THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN

ALGORITHMIQUE

Cours avec 931 exercices et 162 problèmes

4e ÉDITION

Traduit de l'anglais (États-Unis) par OLIVIER ENGLER

DUNOD

L'édition originale de ce livre a été publiée aux États-Unis par The MIT Press, Cambridge, Massachusetts, sous le titre *Introduction to Algorithms, fourth edition*.

First edition 1990.

Illustration de couverture : *Big Red*, Alexander Calder, 1959
© 2026 Calder Foundation, New York / ADAGP, Paris
Digital image Whitney Museum of American Art / Licensed by Scala

Direction et conception graphiques de la couverture :
Nicolas Wiel – Florie Bauduin (graphiste)

Relecture scientifique : Pascal Lafourcade et Thibault Ralet

Édition : Matthieu Daniel – Anne Le Duc, Vanessa Beuneche

Mise en page : Olivier Engler

Fabrication : Damien Naranin

11, rue Paul Bert 92240 Malakoff
www.dunod.com
ISBN : 978-2-10-089189-4

Sommaire

II Tri et statistiques d'ordre

III Structures de données

Préface à l'édition française

C'est avec un très grand plaisir et beaucoup de fierté que nous avons relu la quatrième édition de cet ouvrage de référence.

En effet, le « Cormen » est un des livres de référence concernant l'algorithmique, pour tous les enseignants et les étudiants d'informatique (de la première année du supérieur au Master 2). Ce livre traite des nombreuses facettes de l'algorithmique et est devenu essentiel, tant sur l'aspect théorique (cours) que pratique (exercices et problèmes).

Cette quatrième édition se trouve enrichie de nombreux nouveaux exercices et traite de sujets actuels comme les algorithmes parallèles, en ligne, d'apprentissage automatique, ou les couplages dans les graphes bipartis. Une mise à jour des références bibliographiques a également été effectuée.

Pour cette nouvelle traduction, un travail de relecture approfondi et minutieux a été mené. Il a consisté à reprendre en détail, à vérifier et à améliorer tous les termes techniques du livre, en choisissant les plus fréquemment utilisés dans la littérature actuelle, privilégiant toujours la précision, la simplicité, et la modernité. Nous nous sommes pour cela appuyés sur de nombreux cours universitaires.

Nous avons également privilégié au maximum les notations françaises (pour la multiplication, les probabilités, les intervalles, etc.), tout en restant au plus proche du texte originel.

De plus, quand nous le jugions nécessaire et par souci de clarté, nous avons conservé certaines notations introduites dans cet ouvrage (pour les vecteurs), ainsi que certains termes anglais afin que le lecteur puisse aisément approfondir ses recherches dans la littérature.

Notre méthode de travail a consisté en une relecture individuelle, puis une mise en commun de nos remarques respectives, suivie de discussions (parfois très longues pour un seul terme !).

Nous souhaitons enfin remercier chaleureusement Dunod Éditeur pour la confiance accordée tout au long de ce grand et beau projet, et ce fut un honneur d'avoir été choisis pour mener à bien ce travail.

PASCAL LAFOURCADE
Professeur à l'université Clermont Auvergne
Membre du Laboratoire d'informatique,
de modélisation et d'optimisation des systèmes (LIMOS)
Chercheur en cybersécurité et enseignant en cryptographie, Co-fondateur de la société
ASTEROIDE (Trust4Sign)

THIBAULT RALET
Professeur certifié de Mathématiques au Lycée
International Jeanne d'Arc de Clermont-Ferrand
Diplômé d'un DIU lui permettant d'enseigner la spécialité NSI

Remerciements du traducteur

Le traducteur remercie chaleureusement tous ceux et celles qui ont participé à cette formidable réalisation : Sébastien Mengin pour sa grande TeXnicité, Dave Célestin pour sa scrutation des équations, Stéphanie pour ses relectures scrupuleuses, Pascal et Thibault pour leur expertise algorithmique et bien sûr Anne, Vanessa et Matthieu des éditions Dunod pour leur confiance au long cours.

