agit pas d'une adaptation de la solution précédente, mais d'une approche originale due à l'informaticien australien C. Morgan qui l'utilise comme exemple de construction d'un programme à partir d'une spécification formelle (voir [52], pages 209-216).

L'histogramme est étendu en 0 et $n+1$ de sorte que $h[0] = h[n+1] = -1$. Ces valeurs servent de sentinelles dans la suite.

Soit $P(k) = k \in 0 \,..\, n$ un prédicat. Soit la procédure *AmHMorg*(i, b, j) (où i est un paramètre d'entrée et b et j sont des paramètres de sortie) spécifiée en pré/post par :

Précondition : $P(i)$.

Postcondition : Q défini par $Q \mathrel{\widehat{=}} Q_1$ et Q_2 et Q_3 et Q_4 avec

$Q_1 \mathrel{\widehat{=}} i < j \leqslant n+1$

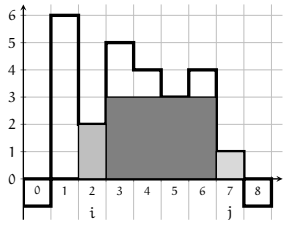
$Q_2 \mathrel{\widehat{=}} h[i] \leqslant \min(h[i+1 \,..\, j-1])$

$Q_3 \mathrel{\widehat{=}} h[i] \geqslant h[j]$

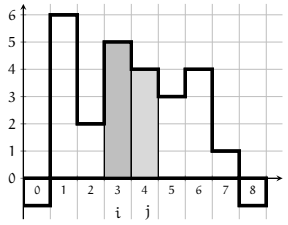
$Q_4 \mathrel{\widehat{=}} b = am(i+1, j)$.

Pour une abscisse i donnée, l'appel *AmHMorg*(i, b, j) fournit deux résultats, j et b. Le premier est la plus petite abscisse (j) supérieure à i telle que d'une part $h[j] \leqslant h[i]$ et d'autre part la portion de l'histogramme $h[i+1 \,..\, j-1]$ est supérieure ou égale à $h[i]$. Notons que la spécification garantit l'existence de j. En effet, il est toujours possible de trouver un tel j puisqu'il existe une abscisse j telle que $h[j]$ est inférieure ou égale à toutes les valeurs précédentes de l'histogramme – c'est $(n+1)$ $(h[n+1] = -1)$ – et que, dans le cas où l'intervalle $i+1 \,..\, j-1$ est vide, l'expression $\min(h[i+1 \,..\, j-1])$ devient $\min(\varnothing)$, qui vaut $+\infty$ $(h[i] \leqslant +\infty)$. Le second résultat, b, est l'aire du plus grand rectangle présent sous la portion de l'histogramme délimitée par l'intervalle $i+1 \,..\, j-1$.

Exemple Reprenons l'exemple de la figure 8.16, page 308, pour évaluer $AmHMorg(2, b, j)$:



Cet appel délivre $j = 7$ et $b = 12$.
Prenons à nouveau l'exemple de la figure 8.16, page 308, pour évaluer $AmHMorg(3, b, j)$:



Cette fois, l'intervalle $i + 1 \,..\, j - 1$ est vide. L'aire b du plus grand rectangle sous la portion $h[4 \,..\, 3]$ est nulle. Cet appel délivre donc $j = 4$ et $b = 0$.

Question 9. Posons $Q_5 \mathrel{\widehat{=}} h[j] \leqslant \min(h[i + 1 \,..\, j - 1])$. Montrer que $Q \Rightarrow Q_5$. Dans la suite, le prédicat $(Q \text{ et } Q_5)$ est noté Q'. 114 - Q 9

Question 10. Montrer qu'au retour de l'appel $AmHMorg(0, b, j)$ j vaut $(n + 1)$ et b est l'aire du plus grand rectangle sous l'histogramme $h[1 \,..\, n]$. 114 - Q 10

Nous recherchons une solution récursive de la forme

```
1.  procédure AmHMorg(i; b, j : modif) pré
2.      i ∈ ℕ et b ∈ ℕ et j ∈ ℕ₁ et
3.      c ∈ ℕ et k ∈ ℕ
4.  début
5.      Initialisation ;
6.      tant que non CA faire
7.          AmHMorg(j, c, k) ;
8.          FinProgression
9.      fin tant que
10. fin
```

Si $I(i, b, j)$ et $CA(i, b, j)$ représentent respectivement l'invariant de la boucle et la condition d'arrêt, la version annotée de la procédure $AmHMorg$ se présente comme suit :

```
1.  procédure AmHMorg(i; b, j : modif) pré
2.      i ∈ ℕ et b ∈ ℕ et j ∈ ℕ₁ et
3.      c ∈ ℕ et k ∈ ℕ
4.  début
5.      P(i)
```

```
6.   Initialisation ;
7.   [I(i, b, j)]
8.   tant que [non CA(i, b, j)] faire
9.     [I(i, b, j) et non CA(i, b, j)]
10.    AmHMorg(j, c, k) ;
11.    [Q'(j, c, k) et I(i, b, j) et non CA(i, b, j)]
12.    FinProgression
13.    [I(i, b, j)]
14.  fin tant que
15.  [I(i, b, j) et CA(i, b, j)]
16.  [Q'(i, b, j)]
17. fin
```

Remarques

1. Puisque i est un paramètre d'entrée, P(i) est un prédicat toujours satisfait.
2. Ces annotations résultent des éléments théoriques fondamentaux de la programmation séquentielle (voir chapitre 3).
3. Dans l'annotation de la ligne 11, le conjoint $Q'(j, c, k)$ est la postcondition résultant de l'appel récursif de la ligne 10. Le reste de la formule ($I(i, b, j)$ **et non** $CA(i, b, j)$) est hérité directement de la précondition de la progression. En effet, l'appel de la ligne 10, *AmHMorg*(j, c, k), ne modifie pas la valeur des variables d'état de la boucle (i, b et j).
4. La ligne 15 est la postcondition « naturelle » de la boucle, tandis que la ligne 16 est la postcondition de la procédure. Nous pouvons logiquement en conclure que
$$I(i, b, j) \text{ et } CA(i, b, j) \Rightarrow Q'(i, b, j). \tag{8.23}$$
5. P et Q' sont des prédicats connus (donnés) ; I et CA sont en revanche inconnus. *Initialisation* et *FinProgression* sont des fragments de code inconnus, à construire à partir de leurs spécifications. Celle de *FinProgression* est le couple de prédicats des lignes 11 (pour la précondition) et 13 (pour la postcondition). Une solution triviale serait de choisir l'action « vide », puisque la postcondition est déjà un conjoint de la précondition. Cependant, ce choix est à exclure car il ne permet pas de montrer que le programme se termine. Il faut rechercher un fragment de code qui « rétablit » l'invariant $I(i, b, j)$.

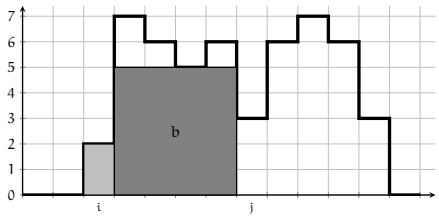
Dans les questions suivantes, nous allons développer progressivement cette boucle en appliquant les principes classiques de construction d'itérations (voir chapitre 3).

114 - Q 11 **Question** 11. Ces principes préconisent de déterminer tout d'abord l'invariant I et la condition d'arrêt CA. En partant de la formule 8.23, faire une proposition pour I et pour CA.

114 - Q 12 **Question** 12. Montrer qu'il est légal d'appeler la procédure *AmHMorg*(j, c, k) à la ligne 10 (autrement dit, que la précondition P(j) est impliquée par le prédicat de la ligne 9).

Question 13. Nous cherchons à présent à déterminer le code correspondant à *FinProgression*. 114 - Q 13

a) Si la configuration suivante



est une instance du prédicat de la ligne 9, quelle est la situation atteinte à la ligne 11 ?

b) Fournir une solution pour le fragment de code *FinProgression*. Quelle est, sur l'exemple considéré, la situation atteinte à la ligne 13 ?

c) Que se serait-il passé si, au lieu d'utiliser la postcondition Q', nous avions simplement utilisé Q ?

Question 14. Fournir une solution pour le fragment de code *Initialisation*. 114 - Q 14

Question 15. Montrer que la procédure se termine. 114 - Q 15

Question 16. Fournir le code de la procédure. 114 - Q 16

Question 17. Dans cette question, nous cherchons à montrer que la complexité de cette solution est en $\Theta(n)$. Nous décidons de dénombrer les appels à la procédure *AmHMorg*. Cette décision est (asymptotiquement) compatible avec le choix initial de dénombrer les conditions puisque s'il y a a appels récursifs à la procédure, il y a $(a+1)$ évaluations de la condition de la boucle. Démontrer par induction la propriété suivante : l'exécution de $AmHMorg(i, c, j)$ entraine $(j-i)$ appels récursifs à *AmHMorg*. 114 - Q 17

Application : recherche de l'aire du plus grand rectangle noir dans une image noir et blanc

Soit une image rectangulaire composée de pixels noirs (1) et blancs (0). On cherche la plus grande sous-image rectangulaire complètement noire. De manière plus formelle, étant donnée une matrice booléenne $T[1..m, 1..n]$ représentant une image, on cherche la valeur de a, aire du plus grand rectangle noir contenu dans T.

Exemple Dans l'image de la figure 8.18, l'aire a du plus grand rectangle est 6.

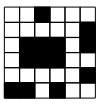


Fig. 8.18 – *Détail d'une image noir et blanc*

114 - Q 18

Question 18. Montrer comment il est possible d'appliquer les algorithmes étudiés auparavant pour résoudre ce problème. On précisera la complexité temporelle de la solution proposée.

CHAPITRE 9

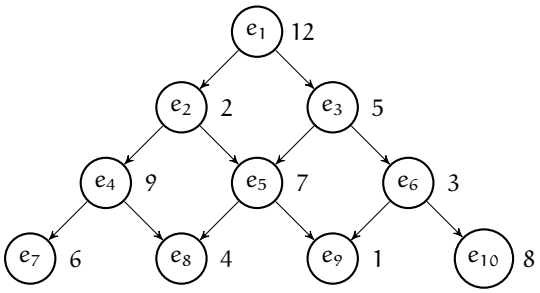
Programmation dynamique

> Se rappeler quelque chose est encore le meilleur moyen de ne pas l'oublier.
>
> *(P. Dac)*

9.1 Présentation

Comme la méthode « Diviser pour Régner », la « Programmation Dynamique » permet de résoudre des problèmes en combinant les solutions de sous-problèmes. Une différence essentielle entre ces deux approches réside dans le fait que la programmation dynamique ne concerne que des problèmes de calcul de valeur optimale d'une grandeur numérique. Par conséquent, la division en sous-problèmes *doit* se traduire par une équation de récurrence. Cette méthode est avantageuse lorsque les sous-problèmes ont eux-mêmes des sous-problèmes en commun (on parle aussi de recouvrement de sous-problèmes) et qu'un sous-problème n'est évalué qu'une seule fois, sa solution étant stockée explicitement (technique de mémoïsation). Une autre différence entre « Diviser pour Régner » et « Programmation Dynamique » tient à la nature des programmes qui leur sont associés : récursifs (en général) dans le premier cas, itératifs dans l'autre.

À titre d'illustration simple, considérons la pyramide de nombres entiers suivante



représentée par un graphe orienté (voir section 1.5, page 22), dont chacun des sommets étiquetés de e_1 à e_{10} porte la valeur figurant à sa droite. On cherche le chemin allant de e_1 (sommet de la pyramide) à l'un quelconque des éléments de la base (e_7, e_8, e_9 ou e_{10}) traversant des nombres dont la somme est maximale. Le calcul du chemin de valeur maximale associé à l'élément e_8 noté $\mathrm{sopt}(e_8)$ est donné par :

$$\mathrm{sopt}(e_8) = 4 + \max(\{\mathrm{sopt}(e_4), \mathrm{sopt}(e_5)\}).$$

De même, pour e_9, on aura :

$$sopt(e_9) = 1 + \max(\{sopt(e_5), sopt(e_6)\})$$

et à leur tour, les calculs de $sopt(e_4)$, $sopt(e_5)$ et $sopt(e_6)$ font appel à des valeurs communes ($sopt(e_2)$ et $sopt(e_3)$). On observe donc que des calculs identiques sont requis pour en effectuer d'autres, ce qui correspond à la présence de sous-problèmes communs.

C'est un des points forts de la programmation dynamique de n'effectuer ces calculs qu'une seule fois grâce à l'utilisation d'une table mémorisant la solution de chaque sous-problème. Dans l'exemple précédent, on stockera dans un tableau $V[1 .. 10]$ les valeurs associées à la récurrence $sopt$. Remarquons que, dans ce problème particulier, si l'on est certain que la valeur recherchée se trouve bien dans V, elle n'est pas localisée *a priori*, mais doit être déterminée comme le maximum du sous-tableau $V[7 .. 10]$.

D'une façon générale, la structure tabulaire choisie va être remplie grâce à une récurrence préalablement établie. Mais il est *indispensable* de procéder de sorte que la valeur de toute cellule relevant d'un terme récurrent puisse être calculée exclusivement à partir de cellules *déjà remplies*. Il faut donc définir une *progression* ou *évolution* du calcul offrant cette garantie. Dans l'exemple précédent, on *doit* faire évoluer le calcul en considérant les éléments par lignes successives de la pyramide, puisque le calcul associé à un élément de la ligne l ($l > 1$) fait appel à un ou deux éléments de la ligne $(l - 1)$.

Un autre aspect de la programmation dynamique est que l'on cherche à résoudre un problème d'optimisation, c'est-à-dire qu'elle vise à trouver une solution qui minimise un coût ou maximise un gain, ce qui est le cas dans l'exemple de la pyramide.

Les problèmes à résoudre doivent cependant satisfaire le *principe d'optimalité* de Bellman [1], selon lequel la [2] solution optimale d'un problème peut s'obtenir à partir des solutions optimales de sous-problèmes. Ce principe s'énonce ainsi :

> Toute *sous-politique* d'une *politique* optimale est elle-même optimale.

Le terme de « politique » est évidemment propre à chaque problème. On peut dès lors procéder selon deux schémas : i) s'assurer d'abord (en général par une démonstration par l'absurde) que le *principe d'optimalité* s'applique à sa solution optimale et rend raisonnable le recours à la programmation dynamique (mais rien n'est garanti malgré tout), ou ii) rechercher une récurrence dont le terme correspond à la grandeur optimale recherchée. Dans les exercices proposés par la suite, à de rares exceptions près, nous nous placerons dans cette seconde approche, puisque l'établissement de la récurrence prouve *de facto* que le principe d'optimalité s'applique.

Nous illustrons maintenant le principe de Bellman. Tout d'abord, nous allons nous interroger sur son applicabilité à deux problèmes « assez proches » de calcul de plus court chemin (au sens de leur longueur) dans un graphe orienté non valué (voir section 1.5, page 22). Un premier exemple (classique) consiste à calculer le plus court chemin entre deux sommets quelconques d'un graphe orienté G. Il est facile de montrer par l'absurde qu'un chemin est de longueur minimale si et seulement si ses sous-chemins sont de longueur minimale. Soit en effet un graphe orienté G et un chemin de a à d de longueur minimale, qui passe par b et c (arcs en trait plein sur la figure 9.1). La longueur de ce chemin est la somme des longueurs des sous-chemins de a à b, de b à c et de c à d. Supposons qu'il

1. Richard Bellman, chercheur à la Rand Corporation, a choisi en 1950 le terme de « programmation dynamique » pour éviter les mots « recherche » et « mathématiques ». Voir [50].
2. Un problème n'a pas nécessairement une solution optimale unique.

existe dans le graphe G un chemin de b à c de coût moindre (en pointillé sur la figure) que le chemin de b à c choisi ; il existe alors un chemin plus court de a à d, ce qui est contraire au fait que le chemin de a à d est de longueur minimale.

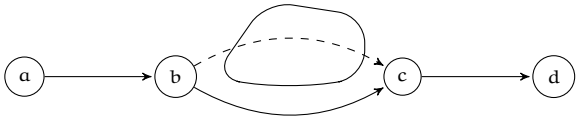


Fig. 9.1 – *Application du principe d'optimalité à la recherche du plus court chemin dans un graphe*

Prenons maintenant un second exemple, la recherche du plus long chemin *sans circuit* entre deux sommets quelconques d'un graphe orienté. Soit le graphe de la figure 9.2. Le plus long chemin sans circuit de a à c est de longueur 2 $\langle a, b, c\rangle$, le plus long chemin sans circuit de a à b est de longueur 2 $\langle a, c, b\rangle$ et le plus long chemin sans circuit de b à c est de longueur 1. Le plus long chemin sans circuit de a à c ne s'obtient donc pas par composition de chemins sans circuit, ce qui montre que le principe d'optimalité n'est pas vérifié dans ce cas.

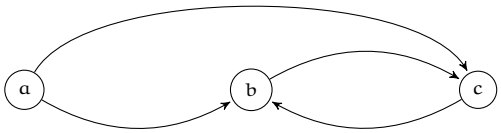


Fig. 9.2 – *Un (contre-)exemple où le principe d'optimalité ne s'applique pas.*

Cependant, les considérations ci-dessus ne *prouvent* pas que la programmation dynamique ne peut pas s'appliquer. En effet, il pourrait exister une autre façon de définir une politique. Néanmoins, les indices sont forts pour penser que l'on ne peut traiter ce problème que par essais successifs (voir chapitre 5), donc au prix d'une complexité élevée.

Reprenons maintenant le problème de la recherche du meilleur chemin (au sens défini auparavant) dans une pyramide de nombres. Ici, au lieu de vérifier l'applicabilité du principe de Bellman *a priori*, nous allons d'emblée chercher à établir une récurrence. Soit la pyramide de hauteur h ayant $n = \left(\sum_{k=1}^{h+1} k\right)$ éléments (figure 9.3). Appelons $v(i)$ la valeur de l'élément e_i, $\mathrm{prg}(i)$ et $\mathrm{prd}(i)$ les numéros des (au plus) deux prédécesseurs possibles (gauche et droit) de e_i sur tout chemin de e_1 à e_i. La valeur $\mathrm{sopt}(i)$ dépend en général de celles de ses deux prédécesseurs et le cas des éléments n'ayant qu'un ou pas de prédécesseur doit être traité séparément, d'où la récurrence :

$$\begin{aligned}
&\mathrm{sopt}(1) = v(1) \\
&\mathrm{sopt}(i) = v(i) + \mathrm{sopt}(\mathrm{prd}(i)) && i \in I_g \\
&\mathrm{sopt}(i) = v(i) + \mathrm{sopt}(\mathrm{prg}(i)) && i \in I_d
\end{aligned}$$

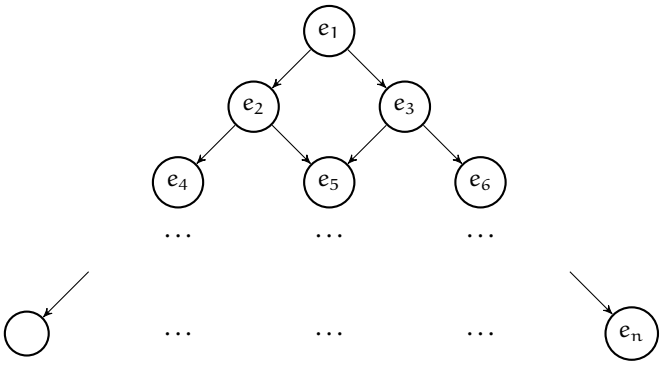


Fig. 9.3 – *Une pyramide de nombres*

$$\text{sopt}(i) = v(i) + \max\left(\left\{\begin{array}{l}\text{sopt}(\text{prg}(i)),\\ \text{sopt}(\text{prd}(i))\end{array}\right\}\right) \qquad 1 < i < n \text{ et } i \notin I_d \text{ et } i \notin I_g$$

où I_d (resp. I_g) désigne l'ensemble des indices de la branche la plus à droite (resp. gauche) de la pyramide. $I_d = \{3, 6, 10, 15, 21, \ldots\}$ est l'ensemble des nombres (entiers) triangulaires (c'est-à-dire de la forme $i = \sum_{k=1}^{j} k$, voir aussi exercice 112, page 299) de l'intervalle $2..n$. $I_g = \{2, 4, 7, 16, 22, \ldots\}$ est quant à lui l'ensemble des entiers successeurs d'un nombre triangulaire de ce même intervalle. Ici, le fait d'avoir pu définir sopt de façon récurrente fait que le principe de Bellman est appliqué.

De façon générale, l'écriture d'un algorithme de programmation dynamique nécessite donc d'exprimer le problème en termes de sous-problèmes de taille inférieure et d'établir une relation de récurrence entre ces derniers (comme dans l'exemple de la pyramide). Souvent, plusieurs récurrences sont possibles (approches correctes différentes, récurrence avant ou arrière pour un même point de vue), que des considérations d'efficacité de mise en œuvre peuvent éventuellement départager.

Une fois la récurrence établie, on détermine une structure tabulaire dans laquelle seront stockées les valeurs associées aux sous-problèmes (donc aux éléments de la récurrence). Cette dernière peut être un vecteur ou une table d'un nombre quelconque de dimensions (en pratique, on dépasse rarement la dimension 2). Le remplissage de cette table va se faire alors de façon itérative en exploitant la relation de récurrence. On voit ainsi que la construction d'un algorithme de programmation dynamique est en fait celle d'un algorithme itératif particulier, dont les éléments constitutifs reposent sur la récurrence et l'exploitation qui en est faite pour progresser dans le remplissage du tableau. Il importe de remarquer qu'un algorithme de programmation dynamique remplit un tableau contenant les coûts ou gains des solutions optimales des sous-problèmes (et finalement du problème) et non la solution elle-même. Ainsi, dans l'exemple de la pyramide de nombres, un élément du tableau sert à stocker le coût optimal associé à un sommet e_i et non le chemin optimal allant du sommet de la pyramide à e_i. Afin de construire une solution optimale[3], il est nécessaire d'enrichir la table au cours de sa construction ; nous parlerons alors de « la méthode du Petit Poucet », puisqu'on laisse « une trace » du choix fait pour obtenir l'optimal de chacun des sous-problèmes. La solution optimale est obtenue en parcourant la

3. S'il y a plusieurs solutions optimales, les retrouver toutes est un peu plus compliqué. En général, trouver une solution optimale quelconque est suffisant.

table dans un ordre approprié, une fois sa construction terminée. Par abus de langage, nous emploierons dans la suite le terme « coût de *la* solution optimale », même si le problème considéré peut admettre plusieurs solutions de même coût optimal. Enfin, il est à noter que la faible complexité (polynomiale) des algorithmes obtenus tient précisément au fait que l'on ne cherche pas à calculer toutes les solutions afin d'en extraire la (une) meilleure (comme le fait la démarche des « Essais Successifs », voir chapitre 5), la programmation dynamique réduisant l'espace de recherche par l'application du principe d'optimalité.

Les algorithmes de programmation dynamique utilisant une structure tabulaire explicite, on complète l'étude de la complexité temporelle des solutions proposées par celle de leur complexité spatiale. Concernant la première, elle s'exprime le plus souvent en termes de conditions évaluées du fait qu'un algorithme relevant de la programmation dynamique recherche une valeur optimale et, à cet effet, repose sur une comparaison de coûts. Dans les exercices où elle n'est pas précisée, l'opération élémentaire est donc l'évaluation de conditions.

9.2 Un exemple : la plus longue sous-séquence commune à deux séquences

Intérêts de cet exemple La recherche de la plus longue sous-séquence commune à deux séquences illustre la démarche décrite ci-dessus sur un exemple de difficulté moyenne quant à l'établissement de la récurrence. Par ailleurs, il sert de base à l'exercice 106 page 278, figurant dans le chapitre 8.

Le problème Étant données deux séquences x et y, le problème est de trouver la longueur de leur(s) plus longue(s) sous-séquence(s) commune(s). Par exemple, pour les chaînes *altitude* et *piteux*, cette longueur vaut 3 car *ite* et *itu* sont leurs plus longues sous-séquences communes.

Comme une séquence x possède $2^{|x|}$ sous-séquences, en y incluant x elle-même et le mot vide ε, une manière « simpliste » de trouver la plus longue sous-séquence commune à deux séquences x et y consiste à calculer et à comparer tous les couples de sous-séquences, ce qui est en complexité $\mathcal{O}(2^{|x|+|y|})$ (l'opération élémentaire étant la comparaison de deux symboles). On va voir que la programmation dynamique procure un moyen de faire très nettement mieux.

Par exemple, reprenons $u\,(=x) =$ *altitude* et $v\,(=y) =$ *piteux*, dont l'une des deux plus longues sous-séquences communes est *itu*. Son préfixe *it* est certainement la plus longue sous-séquence commune à au moins un préfixe de u et un préfixe de v, par exemple à *altit* et *pite*. Pouvons-nous utiliser cette remarque de manière systématique ?

La décomposition en sous-problèmes La solution consiste à définir une famille de sous-problèmes sur lesquels on saura construire une relation de récurrence. Dans notre cas, en appelant « sous-problème » la recherche de la plus longue sous-séquence commune à un préfixe de x et à un préfixe de y, on se ramènerait à un nombre de comparaisons en $\mathcal{O}(|x| \cdot |y|)$ (car $|x|$ (resp. $|y|$) est le nombre de préfixes de x (resp. y), à supposer que la résolution de cet ensemble de sous-problèmes soit suffisante pour résoudre le problème de

départ). C'est ici la difficulté : il faut montrer par récurrence comment cette décomposition permet de résoudre le problème initial.

Notons $|x| = m$ la longueur de x et $|y| = n$ la longueur de y. Commençons par définir $lssc(i,j)$, avec $i \in 0\,..\,n$ et $j \in 0\,..\,m$, comme la longueur de la (d'une) plus longue sous-séquence commune au préfixe de longueur i de y et au préfixe de longueur j de x. Si cette construction peut aller correctement jusqu'à son terme, en augmentant la longueur des préfixes, la solution sera calculée par $lssc(n,m)$. Par exemple, si $u\ (= x) =$ *altitude* et $v\ (= y) =$ *piteux*, $lssc(1,6) = lssc(2,3) = 0$ et $lssc(2,5) = 1$.

Le traitement complet du problème Le problème est maintenant traité de manière complète sous la forme de questions-réponses. La première question a pour but de mettre en évidence que le problème fait appel à trois sous-problèmes de même nature que le problème initial. La seconde est l'étape qui permet de démontrer par récurrence que cette décomposition est suffisante. La troisième concerne la réalisation informatique et l'utilisation de la méthode du « Petit Poucet » pour récupérer la solution optimale. On y aborde aussi les aspects liés à la complexité de l'algorithme *WFlg* obtenu. La dernière question vise à obtenir une version appelée *WFLgAvant* de l'algorithme *WFlg* améliorant la complexité spatiale, mais se limitant au seul calcul de la longueur de la (d'une) plus longue sous-séquence commune à deux séquences. Cet algorithme est utilisé dans l'exercice 106, page 278.