Juin 2026

Avant-propos

Voici peu de temps encore, les personnes ayant entendu le mot « algorithme » étaient presque toujours des informaticiens et informaticiennes ou des mathématiciens et mathématiciennes. Aujourd'hui cependant, alors que l'informatique s'est immiscée dans tous les pans de notre vie quotidienne, ce terme s'est banalisé. Si vous regardez autour de vous, vous trouverez des algorithmes dans les endroits les plus anodins, du four à micro-ondes à la machine à laver, en passant bien sûr par votre ordinateur. Vous vous appuyez sur les recommandations des algorithmes pour écouter de la musique ou pour définir le meilleur itinéraire en voiture. Notre société [1], que cela vous plaise ou non, sollicite même des algorithmes pour suggérer par exemple les peines pour des criminels condamnés. Parfois, c'est littéralement votre vie qui dépend d'eux : pensez aux systèmes de contrôle embarqués dans une voiture ou à ceux de certains équipements médicaux [2]. Le terme « algorithme » s'invite désormais presque chaque jour dans l'actualité.

Voilà pourquoi comprendre la notion d'algorithme ne relève plus seulement de la curiosité d'un étudiant ou d'un expert en informatique, mais bien d'un véritable devoir de citoyen. Une fois que vous aurez compris les algorithmes, vous serez à même d'en expliquer la nature, le fonctionnement et surtout les limites aux autres.

Cet ouvrage offre une introduction complète à la science moderne des algorithmes informatiques. Il en présente un grand nombre et les traite de manière très approfondie, tout en rendant leur conception accessible aussi bien au néophyte qu'au professionnel confirmé. Toutes les analyses sont présentées, certaines simples, d'autres plus complexes. Nous avons essayé d'offrir des explications claires sans sacrifier la richesse du propos ou la rigueur mathématique.

Chaque chapitre présente un algorithme, une technique de conception, un domaine d'application ou un sujet connexe. Les algorithmes sont décrits en français [3], ainsi que dans un pseudocode conçu pour être lisible par toute personne ayant quelques rudiments de programmation. Le livre contient plus de 230 figures dont la plupart comportent plusieurs parties, illustrant le fonctionnement des algorithmes. L'efficacité étant le fil rouge de la conception de cet ouvrage, nous fournissons des analyses détaillées des temps d'exécution de chaque algorithme.

Ce livre est principalement destiné aux étudiants de licence ou de master dans le cadre de cours consacrés aux algorithmes et aux structures de données. Comme il aborde les problématiques de l'ingénierie liées à la conception d'algorithmes, autant que les aspects

1. Note de l'éditeur : les auteurs désignent ici les États-Unis.

2. Pour mesurer à quel point les algorithmes interviennent dans notre quotidien, voyez par exemple l'ouvrage de Fry [162].

3. N.d.T. : cependant, les mots-clés du pseudocode ont été conservés en anglais.

mathématiques, il convient tout aussi bien pour l'autoformation des ingénieurs, développeurs et autres professionnels de l'informatique.

Dans cette quatrième édition, nous avons une nouvelle fois mis à jour l'ensemble du livre. Les modifications couvrent un large éventail de domaines, avec de nouveaux chapitres et sections, et, nous l'espérons, un style plus attrayant.

À l'attention des enseignants

Nous avons conçu cet ouvrage pour qu'il soit à la fois polyvalent et exhaustif. Il s'adapte à différents types de cours, depuis les cours de licence sur les structures de données jusqu'aux cours de master sur les algorithmes. La matière proposée va bien au-delà du contenu classique d'un semestre et il vous appartiendra de sélectionner les chapitres et sections qui correspondent à votre enseignement.

Vous devriez pouvoir organiser votre cours facilement en vous concentrant uniquement sur les chapitres dont vous avez besoin. Nous avons conçu les chapitres indépendants les uns des autres, de telle sorte que vous n'avez pas à vous soucier d'une dépendance inattendue et inutile d'un chapitre par rapport à l'autre. Alors que dans un cours de licence, vous n'utiliserez peut-être que certaines sections d'un chapitre, dans un cours de master, l'intégralité d'un chapitre pourra être abordée.

Chaque section se termine par des exercices, et chaque chapitre se termine par des problèmes. Il y a au total 931 exercices et 162 problèmes. Les exercices sont généralement composés de questions courtes qui portent sur les notions de base. Certains servent à vérifier la compréhension immédiate, mais la plupart sont plus complexes et peuvent être donnés comme devoirs. Les problèmes comprennent des études de cas plus élaborées qui introduisent souvent de nouvelles notions. Ils se composent généralement de plusieurs parties qui guident pas à pas l'étudiant vers la solution.