Question 1. Soit $z = z[1..k]$ une plus longue sous-séquence commune à x et y. Montrer que :

- soit $x[m] = y[n]$ et alors : i) $z[k] = x[m] = y[n]$, et ii) $z[1\,..\,k-1]$ est une plus longue sous-séquence commune à $x[1\,..\,m-1]$ et $y[1\,..\,n-1]$,
- soit $x[m] \neq y[n]$ et alors z est la plus longue entre d'une part la plus longue sous-séquence commune à $x[1..m-1]$ et y, de l'autre la plus longue sous-séquence commune à x et $y[1\,..\,n-1]$.

Réponse 1. On examine successivement les deux cas évoqués dans la question. Prenons tout d'abord le cas $x[m] = y[n] = g$. Appelons w une plus longue sous-séquence commune à $x[1\,..\,m-1]$ et $y[1\,..\,n-1]$. La séquence $w \cdot g$ est une sous-séquence commune à x et y, de longueur supérieure de 1 à celle de w et donc maximale.

Supposons maintenant que $x[m] \neq y[n]$. La plus longue sous-séquence commune à x et y, notée w, ne peut être construite que de deux façons, et on prendra la meilleure. Le premier choix consiste à partir de z, la plus longue sous-séquence commune à $x[1\,..\,m-1]$ et $y[1\,..\,n-1]$, et à regarder ce que provoque l'ajout du dernier symbole de x, $x[m]$. L'autre choix est offert par le cas symétrique, à savoir partir de z et regarder ce que provoque l'ajout de $y[n]$ le dernier symbole de y. Prenons le premier cas : le symbole $x[m]$ trouve ou non un symbole dans y au-delà du dernier ($y[n]$), ayant servi à construire z. Autrement dit, la séquence candidate w est la plus longue sous-séquence commune à x et $y[1\,..\,n-1]$. De façon symétrique, la séquence candidate w est la plus longue sous-séquence commune à $x[1\,..\,m-1]$ et y.

Question 2. En déduire la formule qui permet dans le cas général de calculer $lssc(i,j)$ en fonction de $lssc(i-1,j-1)$, $lssc(i-1,j)$ et $lssc(i,j-1)$, puis donner la récurrence complète de calcul de $lssc$.

Réponse 2. Pour le cas général, on peut s'appuyer sur ce qui a été établi à la question précédente en considérant des préfixes $x[1..j]$ d'une part et $y[1..i]$ d'autre part, non vides. Si $x[j] = y[i]$, une plus longue sous-séquence commune à $x[1..j]$ et $y[1..i]$ est construite comme une plus longue sous-séquence commune à $x[1..j-1]$ et $y[1..i-1]$ rallongée de $x[j] = y[i]$. Sinon, on doit résoudre deux sous-problèmes : trouver une plus longue sous-séquence commune à $x[1..j]$ et $y[1..i-1]$ et une plus longue sous-séquence commune à $x[1..j-1]$ et $y[1..i]$. Ensuite, il suffit de choisir la plus longue des deux, ou l'une quelconque des deux si elles ont la même longueur.
Si on remarque que la plus longue sous-séquence commune à la séquence ε de longueur nulle et toute autre séquence est elle-même ε, on peut déduire la récurrence :

$$
\begin{array}{ll}
lssc(0,j) = 0 & 0 \leqslant j \leqslant m \\
lssc(i,0) = 0 & 1 \leqslant i \leqslant n \\
lssc(i,j) = lssc(i-1,j-1) + 1 & x[j] = y[i] \text{ et } 1 \leqslant i \leqslant n \text{ et } 1 \leqslant j \leqslant m \\
lssc(i,j) = \max\left(\left\{\begin{array}{l} lssc(i,j-1), \\ lssc(i-1,j) \end{array}\right\}\right) & x[j] \neq y[i] \text{ et } 1 \leqslant i \leqslant n \text{ et } 1 \leqslant j \leqslant m.
\end{array}
$$

Question 3. Proposer une évolution du calcul de la récurrence précédente, puis écrire l'algorithme de programmation dynamique appelé *WFlg* qui, étant donné deux séquences x et y de longueurs respectives m et n, calcule la longueur de leur(s) plus longue(s) sous-séquence(s) commune(s) et permet de retrouver une des plus longues sous-séquences communes. Préciser où se trouve la longueur de la solution optimale calculée. Quelles sont les complexités spatiale et temporelle de cet algorithme en prenant l'évaluation de conditions comme opération élémentaire ? Traiter en exemple les séquences $u = abcbdab$ et $v = bdcaba$.

Réponse 3. On va remplir une structure tabulaire $LSSC[0..n, 0..m]$ en lien avec les m symboles de x et les n symboles de y. La longueur d'une des plus longues sous-séquences communes aux séquences x et y se trouvera dans la cellule $LSSC[n, m]$.

Les deux premiers termes de la récurrence se traduisent par l'initialisation à 0 de la ligne et de la colonne d'indice 0 de LSSC. On observe que tout autre élément $LSSC[i, j]$ avec i et j strictement positifs dépend soit de son voisin de la colonne précédente et de la ligne précédente selon le dernier terme de la récurrence, soit de son voisin de la diagonale précédente selon le troisième terme de la récurrence. On peut donc procéder à un calcul colonne par colonne et de bas en haut dans chaque colonne. Il est à noter que l'on pourrait tout aussi bien procéder à un remplissage par ligne ou par diagonale et que ces trois types de remplissage sont les plus communs pour les tableaux à deux dimensions.

Pour retrouver une des plus longues sous-séquences communes par la méthode du « Petit Poucet » (semant des cailloux pour retrouver son chemin), on double le tableau $LSSC[0..n, 0..m]$ par un tableau $CH[0..n, 0..m]$ de « cailloux blancs » construit en parallèle. Chaque case de ce tableau contient une information (1, 2 ou 3) indiquant laquelle des trois possibilités de la récurrence a créé la valeur retenue dans l'élément de LSSC de mêmes indices. Parcourir CH à partir de $CH[n, m]$ permet de retrouver de manière déterministe une plus longue sous-séquence commune à x et y en allant jusqu'à $CH[0, 0]$ (ou toute case d'indice ligne ou colonne égal à 0). S'il y plusieurs possibilités, la sous-séquence retenue dépend de l'ordre des tests effectués dans le programme. On déduit de ce qui précède l'algorithme *WFlg* ci-dessous, qui affiche la valeur de $lssc(n, m)$:

```
1.  constantes
2.     x ∈ chaîne(Σ) et x = ... et y ∈ chaîne(Σ) et y = ... et m = |x| et n = |y|
3.  variables
4.     LSSC ∈ 0 .. n × 0 .. m → ℕ et CH ∈ 0 .. n × 0 .. m → {1, 2, 3}
5.  début
6.     pour j ∈ 0 .. m faire
7.        LSSC[0, j] ← 0 ; CH[0, j] ← 2
8.     fin pour ;
9.     pour i ∈ 1 .. n faire
10.       LSSC[i, 0] ← 0 ; CH[i, 0] ← 3
11.    fin pour ;
12.    pour j parcourant 1 .. m faire
13.       pour i parcourant 1 .. n faire
14.          si x[j] = y[i] alors
15.             LSSC[i, j] ← LSSC[i − 1, j − 1] + 1 ; CH[i, j] ← 1
16.          sinon
17.             si LSSC[i − 1, j] > LSSC[i, j − 1] alors
18.                LSSC[i, j] ← LSSC[i − 1, j] ; CH[i, j] ← 3
19.             sinon
20.                LSSC[i, j] ← LSSC[i, j − 1] ; CH[i, j] ← 2
21.             fin si
22.          fin si
23.       fin pour
24.    fin pour ;
25.    écrire(LSSC[n, m])
26. fin
```

La structure tabulaire utilisée requiert $(n+1)$ lignes et $(m+1)$ colonnes ; on a donc une complexité spatiale en $\Theta(m \cdot n)$. Le nombre de comparaisons effectuées est au minimum de $m \cdot n$ et au maximum de $2 \cdot m \cdot n$ (en négligeant le contrôle des boucles). La complexité temporelle est donc elle aussi en $\Theta(m \cdot n)$.

6	*a*	0 ↓	1 ↙	2 ↓	2 ←	3 ↓	3 ←	4 ↙	4 ←
5	*b*	0 ↓	1 ↓	2 ↙	2 ←	3 ↙	3 ←	3 ←	4 ↙
4	*a*	0 ↓	1 ↙	1 ←	2 ↓	2 ←	2 ←	3 ↙	3 ←
3	*c*	0 ↓	0 ←	1 ↓	2 ↙	2 ←	2 ←	2 ←	2 ←
2	*d*	0 ↓	0 ←	1 ↓	1 ←	1 ←	2 ↙	2 ←	2 ←
1	*b*	0 ↓	0 ←	1 ↙	1 ←	1 ↙	1 ←	1 ←	1 ↙
0	ε	0 ←	0 ←	0 ←	0 ←	0 ←	0 ←	0 ←	0 ←
i	v/u	ε	*a*	*b*	*c*	*b*	*d*	*a*	*b*
	j	0	1	2	3	4	5	6	7

La plus longue sous-séquence commune à $u = abcbdab$ et $v = bdcaba$ trouvée par l'algorithme précédent est $bcba$. Il y en a aussi deux autres : $bcab$ et $bdab$. Le tableau de calcul est donné ci-dessus. La récupération de la plus longue sous-séquence commune se fait en partant de la cellule nord-est du tableau. On en suit les indications de cheminement ($\swarrow$ (resp. $\leftarrow, \downarrow$) correspondant à 1 (resp. 2, 3) dans l'algorithme), ce qui conduit au chemin dont les cellules sont grisées. En conservant les valeurs des symboles associés aux cellules comportant l'indication $\swarrow$, on obtient $abcb$ qui est la chaîne miroir de celle recherchée.

Question 4. Expliquer pourquoi on peut envisager une version de l'algorithme précédent de complexité spatiale moindre, pour autant que l'on ne soit intéressé que par la longueur de la plus longue sous-séquence commune et non par la reconstitution de la sous-séquence associée. Écrire le code de l'opération « **procédure** $WFLgAvant(x, y; P : \mathrm{modif})$ » fournissant dans le vecteur P la longueur de la plus longue sous-séquence commune à $x[0\,..\,m]$ et $y[0\,..\,i]$ pour $i \in 0\,..\,n$. Préciser sa complexité spatiale.

Réponse 4. Comme il a été mentionné, le calcul associé à la cellule (i, j) du tableau LSSC ne fait appel qu'au contenu de ses trois voisines $\mathrm{LSSC}[i-1, j-1]$, $\mathrm{LSSC}[i-1, j]$ et $\mathrm{LSSC}[i, j-1]$. On peut donc faire un calcul utilisant un tableau à deux colonnes, $h[0\,..\,n, 0\,..\,1]$. Le code de la procédure *WFLgAvant* associée est le suivant :

```
1.  procédure WFLgAvant(x, y; P : modif) pré
2.     x ∈ chaîne(Σ) et y ∈ chaîne(Σ) et const m = |x| et const n = |y| et
3.     P ∈ 0 .. n → ℕ et h ∈ 0 .. n × 0 .. 1 → ℕ
4.  début
5.     pour i ∈ 0 .. n faire
6.        h[i, 1] ← 0
7.     fin pour
8.     pour j parcourant 1 .. m faire
9.        pour i ∈ 0 .. n faire
10.          h[i, 0] ← h[i, 1]
11.       fin pour ;
12.       pour i parcourant 1 .. n faire
13.          si x[j] = y[i] alors
14.             h[i, 1] ← h[i − 1, 0] + 1
15.          sinon
16.             h[i, 1] ← max({h[i − 1, 1], h[i, 0]})
17.          fin si
18.       fin pour
19.    fin pour
20.    pour k ∈ 0 .. n faire
21.       P[k] ← h[k, 1]
22.    fin pour
23. fin
```

Grâce à l'utilisation des tableaux h et P, la complexité spatiale de cette procédure est en $\Theta(n)$.

9.3 Ce qu'il faut retenir pour appliquer la programmation dynamique

Comme pour les autres démarches vues auparavant, la construction d'un algorithme de programmation dynamique s'inscrit dans une démarche méthodologique qui en guide l'élaboration. Celle-ci peut être synthétisée par les étapes suivantes :

1. identifier la grandeur numérique g qui va être calculée (celle-ci peut « s'écarter » quelque peu de celle apparaissant initialement dans le problème, voir par exemple les exercices 116 page 331, et 135 page 364),
2. établir une relation de récurrence permettant le calcul de g qui soit complète, c'est-à-dire qui couvre toutes les valeurs admissibles des indices qu'elle comporte,
3. définir la structure tabulaire T associée au calcul (dimensions et taille de chacune d'elles) et vérifier que la solution au problème posé s'y trouvera (l'emplacement de la valeur optimale est généralement connu *a priori*, mais il arrive que ce ne soit pas le cas – voir le problème de la pyramide – et il est alors déterminé au moyen d'un traitement complémentaire),
4. déterminer une évolution du calcul de g qui garantisse la compatibilité avec la programmation dynamique, c'est-à-dire assurant que le calcul d'un élément ne fait appel qu'à des valeurs déjà calculées,
5. si besoin, introduire les éléments nécessaires au calcul de la solution optimale elle-même (« Petit Poucet ») en complétant la structure tabulaire T, afin d'enregistrer le choix permettant d'obtenir la valeur optimale pour chaque cellule de T (le principe de l'algorithme exploitant ensuite cette information afin de construire la (une) solution optimale doit être explicité),
6. préciser la complexité spatiale (liée à la structure tabulaire) et temporelle de la solution proposée (quand cette dernière est exprimée en termes de comparaisons, ce qui est le plus fréquent, les conditions relevant du contrôle des boucles qui n'introduisent qu'un facteur multiplicatif ne seront pas prises en compte),
7. le cas échéant, procéder à des améliorations (limitation du calcul aux seuls éléments nécessaires, réduction de la taille (voire suppression) de la structure tabulaire, etc...),
8. produire le code associé.

9.4 Exercices

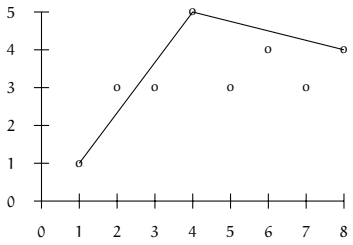
Les exercices qui suivent proposent une variété de sujets pour lesquels la programmation dynamique se révèle pertinente. Ils ont été classés selon différents thèmes : découpe ou partage, problèmes relatifs aux séquences, arbres ou graphes, problèmes liés aux images, jeux, et enfin l'illustration d'un problème pseudo-polynomial. Compte tenu du caractère systématique de la production de l'algorithme à partir de la récurrence et de la stratégie de remplissage de la structure tabulaire choisie, le code de l'algorithme n'est demandé que de façon occasionnelle quand un intérêt particulier le justifie. Enfin, sauf mention contraire explicite, la complexité temporelle se mesure en termes de nombre de conditions évaluées en lien avec les opérations « minimum » et « maximum » de la récurrence.

9.4.1 Découpe - Partage : problèmes à une dimension

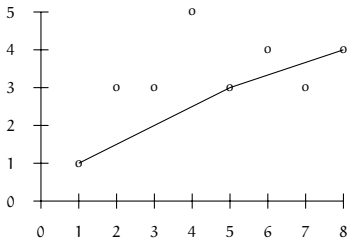
Exercice 115. Approximation d'une fonction échantillonnée par une ligne brisée ⁘ •

Cet exercice illustre la notion d'approximation optimale au moyen de la programmation dynamique. Il montre la réduction très importante que cette méthode apporte au plan de la complexité par rapport à une approche naïve de type « Essais Successifs ».

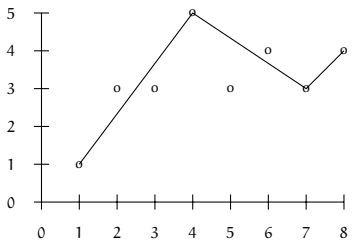
Soit un ensemble P de n points du plan, avec $n \geqslant 1$, dont les abscisses valent $1, 2, \ldots, n$ et dont les ordonnées sont des entiers naturels quelconques. On cherche la meilleure approximation de cet ensemble par une ligne brisée définie comme une suite de segments de droite dont les extrémités sont des points de P. Elle peut se représenter par la suite des abscisses des points sur lesquels elle s'appuie. On impose que le premier nombre vaille 1 et le dernier n, autrement dit que le premier point de P soit le départ du premier segment de droite et son dernier point l'arrivée du dernier segment de droite. Dans les quatre exemples ci-après, les lignes brisées sont successivement : $(1,4,8)$, $(1,5,8)$, $(1,4,7,8)$ et $(1,3,4,8)$.



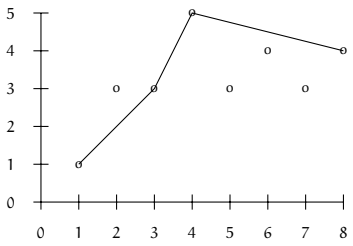
Exemple 1 : $n = 8$, $m = 2$
ligne brisée $(1,4,8)$



Exemple 2 : $n = 8$, $m = 2$
ligne brisée $(1,5,8)$



Exemple 3 : $n = 8$, $m = 3$
ligne brisée $(1,4,7,8)$



Exemple 4 : $n = 8$, $m = 3$
ligne brisée $(1,3,4,8)$.

La qualité de l'approximation de P par une ligne brisée se mesure *pour partie* en calculant la somme sde des distances euclidiennes de chaque point de P au segment de droite dont les extrémités l'encadrent. Par suite, tout point origine ou extrémité d'un segment de l'approximation induit une distance nulle. Pour la première figure, cette somme est composée de deux parties :

— la distance des points $1, 2, 3$ et 4 au segment construit sur les points 1 et 4, soit $(0 + 0.4 + 0.4 + 0) = 0.8$,

— la distance des points $4, 5, 6, 7$ et 8 au segment construit sur les points 4 et 8, soit environ $(0 + 1.75 + 0.5 + 1.25 + 0) = 3.5$.

On ajoute à cette somme un terme positif $(m - 1) \cdot C$, proportionnel au nombre m de segments de droite de la ligne brisée. Dans l'exemple de la première figure, si l'on choisit $C = 2$, ce terme vaut 2, puisque $m = 2$. L'approximation réalisée est d'autant meilleure que le coût $(sde + (m - 1) \cdot C)$ est petit.

Dans le premier exemples, ce coût vaut donc à peu près $0.8 + 3.5 + 2 = 6.3$. On calcule de la même manière, pour l'exemple 3, $sde = (0 + 0.4 + 0.4 + 0) + (0 + 1.1 + 0.3 + 0) + (0 + 0) = 2.2$; le coût de l'approximation est $2.2 + 2 \cdot 2 = 6.2$. Cette seconde approximation est donc un peu meilleure que la précédente. Les second et quatrième exemples conduisent à des approximations de coût respectif, environ 7.9 et 8.1, moins bonnes que la première. La meilleure des quatre approximations considérées est donc la troisième. Cependant, l'approximation constituée des deux segments $(1, 2)$ et $(2, 8)$ fait encore mieux que celle-ci, puisqu'elle a un coût d'environ 5.4.

Étant donnés un ensemble P de n points et C, le problème général posé est de trouver la ligne brisée optimale, c'est-à-dire celle qui minimise le coût $(sde + m \cdot C)$.

115 - Q 1 **Question 1.** Que peut-on dire des approximations quand $n \cdot C$ est très petit (et donc C aussi) ou C très grand ?

115 - Q 2 **Question 2.** Supposons que la ligne brisée de valeur optimale corresponde à la suite croissante d'abscisses $(a_1, a_2, \ldots, a_{m-1}, a_m)$, avec $a_1 = 1$ et $a_m = n$. Notons $appopt(i)$ la valeur optimale que l'on obtient pour approximer l'ensemble des points d'abscisses $(1, \ldots, i)$ par une ligne brisée. Maintenant, notons $k = a_{m-1}$ l'abscisse de départ du dernier segment de la ligne brisée optimale pour l'ensemble P des n points et $sde(k, n)$ la somme des distances des points d'abscisses $(k, \ldots, n)$ à la droite passant par les deux points de P ayant pour abscisses k et n. Montrer que $appopt(n) = sde(k, n) + C + appopt(k)$. Comment calculer $appopt(n)$ sans connaître k, mais en supposant connus $appopt(1) = 0$, $appopt(2)$, ..., $appopt(n - 1)$?

115 - Q 3 **Question 3.** Établir une formule de récurrence permettant de calculer $appopt(i)$ $(1 \leqslant i \leqslant n)$ en fonction de $appopt(1), \ldots, appopt(i - 1)$, de $sde(j, i)$ et de C.

115 - Q 4 **Question 4.** On suppose qu'il existe une fonction *Distance*(j, i, k), avec $j \leqslant k \leqslant i$, qui calcule la distance du point d'abscisse k à la droite construite sur les points j et i. Quels sont les calculs à faire, et dans quel ordre, pour calculer $appopt(n)$? Où le résultat recherché se trouve-t-il dans la structure tabulaire utilisée ?

115 - Q 5 **Question 5.** En déduire un algorithme qui calcule la valeur $appopt(n)$ de l'approximation optimale d'un ensemble P quelconque de n points par une ligne brisée. Quel est sa complexité temporelle en nombre d'appels à la fonction *Distance* ? Que faut-il en penser ?

115 - Q 6 **Question 6.** Comment faudrait-il modifier cet algorithme pour qu'il produise aussi les abscisses des points de la ligne brisée optimale ?

Exercice 116. Le meilleur intervalle (le retour) ○ •

Cet exercice propose une solution alternative à celle élaborée par la démarche DpR dans l'exercice 97, page 259. Il illustre un cas limite (rare) où il existe une solution ne nécessitant aucune structure tabulaire de mémorisation, et donc de complexité spatiale constante.

Rappelons le problème, déjà présenté dans l'exercice 97, page 259. On dispose d'un tableau $T[1\,..\,n]$ $(n \geqslant 1)$ de valeurs positives réelles. Il existe (au moins) deux indices i et j, définissant l'intervalle $i..j$, avec $1 \leqslant i \leqslant j \leqslant n$, tels que la valeur $T[j]-T[i]$ est maximale. On cherche cette valeur maximale appelée valeur du (d'un) meilleur intervalle. Par exemple, si $T = [9, 15, 10, 12, 8, 18, 20, 7]$, le meilleur intervalle de valeur 12 est unique et obtenu pour $i = 5$ et $j = 7$.

Remarque Si le tableau est monotone décroissant, cette valeur est nulle et correspond à n'importe quel intervalle de type $i\,..\,i$ pour $1 \leqslant i \leqslant n$.

On a trouvé une solution de complexité linéaire (en termes de conditions évaluées) dans l'exercice 97 page 259 et l'on souhaite en élaborer une en programmation dynamique de même ordre de complexité si possible (tenter de faire mieux est illusoire).

Question 1. On appelle $\nu mi(k)$ la valeur du (d'un) meilleur intervalle se terminant *exactement* en position k. Préciser comment obtenir la valeur du (d'un) meilleur intervalle du tableau T à partir de $\nu mi(1), \ldots, \nu mi(n)$. Établir une relation de récurrence permettant le calcul de $\nu mi(k)$. 116 - Q 1

Question 2. Déduire de la récurrence précédente une stratégie de remplissage d'une structure tabulaire appropriée. Quelles en sont les complexités temporelle et spatiale ? Vérifier que la complexité temporelle répond bien à l'objectif fixé. 116 - Q 2

Question 3. Comment peut-on procéder pour déterminer le meilleur intervalle lui-même, autrement dit ses bornes ? 116 - Q 3

Question 4. Donner le code du programme associé. 116 - Q 4

Question 5. Appliquer cet algorithme au tableau d'entiers $T[1\,..\,14] = [14, 11, 16, 12, 20, 7, 3, 3, 19, 24, 24, 3, 5, 16]$. 116 - Q 5

Question 6. Proposer une variante (principe et code) de la solution précédente qui améliore la complexité spatiale. 116 - Q 6

Question 7. Comparer les deux solutions pour ce problème : a) version DpR de l'exercice 97, page 259, et b) celle donnée en réponse à la question précédente. 116 - Q 7

Exercice 117. Installation de stations-service ○ •

Cet exercice illustre un problème assez simple pour lequel la valeur associée à la solution optimale recherchée se trouve dans un emplacement prédéterminé de la structure tabulaire. Une attention particulière est portée à l'étape de reconstitution de la solution optimale elle-même.

On obtient la concession de stations-service le long d'une autoroute en construction. La règle est la suivante : la compagnie gestionnaire de l'autoroute indique les n emplacements envisagés pour implanter les stations-service. Chaque emplacement est numéroté par un entier i, et sa position est donnée par son kilométrage à partir de l'entrée de l'autoroute. Pour chaque emplacement possible, la compagnie donne le montant que va rapporter chaque année une station-service placée à cet endroit (en M€). Elle indique également la « distance à préserver » en kilomètres au sens suivant : si l'on décide d'installer une station-service à un emplacement, on n'a pas le droit d'en installer une autre (en direction de l'entrée de l'autoroute) à une distance inférieure ou égale à la distance à préserver. On suppose que les emplacements sont numérotés par distance croissante par rapport à l'entrée de l'autoroute. Par exemple, avec les données suivantes :

Emplacement	Position	Gain annuel	Distance à préserver
1	40	5	0
2	90	7	70
3	130	1	50
4	200	5	120

si l'on choisit d'installer une station à l'emplacement 4, les emplacements 3 et 2 ne peuvent plus être équipés, puisque ces stations-service seraient à une distance inférieure à 120 km. On peut donc seulement réaliser les combinaisons suivantes d'emplacements pour installer les stations-service : $\langle 1\rangle$, $\langle 2\rangle$, $\langle 3\rangle$, $\langle 4\rangle$, $\langle 1,3\rangle$ et $\langle 1,4\rangle$.

117 - Q 1 **Question 1.** Parmi les six configurations possibles de cet exemple, quelle est celle qui rapporte le plus ?

Le problème est de placer les stations-service parmi les n emplacements envisagés, de façon à ce que le rapport (gain) soit maximal. On note $\mathrm{gopt}(i)$ le gain maximal que peuvent rapporter des stations-service si l'on considère seulement les emplacements 1 à i ($\mathrm{gopt}(n)$ est par suite la valeur finale recherchée). Par ailleurs, $e(i)$ désigne le numéro de l'emplacement le plus proche (dans la direction de l'entrée) où il est possible d'implanter une station-service s'il y en a une à l'emplacement i (compte tenu de la distance à préserver). Si on ne peut mettre aucune station-service avant celle en position i, $e(i)$ vaut 0. Enfin, on note $g(i)$ le gain annuel de la station-service implantée à l'emplacement i.

117 - Q 2 **Question 2.** Expliquer pourquoi on doit introduire un emplacement « virtuel » de numéro 0 et préciser la valeur de $g(0)$.

117 - Q 3 **Question 3.** Établir la récurrence de calcul de $\mathrm{gopt}(i)$, gain maximal associé à une implantation optimale relative aux emplacements 0 à i.

Question 4. Préciser les principaux éléments du programme implantant cette récurrence (structure tabulaire et progression de son remplissage, localisation de la valeur optimale recherchée, complexités spatiale et temporelle en nombre de conditions évaluées). 117 - Q 4

Question 5. Expliciter le principe de reconstitution de la configuration optimale. 117 - Q 5

Question 6. L'appliquer à l'exemple : 117 - Q 6

Emplacement	Position	Gain annuel	Distance à préserver
1	20	6	0
2	80	7	70
3	170	2	100
4	200	3	50
5	260	1	80
6	280	5	100
7	340	2	90

Exercice 118. Le voyageur dans le désert ○ •

Cet exercice montre que, selon la fonction de coût considérée, une simple solution gloutonne (voir chapitre 7) suffit ou qu'il faut, au contraire, recourir à une solution faisant appel à une récurrence, demandant donc une spécification un peu plus élaborée. Ici encore, le coût de la solution optimale se trouve à un emplacement prédéterminé de la structure tabulaire.

Un voyageur veut aller d'une oasis à une autre sans mourir de soif. Il connaît la position des puits sur la route (numérotés de 1 à n, le puits numéro 1 (resp. n) étant l'oasis de départ (resp. d'arrivée)). Le voyageur sait qu'il consomme exactement un litre d'eau au kilomètre. Il est muni d'une gourde pleine à son départ. Quand il arrive à un puits, il choisit entre deux possibilités : a) poursuivre sa route, ou b) remplir sa gourde. S'il fait le second choix, il vide sa gourde dans le sable avant de la remplir entièrement au puits afin d'avoir de l'eau fraîche. À l'arrivée, il vide la gourde.

Question 1. Le voyageur veut faire le moins d'arrêts possible. Mettre en évidence une stratégie gloutonne (voir chapitre 7) optimale atteignant cet objectif. 118 - Q 1

Question 2. Le voyageur veut verser dans le sable le moins de litres d'eau possible. Montrer que la stratégie gloutonne précédente est toujours optimale. 118 - Q 2

Question 3. À chaque puits, y compris celui de l'oasis d'arrivée, un gardien lui fait payer autant d'unités de la monnaie locale que le carré du nombre de litres d'eau qu'il vient de verser à l'arrivée du tronçon qu'il a parcouru. Le problème est de choisir les puits où il doit s'arrêter pour payer le moins possible. Montrer avec les données de l'exemple de la question 5 que la stratégie gloutonne précédente n'est plus optimale. 118 - Q 3

Question 4. Construire une solution fondée sur la programmation dynamique dont les éléments sont : 118 - Q 4

- popt(i) : somme minimale payée au total depuis le puits numéro 1 (l'oasis de départ) jusqu'au puits numéro i, étant donné que le voyageur vide sa gourde au puits numéro i,
- d(i, j) : nombre de kilomètres entre le puits numéro i et le puits numéro j,
- D : volume de la gourde

dont on donnera la récurrence, la structure tabulaire utilisée, l'évolution de son remplissage et les complexités temporelle (en nombre de conditions évaluées) et spatiale du programme qui en résulte (le code n'est pas demandé).

118 - Q 5 **Question** 5. Appliquer cette solution avec une gourde de dix litres et des puits situés à 8, 9, 16, 18, 24 et 27 km de l'oasis de départ, l'arrivée étant située à 32 km de l'oasis de départ.

Exercice 119. Formatage d'alinéa ○ •

Cet exercice illustre une application de la programmation dynamique à un problème simple de formatage de texte pour lequel on veut minimiser un coût associé aux espaces apparaissant dans les lignes. On pourra noter de fortes analogies avec l'exercice précédent, en particulier dans la forme de la récurrence.

Pour un logiciel de traitement de texte, on cherche à disposer la suite des mots qui forment un alinéa de manière à répartir au mieux les espaces dans un sens qui sera précisé ultérieurement. Pour simplifier, on considère un texte sans signes de ponctuation, donc constitué uniquement de mots (suites de lettres) et d'espaces. On se donne les règles suivantes :

- chaque mot – insécable – a une longueur égale à son nombre de caractères et inférieure ou égale à celle d'une ligne,
- les mots d'une ligne sont séparés par une espace,
- toute ligne commence par un mot calé à gauche,
- chaque espace a la longueur d'une lettre,
- on ne peut pas dépasser la longueur d'une ligne.

Par exemple, pour des lignes de taille égale à 26 caractères, on a les deux possibilités suivantes, parmi un grand nombre (les espaces sont représentées par le caractère « = ») :

```
Ce=court=texte=est=formaté  | Ce=court=texte=est========
de=deux=façons============  | formaté=de=deux=façons====
différentes===============  | différentes===============
```

Le premier formatage compte quatre espaces sur la première ligne, 14 sur la seconde ligne et 15 sur la troisième. Le second compte 11 espaces sur la première ligne, sept sur la seconde et 15 sur la dernière ligne. Ils comptent tous les deux 33 espaces, et l'on souhaite

départager ce genre d'égalité. À cette fin, on mesure la qualité d'un formatage par son *coût* cf donné comme la somme des carrés du nombre total d'espaces sur chacune des lignes. Le coût du premier formatage vaut $4^2+14^2+15^2=437$, celui du second $11^2+7^2+15^2=395$ et, en conséquence, le second est meilleur que le premier.

Le but de cet exercice est de trouver, pour un texte et une longueur de ligne donnés, un (le) formatage de coût minimal.

Question 1. Que dire de deux formatages d'un même texte utilisant le même nombre de lignes, si l'on prend comme coût cf' le nombre d'espaces et non pas cf ? 119 - Q 1

Question 2. On considère les deux formatages suivants (la longueur de ligne est égale à 26) du texte : « `Ce court texte est formaté de deux façons très différentes` ». 119 - Q 2

`Ce=court=texte=est=formaté`	`Ce=court=texte=est========`
`de=deux=façons=très=======`	`formaté=de=deux=façons====`
`différentes===============`	`très=différentes==========`

Que dire d'un algorithme glouton (voir chapitre 7) consistant à remplir chaque ligne le plus possible ?

Question 3. On prend des lignes de longueur 10. Donner toutes les façons de formater la phrase « `Racine est un musicien` ». 119 - Q 3

Question 4. On considère les N mots $m_1,\dots,m_N$ du texte à formater, leur longueur $lg(m_1),\dots,lg(m_N)$ et L la longueur de la ligne. On note $ml(i,j)$ le coût qui résulte de l'écriture des mots $m_i,\dots,m_j$ sur la même ligne. Compte tenu des principes évoqués précédemment, on distingue deux cas : 119 - Q 4

- on peut mettre $m_i,\dots,m_j$ sur une ligne ($\sum_{k=i}^{j} lg(m_k)+(j-i)\leqslant L$), alors $ml(i,j)=(L-\sum_{k=i}^{j} lg(m_k))^2$, le carré du nombre total d'espaces de la ligne,
- les mots $m_i,\dots,m_j$ ne tiennent pas sur une ligne ($\sum_{k=i}^{j} lg(m_k)+(j-i)>L$), alors $ml(i,j)$ prend une valeur arbitrairement grande, dénotée $+\infty$.

On appelle $fopt(i)$ le coût optimal de l'écriture des mots $m_i,\dots,m_N$ sous la contrainte que m_i soit au début d'une ligne. Donner la formule de récurrence qui calcule $fopt(i)$.

Question 5. En déduire le principe d'un programme qui, connaissant les valeurs de ml, calcule le formatage d'un texte de coût minimal et permet de l'écrire ultérieurement. On en précisera les complexités spatiale et temporelle. 119 - Q 5

Question 6. Traiter l'exemple de la question 3. 119 - Q 6

Question 7. Dans la démarche proposée, on est parti de $fopt(i)$, le coût optimal de l'écriture de $m_i,\dots,m_N$. Aurait-on pu procéder autrement ? 119 - Q 7

Question 8. On pourrait faire d'autres choix concernant la comptabilisation des espaces, par exemple prendre pour une ligne la somme des carrés des nombres d'espaces. Quel impact cela aurait-il sur la solution proposée précédemment ? 119 - Q 8

Exercice 120. Codage optimal

Cet exercice illustre une application de compression optimale de texte étant données des séquences de symboles définissant le code employé. L'établissement de la récurrence est un peu plus ardu que dans les exercices précédents. De plus, on s'intéresse ici à l'explicitation de la procédure de reconstitution du codage optimal.

On dispose d'un ensemble $\mathcal{C}$ de m mots sur un alphabet Σ, tous de longueur inférieure ou égale à k ($\mathcal{C}$ est appelé le *code*). On a d'autre part un autre mot D de longueur n sur l'alphabet Σ, que l'on cherche à coder en utilisant le moins possible d'occurrences de mots de $\mathcal{C}$. Par exemple, si $\mathcal{C} = \{a, b, ba, abab\}$ et $D = bababbaababa$, un codage possible de D est $ba\ ba\ b\ ba\ abab\ a$, utilisant six occurrences de $\mathcal{C}$. Il peut ne pas exister de solution, comme pour le codage de $D = abbc$ avec le code $\mathcal{C} = \{a, bc\}$. Une condition suffisante pour que toute chaîne puisse être codée (et donc admette un codage optimal) est que Σ soit inclus (au sens large) dans $\mathcal{C}$.

120 - Q 1 **Question 1.** Donner une récurrence pour calculer le nombre minimum d'occurrences de mots du code $\mathcal{C}$ nécessaires au codage de D. Par convention, ce nombre minimum vaut $+\infty$ si D ne peut être codé avec $\mathcal{C}$.

120 - Q 2 **Question 2.** L'opération élémentaire étant la comparaison des lettres de l'alphabet Σ, écrire un algorithme en $\mathcal{O}(n \cdot m \cdot k)$ qui trouve le coût du codage optimal et permette de le produire ultérieurement (s'il existe).

120 - Q 3 **Question 3.** Expliciter l'algorithme permettant de reconstituer le codage optimal quand il existe.

120 - Q 4 **Question 4.** Appliquer cet algorithme pour : i) le code $\mathcal{C} = \{a, b, ba, abab\}$ et le mot $D = bababbaababa$, ii) le code $\mathcal{C} = \{a, bc\}$ et le mot $D = abbc$.

Exercice 121. Découpe d'une barre ◦ •

Le seul point de cet exercice méritant une attention particulière concerne l'établissement des récurrences.

Étant données une barre de métal de longueur n (centimètres) et une table des prix de vente unitaires PU croissants des segments de métal pour les longueurs $i = 1, \ldots, n$, on cherche à découper la barre en segments de façon à maximiser son prix de vente. Le tableau qui suit constitue un exemple de table de prix de vente pour des longueurs allant de 1 à 7.

Longueur i du segment	1	2	3	4	5	6	7
Prix de vente PU[i]	3	7	10	13	16	20	24

121 - Q 1 **Question 1.** Sur cet exemple, quelle est la découpe optimale d'une barre de longueur 4 ?

Question 2. Donner une formule de récurrence pour le prix optimal $pvopt(n)$ de la découpe d'une barre de longueur n. 121 - Q 2

Question 3. En déduire le principe d'un algorithme de programmation dynamique qui calcule ce prix. En préciser les complexités temporelle (en nombre de conditions évaluées) et spatiale. 121 - Q 3

Question 4. Comment connaître non seulement le prix optimal, mais aussi la longueur des segments qui composent la découpe optimale ? 121 - Q 4

Question 5. Traiter le cas d'une barre de longueur 7 avec le tableau de prix donné auparavant. 121 - Q 5

Question 6. Comment faire qu'en cas de prix de vente optimal identique entre une possibilité avec découpe et une autre sans découpe, cette dernière soit choisie par l'algorithme ? 121 - Q 6

On suppose maintenant que les segments ne peuvent être produits (et vendus) que pour un certain nombre m de longueurs $LG[1] = 1$ à $LG[m] = p$ avec $p > m$, dont on connaît le prix de vente unitaire $PU[i]$ $(1 \leqslant i \leqslant m)$. Par exemple, pour $m = 4$, on aura $LG[1] = 1, LG[2] = 3, LG[3] = 4$, $LG[4] = 6$ et $PU[1] = 3, PU[2] = 8, PU[3] = 13$, $PU[4] = 20$. De façon générale, toute barre de longueur supérieure à p doit être découpée, ce qui est toujours possible puisque $LG[1] = 1$.

Question 7. Donner la nouvelle récurrence de calcul de $pvopt(n)$, le prix de vente optimal d'une barre de longueur n (entier positif) quelconque. Quelle est la complexité temporelle (en nombre de conditions évaluées) de l'algorithme associé ? 121 - Q 7

Question 8. Comment obtenir maintenant la longueur de chacun des segments constituant la (une) découpe optimale ? 121 - Q 8

Question 9. Traiter le cas d'une barre de longueur 11 à découper avec les données suivantes : 121 - Q 9

Numéro i du segment	1	2	3	4	5	6
Longueur LG[i] du segment i	1	2	4	6	7	9
Prix de vente PU[i] du segment i	2	5	11	15	17	24

9.4.2 Découpe - Partage : problèmes à deux dimensions

Exercice 122. Affectation d'effectifs à des tâches ○ ⁝

Le principal intérêt de cet exercice réside dans l'attention qui doit être portée à l'établissement de la récurrence qui révèle quelques particularités.

On considère n tâches $T_1, \ldots, T_n$, à réaliser en parallèle, chacune pouvant être effectuée avec différents effectifs d'employés (nombres entiers de 1 à k). Tout employé est apte à contribuer à toute tâche T_i, mais une fois affecté à l'une d'elles, il s'y consacre jusqu'à son terme et cesse ensuite son activité. La durée de chaque tâche varie en fonction de l'effectif qui lui est affecté ; $d(i, e)$ la durée de la tâche i réalisée avec e employés est exprimée en unités de temps (nombres entiers de 10 à 200). Pour chaque tâche T_i, la durée $d(i, e)$ diminue au fur et à mesure que l'effectif e qui lui est affecté augmente, sauf si la tâche n'est pas réalisable pour l'effectif imparti, auquel cas elle ne l'est pas non plus pour tout effectif supérieur. À titre d'exemple, avec $n = 4$ et $k = 5$, on peut avoir le tableau de durées ci-dessous :

	effectif e	1	2	3	4	5
$i =$	1	110	90	65	55	$+\infty$
	2	120	90	70	50	40
	3	90	70	65	60	$+\infty$
	4	65	60	55	$+\infty$	$+\infty$

la valeur $+\infty$ indiquant que la tâche ne peut pas être réalisée avec cet effectif. Ici, les tâches $1, 3$ et 4 ne peuvent être effectuées avec cinq employés, et la tâche 4 ne peut même pas l'être avec quatre.

Le problème à résoudre consiste à trouver une affectation optimale d'effectif à chacune des tâches considérées au sens suivant : pour un effectif global E fixé disponible pour l'ensemble des tâches à effectuer, e_i désignant l'effectif assigné à la tâche t_i ($E = \sum_{i=1}^{n} e_i$), la somme des durées $SD = \sum_{i=1}^{n} d(i, e_i)$ est minimale. On suppose que l'effectif global E ne permet pas d'attribuer à chaque tâche l'effectif lui assurant la durée minimale, sinon la solution est triviale. Ainsi, dans l'exemple précédent, on supposera $E < 16$. On appelle $sdopt(i, e)$ le coût associé à l'affectation optimale de e employés aux tâches T_1 à T_i.

122 - Q 1 **Question 1.** Donner la récurrence complète de calcul de $sdopt(i, e)$.

122 - Q 2 **Question 2.** Préciser la structure tabulaire utilisée et l'évolution du calcul permettant de la remplir.

122 - Q 3 **Question 3.** Établir que la complexité spatiale de l'algorithme associé est en $\Theta(n \cdot E)$ et que sa complexité temporelle est en $\mathcal{O}(k^2 \cdot n^2)$ conditions évaluées.

122 - Q 4 **Question 4.** Donner le résultat obtenu pour l'exemple donné auparavant en prenant $E = 10$.

Exercice 123. Produit chaîné de matrices

Cet exercice se situe dans le domaine du calcul numérique et résout de façon élégante et efficace la question cruciale du choix de l'ordre dans lequel effectuer une succession de produits de matrices. La solution fait appel à une récurrence à deux éléments, le coût optimal se trouvant dans un emplacement prédéfini de la structure tabulaire utilisée. L'objectif final n'est pas tant de trouver le coût et le parenthésage optimal que de pouvoir utiliser ces éléments pour construire un programme efficace réalisant le produit de matrices considéré.

On s'intéresse au produit des matrices réelles $M_1 \times M_2 \times \cdots \times M_n$, et on souhaite l'obtenir en effectuant le moins possible de multiplications de nombres réels. Les dimensions de ces matrices sont décrites dans un tableau $D[0\,..\,n]$, avec $M_i[1\,..\,D[i-1], 1\,..\,D[i]]$. On appelle $\mathrm{propt}(i,j)$ le nombre minimal de multiplications pour réaliser le produit de la chaîne partielle de matrices $(M_i \times \cdots \times M_{i+j})$. On cherche donc la valeur $\mathrm{propt}(1, n-1)$.

Question 1. Soit les matrices $M_1[10,20]$, $M_2[20,50]$, $M_3[50,1]$ et $M_4[1,100]$. Comparer le nombre de multiplications quand on fait les opérations dans l'ordre donné par le parenthésage $(M_1 \times (M_2 \times (M_3 \times M_4)))$ avec celui qui découle du parenthésage $((M_1 \times (M_2 \times M_3)) \times M_4)$. 123 - Q 1

Question 2. Donner le nombre de parenthésages possibles pour le produit $M_1 \times \cdots \times M_n$. 123 - Q 2

Question 3. Proposer une récurrence de calcul de $\mathrm{propt}(i,j)$. 123 - Q 3

Question 4. Construire le programme calculant $\mathrm{propt}(1,n)$ (et permettant de trouver le parenthésage optimal associé) après avoir mis en évidence la structure tabulaire utilisée et l'évolution de son remplissage. Quel est l'ordre de grandeur de complexité spatiale et temporelle (en nombre de conditions évaluées) de ce programme ? Comparer la complexité temporelle à celle de la méthode naïve comparant tous les parenthésages. 123 - Q 4

Question 5. Appliquer ce programme sur l'exemple des quatre matrices $M_1[10,20]$, $M_2[20,50]$, $M_3[50,1]$ et $M_4[1,100]$ pour calculer le nombre minimal de multiplications nécessaires au produit $M_1 \times M_2 \times M_3 \times M_4$. 123 - Q 5

Question 6. Expliciter le principe du programme reconstituant le (un) parenthésage optimal. L'illustrer avec l'exemple précédent. 123 - Q 6

Exercice 124. Découpe de planche

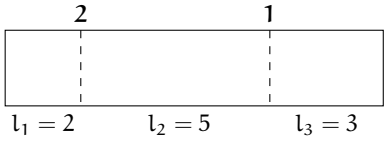
Bien qu'apparemment voisin du problème de découpe de barre, celui-ci se révèle plus compliqué en ce qui concerne l'établissement de la récurrence. Il faut en effet spécifier ici une récurrence à deux variables.

On dispose d'une planche de longueur entière N que l'on veut découper en n segments de longueurs entières $l_1, l_2, \ldots, l_n$ avec $\sum_{i=1}^{n} l_i = N$. Les segments doivent découper la

planche de gauche à droite selon leur indice. Par exemple, si $N = 10$, $l_1 = 2$, $l_2 = 5$ et $l_3 = 3$, les découpes doivent se faire aux abscisses 2 et 7 sur la planche.

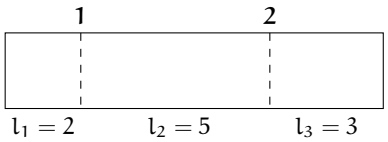
On cherche à minimiser le coût de la découpe, fondé sur le principe suivant : découper en 2 un segment de taille m coûte m unités (il faut transporter le segment de taille m vers la scie). On cherche dans quel ordre pratiquer les découpes pour minimiser le coût total. Dans l'exemple précédent, il n'y a que deux manières de faire :

— Couper d'abord la planche à l'abscisse 7, puis à l'abscisse 2, ce qui coûte $10+7 = 17$ unités. Ceci se représente par le schéma de découpe suivant :



ou par le parenthésage $((l_1\ l_2)\ l_3)$.

— Procéder en sens inverse, ce qui coûte $10+8 = 18$ unités, et se représente par le schéma de découpe :

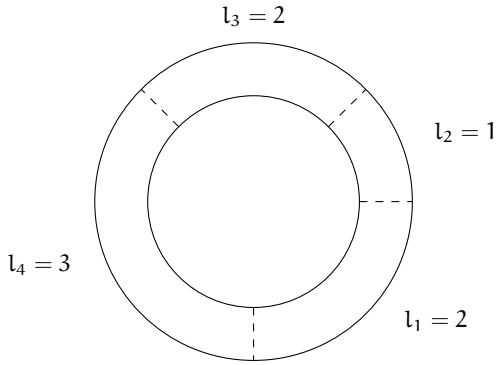


ou par le parenthésage $(l_1\ (l_2\ l_3))$.

124 - Q 1 **Question 1.** Quel est le nombre de découpes distinctes possibles ? À quel autre problème celui-ci fait-il penser en termes de combinatoire ?

124 - Q 2 **Question 2.** Donner la récurrence qui calcule la valeur de la découpe optimale et préciser la complexité de la procédure associée (qui n'est pas demandée). L'appliquer à l'exemple de l'énoncé.

124 - Q 3 **Question 3.** Traiter le problème (récurrence, complexité notamment) quand la planche est circulaire (un beignet plat), comme sur l'exemple ci-dessous :



Exercice 125. Les pilleurs de coffres

L'établissement de la récurrence associée à ce problème repose sur un raisonnement analogue à celui développé dans l'exercice 115 page 329, selon lequel une liste finie non vide possède forcément un dernier élément. Cependant, compte tenu d'une propriété du problème traité ici, on va pouvoir simplifier les calculs, ce qui constitue l'intérêt majeur de cet exercice.

Des voleurs rentrent dans la salle forte de la banque, où les coffres sont alignés le long du mur. Ces voleurs veulent ouvrir tous les coffres en un minimum de temps. Le problème est que les coffres proviennent de fabricants différents, ce qui fait qu'ils ne prennent pas tous le même temps à ouvrir. Chaque coffre est affecté à un voleur, et les voleurs sont tous aussi habiles à la tâche. Ils décident de diviser le mur en secteurs compsés de coffres contigus, et d'affecter un secteur à chacun d'entre eux.

Exemple Pour illustrer, disons qu'il y a trois voleurs et neuf coffres, que l'on peut numéroter de gauche à droite le long du mur et dont le temps d'ouverture, en minutes, se répartit ainsi :

Numéro du coffre	1	2	3	4	5	6	7	8	9
Temps d'ouverture	5	7	3	5	6	3	2	5	3

Question 1. On considère la stratégie consistant à affecter le secteur $(1,2,3)$ au premier voleur, le secteur $(4,5,6)$ au second, et le secteur $(7,8,9)$ au troisième. Les voleurs repartent après 15 minutes, le temps mis par le voleur le plus lent (le premier). Mettre en évidence une solution meilleure que celle qui repose sur cette stratégie. 125 - Q 1

Question 2. Dans le cas général, on a N coffres et p voleurs. On appelle $\mathrm{TOC}(i)$ le temps nécessaire à l'ouverture du i^e coffre par l'un quelconque des voleurs, et $\mathrm{tgopt}(n,k)$ le temps global optimal, c'est-à-dire le temps minimum nécessaire à l'ouverture de n coffres $(1 \leqslant k \leqslant N)$ par k voleurs $(1 \leqslant k \leqslant p)$. Que vaut $\mathrm{tgopt}(n,k)$ quand : i) $n = 1$ (il n'y a qu'un coffre à ouvrir), ii) $k = 1$ (un seul voleur doit ouvrir tous les coffres), iii) $k > n$ (le nombre de voleurs est supérieur au nombre de coffres à ouvrir), iv) $k \leqslant n$? 125 - Q 2

Question 3. En déduire la récurrence permettant le calcul de $\mathrm{tgopt}(N,p)$, le temps minimum nécessaire à l'ouverture de N coffres par p voleurs. 125 - Q 3

Question 4. Décrire la structure tabulaire utilisée par l'algorithme associé, ainsi que la stratégie de son remplissage. Quelles sont les complexités spatiale et temporelle de cet algorithme ? 125 - Q 4

Question 5. L'appliquer à l'exemple. 125 - Q 5

Exercice 126. Trois problèmes d'étagères

On s'intéresse à trois problèmes de rangement de livres dans une étagère, avec des contraintes différentes de rangement selon que l'on fixe le nombre de rayons ou la hauteur ou encore la largeur de l'étagère.

Une étagère sert à ranger des livres d'épaisseurs et hauteurs variées. La profondeur des livres, ainsi que celle de l'étagère, ne jouent aucun rôle ici, elles sont donc ignorées par la suite. L'étagère est constituée d'un ou plusieurs rayons de hauteur fixe ou variable ayant une certaine largeur, sur lesquels sont posés les livres. Pour simplifier, on admet que les planches qui forment les rayons ont une épaisseur nulle. Le rayon du haut est surmonté d'une planche qui définit la hauteur totale de l'étagère.

On va considérer différents problèmes de rangement selon les paramètres fixés au départ (nombre et hauteur(s) des rayons, largeur de l'étagère, ordre des livres en particulier). Dans la première partie, on se donne N livres $B_1, \ldots, B_N$ de différentes hauteurs à ranger dans cet ordre, une largeur donnée de rayon L, et l'on cherche le nombre de rayons (et la hauteur totale minimale) de l'étagère permettant de ranger tous les livres. Dans la partie suivante, on cherche à nouveau à ranger l'intégralité des livres $B_1, \ldots, B_N$ dans cet ordre, dans un nombre fixé K de rayons. Ceux-ci ont même hauteur, ainsi que les livres, et l'on cherche la largeur minimale de l'étagère (et de ses rayons). Dans la troisième et dernière partie, on considère N livres $B_1, \ldots, B_N$ de même hauteur et une étagère ayant K rayons de largeur L donnée. Le problème est ici de déterminer le nombre maximal de livres qui peuvent être rangés en respectant leur ordre.

L'étagère de hauteur minimale

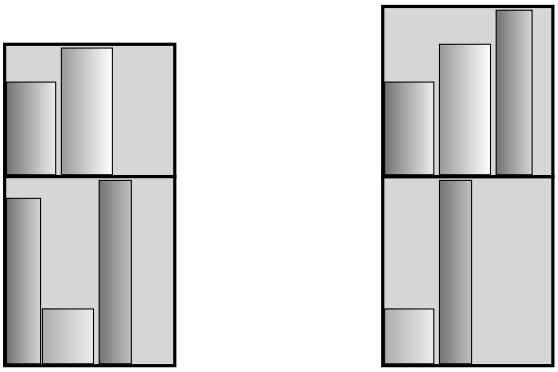
On dispose de N livres $B_1, \ldots, B_N$ que l'on souhaite ranger sur l'étagère avec les contraintes suivantes :

- l'ordre dans lequel les livres sont disposés est fixé : B_1 doit se trouver à l'extrême gauche dans le rayon du haut, B_2 est accolé à B_1 ou, à défaut, à l'extrême gauche du rayon du dessous, et ainsi de suite,
- l'étagère (et donc chacun de ses rayons) a une largeur fixe L, mais le nombre de rayons est réglable de même que la hauteur de chaque rayon,
- chaque livre B_i est caractérisé par sa hauteur h_i et son épaisseur e_i.