Comme pour la troisième édition de cet ouvrage, les solutions d'une petite sélection de problèmes et d'exercices sont accessibles (en anglais) sur le site web, http://mitpress.mit.edu/algorithms/. N'hésitez pas à le consulter pour voir s'il contient la solution à un exercice ou à un problème que vous prévoyez de donner. L'ensemble des solutions publiées étant susceptible de s'étoffer au fil du temps, nous vous recommandons de consulter le site à chaque fois que vous préparez un nouveau cours.

Les sections et exercices marqués d'une étoile (★) sont ceux destinés plutôt aux étudiants de master qu'à ceux de licence. Ce ne sont pas toujours les plus difficiles, mais ils peuvent nécessiter des connaissances mathématiques plus avancées ou une créativité supérieure à la moyenne.

À l'attention des étudiants

Nous espérons que ce manuel vous offrira une introduction agréable au monde des algorithmes. Nous avons essayé de rendre chaque algorithme accessible et intéressant. Pour cela, nous détaillons leur fonctionnement étape par étape, en prenant soin d'expliquer les notions mathématiques nécessaires à leur analyse, et en les illustrant par des figures pour vous aider à visualiser ce qui se passe.

Ce livre étant volumineux, il est probable que votre cursus n'en couvre qu'une partie. S'il doit d'abord servir d'ouvrage de référence pendant votre formation, nous l'avons également conçu suffisamment complet pour qu'il mérite une place de choix dans votre future bibliothèque professionnelle.

Quels sont les prérequis pour la lecture de ce livre ?

- Vous devez avoir un minimum d'expérience en programmation. Vous devez notamment maîtriser les procédures récursives et les structures de données simples, telles que les tableaux et les listes chaînées (bien que la section 10.2 traite des listes chaînées et introduit une variante qui pourrait vous sembler nouvelle).
- Vous devez avoir une certaine aisance avec les démonstrations mathématiques, en particulier les démonstrations par récurrence. Certaines parties du livre reposent sur des connaissances élémentaires en calcul. Notez que la première partie ainsi que les annexes A à D fournissent l'ensemble des outils nécessaires à la compréhension des nombreuses notions mathématiques qui jalonnent l'ouvrage.

Le site web http://mitpress.mit.edu/algorithms/ renvoie vers les solutions (en anglais) de certains problèmes et exercices. N'hésitez pas à comparer ces solutions aux vôtres. Nous vous demandons toutefois de ne pas nous envoyer vos solutions.

À l'attention des professionnels

La grande diversité des sujets abordés dans cet ouvrage en fait un manuel complet consacré aux algorithmes. Chaque chapitre étant relativement indépendant, vous pouvez vous concentrer sur les sujets qui vous intéressent le plus.

La plupart des algorithmes présentés ayant une grande utilité pratique, nous traitons également des questions de mise en œuvre et des aspects techniques. Quant aux rares algorithmes à vocation strictement théorique, nous présentons généralement des alternatives pratiques.

Si vous souhaitez mettre en œuvre un algorithme de ce livre, il vous suffit de traduire notre pseudocode dans votre langage de programmation préféré. Nous avons conçu le pseudocode de manière à présenter chaque algorithme de façon claire et concise. Par conséquent, nous n'abordons pas la gestion des erreurs et d'autres questions d'ingénierie logicielle qui nécessitent des hypothèses spécifiques à propos de votre environnement de programmation. Nous essayons de présenter chaque algorithme de manière simple et directe, sans laisser les

particularités d'un langage de programmation obscurcir son essence. Si vous êtes habitué à commencer l'indexation des tableaux à 0, vous serez peut-être un peu gêné par notre manière d'indicer les tableaux à partir de 1. Il vous suffira de retrancher 1 à vos indices, ou simplement de surdimensionner vos tableaux pour laisser la position 0 inutilisée.

Nous avons conscience que, si vous utilisez ce livre hors du cadre universitaire, vous ne pourrez peut-être pas vérifier vos réponses aux exercices en l'absence d'enseignant. Le site web, http://mitpress.mit.edu/algorithms/, renvoie vers les solutions de certains problèmes et exercices pour que vous puissiez vérifier votre raisonnement. Merci de ne pas nous envoyer vos solutions.

À l'attention de nos collègues

Nous avons fourni une bibliographie exhaustive et des références à la littérature actuelle du domaine. Chaque chapitre se termine par une série de notes qui fournissent des détails historiques et des références. Ces notes ne constituent toutefois pas une référence exhaustive dans le domaine des algorithmes. Bien que cela puisse paraître difficile à croire pour un ouvrage de cette taille, les contraintes d'espace nous ont empêchés d'inclure de nombreux algorithmes intéressants.

Malgré les innombrables demandes des étudiants réclamant les solutions aux problèmes et aux exercices, nous nous sommes résolus à ne pas citer de références à leur sujet. Le but est d'éviter que les étudiants ne soient tentés de consulter une solution plutôt que de la découvrir par eux-mêmes.

Modifications apportées dans cette quatrième édition

Comme nous l'avions souligné lors des deuxième et troisième éditions, selon le point de vue, le livre a soit peu, soit beaucoup changé. Un rapide coup d'œil à la table des matières montre que la plupart des chapitres et sections de la troisième édition restent présents dans cette quatrième édition. Nous avons supprimé trois chapitres et plusieurs sections, pour en ajouter trois ainsi que plusieurs nouvelles sections dans les chapitres existants.

Nous avons conservé l'organisation hybride des trois premières éditions. Plutôt que d'organiser les chapitres uniquement par domaine de problèmes ou uniquement selon les techniques, cet ouvrage combine des éléments des deux. Il contient des chapitres basés sur les techniques diviser-pour-régner, de programmation dynamique, d'algorithmes gloutons, d'analyse amortie, d'augmentation des structures de données, de NP-complétude et d'algorithmes d'approximation.

Mais l'ouvrage comporte également des sections entières sur le tri, les structures de données pour les ensembles dynamiques et les algorithmes pour les graphes. Nous avons en effet constaté que, même si vous devez savoir comment appliquer les techniques de conception et d'analyse d'algorithmes, il est rare que les problèmes signalent d'eux-mêmes quelles sont les techniques les plus adaptées pour les résoudre.

Certaines des modifications apportées à la quatrième édition s'appliquent de manière générale à l'ensemble du livre, tandis que d'autres concernent des chapitres ou des sections particuliers.

Voici un résumé des principales modifications globales :

- Nous avons créé 140 nouveaux exercices et 22 nouveaux problèmes et amélioré un grand nombre des anciens exercices et problèmes, souvent à la suite des commentaires des lecteurs. (Merci à tous les lecteurs qui ont fait des suggestions.)
- Les procédures en pseudocode apparaissent sur fond grisé pour être plus faciles à repérer, même si elles n'apparaissent pas toujours à la première mention dans le texte. Dans ce cas, un renvoi précis vous oriente vers la page correspondante. De même, toute référence croisée à une équation, un théorème, un lemme ou un corollaire numéroté s'accompagne du numéro de page.
- Nous avons écarté les sujets rarement enseignés. Nous avons supprimé dans leur intégralité les chapitres consacrés aux tas de Fibonacci, aux arbres de van Emde Boas et à la géométrie algorithmique. En outre, les éléments suivants ont été supprimés : le problème du sous-tableau maximal, l'implémentation des pointeurs et des objets, le hachage parfait, les arbres binaires de recherche construits aléatoirement, les matroïdes, les algorithmes pousser-réétiqueter pour le flot maximal, la méthode itérative de transformation de Fourier rapide, les détails de l'algorithme du simplexe pour la programmation linéaire et la factorisation des entiers. Le contenu supprimé reste accessible sur notre site web : http://mitpress.mit.edu/algorithms/.
- Nous avons revu l'ensemble du livre afin de rendre le texte plus clair. Nous pensons qu'il est essentiel que les sciences et l'ingénierie, y compris notre propre domaine, l'informatique, soient ouvertes à tous. (Le seul passage qui nous a posé problème se trouve au chapitre 13, qui nécessite un terme pour désigner le frère ou la sœur d'un parent. Comme les langues française et anglaise ne disposent pas d'un tel terme neutre, nous avons malheureusement conservé le terme « oncle ».)
- Les notes de chapitre, la bibliographie et l'index ont été mis à jour afin de refléter l'essor considérable du domaine des algorithmes depuis la troisième édition.
- Nous avons corrigé les erreurs et publié la plupart des corrections sur notre site web. Celles qui nous ont été signalées alors que nous étions en pleine préparation de cette édition n'ont pas été publiées sur le site, mais ont été corrigées dans cette édition. (Merci encore aux lecteurs vigilants pour leurs retours.)