Avec des livres d'épaisseur e_i et de hauteur h_i suivantes :

i	1	2	3	4	5
e_i	3	3	2	3	2
h_i	5	7	9	3	10

la figure ci-après donne deux façons différentes de ranger $N = 5$ livres sur une étagère de largeur $L = 10$.



La hauteur totale de la première étagère est $7 + 10 = 17$, celle de la seconde $9 + 10 = 19$.

Le problème est de trouver comment disposer les rayons de l'étagère pour qu'elle ait une hauteur totale minimale (peu importe le nombre de rayons).

Question 1. On note $hgmin(i)$ la hauteur globale minimale d'une étagère de largeur L sur laquelle sont rangés les livres $B_i, \ldots, B_N$ $(1 \leqslant i \leqslant N)$, avec la convention $hgmin(N+1) = 0$ pour la hauteur de l'étagère dans laquelle on ne met aucun livre. On note $hr(i,j)$ la hauteur du rayon où est rangée la suite de livres commençant par B_i et se terminant par B_j (avec $j \geqslant i$). On a : 126 - Q 1

$$hr(i,j) = \begin{cases} \max\limits_{k \in i..j} (h_k) & \text{si } \sum\limits_{k=i}^{j} e_k \leqslant L \\ +\infty & \text{sinon.} \end{cases}$$

Trouver une relation de récurrence descendante (ou arrière) définissant $hgmin(i)$.

Question 2. En déduire les caractéristiques d'un algorithme de programmation dynamique pour calculer $hgmin(1)$. En donner la complexité temporelle. 126 - Q 2

Question 3. Appliquer cet algorithme sur l'exemple suivant : 126 - Q 3

i	1	2	3	4	5	6
e_i	1	2	1	1	2	1
h_i	1	2	5	4	3	1

avec $L = 4$.

L'étagère de largeur minimale

On veut maintenant ranger la totalité des livres $B_1, \ldots, B_N$ dans une étagère ayant K (fixé) rayons de largeur L réglable. L'ordre dans lequel les livres sont disposés est imposé comme précédemment (B_1 doit se trouver à l'extrême gauche du rayon du haut, B_2 est serré à droite de B_1 ou à l'extrême gauche du rayon du dessous, etc...). Chaque livre B_i est caractérisé par son épaisseur e_i, et, sans perte de généralité, tous les livres sont supposés de même hauteur.

On cherche la valeur L la plus petite possible qui permet de ranger tous les livres $B_1, \ldots, B_N$. Autrement dit, il s'agit de partitionner $B_1, \ldots, B_N$ en K sections de telle sorte que la largeur de la plus large des sections soit la plus petite possible. Par exemple, avec $K = 3$ rayons et $N = 6$ livres dont les largeurs sont les suivantes :

i	1	2	3	4	5	6
e_i	5	3	4	1	3	2

le rangement optimal est obtenu en mettant le premier livre sur le rayon du haut, les second et troisième livres sur le suivant et les trois derniers sur le rayon du bas. La largeur du rayon le plus large vaut 7. Notons que si l'ordre des livres n'avait pas d'importance, un rangement sur trois étagères de largeur 6 serait possible, par exemple en mettant les livres B_1 et B_4 sur la première étagère, les livres B_2 et B_5 sur la seconde et, enfin, les livres B_3 et B_6 sur la dernière.

On note $lgmin(n, k)$, avec $1 \leqslant n \leqslant N$ la largeur minimale qui permet de ranger les n premiers livres $B_1, \ldots, B_n$ dans une étagère composée de k rayons (qui sont donc de largeur $lgmin(n, k)$). On note $ep(i, j) = e_i + \cdots + e_j$ la somme des épaisseurs des livres $B_i, \ldots, B_j$.

126 - Q 4 **Question 4.** Montrer que, pour tout n, avec $1 \leqslant n \leqslant N$, on a $lgmin(n, 1) = ep(1, n)$.

126 - Q 5 **Question 5.** Soit une situation où l'on a n livres et k rayons, avec $n \leqslant k$. Montrer que l'on peut faire en sorte qu'il n'y ait qu'un seul livre par rayon. Établir que par conséquent $lgmin(n, k) = \max(\{e_1, \ldots, e_n\})$.

126 - Q 6 **Question 6.** On considère maintenant un rangement optimal de n livres sur k rayons, avec $n > k$, le dernier rayon contenant les livres $B_m, \ldots, B_n$. Montrer qu'alors on a soit $lgmin(n, k) = lgmin(m-1, k-1)$, soit $lgmin(n, k) = ep(m, n)$.

126 - Q 7 **Question 7.** Déduire des questions précédentes une relation de récurrence pour calculer $lgmin(N, K)$.

126 - Q 8 **Question 8.** Préciser l'évolution du calcul effectué par l'algorithme de programmation dynamique associé et montrer que sa complexité temporelle est en $\mathcal{O}(K \cdot N^2)$.

126 - Q 9 **Question 9.** Traiter l'exemple donné précédemment, où $N = 6$ (livres) et $K = 3$ (rayons).

Un maximum de livres dans une étagère

On dispose d'une étagère de largeur L composée de K rayons de la même hauteur (L et K fixés) et de N livres $B_1, \ldots, B_N$ d'épaisseur $e_1, \ldots, e_N$, tous de la même hauteur (légèrement inférieure à celle des rayons de la bibliothèque). Les livres sont numérotés de 1 à N, par ordre alphabétique des auteurs.

On suppose que tous les livres ne peuvent tenir sur l'étagère (notamment si $\sum_{i=1}^{N} e_i > K \cdot L$) et on désire ranger dans l'étagère un maximum M de livres parmi les N, en préservant leur ordre alphabétique. Par exemple, s'il y a quatre livres, de largeurs données ci-après :

i	1	2	3	4
e_i	3	3	2	2

que $L = 5$ et $K = 2$, on ne peut ranger que trois ($M = 3$) des quatre livres de l'une des façons suivantes :

- B_1 sur la première étagère, B_2 et B_3 sur la seconde,
- B_1 sur la première étagère, B_2 et B_4 sur la seconde,
- B_1 sur la première étagère, B_3 et B_4 sur la seconde,
- B_2 sur la première étagère, B_3 et B_4 sur la seconde,
- B_1 et B_3 sur la première étagère, B_4 sur la seconde,
- B_2 et B_3 sur la première étagère, B_4 sur la seconde.

Pourtant, l'épaisseur totale des livres est 10, et si l'on avait le droit de changer leur ordre, on pourrait tous les ranger en mettant par exemple B_1 et B_3 sur la première étagère, B_2 et B_4 sur la seconde.

Notons $lgnec(i, j)$ la largeur minimale nécessaire au rangement d'un sous-ensemble ordonné de j livres parmi i livres. Si ces j livres sont répartis sur plusieurs étagères, il faut prendre en compte dans $lgnec(i, j)$ l'éventuelle place perdue à la fin des premières rangées (mais pas celle perdue sur la dernière). En reprenant l'exemple précédent, la valeur 7 est obtenue pour $lgnec(4, 3)$ avec les deux derniers placements de la liste ci-dessus.

Question 10. On suppose tout d'abord que l'on a un seul rayon dans l'étagère ($K = 1$). Quelle est la combinatoire *a priori* du problème ? 126 - Q 10

Question 11. Donner une formule de récurrence pour calculer $lgnec(i, j)$. En déduire le principe de l'algorithme de programmation dynamique associé, la façon de déterminer la valeur M cherchée et la complexité temporelle de cet algorithme. Traiter l'exemple précédent avec les quatre livres d'épaisseurs $3, 3, 2$ et 2 et $L = 5$. 126 - Q 11

Question 12. On considère maintenant le cas où l'on a $K = 2$. Donner une formule de récurrence pour calculer $lgnec(i, j)$. En déduire le principe de l'algorithme de programmation dynamique associé, la façon de déterminer la valeur M cherchée et la complexité temporelle de cet algorithme. Traiter l'exemple précédent avec les quatre livres d'épaisseurs $3, 3, 2$ et 2, et $L = 5$. 126 - Q 12

Question 13. Écrire un algorithme pour K quelconque ($K \geqslant 1$). Préciser la façon de déterminer la valeur M recherchée. Donner les complexités temporelle et spatiale de cet algorithme. L'appliquer avec $L = 5$ et $K = 3$ aux huit livres dont l'épaisseur est donnée ci-dessous : 126 - Q 13

i	1	2	3	4	5	6	7	8
e_i	3	3	1	2	4	2	3	4

Exercice 127. Distribution de skis

Cet exercice traite d'un problème d'affectation optimale de ressources avec deux critères d'optimalité « voisins ». La clé de l'optimalité réside dans une propriété simple de la fonction d'attribution des paires de skis aux skieurs. Dans le cas particulier où l'on a autant de skieurs que de paires de skis, il s'avère que la solution peut être atteinte par un procédé glouton (voir chapitre 7). Dans le cas général d'une résolution par programmation dynamique, une simplification des calculs est étudiée.

On dispose de m paires de skis qu'il faut attribuer à n skieurs, avec $m \geqslant n \geqslant 1$. Une paire de skis – et une seule – doit être attribuée à chaque skieur et on désire maximiser la satisfaction globale des skieurs, sachant qu'un skieur est d'autant plus satisfait que la longueur de la paire de skis qu'on lui attribue est proche de sa taille.

Plus formellement, notons $h_1, \ldots, h_n$ les tailles de skieurs et $s_1, \ldots, s_m$ les longueurs de skis. Il s'agit de trouver une fonction injective $fa \in 1..n \to 1..m$ optimale. Elle doit maximiser la valeur de la satisfaction globale associée à l'attribution définie par fa, donc minimiser la somme des écarts (en valeur absolue) :

$$sde(n, m, fa) = \sum_{k=1}^{n} |h_k - s_{fa(k)}|.$$

Sans perte de généralité, on supposera ordonnées les longueurs de skis ($s_1 \leqslant \ldots \leqslant s_m$) et les tailles de skieurs ($h_1 \leqslant \ldots \leqslant h_n$).

127 - Q 1 **Question 1.** Donner la combinatoire *a priori* de ce problème.

127 - Q 2 **Question 2.** Montrer qu'une fonction fa monotone permet toujours d'atteindre une affectation optimale.

127 - Q 3 **Question 3.** En déduire qu'en présence d'autant de skieurs que de paires de skis ($n = m$), un procédé glouton (à expliciter) permet de résoudre le problème.

127 - Q 4 **Question 4.** Notons $affopt(i, j)$ la somme des écarts (en valeur absolue) correspondant à l'affectation optimale qui utilise les skis de rang 1 à j pour équiper les skieurs de rang 1 à i, avec $1 \leqslant i \leqslant j$. Soit la paire de skis de rang j est attribuée à un skieur, soit elle n'a pas été attribuée. Montrer que, dans le premier cas, c'est obligatoirement au skieur de rang i que la paire de skis de rang j doit être attribuée.

127 - Q 5 **Question 5.** En déduire la récurrence complète définissant $affopt(i, j)$.

127 - Q 6 **Question 6.** Donner le principe d'un algorithme de programmation dynamique mettant en œuvre ce calcul. Quelles en sont les complexités spatiale et temporelle (nombre de conditions évaluées) ? Préciser comment peut être déterminée la (une) fonction fa optimale.

127 - Q 7 **Question 7.** Résoudre le problème avec $m = 5$, $n = 3$, les longueurs de skis $s_1 = 158, s_2 = 179, s_3 = 200, s_4 = 203, s_5 = 213$, et les tailles de skieurs $h_1 = 170, h_2 = 190, h_3 = 210$.

127 - Q 8 **Question 8.** Proposer une stratégie remplissant le tableau associé à $affopt$ de façon partielle.

Question 9. On considère maintenant un nouveau critère d'optimalité. Il s'agit de trouver une fonction injective $fa \in 1..n \to 1..m$ optimale, au sens de la minimisation du plus grand des écarts entre la taille du skieur et la longueur de la paire de skis qui lui est attribuée, soit : 127 - Q 9

$$pgde(n, m, fa) = \max_{k \in 1..n} (|h_k - s_{fa(k)}|).$$

La solution précédente peut-elle être adaptée ?

Exercice 128. Lâchers d'œufs par la fenêtre (le retour)

Cet exercice revient sur le problème traité dans l'exercice 112 page 299, au chapitre « Diviser pour Régner ». Cependant, on le renforce ici en exigeant que la garantie de déterminer la résistance des œufs soit obtenue avec le moins de lâchers possible dans le pire cas. Après avoir examiné deux approches fondées sur des récurrences, on les compare entre elles et à la solution de type « Diviser pour Régner » appelée radixchotomie. *Enfin, on établit un lien entre le problème de lâchers d'œufs et celui de l'identification de tout nombre d'un intervalle d'entiers par un nombre fixé de questions et un nombre limité de réponses négatives.*

Rappelons tout d'abord le problème *Lâchers1*, présenté dans l'exercice 112 page 299, au chapitre « Diviser pour Régner ». On dispose d'œufs tous identiques et on cherche à connaître leur résistance, c'est-à-dire la hauteur (nombre f d'étages) à partir de laquelle ils se cassent si l'on les laisse tomber par la fenêtre d'un immeuble. Un œuf qui ne s'est pas cassé peut être réutilisé, alors que s'il s'est cassé, on l'écarte définitivement. Étant donnés un immeuble de n ($n \geqslant 1$) étages et un nombre initial k ($k \geqslant 1$) d'œufs, on cherche la valeur de f. Si les œufs ne se brisent pas même lâchés du dernier étage, la valeur de f est fixée à $n+1$, ce qui revient à considérer que les œufs se cassent forcément avec un immeuble ayant un étage de plus que réellement. L'un des objectifs de l'exercice 112 page 299, était aussi de limiter le nombre de lâchers pour n et k donnés, tout en garantissant la détermination de f puisque le nombre de lâchers était l'opération élémentaire considérée.

On considère maintenant le problème *Lâchers2* très voisin du précédent. On souhaite toujours garantir la détermination de f pour un couple (n, k) fixé, mais avec un nombre de lâchers d'œufs minimal *dans le pire cas*. Ainsi, après avoir lâché un des œufs disponibles du quatrième des dix étages d'un immeuble, on considèrera le plus grand des nombres de lâchers nécessaires à l'exploration des trois premiers étages d'une part, des six derniers de l'autre, selon qu'il y a casse ou non de l'œuf. De plus, à des fins de simplification, on étend le traitement au cas des immeubles de taille nulle ($n = 0$).

Une approche « directe »

Question 1. Appelons $nblmin(i, j)$ le nombre minimum de lâchers nécessaires à la détermination de f dans le *pire cas*, sachant que l'on dispose de i œufs et que l'immeuble 128 - Q 1

possède j étages. Montrer que $nblmin$ peut être défini par la récurrence :

$$\begin{array}{ll}
nblmin(i,0) = 0 & 1 \leqslant i \leqslant k \\
nblmin(i,1) = 1 & 1 \leqslant i \leqslant k \\
nblmin(1,j) = j & 1 < j \leqslant n \\
nblmin(i,j) = 1 + \min\limits_{p \in 1..j} \left(\max \left(\left\{ \begin{array}{l} nblmin(i-1,p-1), \\ nblmin(i,j-p) \end{array} \right\} \right) \right) & \left\{ \begin{array}{c} 1 < i \leqslant k \\ \textbf{et} \\ 1 < j \leqslant n \end{array} \right. .
\end{array}$$

128 - Q 2 **Question** 2. Proposer le principe d'un algorithme *LâchDyn1* dérivé de façon canonique de la récurrence précédente stockant les valeurs de $nblmin$ dans le tableau NBLM[1..k, 0..n]. Quelle en est la complexité temporelle ?

128 - Q 3 **Question** 3. Utiliser cet algorithme pour calculer $nblmin(3,8)$. Vérifier que $nblmin(i,j)$ est croissant avec j sur cet exemple, et le montrer dans le cas général. En déduire le principe d'un algorithme *LâchDyn2* calculant $nblmin(k,n)$ en $\mathcal{O}(k \cdot n \cdot \log_2(n))$. Qu'en conclut-on quant à l'intérêt de l'algorithme *LâchDyn1* ?

128 - Q 4 **Question** 4. Afin de déterminer une séquence de lâchers associée à toute valeur NBLM[i, j] (ou tout couple (i, j), tel que $i \geqslant 1, j \geqslant 0$), on double NBLM d'un tableau CH[1 .. k, 0 .. n] dans lequel CH[i, j] vaut :

- la (une) valeur de p associée à la valeur optimale NBLM[i, j], pour $2 \leqslant i \leqslant k$ et $2 \leqslant j \leqslant n$,
- la valeur 1 pour $j = 1$ et $1 \leqslant i \leqslant k$ d'une part, $1 < j \leqslant n$ et $i = 1$ d'autre part.

Remarque Les cellules CH[i, 0] ne présentent pas d'intérêt puisqu'alors il ne reste pas d'étages à examiner.

Expliquer comment le tableau CH est utilisé pour la détermination d'une séquence de lâchers correspondant à un couple (i, j) fixé.

Par la suite, on prend le tableau CH[1 .. 3, 0 .. 8] ci-après :

j	0	1	2	3	4	5	6	7	8
i = 1	/	1	1	1	1	1	1	1	1
2	/	1	1	2	1	2	3	1	2
3	/	1	1	2	1	2	3	4	1

Donner la séquence de lâchers pour : i) $k = 2, n = 8, f = 5$, ii) $k = 2, n = 6, f = 3$, iii) $k = 2, n = 4, f = 5$, iv) $k = 2, n = 5, f = 1$.

Une solution passant par un problème « voisin »

128 - Q 5 **Question** 5. On considère maintenant le problème *Lâchers3* du calcul de la hauteur maximale d'immeuble (exprimée en nombre d'étages) $himax(i,j)$, pour laquelle la valeur de f peut être identifiée à coup sûr avec au plus i œufs et j lâchers. Établir la récurrence

calculant $himax(k, nbl)$. Quelle différence y a-t-il entre celle-ci et celle proposée pour le problème *Lâchers2* ? Calculer la valeur $himax(4, 12)$.

Question 6. Discuter la cohérence des valeurs $nblmin(i, j)$ (ou NBLM[i, j]) et $himax(i, l)$ (ou HIM[i, l]), où i est le nombre d'œufs, j le nombre d'étages de l'immeuble et l le nombre de lâchers. 128 - Q 6

Question 7. Montrer comment utiliser himax pour résoudre le problème *Lâchers2* (identification de f à coup sûr en un nombre minimal de lâchers au pire avec k œufs pour un immeuble de n étages). Expliciter le principe de l'algorithme *LâchDyn3* réalisant le calcul et en donner la complexité temporelle. 128 - Q 7

Question 8. Expliciter le principe de l'algorithme de reconstitution de la séquence de lâchers associée à un nombre k d'œufs et à un immeuble de n étages à partir du tableau HIM. 128 - Q 8

Question 9. Appliquer cet algorithme pour : i) $k = 2, n = 9, f = 5$, ii) $k = 2, n = 7, f = 3$, ces deux cas correspondant à ceux de la question 4, page 348. Que constate-t-on quant aux séquences de lâchers produites ? L'appliquer également à $k = 3, n = 42, f = 4$. 128 - Q 9

Un calcul alternatif de $himax(i, j)$

Question 10. Montrer que l'on a la propriété suivante : 128 - Q 10

$$himax(i, j) = \sum_{p=0}^{i} C_j^p \qquad 1 \leqslant j \leqslant n, 1 \leqslant i \leqslant k.$$

Question 11. En déduire le principe de l'algorithme *LâchDyn4* calculant le nombre minimal de lâchers *au pire* avec k œufs et un immeuble de n étages (au sens du problème *Lâchers2*) de complexité temporelle $\Theta(k \cdot \log_2(n))$, sachant que l'on a d'une part l'identité $C_n^{p+1} = (C_n^p \cdot (n - p))/(p + 1)$ pour $0 \leqslant p \leqslant n$ et $n \geqslant 0$, d'autre part $himax(i, r) \geqslant hmax(i, r - 1)$ pour tout $r \geqslant 1$ (qui se déduit de $C_r^p \geqslant C_{r-1}^p$). 128 - Q 11

Question 12. Expliciter la phase de reconstitution de la séquence de lâchers associée à $himax(i, j)$. Préciser sa complexité et la comparer à celle de la procédure donnée pour la question 8. 128 - Q 12

Synthèse : choix d'une méthode

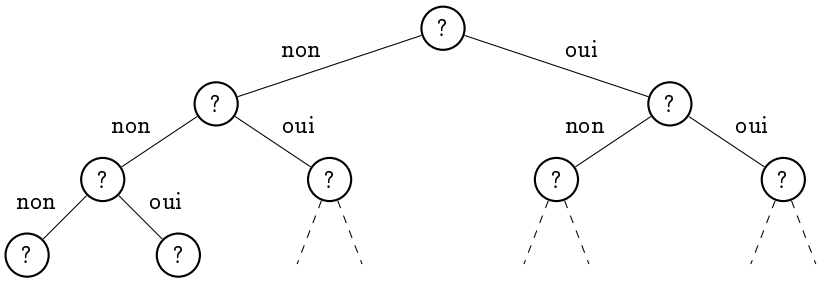
Question 13. On va maintenant confronter les différentes méthodes de résolution du problème de lâcher d'œufs : approche DpR appelée *radixchotomie* dans l'exercice 112 page 299, et algorithmes fondés sur la programmation dynamique *LâchDyn2*, *LâchDyn3* ou *LâchDyn4*. Argumenter le choix d'une de ces approches (on pourra ignorer l'aspect lié au calcul de la séquence de lâchers). 128 - Q 13

Question 14. On considère les configurations $k = 2, n = 3, 1 \leqslant f \leqslant 4$. Donner la séquence de lâchers obtenue d'une part au moyen du tableau CH donné en question 4, pour l'algorithme *LâchDyn2*, d'autre part avec la *radixchotomie*. Qu'en conclut-on quant au nombre de lâchers nécessaires à la détermination de f ? 128 - Q 14

Identification exhaustive des éléments d'un intervalle

128 - Q 15 **Question** 15. On considère l'intervalle 1 .. N et on veut pouvoir identifier tout nombre x de cet intervalle avec au plus Q questions du type « x est-il supérieur à v ? » et au plus RN réponses négatives. Quel lien ce problème, appelé *IdentInterv*, entretient-il avec le problème *Lâchers3* ? Quand admet-il (au moins) une solution ? Quand c'est le cas, comment identifie-t-on la (une) séquence de questions répondant au problème ?

128 - Q 16 **Question** 16. On choisit de représenter la solution au problème *IdentInterv* par un arbre binaire dont chaque nœud est étiqueté par la valeur v de la question posée et chaque feuille est la valeur identifiée. Compléter (étiqueter) l'arbre partiel ci-dessous pour le cas $N = 299, Q = 12, RN = 3$, la racine correspondant à la première question à poser.



128 - Q 17 **Question** 17. Si on considère non plus l'intervalle 1 .. 299, mais 1 .. 296, quelle est la première question à poser ?

128 - Q 18 **Question** 18. Caractériser les longueurs d'intervalle pour lesquelles l'arbre des questions est unique.

9.4.3 Graphes - Arbres

On traite maintenant d'un certain nombre de problèmes liés aux graphes et arbres. Les définitions et notions associées sont données dans les sections 1.5 page 22, et 1.6 page 30.

Exercice 129. Chemin de valeur minimale dans un graphe particulier 8 •

Cet exercice aborde le problème du cheminement dans un graphe orienté valué spécifique : un graphe sans circuit n'ayant qu'un seul point d'entrée et un seul point de sortie. On cherche un chemin de valeur minimale entre ces deux sommets. L'intérêt de l'exercice réside dans l'absence de restrictions sur la valuation du graphe, mais aussi dans la faible complexité de la solution qui est construite.

Soit un graphe G non valué ayant les caractéristiques suivantes :

— il n'y a pas de boucle,

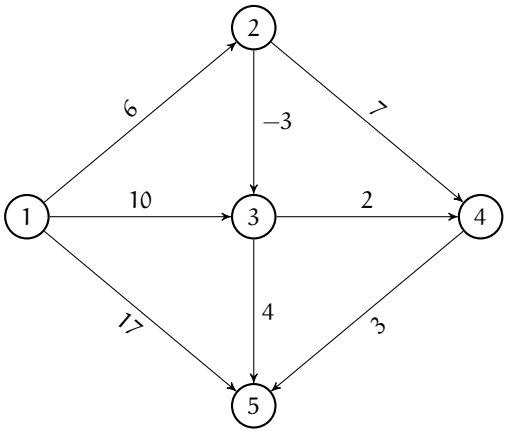


Fig. 9.4 – *Un graphe conforme*

— les n nœuds de G sont étiquetés par les entiers de 1 à n,

— si l'arc (i, j) existe dans G, alors $i < j$.

Le graphe G est dit « de numérotation conforme » ou encore « conforme ».

Question 1. Montrer qu'un graphe conforme G ne comporte aucun circuit et qu'il possède nécessairement *au moins* un point d'entrée (sommet sans prédécesseur) et un point de sortie (sommet sans successeur). 129 - Q 1

Dans la suite, on considère un graphe orienté valué $GV = (N, V, P)$ ayant en outre les propriétés suivantes :

— le sommet 1 est point d'entrée et le sommet n est point de sortie,

— tout autre sommet possède au moins un prédécesseur et un successeur,

— chaque arc possède une valeur réelle *quelconque*, GV étant représenté par une matrice MGV, à n lignes et n colonnes, dans laquelle la valeur de l'élément $MGV[i, j]$ correspond à celle de l'arc (i, j) ($+\infty$ est la valeur conventionnelle utilisée si cet arc n'existe pas).

Question 2. Donner la récurrence de calcul du nombre de chemins de 1 à n dans un graphe valué GV respectant les conditions précédentes, en définissant $nbchm(j)$ comme le nombre de chemins de 1 à j. Donner la valeur de $nbchm(5)$ pour le graphe de la figure 9.4. 129 - Q 2

Question 3. Donner les caractéristiques (récurrence, structure tabulaire, stratégie de remplissage) d'un algorithme de programmation dynamique qui calcule la valeur du (d'un) chemin de valeur minimale entre les nœuds 1 et n. Quelles sont ses complexités temporelle et spatiale ? 129 - Q 3

Question 4. Traiter l'exemple du graphe de la figure 9.4. 129 - Q 4

Exercice 130. Chemins de valeur minimale depuis une source – Algorithme de Bellman-Ford

L'algorithme de Bellman-Ford est un des grands classiques des algorithmes de programmation dynamique relatifs aux graphes orientés valués, dont il existe de nombreuses variantes. Il traite un problème plus général que le précédent, puisqu'ici on cherche la valeur des chemins optimaux entre un sommet origine donné et tout autre sommet. Son principal intérêt est d'être moins contraignant que celui de Dijkstra (voir exercice 77, page 210) quant aux valeurs portées par les arcs. Dans cet exercice, on prend le parti initial d'une construction d'algorithme issue de la programmation dynamique. D'autres algorithmes, plus efficaces ou résolvant un problème « voisin », sont également abordés.

Dans la suite, le graphe valué (sur $\mathbb{R}$) $GV = (N, V, P)$ considéré ne possède aucune boucle, et ses sommets sont étiquetés de 1 à n. Le sommet 1 joue un rôle particulier et est appelé *source*, aussi dénoté sc. On recherche la valeur du (d'un) chemin de valeur minimale entre sc et tout autre sommet du graphe GV ($+\infty$ s'il n'existe pas de chemin). La présence éventuelle d'un circuit de valeur positive ne gêne pas, puisqu'un chemin de valeur minimale ne peut inclure un tel circuit (il existe un chemin sans ce circuit de valeur moindre). Il en va de même d'un circuit de valeur négative dont les sommets ne sont pas atteignables à partir de sc comme dans le graphe de la figure 9.5.

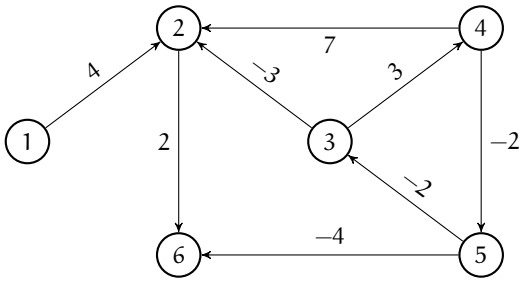


Fig. 9.5 – *Exemple de graphe possédant un circuit ($\langle 3,4,5,3\rangle$) de valeur négative (-1) non atteignable depuis le sommet* 1.

Ici, il n'existe pas de chemin du sommet 1 aux sommets 3, 4 et 5 et la valeur du chemin optimal de 1 à 3, 4 et 5 est $+\infty$. Le seul cas problématique est celui d'un circuit de valeur négative atteignable depuis sc, puisqu'alors la valeur du chemin optimal de sc à tout sommet du circuit est asymptotiquement $-\infty$ (cas apparaissant en remplaçant l'arc $(5,6)$ par $(6,5)$ dans le graphe de la figure 9.5.

Pour le moment, on suppose le graphe GV exempt de circuit(s) de valeur négative atteignable(s) depuis la source. S'il existe (au moins) un chemin élémentaire de la source à un autre sommet s_{i_p}, il existe (au moins) un chemin de valeur minimale de sc à s_{i_p}. Soit $ch_1 = \langle sc, s_{i_1}, \ldots, s_{i_p}\rangle$ ($p \geqslant 1$) un chemin optimal de sc à s_{i_p}, alors $ch_2 = \langle sc, s_{i_1}, \ldots, s_{i_{p-1}}\rangle$ est un chemin optimal de sc à $s_{i_{p-1}}$ (principe d'optimalité de Bellman) qui a la propriété d'avoir un arc de moins que ch_1. De ce constat, vient l'idée de définir une récurrence

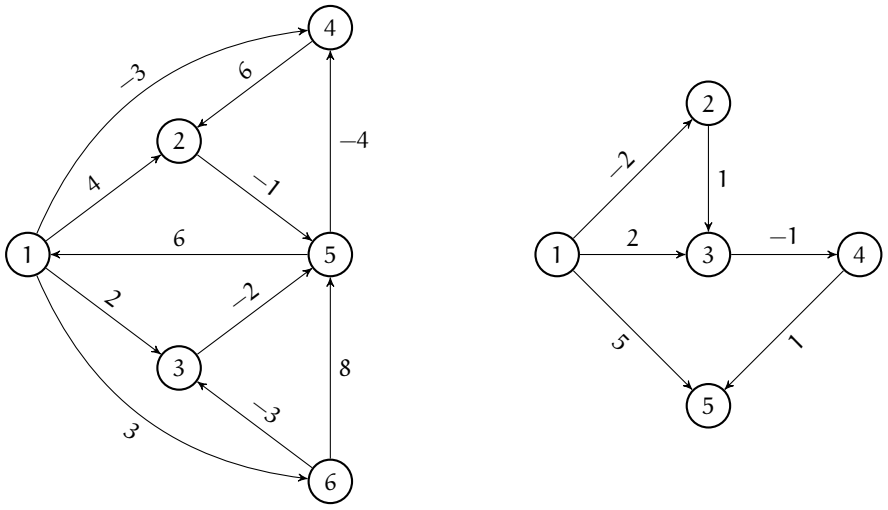


Fig. 9.6 – *Deux graphes pour la question 3*

portant sur le nombre d'arcs des chemins optimaux. On note $valchvmin(s, i)$ la valeur du (d'un) chemin optimal de la source au sommet s comportant au plus i arcs.

Question 1. Donner la formule de récurrence définissant $valchvmin(s, i)$. 130 - Q 1

Première version de l'algorithme

Question 2. On considère un graphe valué GV ayant n sommets et m arcs représenté par la table de ses arcs $AGV[1 .. m, 1 .. 2]$ et le vecteur de ses valeurs $PGV[1 .. m]$. $AGV[p, 1]$ est l'origine de l'arc p et $AGV[p, 2]$ son extrémité, alors que $PGV[p]$ est la valeur de l'arc p. En déduire la version de l'algorithme de Bellman-Ford utilisant la structure tabulaire $VCHVMIN[1..n, 1..2]$. Donner sa complexité temporelle en termes de nombre de conditions évaluées. 130 - Q 2

Question 3. Appliquer cet algorithme aux deux graphes de la figure 9.6. 130 - Q 3

Question 4. Expliciter le principe d'une solution permettant de reconstruire le (un) chemin optimal de la source sc à tout autre sommet. L'illustrer sur le graphe GV_1 de la figure 9.6. 130 - Q 4

Variante avec calcul « sur place »

Question 5. On envisage l'algorithme suivant : 130 - Q 5

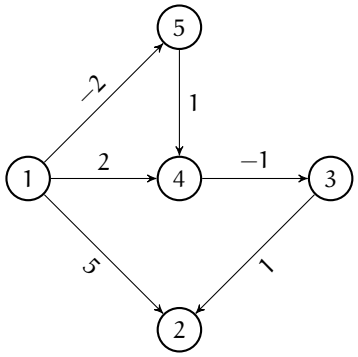
1. **constantes**
2. $n \in \mathbb{N}_1$ **et** $n = \ldots$ **et** $m \in \mathbb{N}_1$ **et** $m = \ldots$ **et**

```
3.  AGV ∈ 1..m × 1..2 → ℕ1 et AGV = [...] et PGV ∈ 1..m → ℝ et
4.  PGV = [...] et EstSansCircuitNégatif(GV)
5.  /% AGV est la matrice associée aux arcs du graphe GV considéré et VGV
    le vecteur donnant leur valeur ; l'arc (t,s) de valeur v est représenté par
    AGV[k,1] = t, AGV[k,2] = s, PGV[k] = v ; VCHVMIN est le vecteur des
    valeurs de chemin de valeur minimale de la source (sommet 1) à tout
    autre sommet ; EstSansCircuitNégatif(GV) indique que GV est exempt
    de circuit de valeur négative %/
6.  variables
7.    VCHVMIN ∈ 1..n → ℝ
8.  début
9.    VCHVMIN[1] ← 0 ;
10.   pour s parcourant 2..n faire
11.     VCHVMIN[s] ← +∞
12.   fin pour ;
13.   pour i parcourant 1..n − 1 faire
14.     /% calcul (mise à jour) de la valeur du chemin de valeur minimale pour
        tout sommet autre que 1 %/
15.     pour a ∈ 1..m faire
16.       VCHVMIN[AGV[a,2]] ←
17.                   min({ VCHVMIN[AGV[a,2]],
                            VCHVMIN[AGV[a,1]] + PGV[a] })
18.     fin pour
19.   fin pour ;
20.   écrire(VCHVMIN)
21. fin
```

Quel est le principal avantage de cette version ?

130 - Q 6 **Question** 6. Expliquer pourquoi cet algorithme résout lui aussi le problème des chemins de valeur minimale de la source (sommet 1) à tout autre sommet. Le vérifier sur les graphes GV_1 et GV_2 en prenant dans l'ordre les arcs $(5,1)$, $(1,2)$, $(4,2)$, $(1,3)$, $(6,3)$, $(1,4)$, $(5,4)$, $(2,5)$, $(3,5)$, $(6,5)$, $(1,6)$ pour GV_1 et $(1,2)$, $(1,3)$, $(2,3)$, $(3,4)$, $(1,5)$, $(4,5)$ pour GV_2.

130 - Q 7 **Question** 7. En prenant les arcs dans l'ordre $(1,2)$, $(3,2)$, $(4,3)$, $(1,4)$, $(5,4)$, $(1,5)$, appliquer cet algorithme au graphe GV_3 suivant :



Quel(s) commentaire(s) l'utilisation de cet algorithme sur les graphes GV_1, GV_2 et GV_3 inspire-t-elle ?

Question 8. Pourrait-on améliorer cet algorithme ? 130 - Q 8

Question 9. Comparer l'interprétation de la valeur VCHVMIN[s] après son calcul au pas j avec son homologue dans l'algorithme donné en réponse à la question 2. 130 - Q 9

Aspects complémentaires

Question 10. Comment pourrait-on compléter ces algorithmes pour qu'ils puissent aussi rendre un booléen précisant s'il y a ou non (au moins) un circuit de valeur négative atteignable depuis la source ? 130 - Q 10

Question 11. Comment résoudre le problème de recherche du (d'un) chemin de valeur minimale entre tout sommet et un sommet donné appelé *puits* (par opposition à source) ? 130 - Q 11

Question 12. Que penser du problème de recherche du (d'un) chemin de valeur maximale d'une source donnée à tout sommet ? 130 - Q 12

Exercice 131. Chemins de valeur minimale – Algorithme de Roy-Warshall et algorithme de Floyd – Algèbres de chemins

L'algorithme de Floyd est lui aussi l'un des grands classiques des algorithmes de programmation dynamique relatifs aux graphes orientés valués. Il concerne un problème de cheminement plus général que celui de l'exercice précédent, puisqu'ici on considère les chemins optimaux pour tous les couples de sommets. L'intérêt de cet exercice est double : 1) l'algorithme de Floyd est construit comme une adaptation de l'algorithme de Roy-Warshall, qui calcule la fermeture transitive d'un graphe (mais ne relève pas de la programmation dynamique à proprement parler puisqu'il n'y a pas recherche d'optimum), et 2) il sert de base à une famille d'algorithmes de calcul de chemins optimaux à des sens divers : le plus court, le plus long, celui de probabilité ou de capacité minimale (ou maximale), etc...

Préliminaire : existence de chemins, fermeture transitive et algorithme de Roy-Warshall

On considère un graphe non valué $G = (N, V)$ où :

- il n'y a pas de boucle,
- les sommets sont étiquetés de 1 à n.

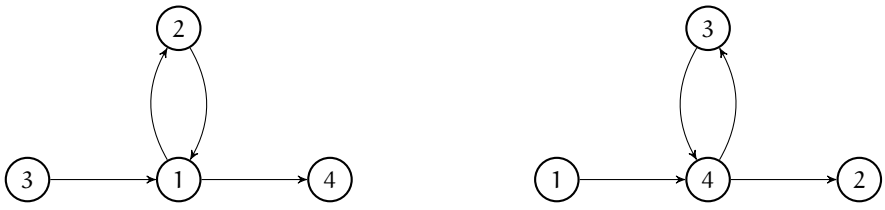
On s'intéresse tout d'abord au calcul de la fermeture transitive G^+ de G, c'est-à-dire au graphe G^+ tel que l'existence d'un chemin $\langle x * y \rangle$ dans G s'y traduit par la présence de l'arc (x, y). L'algorithme de Roy-Warshall réalisant ce calcul repose sur une récurrence

portant sur le numéro maximal des sommets *intermédiaires* apparaissant dans les chemins construits à une étape donnée. À l'étape i, on introduit l'arc (x,y) dans G^+ si les deux arcs (x,i) et (i,y) sont présents dans G^+. Ceci exprime le fait que si dans G il y a un chemin de x à i et un chemin de i à y avec des sommets intermédiaires de numéro au plus égal à $(i-1)$, on a bien un chemin $\langle x * i * y\rangle$ dont le numéro des sommets intermédiaires n'excède pas i. En notant $chemin(x,y,i)$ le prédicat représentant l'existence dans G d'un tel chemin, on a la récurrence :

$$\begin{array}{ll} chemin(x,y,0) = (x,y) \in V & 1 \leqslant x \leqslant n \text{ et } 1 \leqslant y \leqslant n \\ chemin(x,y,i) = \left(\begin{array}{l} chemin(x,y,i-1) \text{ ou} \\ \left(\begin{array}{l} chemin(x,i,i-1) \text{ et} \\ chemin(i,y,i-1) \end{array} \right) \end{array} \right) & \left\{ \begin{array}{l} 1 \leqslant i \leqslant n \text{ et} \\ 1 \leqslant x \leqslant n \text{ et} \\ 1 \leqslant y \leqslant n \end{array} \right. . \end{array}$$

131 - Q 1 **Question** 1. Montrer que cette récurrence prend en compte (on dit aussi « voit ») *tous* les chemins et circuits *élémentaires*, même si au final elle se limite à leurs origines et extrémités.

131 - Q 2 **Question** 2. Concernant les chemins *non élémentaires*, certains sont « vus », mais leur prise en compte dépend de la numérotation des sommets. Les graphes G_1 et G_2 :



G_1 G_2

sont identiques, à la numérotation des sommets près. Expliquer pourquoi on « voit » le chemin $\langle 3,1,2,1,4\rangle$ dans G_1, mais pas son homologue $\langle 1,4,3,4,2\rangle$ dans G_2.

131 - Q 3 **Question** 3. Écrire le code de l'algorithme dérivé de façon canonique de cette récurrence, en adoptant la représentation matricielle MG (resp. MG^+) du graphe $G = (N,V)$ (resp. $G^+ = (N,V^+)$). Quelles en sont les complexités spatiale et temporelle en prenant l'accès aux graphes G et G^+ comme opération élémentaire ?

131 - Q 4 **Question** 4. Vu que le calcul de $chemin(x,y,i)$ n'utilise que des éléments de dernier indice $(i-1)$, on peut se contenter de deux tableaux à deux dimensions $1..n \times 1..n$, l'un relatif à i et l'autre à $(i-1)$. Mais, en fait, l'algorithme de Roy-Warshall n'utilise qu'un tableau $1..n \times 1..n$ et effectue un calcul « sur place ». Expliquer pourquoi une telle structure suffit au calcul et établir la nouvelle récurrence résultant de cette simplification.

131 - Q 5 **Question** 5. On peut aussi améliorer la complexité temporelle en se débarrassant de la disjonction présente dans la récurrence et en remarquant que l'absence d'un chemin de x à i induit celle d'un chemin de x à y passant par i. Donner l'algorithme final (de Roy-Warshall) tenant compte de toutes les remarques précédentes et en préciser les complexités spatiale et temporelle (en nombre d'accès aux graphes).

L'algorithme de Floyd

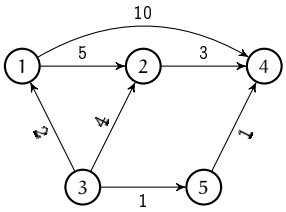
Le problème résolu par l'algorithme de Floyd est le calcul de la valeur du (d'un) chemin optimal, c'est-à-dire de valeur minimale pour tout couple de sommets d'un graphe orienté valué $GV = (N, V, P)$. Autrement dit, si dans GV on a plusieurs chemins d'origine s_i et d'extrémité s_j, on conservera la valeur de celui de moindre valeur pour le couple (s_i, s_j).

Question 6. Le principe de l'algorithme de Floyd consiste à tirer parti de l'algorithme de Roy-Warshall, en l'adaptant, pour autant que le problème puisse être résolu sur l'espace des chemins élémentaires. Que dire de la présence de circuits de valeur positive, nulle ou négative dans le graphe GV ? 131 - Q 6

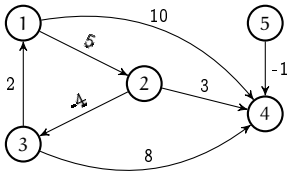
Question 7. Proposer une récurrence permettant de calculer la valeur du (d'un) chemin optimal pour tout couple de sommets (x, y) du graphe valué GV ne possédant aucun circuit posant problème. 131 - Q 7

Question 8. En déduire le code de l'algorithme de Floyd. Préciser ses complexités spatiale et temporelle. 131 - Q 8

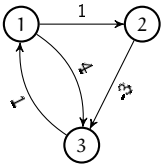
Question 9. Appliquer cet algorithme sur les graphes GV_1 et GV_2 donnés plus loin. L'appliquer également au graphe GV_3 ci-après en relâchant la précondition sur les circuits de valeur négative. Dans ce contexte, comment caractériser la présence de circuit(s) de valeur négative ? 131 - Q 9



GV_1



GV_2



GV_3

Question 10. Donner deux façons différentes, fondées sur le principe du « Petit Poucet », permettant d'identifier un (le) chemin optimal pour tout couple de sommets. Les appliquer au graphe GV_2. 131 - Q 10

Algèbres de chemins

On vient de traiter une adaptation de l'algorithme de Roy-Warshall et on peut en imaginer d'autres qui soient « intéressantes ». Par analogie avec l'algorithme de Floyd, une telle adaptation est envisageable à deux conditions :

- Le problème considéré consiste à rechercher une valeur optimale relative à tous les couples de sommets d'un graphe valué. Ceci permet, le cas échéant, de répondre au problème posé pour des couples spécifiques de sommets, par exemple : i) d'un sommet fixé à tout autre sommet, ii) de tout sommet $x \neq s_k$ à un sommet fixé s_k, ou encore iii) d'un sommet fixé s_i à un autre sommet fixé s_j.
- Le problème peut être résolu sur l'espace des chemins élémentaires, c'est-à-dire qu'aucun circuit n'est susceptible de compromettre le résultat délivré par le programme adapté.

Dans l'algorithme de Roy-Warshall, on « manipule » les chemins au moyen de deux opérations, la disjonction (« ou » logique) et la conjonction (« et » logique), comme l'illustre la récurrence donnée dans l'énoncé. Ces opérateurs ont été remplacés dans l'algorithme de Floyd par deux autres (minimum et addition) qui permettent le calcul approprié des chemins de valeur minimale. On parle d'*algèbre de chemins* associée au problème traité.

131 - Q 11 **Question** 11. Pour chacun des problèmes suivants, discuter la possibilité d'adapter l'algorithme de Roy-Warshall en précisant : 1) le couple d'opérateurs s'appliquant aux chemins, 2) l'initialisation du graphe utilisé pour calculer le résultat recherché, et 3) les types de circuit posant problème :

- la valeur du chemin de valeur maximale pour tout couple de sommets,
- la longueur du chemin le plus court pour tout couple de sommets (la longueur d'un chemin étant le nombre d'arcs qui le composent),
- la longueur du chemin le plus long pour tout couple de sommets,
- sachant que la valeur d'un arc représente une probabilité et que celle du chemin correspondant à la concaténation de plusieurs arcs est le produit des probabilités qui leur sont attachées, la probabilité du chemin de probabilité minimale (resp. maximale) pour tout couple de sommets,
- sachant que la valeur d'un arc représente une capacité (la capacité d'un chemin est celle de l'arc de moindre capacité le composant)
  a) la capacité du chemin de capacité minimale pour tout couple de sommets,
  b) la capacité du chemin de capacité maximale pour tout couple de sommets.

131 - Q 12 **Question** 12. Donner le principe de l'algorithme calculant le nombre de chemins pour tout couple de sommets, puis son code.

Deux variantes du problème des chemins de valeur minimale

131 - Q 13 **Question** 13. Pour tout couple de sommets, on souhaite calculer la valeur du (d'un) chemin de valeur minimale n'empruntant pas le *sommet intermédiaire* de numéro k donné. Préciser à quelle condition ce problème peut être résolu par un algorithme adapté de celui de Floyd, puis donner la récurrence associée.

131 - Q 14 **Question** 14. Pour tout couple de sommets (x, y), on souhaite calculer la valeur du (d'un) chemin de valeur minimale passant par le *sommet intermédiaire* de numéro k donné. Que penser d'une adaptation de l'algorithme de Floyd ?

Exercice 132. Chemin de coût minimal dans un tableau ○ •

Dans cet exercice, on s'intéresse à un problème de cheminement dans un tableau carré. On va voir qu'il se reformule comme un problème de cheminement dans un graphe valué, ce qui justifie cette place dans le chapitre sur la programmation

dynamique. La solution développée ici sera comparée aux autres algorithmes de cheminement dans les graphes étudiés précédemment dans ce chapitre.

On considère un tableau $TJ[1\,..\,n, 1\,..\,n]$ $(n > 1)$ et on s'intéresse au calcul du meilleur chemin allant de la case $(n, 1)$ à la case $(1, n)$ de TJ sachant que :

- chaque case est dotée d'une valeur (un entier positif, négatif ou nul), appelée pénalité par la suite, et que le coût d'un chemin du tableau est la somme des valeurs des cases qu'il emprunte,
- un meilleur chemin est un chemin de coût minimal,
- sans perte de généralité, on suppose les lignes numérotées de 1 à n du haut vers le bas et de la gauche vers la droite,
- on autorise les déplacements suivants :

  a) $(i, j) \longrightarrow (i, j+1)$ $(\rightarrow)$ si la pré-condition $(j + 1 \leqslant n)$ est satisfaite,

  b) $(i, j) \longrightarrow (i-1, j-1)$ $(\nwarrow)$ si la pré-condition $((1 \leqslant i-1 \leqslant n)$ **et** $(1 \leqslant j-1 \leqslant n))$ est vérifiée,

  c) $(i, j) \longrightarrow (i-1, j+1)$ $(\nearrow)$ si la pré-condition $((1 \leqslant i-1 \leqslant n)$ **et** $(1 \leqslant j+1 \leqslant n))$ est valide.

On notera qu'avec ces déplacements, aucun chemin de la case $(n, 1)$ à la case $(1, n)$ ne peut comporter de circuits.

Exemple de tableau et de cheminement.

	1	2	3	4	5	6
1	−1	−1	5	2	1	3
2	1	1	−1	0	0	0
3	−1	−2	−3	7	0	6
4	0	−5	2	0	0	6
5	1	−2	3	1	5	−3
6	2	5	4	0	−2	7

Le chemin emprunté ci-dessus est : $\langle(6,1), (6,2), (6,3), (6,4), (5,5), (4,4), (4,5), (3,6), (2,5), (1,4), (1,5), (1,6)\rangle$. Son coût est de 28.

On veut calculer par programmation dynamique la valeur $ccm(n, 1)$ représentant le coût d'un chemin de coût minimal allant de la case $(n, 1)$ à la case $(1, n)$.

Question 1. Reformuler le problème comme la recherche d'un chemin de valeur minimale dans un graphe orienté valué particulier. Expliciter le graphe associé au tableau TJ ci-après : 132 - Q 1

	1	2	3	4
1	2	−4	1	0
2	−6	2	−1	3
3	5	−2	−3	3
4	0	10	2	7

132 - Q 2 **Question** 2. Connaissant les valeurs associées aux cases du tableau TJ, établir la formule de récurrence pour calculer $ccm(i,j)$, le coût associé au (à un) meilleur chemin allant de la case (i,j) à la case $(1,n)$ avec $1 \leqslant i \leqslant n, 1 \leqslant j \leqslant n$.

132 - Q 3 **Question** 3. Expliciter l'évolution du calcul effectué par l'algorithme mettant en œuvre ces formules. Quelles sont les complexités spatiale et temporelle (en nombre de conditions évaluées) de l'algorithme de calcul de $ccm(i,j)$. Comparer la complexité temporelle à celle des algorithmes « généraux » de cheminement dans un graphe valué vus dans les exercices 130 page 352, et 131 page 355.

132 - Q 4 **Question** 4. Préciser comment pourra être reconstitué le meilleur chemin.

132 - Q 5 **Question** 5. Donner les valeurs de ccm, ainsi que le chemin optimal, dans le cas du tableau TJ de la question 1.

132 - Q 6 **Question** 6. Qu'aurait-on obtenu si les valeurs des cases $(4,1)$ et $(1,4)$ de TJ avaient été respectivement 9 et -5 ?

132 - Q 7 **Question** 7. Sur quelle autre récurrence aurait pu être fondée la solution de ce problème ?

Exercice 133. Arbres binaires de recherche pondérés

Les arbres binaires de recherche (abr) sont une structure de données (voir chapitre 1) permettant de gérer l'ordre sur les valeurs portées par les nœuds de l'arbre, par exemple les valeurs inférieures (resp. supérieures) à celle de la racine dans le sous-arbre gauche (resp. droit). On étudie ici un problème particulier de recherche de valeur dans un abr, avec une hypothèse probabiliste sur les valeurs qui s'y trouvent. On met en évidence une certaine similitude entre cet exercice et le produit chaîné de matrices (exercice 123, page 339).

On s'intéresse aux *arbres binaires de recherche* (abr) dont les valeurs sont les entiers $x_1, x_2, \ldots, x_n$ tels que $x_1 < x_2 < \cdots < x_n$. Il existe de nombreuses manières différentes de procéder pour les construire. Par exemple, dans le cas particulier où pour tout i de 1 à 5, $x_i = i$, au moins deux abr sont possibles (voir figure 9.7).

Pour simplifier, on assimile tout nœud d'un tel abr à la valeur x_i qu'il renferme. Toute valeur x_i a la probabilité $p(x_i)$ d'être recherchée. On appelle *coût* d'un abr A la valeur $cabr(A) = \sum_{k=1}^{n} p(x_k) \cdot (d_k + 1)$, où d_k est la profondeur de x_k dans l'abr A (la profondeur de la racine étant 0). La valeur $cabr(A)$ est en fait l'espérance du nombre de comparaisons à effectuer pour trouver un élément existant dans l'abr A. On cherche à construire l'abr de coût minimal, connaissant les couples $(x_i, p(x_i))$. Pour les abr de la figure 9.7, les coûts respectifs valent :

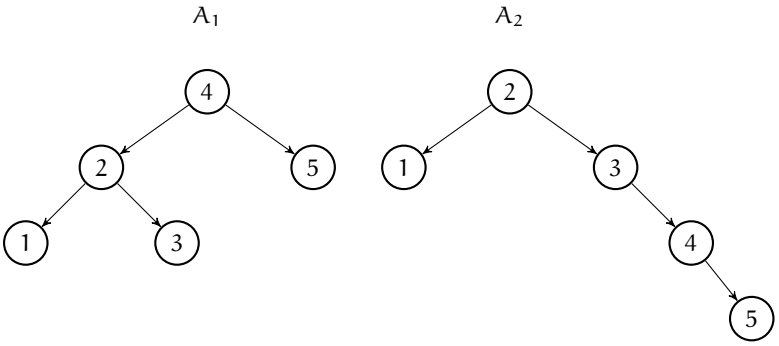


Fig. 9.7 – A_1 *et* A_2, *deux* abr *possibles avec les valeurs* $1,2,3,4,5$

$3\cdot p(1)+2\cdot p(2)+3\cdot p(3)+p(4)+2\cdot p(5)$ pour A_1,
$2\cdot p(1)+p(2)+2\cdot p(3)+3\cdot p(4)+4\cdot p(5)$ pour A_2.