Les modifications spécifiques apportées à la quatrième édition sont les suivantes :

- Nous avons renommé le chapitre 3 et ajouté une section donnant un aperçu de la notation asymptotique, avant d'en présenter les définitions formelles.
- Le chapitre 4 a été profondément remanié afin d'améliorer ses fondements mathématiques et de le rendre plus robuste et intuitif. La notion de récurrence algorithmique a été introduite, et la question d'ignorer les parties entières inférieures et supérieures dans les récurrences a été traitée de manière plus rigoureuse. Le deuxième cas du

théorème maître intègre des facteurs poly-logarithmiques, et une preuve rigoureuse d'une version « continue » du théorème maître est désormais fournie. Nous présentons également la méthode puissante et générale d'Akra-Bazzi (sans la démonstration).

- L'algorithme déterministe des statistiques d'ordre présenté au chapitre 9 est légèrement différent, et les analyses des algorithmes aléatoires et déterministes des statistiques d'ordre ont été remaniées.
- Outre les piles et les files, la section 10.1 traite des méthodes de stockage des tableaux et des matrices.
- Le chapitre 11 sur les tables de hachage comprend une approche moderne des fonctions de hachage. Il met également l'accent sur le sondage linéaire comme méthode efficace pour résoudre les collisions lorsque le matériel sous-jacent met en œuvre la mise en cache pour favoriser les recherches locales.
- Pour remplacer les sections sur les matroïdes au chapitre 15, nous avons étoffé un problème de la troisième édition portant sur la mise en cache hors-ligne afin d'en faire une section complète.
- La section 16.4 contient désormais une explication plus intuitive des fonctions potentielles permettant d'analyser le doublement et la réduction de moitié des tables.
- Le chapitre 17 sur l'augmentation des structures de données a été déplacé de la partie III à la partie V, reflétant notre opinion selon laquelle cette technique dépasse le cadre des notions de base.
- Le chapitre 25 est un nouveau chapitre consacré au couplage dans les graphes bipartis. Il présente des algorithmes permettant de trouver un couplage de cardinalité maximale, de résoudre le problème du mariage stable et de trouver un couplage de poids maximal (mieux connu sous le nom de « problème d'affectation »).
- Le chapitre 26, consacré aux calculs parallèles, a été mis à jour avec une terminologie moderne, y compris le titre du chapitre.
- Le chapitre 27, qui traite des algorithmes en ligne, est un autre nouveau chapitre. Dans un algorithme en ligne, les données d'entrée arrivent au fur et à mesure, plutôt que d'être disponibles dans leur intégralité dès le début d'exécution de l'algorithme. Le chapitre décrit plusieurs exemples d'algorithmes en ligne, notamment la recherche du temps d'attente d'un ascenseur avant de décider de prendre les escaliers, la maintenance d'une liste chaînée via l'heuristique de renvoi sur le devant (*move-to-front*) et l'évaluation des politiques de remplacement pour les caches.
- Dans le chapitre 29, nous avons supprimé la présentation détaillée de l'algorithme du simplexe car elle était trop mathématique sans vraiment transmettre beaucoup d'idées algorithmiques. Le chapitre se concentre désormais sur l'aspect clé de la modélisation des problèmes sous forme de programmes linéaires, ainsi que sur la propriété essentielle de dualité de la programmation linéaire.
- Dans le chapitre 32, la nouvelle section 32.5 ajoute au chapitre sur le couplage de chaînes la structure simple mais puissante du tableau de suffixes.

- Le chapitre 33, consacré à l'apprentissage automatique, est le troisième nouveau chapitre. Il présente plusieurs méthodes de base utilisées dans l'apprentissage automatique : le regroupement d'éléments similaires, les algorithmes à majorité pondérée et la descente de gradient pour trouver le minimiseur d'une fonction.
- La section 34.5.6 résume les stratégies de réduction en temps polynomial pour montrer que les problèmes sont NP-difficiles.
- La preuve de l'algorithme d'approximation pour le problème de couverture par ensembles a été révisée (section 35.3).

Site web

Pour plus d'informations, d'exercices corrigés et de ressources complémentaires, rendez-vous sur notre site officiel : http://mitpress.mit.edu/algorithms/. Le site renvoie vers une liste d'errata, des éléments de la précédente édition non repris ici, des solutions pour une sélection d'exercices et de problèmes, des implémentations Python de nombreux algorithmes présentés dans cet ouvrage, une liste expliquant les blagues ringardes à propos des noms des professeurs (eh oui), et diverses autres ressources en constante évolution. Le site vous indique également comment signaler des erreurs ou faire des suggestions.

Conditions de production du livre

Comme pour les éditions précédentes, la mise en page a été réalisée avec le langage de description LaTeX2. Nous avons utilisé la police Times pour la partie texte et les polices MathTime Professional II pour la notation mathématique. L'index a été généré par notre propre programme en C, « Windex », et la bibliographie produite à l'aide de BIBTEX.