Question 1. Quel est le nombre d'abr contenant les valeurs $x_1,\dots,x_n$? 133 - Q 1

Question 2. Soit $sag(A)$ (resp. $sad(A)$) le sous-arbre gauche (resp. droit) d'un abr A et $spr(A)$ la somme des probabilités associées aux valeurs des nœuds de A. Si A est vide, on pose $spr(A)=0$. Montrer que : 133 - Q 2

$$cabr(A)=cabr(sag(A))+cabr(sad(A))+spr(A).$$

Le vérifier sur les abr A_1 et A_2 de la figure 9.7, page 361. En déduire que les sous-arbres gauche et droit d'un abr de coût minimal sont eux-mêmes de coût minimal, ce qui valide le principe d'optimalité de Bellman.

Question 3. On remarquera que dans le contexte de cet exercice, le sous-arbre gauche (resp. droit) de tout abr contient des valeurs d'indices consécutifs. On appelle $A_{i,t}$ l'abr dont les valeurs sont $x_i,\dots,x_{i+t-1}$, et on pose la notation $sp(i,t)=spr(A_{i,t})$. Donner la récurrence complète de calcul de $copt(1,n)$, le coût de l'abr $A_{1,n}$ optimal. 133 - Q 3

Question 4. Expliciter le principe de l'algorithme de programmation dynamique mettant en œuvre le calcul de $copt(1,n)$. Donner les complexités spatiale et temporelle de cet algorithme. Commenter le gain apporté par cette solution par rapport à la question 1. 133 - Q 4

Question 5. On considère les couples de valeurs (x_i,p_i) figurant dans le tableau suivant : 133 - Q 5

i	1	2	3	4	5
x_i	1	2	3	4	5
p_i	0.05	0.1	0.2	0.15	0.5

Calculer $copt(1,5)$ et fournir l'abr $A_{1,5}$ optimal.

Question 6. Situer ce problème vis-à-vis du produit chaîné de matrices (exercice 123, page 339). 133 - Q 6

Exercice 134. Ensemble indépendant de poids maximal dans un arbre ○ ⁝

Cet exercice se situe dans le cadre de l'exploration d'un arbre. Il présente plusieurs singularités par rapport à la quasi-totalité de ceux du chapitre : i) les éléments de la récurrence ne seront pas stockés dans une structure tabulaire, mais directement dans l'arbre, ii) l'algorithme construit n'est pas itératif mais récursif, et iii) la construction de la solution optimale elle-même se fait en même temps que le calcul de la valeur qui lui est associée.

Soit un arbre a non vide, non ordonné (dont les fils d'un nœud sont considérés comme un ensemble de nœuds). Chaque nœud u (feuilles comprises) possède un poids (entier positif) noté $\mathrm{pds}(u)$. On définit le poids $P(\mathcal{S})$ d'un sous-ensemble $\mathcal{S}$ de nœuds de a comme la somme des poids des nœuds qui le composent :

$$P(\mathcal{S}) = \sum_{u \in \mathcal{S}} \mathrm{pds}(u).$$

On dit que deux nœuds u et v sont *adjacents* quand u est le père de v ou quand v est le père de u. Un ensemble de deux nœuds ou plus est dit *indépendant* s'il ne contient aucun couple de nœuds adjacents. On recherche $\mathcal{S}^*$, un sous-ensemble indépendant de poids maximal des nœuds de a et son poids $P(\mathcal{S}^*)$. La figure 9.8 donne un exemple d'arbre et de sous-ensemble indépendant.

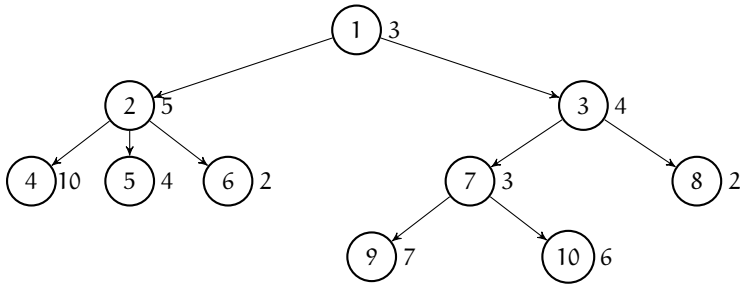


Fig. 9.8 – *Un exemple d'arbre. Le poids d'un nœud figure à côté de son identifiant qui est encerclé. L'ensemble indépendant* $\{1, 4, 5, 6, 7, 8\}$ *a pour poids* 24.

Dans la suite, on considère des arbres non ordonnés quelconques où chaque nœud u a la structure suivante :

1. identifiant id,
2. poids pds,
3. valeur optimale $v\mathrm{so}$ du sous-arbre de racine u,
4. ensemble $e\mathrm{so}$ des identifiants des nœuds du sous-ensemble indépendant de poids maximal de racine u,
5. ensemble fls des identifiants des fils de u.

Ce type d'arbre est défini de façon inductive, comme les arbres binaires (voir section 1.6, page 30). Le programme qui suit illustre l'utilisation de l'arbre de la figure 9.9 page 363 :

```
1.  constantes
2.     aqe  =  {/} ∪ {(id, pds, vso, eso, fls) | id ∈ ℕ1 et pds ∈ ℕ1 et
3.                          vso ∈ ℕ1 et eso ⊂ ℕ1 et fls ⊂ aqe}
4.  variables
5.     t ∈ aqe
6.  début
7.     t ← (1, 7, 15, {1, 6},
8.          {(2, 1, 1, {2}, /), (3, 4, 8, {6}, {(6, 8, 8, {6}, /)}), (4, 2, 2, {4}, /), (5, 2, 2, {5}, /)}) ;
9.     pour e ∈ t.fls faire
10.       écrire(le nœud d'identité , e.id, a le poids , e.pds)
11.    fin pour
12. fin
```

Ainsi, aqe étant l'ensemble de tous les arbres ayant la structure retenue, ce programme écrit l'identifiant et le poids de chacun des fils de la racine de l'arbre t de la figure 9.9, page 363.

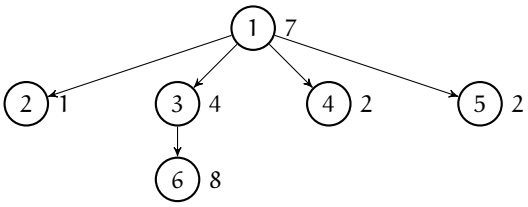


Fig. 9.9 – *L'arbre du programme exemple où ne figurent que les identifiants et les poids des différents nœuds.*

Soit u un nœud de a dont les fils sont $v_1, \ldots, v_c$ et les petits-fils $w_1, \ldots, w_g$. On note $\mathcal{S}^*_u$ un ensemble indépendant de poids maximal pour le sous-arbre de racine u.

Question 1. Montrer que : 134 - Q 1

— si $u \notin \mathcal{S}^*$, alors $\mathcal{S}^*_u = \mathcal{S}^*_{v_1} \cup \cdots \cup \mathcal{S}^*_{v_c}$, où $\mathcal{S}^*_{v_i}$ est un ensemble indépendant de poids maximal pour l'arbre de racine v_i,

— si $u \in \mathcal{S}^*$, alors $\mathcal{S}^*_u = \{u\} \cup \mathcal{S}^*_{w_1} \cup \cdots \cup \mathcal{S}^*_{w_g}$, où $\mathcal{S}^*_{w_i}$ est un ensemble indépendant de poids maximal pour l'arbre de racine w_i.

Question 2. En déduire une relation de récurrence permettant de calculer le poids de $\mathcal{S}^* = \mathcal{S}^*_r$, le sous-ensemble indépendant de poids maximal de l'arbre a de racine r. 134 - Q 2

Question 3. On suppose disponible l'opération « **procédure** *Collecte*(a ; : **modif** vof, vopf, eof, eopf) » délivrant pour l'arbre a de racine r : 134 - Q 3

— vof la valeur $\sum_{u \in fils(r)} P(\mathcal{S}^*_u)$,

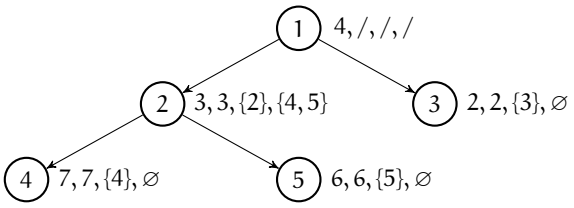
— vopf la valeur $\sum_{u \in petits-fils(r)} P(\mathcal{S}^*_u)$,

— eof les identifiants des nœuds de l'ensemble indépendant de poids maximal de chacun des fils de r,

— eopf les identifiants des nœuds de l'ensemble indépendant de poids maximal de chacun des petits-fils de r.

Cette procédure doit être appelée sur un arbre, dont les cinq composants num, pds, vso, eso et fls sont renseignés pour tout nœud autre que la racine. Ainsi, cette procédure se limite à extraire les valeurs qu'elle doit rendre, sans calcul à proprement parler.

Exemple Avec l'arbre



où tout nœud est étiqueté par son identifiant et complété par ses quatre autres composants ; l'appel de *Collecte* retourne 5 pour vof, 13 pour vopf, $\{2, 3\}$ pour eof et $\{4, 5\}$ pour eopf.

Expliciter un algorithme de programmation dynamique pour calculer simultanément $\mathcal{S}^*$ et son poids. Quelle en est la complexité temporelle en prenant la visite d'un nœud comme opération élémentaire ?

134 - Q 4 **Question** 4. Appliquer cet algorithme à l'arbre de la figure 9.8 page 362, pour en déterminer le (un) sous-ensemble indépendant de poids maximal.

9.4.4 Séquences

Exercice 135. Plus longue sous-séquence croissante ⚷ •

Outre le fait qu'il s'agit d'un problème classique sur les séquences, cet exercice illustre un cas où la valeur optimale recherchée ne se trouve pas à un emplacement prédéterminé de la structure tabulaire utilisée.

On travaille sur des séquences de nombres entiers positifs de longueur au moins égale à 2. Par exemple, une telle séquence est $u = \langle 11, 5, 2, 8, 7, 3, 1, 6, 4, 2\rangle$, de longueur 10. On note de manière générale $x = \langle x[1], \ldots, x[i], \ldots, x[n]\rangle$ une séquence de longueur $n \geqslant 1$. On appelle *sous-séquence croissante* (SSC) de x une séquence de longueur inférieure ou égale à n, dont :

— les éléments sont pris de gauche à droite dans x,

— les éléments croissent strictement de gauche à droite.

Par exemple, $\langle 2, 3, 4\rangle$ et $\langle 1, 6\rangle$ sont des SSC de u. La première est particulière, puisque ses éléments se suivent dans u. On dit qu'elle est une sous-séquence croissante *contiguë* (SSCC) de u.

Le but de cet exercice est de trouver la longueur des plus longues SSCC et SSC d'une séquence quelconque x, notées $\mathrm{lsscc}(x)$ et $\mathrm{lssc}(x)$. Par exemple, les plus longues SSCC de u sont $\langle 2,8\rangle$ et $\langle 1,6\rangle$, donc $\mathrm{lsscc}(u) = 2$. Les plus longues SSC de u sont $\langle 2,3,4\rangle$ et $\langle 2,3,6\rangle$, ce qui conduit à $\mathrm{lssc}(u) = 3$. L'opération élémentaire pour l'évaluation de la complexité est la comparaison des nombres de la séquence x considérée.

Question 1. Montrer que, pour toute séquence x, on a : $\mathrm{lssc}(x) \geqslant \mathrm{lsscc}(x)$. 135 - Q 1

Question 2. Donner le principe d'un algorithme en $\Theta(n)$ pour calculer $\mathrm{lsscc}(x)$. 135 - Q 2

Question 3. Pourquoi n'est-il pas possible de calculer $\mathrm{lssc}(x)$ en $\Theta(n)$ avec un algorithme analogue au précédent ? 135 - Q 3

Question 4. On va construire un algorithme de programmation dynamique qui calcule $\mathrm{lssct}(i)$ la longueur de la plus longue sous-séquence de x dont *le dernier élément* est $x[i]$. Dans l'exemple précédent, on a : 135 - Q 4

i	1	2	3	4	5	6	7	8	9	10
u[i]	11	5	2	8	7	3	1	6	4	2
lssct(i)	1	1	1	2	2	2	1	3	3	2

Connaissant $\mathrm{lssct}(1), \ldots, \mathrm{lssct}(n)$, le calcul de $\mathrm{lssc}(x)$ est immédiat par une boucle recherchant le maximum de $\mathrm{lssct}(i)$ pour $i \in 1\,..\,n$. Donner la récurrence définissant $\mathrm{lssct}(i)$. En déduire le programme de calcul de $\mathrm{lssc}(x)$ pour toute séquence x de longueur n, permettant de plus d'identifier ultérieurement une sous-séquence de cette longueur. On en précisera les complexités spatiale et temporelle.

Question 5. Appliquer cet algorithme sur la séquence $u = \langle 11,5,2,8,7,3,1,6,4,2\rangle$. 135 - Q 5

Exercice 136. Plus courte sur-séquence commune 8 •

Cet exercice est assez semblable à celui traité en exemple dans l'introduction de ce chapitre et peut être vu comme « son inverse ». L'intérêt principal porte sur l'établissement de la récurrence et de la propriété liant la longueur d'une plus longue sous-séquence et d'une plus courte sur-séquence communes à deux séquences.

En début de chapitre, on a étudié le problème de la recherche de la plus longue sous-séquence commune à deux séquences et on s'intéresse maintenant à celui de la détermination de la plus courte sur-séquence commune à deux séquences. Par exemple, si $u =$ *actuel* et $v =$ *actionne*, la séquence *anticonstitutionnellement* est une sur-séquence qui leur est commune de longueur 25. Cependant, une de leurs plus courtes sur-séquences communes est *actuionnel* de longueur 10, et leur unique plus longue sous-séquence commune est *acte* de longueur 4.

Soit x et y deux séquences. On note $\mathrm{lssc}(i,j)$ (resp. $\mathrm{cssc}(i,j)$ la longueur de la (d'une) plus longue sous-séquence (resp. plus courte sur-séquence) commune aux préfixes de longueur j de x et de longueur i de y.

136 - Q 1 **Question** 1. En s'inspirant de celle établie dans l'exercice traité en début de chapitre, donner une récurrence complète de calcul de cssc.

136 - Q 2 **Question** 2. En déduire le principe d'un algorithme calculant la longueur de la plus courte sur-séquence commune aux séquences x et y (et cette sur-séquence elle-même). En préciser les complexités spatiale et temporelle.

136 - Q 3 **Question** 3. Appliquer cet algorithme aux séquences $u = \textit{vache}$ et $v = \textit{veau}$.

136 - Q 4 **Question** 4. Montrer que pour tout couple de séquences x, y tel que $|x| = m$ et $|y| = n$, on a :

$$\text{lssc}(n, m) + \text{cssc}(n, m) = m + n.$$

Le vérifier pour $u = \textit{vache}$ ($n = |u| = 5$) et $v = \textit{veau}$ ($m = |v| = 4$).

Exercice 137. Distance entre séquences : algorithme de Wagner et Fischer

Cet exercice peut être vu comme une variante de celui abordé dans l'introduction de ce chapitre. Il trouve son application dans le domaine des traitements de chaînes de caractères et du génome.

Définitions

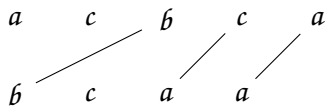
On admet que l'on peut insérer autant de symboles ε que l'on veut dans une séquence x sans en changer la signification, à savoir celle de la séquence sans symboles ε ; on appelle « super-séquence » de x la séquence x' avec de telles insertions. Par exemple, si $u = \textit{bcaa}$, une super-séquence de u est $\varepsilon\textit{bc}\varepsilon\varepsilon\textit{aa}$. Par abus de langage, on dira que la longueur de cette super-séquence est 7.

Soit deux séquences x et y et deux super-séquences de x et de y de mêmes longueurs construites sur l'alphabet Σ. On appelle « alignement » entre x et y la mise en correspondance lettre à lettre des deux super-séquences. Par exemple, entre les séquences $u = \textit{bcaa}$ et $v = \textit{acbca}$, on peut créer l'alignement :

$$\begin{array}{cccccc} \varepsilon & \varepsilon & \textit{b} & \textit{c} & \textit{a} & \textit{a} \\ | & | & | & | & | & | \\ \textit{a} & \textit{c} & \textit{b} & \varepsilon & \textit{c} & \textit{a} \end{array}$$

avec $u' = \varepsilon\varepsilon\textit{bcaa}$ et $v' = \textit{acb}\varepsilon\textit{ca}$.

Une formulation alternative de l'alignement entre deux séquences est celle de « trace », dans laquelle on utilise les séquences sans y insérer de caractère ε. La trace correspondant à l'exemple précédent est :



Une trace doit être telle que deux traits d'association entre lettres ne se croisent jamais. Sous cette contrainte, on peut construire un[4] alignement équivalent à une trace et construire de façon unique un alignement à partir d'une trace. Un alignement ou une trace peut s'interpréter comme une suite d'*opérations élémentaires d'édition* entre séquences : insertions, suppressions et transformations de lettres pour former la seconde séquence à partir de la première. Dans l'exemple précédent, l'alignement s'interprète comme la suite de transformations suivante :

1. insertion de a
2. insertion de c
3. transformation de b en b
4. suppression de c
5. transformation de a en c
6. transformation de a en a

Pour donner une valeur aux coûts des insertions, des suppressions et des transformations, on utilise une matrice δ de nombres réels positifs ou nuls définie sur $(|\Sigma|+1)\times(|\Sigma|+1)$ qui correspond à une distance : elle est symétrique, de diagonale nulle et vérifie l'inégalité triangulaire. La valeur d'un élément de cette matrice peut s'interpréter comme le coût pour transformer un symbole en l'autre ou comme le coût de suppression et d'insertion pour chaque symbole. Par exemple, sur l'alphabet Σ constitué des trois lettres a, b et c, une telle matrice pourrait être :

	ε	a	b	c
ε	0	1	1	1
a	1	0	1.5	1.2
b	1	1.5	0	1.7
c	1	1.2	1.7	0

Dans cet exemple, le coût de suppression de a est 1 ($\delta[a,\varepsilon]=1$), le coût d'insertion de b est 1 ($\delta[\varepsilon,b]=1$) et le coût de transformation de a en c est 1.2 ($\delta[a,c]=1.2$).

On appelle *coût d'un alignement* la somme des coûts élémentaires des opérations qui le constituent. Le coût de l'alignement :

$$\begin{array}{cccccc} \varepsilon & \varepsilon & b & c & a & a \\ | & | & | & | & | & | \\ a & c & b & \varepsilon & c & a \end{array}$$

est donc : 1 (insertion de a) + 1 (insertion de c) + 0 (transformation de b en b) + 1 (suppression de c) + 1.2 (transformation de a en c) + 0 (transformation de a en a) = 4.2. Un autre alignement entre les mots $u=bcaa$ et $v=acbca$ est par exemple :

4. Parfois plusieurs, mais ils ont la même interprétation.

$$\begin{array}{ccccc} b & c & a & \varepsilon & a \\ | & | & | & | & | \\ a & c & b & c & a \end{array}$$

pour un coût (moindre) de 1.5 (transformation de b en a) $+ 0$ (transformation de c en c) $+ 1.5$ (transformation de a en b) $+ 1$ (insertion de c) $+ 0$ (transformation de a en a) $= 4$.

On remarquera qu'un alignement associant des paires de symboles ε ne présente pas d'intérêt. En effet, un tel alignement $algn$ est équivalent au sens du coût à un autre alignement $algn'$ privé des paires de symboles ε, puisque pour toute matrice δ, on a $\delta(\varepsilon, \varepsilon) = 0$. Par exemple, l'alignement $algn$:

$$\begin{array}{cccccccc} \varepsilon & \varepsilon & \varepsilon & b & c & a & \varepsilon & a \\ | & | & | & | & | & | & & \\ \varepsilon & a & c & b & \varepsilon & c & \varepsilon & a \end{array}$$

a le même coût que l'alignement $algn'$:

$$\begin{array}{cccccc} \varepsilon & \varepsilon & b & c & a & a \\ | & | & | & | & | & | \\ a & c & b & \varepsilon & c & a \end{array}$$

Dans la suite, on ne considère que des alignements n'associant aucune paire de symboles ε.

Le problème

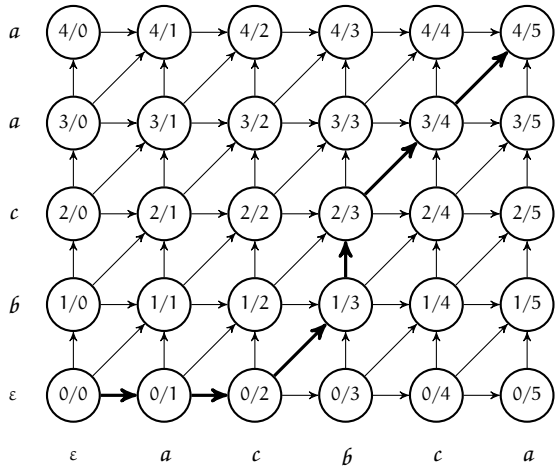
Le problème est de trouver le coût d'un alignement optimal, c'est-à-dire le moins coûteux, entre deux séquences x et y. On appelle $\Delta(x, y)$ le coût d'un (de l')alignement optimal entre les séquences x et y, et $calopt(i, j)$ le coût de l'alignement optimal entre le préfixe de longueur i de y et le préfixe de longueur j de x. On cherche donc la valeur $\Delta(x, y) = calopt(|y|, |x|) = calopt(n, m)$.

137 - Q 1

Question 1. Cette question vise à établir une relation de récurrence du calcul de $calopt(i, j)$. Pour ce faire, on va reformuler le problème comme un problème de recherche de chemin de valeur minimale dans un graphe. Cependant, compte tenu de la forme particulière du graphe à traiter, on va développer une solution spécifique (adaptée à la « topologie » particulière du graphe), comme cela a déjà été fait dans l'exercice 132, page 358. On remarque en effet qu'un alignement peut s'interpréter comme un chemin dans un graphe, ainsi que l'illustre l'exemple qui suit. Le chemin entre le nœud étiqueté $(0/0)$ et le nœud étiqueté $(4/5)$ (arcs en gras) dans le graphe ci-après représente l'alignement :

$$\begin{array}{cccccc} a & c & b & \varepsilon & c & a \\ | & | & | & | & | & | \\ \varepsilon & \varepsilon & b & c & a & a \end{array}$$

entre les séquences $u = acbca$ et $v = bcaa$.



a) Définir dans le cas général un tel graphe pour deux séquences quelconques x et y (donner en particulier la valeur attribuée aux arcs) et montrer qu'un alignement optimal entre deux séquences correspond au calcul d'un chemin de valeur minimale dans ce graphe.

b) Calculer le nombre d'alignements différents entre deux séquences x et y.

c) Compte tenu de la forme particulière du graphe, donner une relation de récurrence pour calculer $calopt(i,j)$ comme la valeur d'un chemin de valeur minimale entre le nœud $(0/0)$ et le nœud (i/j) de ce graphe.

Question 2. Donner l'algorithme (dit de Wagner et Fischer – *WF*) qui calcule le coût $\Delta(x,y)$ de l'(un des) alignement(s) de coût minimal entre deux séquences quelconques, connaissant la matrice δ. Quelles en sont les complexités spatiale et temporelle en fonction de $m = |x|$ et $n = |y|$? Quels algorithmes « standard » de calcul de plus court chemin aurait-on pu envisager d'utiliser ? Situer leur complexité temporelle par rapport à celle de l'algorithme *WF*. 137 - Q 2

Question 3. On prend l'alphabet du français, avec : 137 - Q 3

- pour toute lettre α : $\delta[\alpha, \varepsilon] = \delta[\varepsilon, \alpha] = 2$;
- pour toute lettre α : $\delta[\alpha, \alpha] = 0$;
- si α et β sont deux consonnes ou deux voyelles différentes : $\delta[\alpha, \beta] = \delta[\beta, \alpha] = 1$;
- si α est une consonne et β une voyelle : $\delta[\alpha, \beta] = \delta[\beta, \alpha] = 3$.

Calculer $\Delta($*coquine*, *malin*$)$.

Question 4. Comment est-il possible de reconstituer un alignement optimal ? Expliciter un (l') alignement optimal pour l'exemple précédent, ainsi que pour les chaînes $u =$ *est* et $v =$ *rien*. 137 - Q 4

Question 5. Montrer que l'on peut trouver une solution où la complexité en espace est réduite à $\Theta(n)$. Écrire le programme correspondant, appelé Wagner et Fischer « linéaire » (*WFL*). 137 - Q 5

137 - Q 6 **Question 6.** Si $\overline{x}$ et $\overline{y}$ désignent les séquences miroir de x et y, comment calculer un alignement optimal entre $\overline{x}$ et $\overline{y}$ en fonction d'un alignement optimal entre x et y ?

137 - Q 7 **Question 7.** Montrer que, puisque δ définit une distance, alors Δ définit aussi une distance (d'où le titre de cet exercice). Comment tirer parti de la propriété de symétrie pour améliorer la complexité spatiale de l'algorithme *WFL* ?

Exercice 138. Dissemblance entre chaînes

Cet exercice est un problème classique sur les séquences, dans lequel on cherche à déterminer un coût optimal de transformation d'une séquence en une autre. La base des associations entre symboles des séquences diffère quelque peu de celle de l'exercice précédent, ce qui constitue en partie l'originalité de cet exercice.

On cherche à calculer une association optimale entre deux chaînes définies sur un alphabet Σ. À cet effet, on dispose d'une distance δ sur Σ qui se représente par une matrice définie sur $|\Sigma| \times |\Sigma|$, symétrique, de diagonale nulle et vérifiant l'inégalité triangulaire. On appelle « association » entre deux chaînes $x = x[1], \ldots, x[m]$ et $y = y[1], \ldots, y[n]$ une suite de couples (k, l), où k indice un symbole de x et l un symbole de y, qui doit respecter les contraintes suivantes :

- aucun symbole ne peut être détruit ou inséré, à chaque symbole de x doit donc en correspondre au moins un dans y et réciproquement,
- si plusieurs symboles de x (respectivement y) correspondent à un symbole de y (respectivement x), ils doivent être contigus,
- le couple (k, l) est suivi du couple (k', l') tel que $k' = k + 1$ ou (non exclusif) $l' = l + 1$.

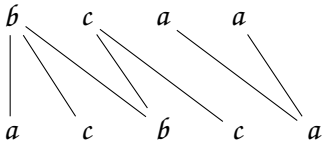
Par exemple, entre les séquences $u = bcaa$ et $v = acbca$, on peut établir, parmi beaucoup d'autres, l'association :

$$\langle (b, a), (b, c), (b, b), (c, b), (c, c), (a, a), (a, a) \rangle$$

qui se décrit sur les indices par :

$$\langle (1,1), (1,2), (1,3), (2,3), (2,4), (3,5), (4,5) \rangle$$

ou par la figure qui suit :



De manière plus formelle, une association est une suite de couples d'indices telle que :

- le premier terme de la suite est le couple $(1, 1)$,

— le dernier terme de la suite est le couple (m, n) ,

— le terme (k, l), à l'exception de (m, n), ne peut être suivi que de l'un des trois termes $(k, l+1)$, $(k+1, l)$ ou $(k+1, l+1)$.

À chaque couple (k, l) composant une association correspond une valeur de la matrice δ : la distance entre les lettres de rang k ($x[k]$) et l ($y[l]$) dans Σ. Le *coût de l'association* est défini comme la somme des valeurs de tous ses couples. Par exemple, pour la matrice δ suivante définie sur $\Sigma = \{a, b, c\}$:

	a	b	c
a	0	2	1.5
b	2	0	1
c	1.5	1	0

l'association vue auparavant entre les séquences $u = bcaa$ et $v = acbca$:

$$\langle (1,1), (1,2), (1,3), (2,3), (2,4), (3,5), (4,5) \rangle$$

a comme coût :

$$\delta[b, a] + \delta[b, c] + \delta[b, b] + \delta[c, b] + \delta[c, c] + \delta[a, a] + \delta[a, a] = 2 + 1 + 0 + 1 + 0 + 0 + 0 = 4.$$

La *dissemblance entre deux séquences* est définie comme le coût de l'association qui a le coût le plus faible parmi toutes les associations possibles entre ces deux séquences. On cherche un algorithme *efficace* pour la calculer.

Question 1. Quelle est la dissemblance entre les séquences $u = a$ et $v = aa$ pour la matrice δ donnée précédemment ? Entre les séquences $u = aab$ et $v = abb$? Entre les séquences $u = ab$ et $v = bac$? Donner un exemple non trivial d'un couple de séquences de dissemblance nulle. 138 - Q 1

Question 2. Proposer une relation de récurrence calculant la dissemblance entre deux séquences x et y. *Indication* : utiliser le fait que, pour tout couple de séquences, le dernier symbole de la première est associé au dernier symbole de la seconde. 138 - Q 2

Question 3. Définir la structure de données à utiliser et la progression de son remplissage, puis écrire le programme effectuant le calcul de la dissemblance entre deux séquences. Quels en sont les complexités spatiale et temporelle ? 138 - Q 3

Question 4. Appliquer l'algorithme avec $u = acbca$ et $v = bcaa$ pour la matrice δ donnée auparavant. 138 - Q 4

Question 5. Expliquer comment reconstituer une (l')association optimale. En donner une pour l'exemple précédent. 138 - Q 5

Question 6. Comment un tel programme pourrait-il servir dans un correcteur orthographique de traitement de texte (dont on mettra en évidence les limites) ? 138 - Q 6

Exercice 139. Plus lourd et moins balourd

Cet exercice présente deux intérêts principaux. Le premier réside dans le fait qu'il peut être résolu en le reformulant comme un problème de séquences. Le second a trait à la progressivité des questions amenant de la résolution d'un cas restreint à celle du cas général.

On considère un ensemble E de n ($n \geqslant 2$) éléphants. Outre son numéro i ($i \in 1..n$), chaque éléphant est représenté par un triplet $(\mathrm{pds}(i), \mathrm{int}(i), \mathrm{val}(i))$ où $\mathrm{pds}(i)$ est le poids de l'éléphant i, $\mathrm{int}(i)$ est une mesure de son intelligence (plus son intelligence est grande, moins l'éléphant est balourd) et $\mathrm{val}(i)$ est sa valeur marchande. On cherche le (l'un des) sous-ensemble(s) S de E qui satisfait aux conditions suivantes :

1. pour tout couple (i,j) de S, $(\mathrm{pds}(i) < \mathrm{pds}(j)) \Leftrightarrow (\mathrm{int}(i) < \mathrm{int}(j))$,
2. il n'existe pas de couple (i,j) de S avec $i \neq j$, tel que $\mathrm{pds}(i) = \mathrm{pds}(j)$ et $\mathrm{int}(i) = \mathrm{int}(j)$,
3. la valeur $\sum_{i \in S} \mathrm{val}(i)$ est maximale (autrement dit, pour tout sous-ensemble T qui vérifie les deux conditions ci-dessus : $\sum_{i \in S} \mathrm{val}(i) \geqslant \sum_{i \in T} \mathrm{val}(i)$).

Par exemple, pour l'ensemble $E = \{(1,2300,7,10), (2,2000,14,80), (3,2800,13,40), (4,2100,11,50), (5,2500,6,20), (6,2600,9,15), (7,2000,17,50)\}$, huit sous-ensembles d'au moins deux éléphants vérifient les deux premières conditions : $\{1,3\}$, $\{1,6\}$, $\{3,4\}$, $\{3,5\}$, $\{3,6\}$, $\{5,6\}$, $\{1,3,6\}$, $\{3,5,6\}$. Parmi eux, le meilleur est $\{3,4\}$ pour une valeur de 90 (qui excède la valeur de tout éléphant pris isolément, en particulier $\{2\}$).

139 - Q 1 **Question 1.** Donner le principe d'une solution par essais successifs. Quelle en est la complexité au pire en nombre de conditions évaluées ?

On envisage de ramener la résolution de ce problème d'ensembles (et sous-ensembles) à la recherche d'une sous-séquence optimale commune à deux séquences d'éléphants x et y. Plus précisément, x et y ont même longueur n et sont construites de façon appropriée (voir questions ultérieures) à partir de l'ensemble d'éléphants E. Au final, la sous-séquence commune résultat est vue comme un raffinement du sous-ensemble S initialement désiré.

139 - Q 2 **Question 2.** Énoncer une condition nécessaire (resp. suffisante) sur l'ordre dans lequel doivent apparaître les éléphants i et j dans les séquences x et y pour qu'ils puissent tous deux (resp. pour qu'un seul au plus puisse) appartenir à une sous-séquence commune à x et y.

On va tout d'abord traiter le cas particulier où tous les éléphants présents dans E ont des intelligences et des poids différents.

139 - Q 3 **Question 3.** Exprimer le problème posé comme l'identification d'une sous-séquence optimale (dans un sens à préciser) commune aux séquences d'éléphants x et y (que l'on explicitera).

139 - Q 4 **Question 4.** En déduire le principe d'une solution fondée sur la programmation dynamique permettant de résoudre ce problème en mettant en évidence la récurrence utilisée. On précisera la structure tabulaire retenue et l'évolution de son remplissage.

Question 5. Donner les complexités temporelle (en nombre de conditions évaluées) et spatiale de l'algorithme résultant (algorithme qui n'est pas demandé), puis comparer sa complexité temporelle à celle de la solution obtenue par essais successifs. 139 - Q 5

Pour les questions à suivre, on impose que l'algorithme s'applique à deux séquences de taille n, c'est-à-dire relatives à l'intégralité des éléphants de l'ensemble E. Ces séquences résulteront d'un pré-traitement fondé sur des tris appropriés dont il y aura lieu de préciser les critères (ou clés de tri).

On considère maintenant le cas d'un ensemble E pouvant contenir un sous-ensemble (de cardinal supérieur ou égal à 2) d'éléphants de type E_{mpid} de même poids, mais d'intelligences différentes (on pourrait aussi bien s'intéresser au cas symétrique d'un sous-ensemble de type E_{mipd} d'éléphants de même intelligence, mais de poids différents).

Question 6. Comment doivent être construites les deux séquences d'éléphants issues de E pour que la méthode développée dans les questions 3 et 4 conduise à un résultat correct ? 139 - Q 6

On prend maintenant en compte une situation où E peut contenir un sous-ensemble d'éléphants de type E_{mpi} ayant même poids et même intelligence (mais aucun sous-ensemble de type E_{mpid} ou E_{mipd}). En vertu de la seconde condition d'appartenance à l'ensemble solution S, au plus un éléphant de E_{mpi} peut y être intégré.

Question 7. Comment résoudre alors le problème posé ? 139 - Q 7

Question 8. Synthétiser le traitement d'un ensemble E quelconque d'éléphants et en préciser la complexité temporelle en nombre de conditions évaluées. 139 - Q 8

Question 9. Traiter l'exemple constitué de l'ensemble d'éléphants $E = \{(1, 1500, 15, 32), (2, 1200, 25, 27), (3, 1400, 22, 17), (4, 1000, 20, 20), (5, 1500, 15, 10), (6, 1800, 26, 15), (7, 1500, 15, 8), (8, 1400, 12, 23)\}$. 139 - Q 9

On révise maintenant les contraintes imposées à l'ensemble solution S en abandonnant la seconde condition. On peut donc désormais trouver dans S plusieurs éléphants de mêmes poids et intelligence. On adopte ainsi une acception plus « large » en autorisant aussi les éléphants « aussi lourds que balourds ».

Question 10. Comment procéder pour résoudre le problème posé en présence d'un ensemble E quelconque d'éléphants ? 139 - Q 10

Question 11. Traiter l'exemple de la question 9. 139 - Q 11

9.4.5 Images

Exercice 140. Triangulation optimale d'un polygone convexe

Cet exercice traite d'une question dont une application se situe dans le domaine de l'imagerie 3D. L'établissement de la récurrence nécessite au préalable de trouver une « bonne » stratégie de triangulation, ce qui constitue un des points clés de la résolution.

Un polygone $\mathcal{P}$ du plan possédant n sommets ($n \geqslant 3$) est par définition *convexe* si et seulement si, quand on construit une droite sur deux sommets consécutifs quelconques, les $(n-2)$ sommets restants sont du même côté de cette droite. Une *corde* de polygone convexe est définie comme le segment joignant deux sommets non adjacents. Une *triangulation* d'un polygone convexe $\mathcal{P}$ est un ensemble de cordes tel que :

- deux cordes ne se coupent pas,
- les cordes divisent complètement le polygone en triangles.

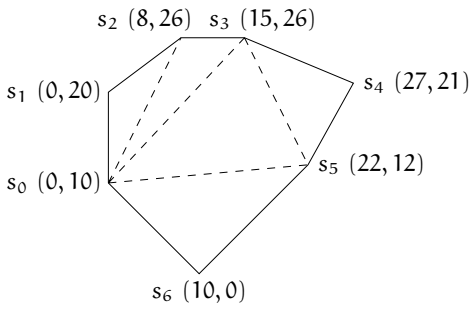


Fig. 9.10 – *Un heptagone et une triangulation de valeur approximativement égale à* 77.56

À partir des coordonnées des n sommets d'un polygone convexe $\mathcal{P}$, on définit la *longueur* d'une triangulation de $\mathcal{P}$ comme la somme des longueurs des cordes qui la composent. Le problème est le suivant : étant donné un polygone convexe $\mathcal{P}$, trouver une *triangulation minimale* de $\mathcal{P}$, c'est-à-dire une triangulation de $\mathcal{P}$ de longueur minimale. Nous supposerons par la suite que le polygone étudié possède n sommets étiquetés dans le sens des aiguilles d'une montre (appelé aussi sens rétrograde ou contraire du sens trigonométrique) notés $s_0, s_1, \ldots, s_{n-1}$.

Une première stratégie de triangulation (appelée *UnTrUnPol*) venant assez naturellement à l'esprit consiste à séparer un polygone à n côtés ($n > 3$) en un triangle et un polygone ayant $(n-1)$ côtés. Une telle approche présente l'inconvénient majeur d'amener à considérer plusieurs fois la même triangulation. Par exemple, avec l'heptagone de la figure 9.10, la triangulation partielle de la figure 9.11 est obtenue : i) en séparant le polygone initial en un triangle de sommets s_0, s_1 et s_2 et un polygone de sommets $s_0, s_2, s_3, s_4, s_5, s_6$,

puis en séparant ce dernier polygone en un triangle de sommets s_3, s_4 et s_5 et un polygone de sommets s_0, s_2, s_3, s_5, s_6, mais aussi ii) en procédant à l'inverse en séparant le polygone initial en un triangle de sommets s_3, s_4 et s_5 et un polygone de sommets $s_0, s_1, s_2, s_3, s_5, s_6$, puis en séparant ce dernier polygone en un triangle de sommets s_0, s_1 et s_2 et un polygone de sommets s_0, s_2, s_3, s_5, s_6.Avec cette démarche, le nombre de triangulations $nbtr1(n)$ examinées pour un polygone de n côtés est donné par :

$$\begin{aligned} &nbtr1(3) = 1 \\ &nbtr1(4) = 2 \\ &nbtr1(n) = n \cdot nbtr1(n-1) \qquad n \geqslant 5 \end{aligned}$$

soit $nbtr1(n) = n!/12$ pour $n > 3$, ce qui est « pire » qu'exponentiel.

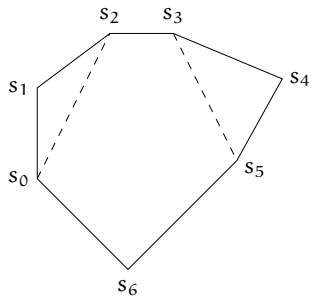


Fig. 9.11 – *Une triangulation partielle de l'heptagone de la figure 9.10*

Il faut donc chercher une stratégie alternative à *UnTrUnPol* évitant l'écueil précédent, c'est-à-dire prenant toujours en compte *toutes* les triangulations possibles (complétude), mais sans doublons (minimalité). On remarque tout d'abord que tout côté du polygone initial appartient à un et un seul des triangles d'une triangulation. On se fixe un côté noté (s_i, s_{i+1}) et on considère la stratégie *UnTrDeuxPol* consistant à tracer depuis tout sommet autre que s_i et s_{i+1}, un triangle dont (s_i, s_{i+1}) est un côté, ce qui est résumé dans la figure 9.12. Nous laissons au lecteur le soin de vérifier que l'ensemble des triangulations obtenues possède bien les deux propriétés voulues (complétude et minimalité).

Question 1. Calculer le nombre $nbtr2(n)$ de triangulations engendrées par la stratégie *UnTrDeuxPol* pour un polygone ayant n côtés. L'exprimer comme un nombre de Catalan (voir page 10) et le comparer à $nbtr1(n)$, le nombre de triangulations obtenu avec la stratégie *UnTrUnPol*. À quels autres problèmes de ce chapitre $nbtr2(n)$ fait-il penser ? 140 - Q 1

Question 2. La stratégie *UnTrDeuxPol* proposée vaut en particulier si l'on choisit comme côté de référence (s_{n-1}, s_0) qui décompose le polygone initial en : 140 - Q 2

- un triangle de sommets s_0, s_j ($j \in 1 \,..\, n-2$) et s_{n-1},
- un polygone de sommets s_0 à s_j (inexistant pour $j = 1$),
- un polygone de sommets s_j à s_{n-1} (inexistant pour $j = n-2$).

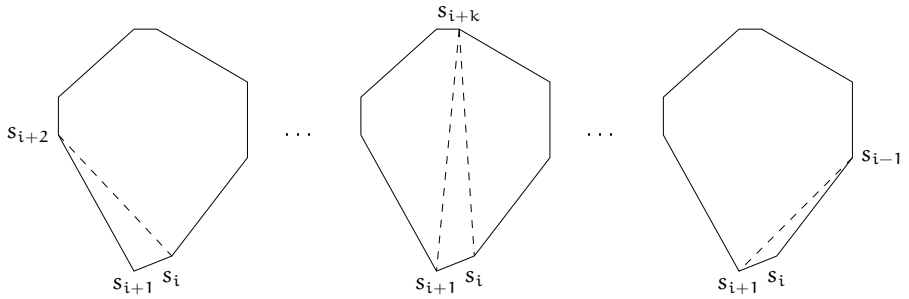
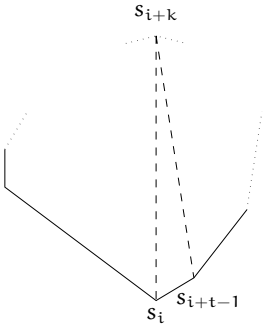


Fig. 9.12 – *La stratégie de triangulation retenue*

Les deux polygones ainsi engendrés ayant des sommets de numéros croissants, on est libéré de la gestion de questions liées à la circularité du problème. Donc, de façon générale, on va considérer la triangulation minimale du polygone de sommets $s_i, \ldots, s_{i+t-1}$ ayant t côtés tel que $i+t-1 < n$, en prenant le côté de référence (s_i, s_{i+t-1}), comme le montre la figure suivante :

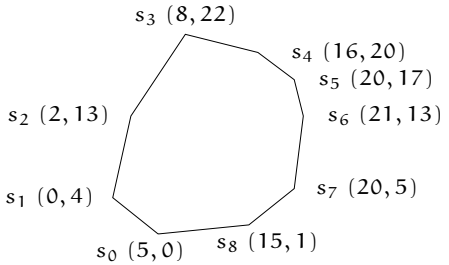


où k varie de 2 à $(t-2)$. Expliquer pourquoi le polygone $\mathcal{P}$ doit être convexe pour que cette stratégie soit convenable.

140 - Q 3 **Question** 3. On appelle $lgtrmin(i,t)$ la longueur de la (d'une) triangulation optimale du polygone de sommets $s_i, \ldots, s_{i+t-1}$ avec $i+t-1 < n$. Établir la récurrence de calcul de $lgtrmin(i,t)$.

140 - Q 4 **Question** 4. Donner les éléments (structure tabulaire, stratégie de remplissage, emplacement de la solution recherchée) d'un algorithme découlant de la formule précédente. En préciser les complexités spatiale et temporelle.

Question 5. Traiter l'exemple du polygone ci-dessous : 140 - Q 5
briques



en calculant non seulement la valeur de la triangulation optimale, mais aussi l'identification des cordes la composant.

Exercice 141. Plus grand carré noir

L'intérêt de cet exercice réside dans la comparaison entre deux approches pour résoudre le problème posé, l'une itérative, l'autre fondée sur la programmation dynamique.

Soit une image rectangulaire de largeur n et de hauteur m composée de pixels noirs (1) et blancs (0) représentée par la matrice IMG$[1..m, 1..n]$. On cherche le côté c de la plus grande sous-image carrée de IMG complètement noire. Par exemple, dans l'image de la figure 9.13, où $m = 6$ et $n = 8$, le plus grand carré noir est unique. Il a pour côté 3 et s'étend sur les lignes 2 à 4 et les colonnes 2 à 4 (avec la convention de numérotation des lignes de bas en haut et des colonnes de gauche à droite qui est utilisée tout au long de l'exercice).

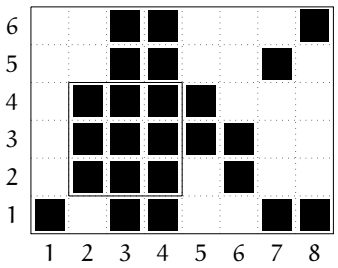


Fig. 9.13 – *Une image noir et blanc 6×8 et son plus grand carré noir (encadré)*

Question 1. On va tout d'abord construire une solution itérative en utilisant la procédure définie dans l'exercice 114 page 306, calculant le côté du plus grand carré sous un histogramme. Préciser le principe de cette solution et en déterminer la complexité temporelle 141 - Q 1

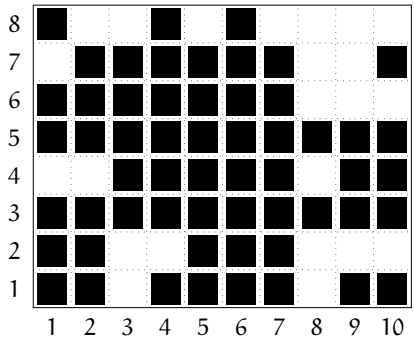


Fig. 9.14 – *Une image noir et blanc* 8×10

en termes de nombre de conditions évaluées. L'appliquer à l'image de la figure 9.14, page 378.

141 - Q 2 **Question** 2. On entreprend maintenant la résolution du problème dans le cadre de la programmation dynamique. On appelle $\mathrm{cpgcn}(i,j)$ le côté du plus grand carré noir de coin nord-ouest de coordonnées (i,j). Établir la récurrence calculant cpgcn.

141 - Q 3 **Question** 3. Expliciter la structure tabulaire à utiliser pour la mise en œuvre et la stratégie permettant de la remplir.

141 - Q 4 **Question** 4. Donner l'algorithme de programmation dynamique correspondant, en précisant ses complexités spatiale et temporelle. Comparer la complexité temporelle à celle de la solution itérative.

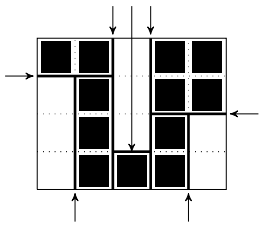
141 - Q 5 **Question** 5. Appliquer cet algorithme à l'image de la figure 9.14.

Exercice 142. Segmentation d'une image

L'originalité de cet exercice réside dans le fait qu'il est le seul de cet ouvrage dans lequel les complexités spatiale et temporelle d'un algorithme de programmation dynamique sont certes polynomiales, mais de degré supérieur à 3.

On dispose d'une image binaire sous forme de matrice de pixels $\mathrm{IMG}[1..m, 1..n]$ (m lignes, n colonnes), dont les éléments valent 0 (blanc) ou 1 (noir). On cherche à partitionner (ou segmenter) l'image en un ensemble de rectangles entièrement noirs ou entièrement blancs. La technique utilisée est celle de la « guillotine », dont la règle est la suivante : étant donné un rectangle, on a le droit de le partager en deux rectangles plus petits soit par un trait horizontal, soit par un trait vertical. Le problème est de trouver le nombre minimal de coups de guillotine pour séparer complètement les pixels noirs des pixels blancs.

Dans l'exemple ci-après d'une image 4×5, la segmentation proposée (qui n'est pas réputée optimale) requiert sept coups de guillotine (traits repérés par une flèche).



Question 1. Représenter la segmentation précédente par un arbre dont chaque nœud (feuilles comprises) est associé à l'un des rectangles (par exemple en coordonnées cartésiennes). L'arbre est-il unique ? Combien de nœuds internes (non feuilles) possède-t-il ? Quelle propriété ont les feuilles ? 142 - Q 1

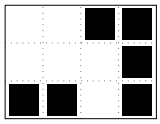
Question 2. Comment peut-on introduire dans cet arbre une numérotation des coups de guillotine ? 142 - Q 2

Question 3. Établir une relation de récurrence pour calculer le nombre minimal de coups de guillotine nécessaires pour segmenter une image quelconque de m lignes et n colonnes $(m, n \geqslant 1)$. 142 - Q 3

Question 4. Proposer une structure tabulaire et expliciter la stratégie gouvernant son remplissage. 142 - Q 4

Question 5. Donner le code de l'algorithme de segmentation optimale. Quelles en sont les complexités spatiale et temporelle ? 142 - Q 5

Question 6. Appliquer l'algorithme à l'image 3×4 ci-après : 142 - Q 6



9.4.6 Jeux

Exercice 143. Empilement de briques ⁸ ⁝

Cet exercice, comme quelques autres, nécessite une étape préalable d'analyse menant à une nouvelle formulation du problème. En effet, le problème initial considère un nombre infini de briques, qui ne peut donc être géré d'un point de vue algorithmique. Après avoir identifié un nombre utile fini de briques, il devient assez aisé d'établir la récurrence servant de base à la résolution.

On dispose de briques de n types différents, et pour chaque type d'un nombre illimité d'exemplaires. Une brique de type i est un parallélépipède rectangle dont les côtés ont pour longueur c_i^1, c_i^2 et c_i^3, avec $c_i^1 \leqslant c_i^2 \leqslant c_i^3$.

On cherche à faire un empilement de hauteur maximale. On commence par poser une brique sur une de ses faces, puis une seconde sur la première, *les côtés parallèles*, puis une troisième, etc. La contrainte est que l'on ne peut poser une brique sur le tas en construction que si la face que l'on pose est strictement incluse dans les deux dimensions dans la face supérieure de la brique précédente. Autrement dit, à chaque nouvelle brique, l'empilement rétrécit *strictement*.

143 - Q 1 **Question 1.** Montrer qu'il y a au plus trois façons réellement différentes de poser une nouvelle brique sur une brique déjà posée.

143 - Q 2 **Question 2.** Montrer que, dans un empilement, il ne peut y avoir au plus que deux briques d'un type donné.

143 - Q 3 **Question 3.** Reformuler le problème comme un empilement optimal, avec un choix non plus parmi un nombre illimité de briques, mais parmi $3n$ objets différents.

143 - Q 4 **Question 4.** Donner la formule de récurrence permettant de construire l'empilement le plus haut. Préciser la structure tabulaire utilisée dans l'algorithme associé et la progression du calcul. Donner la complexité temporelle de cet algorithme en nombre de conditions évaluées.

143 - Q 5 **Question 5.** Traiter la situation avec les trois briques B1, B2 et B3 de dimensions respectives $10 \times 12 \times 16$, $8 \times 9 \times 18$ et $4 \times 6 \times 25$.

Exercice 144. Gain maximum au jeu patagon ○ •

Dans cet exercice, on revient sur le jeu patagon déjà abordé dans l'exercice 18, page 45. L'objectif de ce jeu est de maximiser le gain du joueur qui choisit des valeurs dans un tableau de façon successive (ou tire des cartes « valuées » disposées face exposée), sans pouvoir en prendre deux consécutives.

On a vu à l'exercice 18 page 45, la définition du jeu patagon. Sous une forme un peu différente, on peut dire qu'il consiste, étant donné un tableau $T[1..n]$ d'entiers strictement positifs, à trouver la valeur maximale, $sopt(n) = \sum_{i \in I} T[i]$, obtenue à partir d'un ensemble I d'indices entre 1 et n ($n > 1$) tel que I ne contient pas deux indices consécutifs.

Dans l'exercice 18 page 45, on a étudié le nombre de façons « raisonnables » de jouer noté nfr, autrement dit la combinatoire du jeu. Celle-ci se révèle exponentielle puisque avec un tableau T de taille n, nfr est l'entier le plus proche de $(1.324\cdots)^{n-1}/1.045\cdots$. On va montrer que l'on peut trouver la valeur $sopt(n)$ sans examiner l'ensemble exhaustif de ces candidats.

Question 1. Que penser de l'algorithme glouton qui procède par paire de valeurs de T et choisit, pour autant que la contrainte soit satisfaite, la plus grande valeur ? 144 - Q 1

Question 2. Établir une relation de récurrence complète de calcul de $sopt(n)$. 144 - Q 2

Question 3. En déduire un algorithme de programmation dynamique qui calcule sopt et permet de produire ultérieurement un ensemble I associé. Quelle en est la complexité temporelle ? La situer par rapport à nfr. 144 - Q 3

Question 4. Expliciter le calcul de l'ensemble d'indices I optimal. 144 - Q 4

Question 5. Traiter l'exemple $n = 9$, $T = [2, 5, 7, 3, 1, 4, 1, 8, 4]$. 144 - Q 5

Exercice 145. Vaporisation des robots ○ •

Dans ce jeu de stratégie, le but est d'éliminer un maximum de robots arrivant par vagues successives. L'outil de destruction dont on dispose est d'autant plus puissant que le temps de charge est long, mais quand il est déclenché, il ne détruit que des robots de la vague venant d'arriver.

Dans ce jeu, il s'agit d'éliminer un maximum des robots ennemis qui attaquent une planète. Les robots arrivent par groupes, chaque groupe de robots étant espacé d'une unité de temps (ou instant). On sait combien vont arriver et à quels instants. Par exemple, à l'instant 1 va arriver un robot, suivi de dix à l'instant 2, de dix autres à l'instant 3 et d'un dernier robot qui arrivera seul à l'instant 4.

La défense de la planète est constituée d'une machine à vaporiser les robots (MVR). Quand, à l'instant t, la MVR est chargée au niveau M, elle est capable de faire disparaître dans l'atmosphère M des N robots arrivant à cet instant (tous, si $M \geqslant N$). Ainsi, si dans l'exemple d'attaque décrit précédemment la MVR est déclenchée à l'instant 3 avec un niveau de charge $M = 4$, elle fera disparaître quatre des dix robots arrivant à l'instant 3. Si elle n'est déclenchée qu'à l'instant 4, le seul robot arrivant à cet instant sera vaporisé.

Il est à noter que la MVR se décharge complètement à chaque tir, quelle que soit la taille du groupe de robots venant d'arriver. Il faut alors la laisser se recharger ; son niveau de charge est une fonction croissante du temps. La fonction de chargement de la machine dont on dispose, est par exemple la suivante :

i	1	2	3	4	5
f(i)	1	2	4	8	14

Dans cet exemple, il faut trois unités de temps à la MVR pour se recharger au niveau 4, quatre unités pour se recharger au niveau 8, etc...

Plusieurs solutions s'offrent au joueur pour gérer le rechargement et la mise en action de la MVR. Toujours dans l'exemple précédent, on peut choisir de déclencher la vaporisation à chaque instant. Au départ (instant 0), la machine n'a aucune capacité. À l'instant 1, la machine est rechargée au niveau 1, déclenchée et vaporise l'unique robot venant d'arriver. Ensuite, elle vaporise un des dix robots arrivés à l'instant 2, puis un des dix robots arrivés à l'instant 3, et enfin le seul robot arrivant à l'instant 4. Au total, avec cette stratégie, seulement quatre robots sont détruits. Si on avait attendu l'instant 3 pour le premier tir, la MVR aurait été rechargée au niveau 4 et aurait éliminé quatre robots. En la déclenchant encore à l'instant 4 (auquel la MVR est chargée au niveau 1), elle vaporise le robot arrivant à cet instant. Au total, cinq robots sont détruits, ce qui est mieux que les quatre de la première option.

Le problème est de trouver la tactique optimale de recharge et de déclenchement de la MVR qui permet, étant donnée une attaque de robots, d'en vaporiser le maximum. Pour formaliser, notons $\mathrm{qrob}(i)$ la quantité strictement positive de robots qui arrivent à chaque instant i et $f(j)$ le niveau de rechargement de la MVR quand on attend j instants. Si la machine est déclenchée à l'instant i avec un niveau M, le nombre de robots vaporisés est égal à $\min(\{\mathrm{qrob}(i), M\})$. Notons $\mathrm{nbrvopt}(j)$ le nombre de robots vaporisés en suivant la tactique optimale, en supposant que la séquence des robots s'arrête à l'instant j. On cherche donc $\mathrm{nbrvopt}(n)$, où n est l'instant d'arrivée du dernier groupe de robots.

145 - Q 1 **Question 1.** Commenter la situation où la fonction de chargement de la MVR est $f(i) = i$.

145 - Q 2 **Question 2.** Définir $\mathrm{nbrvopt}(j)$ $(j > 0)$ en fonction de $\mathrm{nbrvopt}(i)$ pour $0 \leqslant i < j$, $\mathrm{qrob}(j)$ et f. En déduire la récurrence complète permettant le calcul de $\mathrm{nbrvopt}(n)$.

145 - Q 3 **Question 3.** Spécifier la structure tabulaire à utiliser par un algorithme de programmation dynamique résolvant ce problème et la progression de son remplissage. Quelles sont les complexités temporelle et spatiale de cet algorithme ?

145 - Q 4 **Question 4.** L'appliquer à l'exemple suivant :

$n = 6$, $\mathrm{qrob}(1) = 3$, $\mathrm{qrob}(2) = 5$, $\mathrm{qrob}(3) = 2$, $\mathrm{qrob}(4) = 4$, $\mathrm{qrob}(5) = 3, \mathrm{qrob}(6) = 2$, $f(1) = 1$, $f(2) = 2$, $f(3) = 4$, $f(4) = 6$, $f(5) = 9$, $f(6) = 12$.

Exercice 146. Jeu des extrêmes ○ ⁝

Cet exercice concerne un jeu à deux joueurs pour lequel deux stratégies de jeu sont successivement étudiées. La première vise à déterminer le gain maximal du joueur commençant en premier. Ce gain constitue une borne supérieure à celui qui peut être atteint dans l'autre stratégie (classique) dans laquelle chaque joueur cherche à amasser un gain maximal. Les algorithmes obtenus dans chacun des cas se révèlent simples et assez voisins.

On a deux joueurs et, sur une table, une ligne de $2n$ ($n \geqslant 1$) cartes avec un nombre écrit sur chacune d'elles. L'ensemble des cartes est à tout moment visible des deux joueurs. Chaque joueur, à tour de rôle, prend une des deux cartes situées aux extrémités de la ligne. La carte disparaît alors du jeu, et le gain du joueur augmente du nombre écrit sur la carte.

Approche collaborative

On se pose d'abord la question de savoir quel gain maximal pourrait récupérer le joueur qui joue le premier, ce qui revient à supposer que son adversaire collabore au maximum avec lui.

Question 1. Montrer sur un exemple que le gain maximal n'est pas toujours égal à la somme des n plus grandes valeurs des cartes. 146 - Q 1

Question 2. Donner une récurrence qui permet de calculer le gain final maximal du joueur qui joue le premier. 146 - Q 2

Question 3. En déduire un algorithme de complexités spatiale et temporelle en $\Theta(n^2)$ pour calculer cet optimum. 146 - Q 3

Question 4. Fournir une trace de l'exécution de cet algorithme avec la ligne de cartes : 146 - Q 4

12	7	6	10	8	5

Question 5. Montrer que, dans cette approche, le joueur jouant en premier ne peut faire moins bien que match nul avec l'autre joueur. 146 - Q 5

Approche compétitive

On suppose maintenant que chacun des deux joueurs cherche à gagner.

Question 6. Donner une récurrence qui permet de calculer le gain final maximal du joueur qui joue le premier, en tenant compte du fait que chaque joueur cherche à maximiser son propre gain. 146 - Q 6

Question 7. Décrire le principe de l'algorithme calculant cet optimum. Préciser où se trouvent, dans la structure tabulaire utilisée, le gain maximal du joueur débutant le jeu et celui de son adversaire. Situer l'algorithme par rapport à celui de la question 3. 146 - Q 7

Question 8. Appliquer cet algorithme sur la ligne de six cartes donnée précédemment. 146 - Q 8

Question 9. Proposer une stratégie gloutonne selon laquelle le joueur jouant en premier gagne ou fait match nul. En déduire qu'il en va de même avec la stratégie fondée sur la programmation dynamique. La stratégie gloutonne conduit-elle au gain maximal ? 146 - Q 9

9.4.7 Problèmes pseudo-polynomiaux

Exercice 147. Le petit commerçant ☼ ⁝

Dans cet exercice qui est un standard, on étudie la composition d'une somme fixée avec un système monétaire donné. On cherche principalement à construire une solution de type programmation dynamique telle que le nombre de pièces rendu est minimal, et ce pour un système monétaire quelconque. Cette solution se révèle simple et efficace pour autant que l'on reste « raisonnable » quant aux montants considérés.

On s'intéresse au rendu de monnaie (avec des pièces uniquement) lorsqu'un client paie un petit commerçant avec une somme supérieure au montant de son achat. Le problème est d'arriver exactement à une somme N donnée en choisissant dans la caisse un multiensemble de pièces dont chacune possède une valeur fixée. Par exemple, dans le système numéraire de la zone euro, si le client effectue un achat de 8€10 et donne 10€, le problème consiste pour le commerçant à composer un multiensemble de pièces qui totalise 1€90. Il y a un grand nombre de solutions, parmi lesquelles :

- une pièce de 1€, une pièce de 50c, deux pièces de 20c,
- deux pièces de 50c, quatre pièces de 20c, deux pièces de 5c,
- 19 pièces de 10c, etc...

On appelle $\mathcal{C} = \{c_1, \ldots, c_n\}$ l'ensemble des pièces du système monétaire utilisé comportant n pièces différentes. On suppose que le commerçant dispose d'un nombre illimité de chacune d'entre elles. La pièce c_i a pour valeur d_i. Dans la zone euro, on a l'ensemble $\mathcal{C}$ de taille 8, avec les valeurs : $d_1 = 2€$, $d_2 = 1€$, $d_3 = 50c$, $d_4 = 20c$, $d_5 = 10c$, $d_6 = 5c$, $d_7 = 2c$, $d_8 = 1c$, ou, en centimes : $d_1 = 200$, $d_2 = 100$, $d_3 = 50$, $d_4 = 20$, $d_5 = 10$, $d_6 = 5$, $d_7 = 2$, $d_8 = 1$. Pour reprendre l'exemple précédent, la première solution peut se noter par le multiensemble $[\![c_2, c_3, c_4, c_4]\!]$ ou encore par un vecteur de dimension n indiquant combien de pièces de chaque type ont été prises pour la solution, ici $[0, 1, 1, 2, 0, 0, 0, 0]$.

Pour achever de définir le problème, on va supposer que le commerçant cherche à redonner le moins de pièces possibles. On peut donc l'appeler RLMMO comme « rendre la monnaie de manière optimale » et l'énoncer comme suit. On se donne un ensemble $\mathcal{C}$. À chaque élément c_i de $\mathcal{C}$ est associée une valeur d_i, un nombre entier strictement positif, tout comme N, la somme à rendre. Trouver un multiensemble S composé d'éléments de $\mathcal{C}$ tel que :

- la somme des valeurs des éléments de S vaille exactement N,
- le nombre des éléments de S soit minimum.

S'il n'existe aucun multiensemble répondant au premier des deux critères ci-dessus, le problème est déclaré insoluble. Cette situation peut survenir notamment si le système monétaire ne comporte pas de pièce de valeur unitaire.

Un algorithme glouton rapide, mais pas toujours exact

La méthode employée en général par un commerçant peut se décrire ainsi : utiliser les pièces par valeur décroissante, en prenant le plus possible de chacune d'elles. C'est cet algorithme glouton qui, dans l'exemple introductif, produit la première solution $[\![c_2, c_3, c_4, c_4]\!]$.

Question 1. Montrer qu'il ne résout pas le problème RLMMO quand $\mathcal{C} = \{c_1, c_2, c_3\}$, avec $d_1 = 6$, $d_2 = 4$, $d_3 = 1$ et $N = 8$. Trouver un autre couple $(\mathcal{C}, N)$ non trivialement déduit de celui-ci pour lequel cet algorithme ne convient pas non plus. 147 - Q 1

Note On peut montrer que cet algorithme résout le problème RLMMO seulement quand $\mathcal{C}$ présente certaines propriétés que possède en particulier le système de pièces européen ou les systèmes du type $\{1, 2, 4, 8, \ldots\}$. On ne s'intéresse pas ici à ces propriétés.

Un algorithme exact

En s'inspirant de la méthode utilisée dans l'exercice 20 page 47, relatif aux pièces jaunes, on définit $nbpmin(i, j)$ comme le nombre minimal de pièces nécessaires pour former la somme j en ne s'autorisant que le sous-ensemble des pièces $\{c_1, \ldots, c_i\}$. Si c'est impossible, $nbpmin(i, j)$ prend une valeur arbitrairement grande. On cherche donc $nbpmin(n, N)$. Les pièces de $\mathcal{C}$ ne sont pas supposées rangées par ordre décroissant (ou croissant).

Question 2. Donner la récurrence complète définissant $nbpmin$. 147 - Q 2

Question 3. En déduire le principe d'un algorithme pseudo-polynomial (voir section 2.1.8, page 56) dont on précisera la complexité temporelle, fondé sur la programmation dynamique, déterminant le nombre de pièces que comporte la solution optimale. 147 - Q 3

Question 4. L'appliquer pour $N = 12$ avec le système monétaire $\mathcal{C} = \{c_1, c_2, c_3\}$, avec $d_1 = 4, d_2 = 5, d_3 = 1$. 147 - Q 4

Question 5. Comment compléter l'algorithme pour savoir quelles pièces sont rendues et en quelles quantités ? 147 - Q 5

Notations

$exp_1 \mathrel{\widehat{=}} exp_2$	définition
$\forall\ ident\ \cdot\ exp_1 \Rightarrow exp_2$	quantification universelle
$\exists\ ident\ \cdot\ exp_1$ et exp_2	quantification existentielle
$\sum_{exp_1} exp_2$	Variante du quantificateur d'addition. Somme des valeurs exp_2 lorsque exp_1 est satisfaite.
$\#ident \cdot exp$	comptage : nombre de fois où le prédicat exp est satisfait
$\lceil exp \rceil$	opérateur « plafond » : plus petit entier supérieur ou égal à exp
$\lfloor exp \rfloor$	opérateur « plancher » : plus grand entier inférieur ou égal à exp
$\lvert exp \rvert$	valeur absolue ou taille d'une entité (liste, sac, etc.)
$\{exp_1, \ldots, exp_n\}$	ensemble défini en extension
$\{ListeIdent \mid exp\}$	ensemble défini en compréhension
$exp_1 .. exp_2$	intervalle de relatifs
$exp_1 - exp_2$	soustraction d'ensembles
$exp_1 \times exp_2$	produit cartésien
(exp_1, exp_2)	couple (élément d'un produit cartésien)
$(exp_1, \ldots, exp_n)$	nuplet (extension de la notion de couple)
$exp_1 \circ exp_2$	composition de relations
$exp_1 \rightarrow exp_2$	exp_1 : intervalle ou produit cartésien d'intervalles, exp_2 : ensemble quelconque, ensemble des fonctions de exp_1 dans exp_2
$ident[exp_1, \ldots, exp_n]$	élément du tableau $ident$, à n dimensions
$ident[exp .. exp]$	tranche de tableau à une dimension
$[exp_1, \ldots, exp_n]$	constante de tableau à une dimension
$\begin{bmatrix} exp_{1,1} & \cdots & exp_{1,n} \\ \vdots & \cdots & \vdots \\ exp_{m,1} & \cdots & exp_{m,n} \end{bmatrix}$	constante de tableau à deux dimensions
$\emptyset$	constante représentant le sac vide
$exp_1 \sqsubseteq exp_2$	prédicat d'appartenance à un sac
$[\![exp_1, \ldots, exp_n]\!]$	définition d'un sac en extension
$exp_1 \dot{-} exp_2$	soustraction de sacs
$exp_1 \sqcap exp_2$	intersection de sacs
$exp_1 \sqcup exp_2$	union de sacs
$exp_1 \sqsubset exp_2$	prédicat d'inclusion stricte de sacs
$exp_1 \not\sqsubset exp_2$	prédicat de non inclusion stricte de sacs
$exp_1 \sqsubseteq exp_2$	prédicat d'inclusion au sens large de sacs
$exp_1 \not\sqsubseteq exp_2$	prédicat de non inclusion au sens large de sacs
$\langle exp_1, \ldots, exp_n \rangle$	liste de n valeurs
$\mathbb{B}$	ensemble des booléens ($\{$**vrai**, **faux**$\}$)
$\mathbb{C}$	nombres complexes
$\mathrm{card}(exp)$ |	cardinal d'un ensemble
chaîne	ensemble des chaînes de caractères
chaîne(exp)	ensemble des chaînes de caractères sur le vocabulaire exp
$\mathrm{codom}(exp)$	codomaine de la relation exp
$\mathrm{dom}(exp)$	domaine de la relation exp
$\mathrm{im}(exp)$	partie imaginaire du complexe exp
$\mathrm{max}(exp)$	plus grand élément d'un ensemble numérique (si vide : $-\infty$)

$\max_{exp_1}(exp_2)$	Quantificateur max. Plus grand élément d'une expression exp_2 lorsque exp_1 est satisfaite.
$\min(exp)$	plus petit élément d'un ensemble numérique (si vide : ∞)
$\min_{exp_1}(exp_2)$	Quantificateur min. Plus petit élément d'une expression exp_2 lorsque exp_1 est satisfaite.
$\mathrm{mult}(exp_1, exp_2)$	multiplicité de l'élément exp_1 dans le sac exp_2
$\mathbb{N}$	entiers naturels
$\mathbb{N}_1$	$\mathbb{N} - \{0\}$
$\mathrm{pred}(exp_1, exp_2)$	prédécesseur de l'élément exp_1 dans la relation binaire exp_2
$\mathbb{P}(exp)$	ensemble des parties finies de l'ensemble exp
$\mathbb{R}$	réels numériques
$\mathbb{R}_+$	réels positifs ou nuls numériques
$\mathbb{R}_+^*$	réels strictement positifs numériques
$\mathrm{re}(exp)$	partie réelle du complexe exp
$\mathrm{sac}(exp)$	ensemble des sous-sacs finis du sac exp
soit $v_1, \ldots, v_2$ **tel que** exp **début** instr **fin**	instruction qui permet de spécifier les variables $v_1, \ldots, v_n$ et de localiser leurs déclarations
$\mathrm{smax}(exp)$	plus grand élément d'un sac numérique (si vide : $-\infty$)
$\mathrm{smin}(exp)$	plus petit élément d'un sac numérique (si vide : ∞)
$\mathrm{succ}(exp_1, exp_2)$	successeur de l'élément exp_1 dans la relation binaire exp_2
$\mathbb{Z}$	entiers relatifs
$\lceil x \rceil$	pour x réel, $\lceil x \rceil \mathrel{\widehat{=}} \min(\{m \in \mathbb{Z} \mid m \geqslant x\})$ (voir [31])
$\lfloor x \rfloor$	pour x réel, $\lfloor x \rfloor \mathrel{\widehat{=}} \max(\{m \in \mathbb{Z} \mid m \leqslant x\})$ (voir [31])
$\bigcup_{exp1} exp2$	quantificateur généralisant l'opérateur $\cup$ (union ensembliste)
$\bigcap_{exp1} exp2$	quantificateur généralisant l'opérateur $\cap$ (intersection ensembliste)

Remarques

1. La notation d'ensemble défini en compréhension ($\{\mathrm{ListeIdent} \mid exp\}$) est utilisée pour définir des « enregistrements ». Ainsi, $\{x, y \mid x \in \mathbb{R}$ **et** $y \in \mathbb{R}\}$ définit l'ensemble des points du plan, $pt \in \{x, y \mid x \in \mathbb{R}$ **et** $y \in \mathbb{R}\}$ déclare une variable (ou une constante) ayant comme premier champ l'« abscisse » x et comme second champ l'« ordonnée » y. Par convention, $pt.x$ (resp. $pt.y$) désigne alors cette abscisse (resp. cette ordonnée).
2. La définition des structures inductives (listes, arbres binaires, etc.) se fait également à partir de la notion d'ensembles définis en compréhension. Ainsi, $liste = \{/\} \cup \{val, suiv \mid val \in \mathbb{N}$ **et** $suiv \in liste\}$ définit $liste$ comme l'union entre la liste vide (notée /) et l'ensemble des couples constitués d'un entier et d'une liste d'entiers. Il est nécessaire d'ajouter que l'on ne s'intéresse qu'aux structures *finies* et qu'une structure $liste$ ainsi définie est le *plus petit* ensemble satisfaisant l'équation en *liste* $\mathrm{liste} = \{/\} \cup \{val, suiv \mid val \in \mathbb{N}$ **et** $suiv \in \mathrm{liste}\}$.
3. L'opérateur de comptage # délivre un entier naturel. Ainsi, si elle est définie, l'expression $\#i{\cdot}(i \in 1..10$ **et** $T[i] = 0)$ dénombre les 0 de la tranche $T[1..10]$ du tableau T.
4. Par abus de notation, dans certains programmes, la rubrique **variables** (resp. **constantes**) contient, outre la déclaration des variables (resp. des constantes), une proposition qui tient lieu de *précondition* (resp. de *contrainte*).

Liste des exercices

Bibliographie

[1] J.-R. Abrial, *The B-Book*, Cambridge University Press, 1996.

[2] A. Arnold et I. Guessarian, *Mathématiques pour l'informatique*, Masson, 1992.

[3] J. Arsac, *Premières leçons de programmation*, Cédic/F. Nathan, 1980.

[4] J. Arsac, *Les bases de la programmation*, Dunod, 1983.

[5] J. Arsac, *Préceptes pour programmer*, Dunod, 1991.

[6] O. Arsac-Mondou, C. Bourgeois-Camescasse et M. Gourtray, *Premier livre de programmation*, Cédic/F. Nathan, 1982.

[7] O. Arsac-Mondou, C. Bourgeois-Camescasse et M. Gourtray, *Pour aller plus loin en programmation*, Cédic/F. Nathan, 1983.

[8] S. Baase et A. V. Gelder, *Computer Algorithms*, Addison-Wesley Longman, 2000.

[9] B. Baynat, P. Chrétienne, C. Hanen, S. Kedad-Sidhoum, A. Munier-Kordon et C. Picouleau., *Exercices et problèmes d'algorithmique*, Dunod, 2007.

[10] G. Beauquier, J. Berstel et P. Chrétienne, *Eléments d'algorithmique*, Masson, 1992.

[11] J. Bentley, *Programming Pearls*, Addison-wesley, 1986.

[12] J. Bentley, *More Programming Pearls. Confessions of a Coder*, Addison-wesley, 1988.

[13] P. Berlioux et P. Bizard, *Algorithmique*, Dunod, 1983.

[14] L. Bougé, C. Kenyon, J.-M. Muller et Y. Robert, *Algorithmique, exercices corrigés*, Ellipses, 1993.

[15] G. Brassard et P. Bradley, *Fundamentals of Algorithmics*, Prentice-Hall, 1996.

[16] E. Cohen, *Programming in the 1990's, an Introduction to the Calculation of Programs*, Springer-Verlag, 1990.

[17] T. Cormen, C. Leiserson, C. Stein et R. Rivest, *Introduction à l'algorithmique*, Dunod, 2002.

[18] J. Courtin et I. Kowarsky, *Introduction à l'algorithmique et aux structures de données, Volume 2*, Dunod, 1995.

[19] J. Courtin et I. Kowarsky, *Introduction à l'algorithmique et aux structures de données, Volume 1*, Dunod, 1998.

[20] M. Crochemore, C. Hancart et T. Lecroq, *Algorithmique du texte*, Vuibert, 2001.

[21] A. Darte et S. Vaudenay, *Algorithmique et optimisation. Exercices corrigés*, Dunod, 2001.

[22] J.-P. Delahaye, *La suite du lézard et autres inventions*, Pour La Science, No 353, (2007).

[23] E. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.

[24] E. Dijkstra et W. Feijen, *A Method of Programming*, Addison-Wesley, 1988.

[25] A. Ducrin, *Programmation, Tome 1. Du problème à l'algorithme*, Dunod, 1984.

[26] A. Ducrin, *Programmation, Tome 2. De l'algorithme au programme*, Dunod, 1984.

[27] J. Edmonds, *How to Think about Algorithms*, Cambridge, 2008.

[28] C. Froidevaux, M.-C. Gaudel et M. Soria, *Types de données et algorithmes*, McGraw-Hill, 1990.

[29] M. Gondran et M. Minoux, *Graphes et algorithmes*, Lavoisier Tec & Doc, 2009.

[30] M. Goodrich et R. Tamassia, *Algorithm Design*, Wiley, 2001.

[31] R. Graham, D. Knuth et O. Patashnik, *Mathématiques concrètes : fondations pour l'informatique*, International Thomson Publishing France, 1998.

[32] D. Gries, *The Science of Programming*, Springer, 1983.

[33] D. Gries et F. Schneider, *A Logical Approach to Discrete Mathematics*, Springer, 1993.

[34] D. Gusfield, *Algorithms on Strings, Trees and Sequences*, Cambridge University Press, 1997.

[35] M. Guyomard, *Spécification et raffinement en B : deux exemples pédagogiques*, International B Conference, APCB (2002).

[36] M. Guyomard, *Structures de données et méthodes formelles*, Springer, 2011.

[37] M. Guyomard, *Développement informatique = spécification + programmation. une démarche méthodologique pour concevoir des algorithmes*, Support de cours stage du groupe Liesse. Enssat, Université de Rennes 1. Mars 2014. Disponible sur le site http ://www.enssat.fr/uploads/site/documents/liesse/programmationCPGE.pdf, 2014.

[38] C. Hoare, *Procedures and Parameters : An Axiomatic Approach*, *in* Proceedings of the Symposium on Semantics of Algorithmic Languages, 1971.

[39] E. Horowitz, S. Sahni et S. Rajasekaran, *Computer Algorithms*, Computer Science Press, 1998.

[40] R. Johnsonbaugh et M. Schaeffer, *Algorithms*, Pearson, Prentice-Hall, 2004.

[41] J. Julliand, *Cours et exercices corrigés d'algorithmique*, Vuibert, 2010.

[42] A. Kaldewaij, *Programming : the Derivation of Algorithms*, Prentice Hall, 1990.

[43] J. Kleinberg et E. Tardos, *Algorithm Design*, Addison Wesley, 2006.

[44] D. Knuth, *The Art of Computer Programming*, Addison-Wesley, 2015.

[45] T. Lecroq, C. Hancart et M. Crochemore, *Algorithmique du texte*, Vuibert, 2001.

[46] A. Levitin, *The Design and Analysis of Algorithms*, Addison-Wesley, 2003.

[47] J.-M. Léry, *Algorithmique. Applications en C*, Pearson, 2005.

[48] U. Manber, *Introduction to Algorithms : a Creative Approach*, Addison Wesley, 1989.

[49] B. Meyer, *Introduction à la théorie des langages de programmation*, InterEditions, 1997.

[50] M. Minoux, *Programmation Mathématique. Théorie et Algorithmes*, Lavoisier, 2008.

[51] A. Mirzaian, *A Halving Technique for the Longest Sluttering Sequence*, Information Processing Letters, 26 (1987), p. 71–75.

[52] C. Morgan, *Programming from Specifications*, Prentice-Hall, 1990.

[53] P. Naudin et C. Quitté, *Algorithmique algébrique*, Masson, 1992.

[54] R. Neapolitan et K. Naimipour, *Foundations of Algorithms*, Jones and Barlett, 2004.

[55] I. Parberry, *Problems on Algorithms*, Prentice-Hall, 1995.

[56] M. Quercia, *Nouveaux exercices d'algorithmique*, Vuibert, 2000.

[57] S. Russel et P. Norvig, *Intelligence artificielle*, Pearson, 2011.

[58] R. Sedgewick, *Algorithmes en C++*, Pearson, 2004.

[59] J. D. Smith, *Design and Analysis of Algorithms*, PWS-Kent, 1989.

[60] C. Villani, *Théorème vivant*, Grasset, 2012.

[61] N. Wirth, *Systematic Programming. An Introduction*, Prentice-Hall, 1973.

[62] N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.

Index

Q

R