Notre appel lancé à Apple dans l'avant-propos de la troisième édition pour mettre à jour le logiciel MacDraw Pro pour macOS 10 est resté sans suite. Nous avons donc continué à dessiner des illustrations sur des Mac pré-Intel fonctionnant sous MacDraw Pro dans l'environnement Classic des anciennes versions de macOS 10. La plupart des expressions mathématiques apparaissant dans les illustrations ont été mises en page avec le package psfrag pour LaTeX2.

Remerciements des auteurs

Depuis le démarrage de la toute première édition en 1987, nous avons collaboré de façon continue et fructueuse avec les équipes du MIT Press : directeurs, éditeurs, responsables de production… Leur soutien n'a jamais failli. Nous tenons à remercier tout particulièrement nos éditrices Marie Lee, qui nous a supportés avec patience, et Elizabeth Swayze, qui nous a menés jusqu'à la ligne d'arrivée. Merci également à la directrice Amy Brand et à Alex Hoopes.

Comme pour la précédente édition, nous étions géographiquement dispersés, travaillant au département d'informatique du Dartmouth College, au laboratoire d'informatique et

d'intelligence artificielle du MIT, au département d'ingénierie électrique et d'informatique du MIT, ainsi qu'au département d'ingénierie industrielle et de recherche opérationnelle, au département d'informatique et à l'institut des sciences des données de l'université Columbia. Pendant la pandémie de COVID-19, nous avons principalement travaillé à domicile. Nous remercions nos universités et nos collègues respectifs de nous avoir fourni des environnements aussi favorables et stimulants. Au moment de terminer ce projet d'édition, ceux d'entre nous, qui ne sont pas à la retraite, sont impatients de retourner dans leurs universités respectives, maintenant que la pandémie semble s'atténuer.

Julie Sussman, P.P.A., nous a une fois de plus tirés d'affaire en effectuant la révision technique dans des délais très courts. Sans elle, ce livre serait truffé d'erreurs (ou, en comporterait bien plus) et serait beaucoup moins lisible. Julie, nous te serons éternellement reconnaissants. Les erreurs qui subsistent relèvent de la responsabilité des auteurs (et ont probablement été introduites après la relecture de Julie).

Des dizaines d'erreurs présentes dans les éditions précédentes ont été corrigées lors de la création de cette édition. Nous remercions nos lecteurs, trop nombreux pour en dresser une liste, qui nous ont signalé des erreurs et suggéré des améliorations au fil des ans.

Nous avons bénéficié d'une aide considérable pour la préparation de certains des nouveaux éléments de cette édition. Neville Campbell (indépendant), Bill Kuszmaul du MIT et Chee Yap de l'université de New York ont fourni de précieux conseils concernant le traitement des récurrences au chapitre 4. Yan Gu de l'université de Californie à Riverside a fourni des commentaires sur les algorithmes parallèles au chapitre 26. Rob Shapire de Microsoft Research a modifié notre approche du matériel sur l'apprentissage automatique grâce à ses commentaires détaillés sur le chapitre 33. Qi Qi, du MIT, nous a aidés à analyser le problème de Monty Hall (problème C-1). Molly Seaman et Mary Reilly, du MIT Press, et Wojciech Jarosz, du Dartmouth College, nous ont aidé dans la conception de nos nouvelles illustrations. Yichen (Annie) Ke et Linda Xiao, qui ont depuis obtenu leur diplôme à Dartmouth, ont aussi participé à la création des illustrations. Linda a également produit de nombreuses implémentations Python disponibles sur le site web du livre.

Enfin, nous remercions nos épouses, Wendy Leiserson, Gail Rivest, Rebecca Ivry et feu Nicole Cormen, ainsi que nos familles. La patience et les encouragements de ceux qui nous aiment ont rendu ce projet possible. Nous leur dédions donc ce livre avec affection.

THOMAS H. CORMEN — *Lebanon, New Hampshire*
CHARLES E. LEISERSON — *Cambridge, Massachusetts*
RONALD L. RIVEST — *Cambridge, Massachusetts*
CLIFFORD STEIN — *New York, New York*

Juin 2021

Partie I Introduction

Pour pouvoir créer des algorithmes et analyser leur efficacité, vous devez être capable de décrire leur conception et leur fonctionnement. Vous devez aussi disposer des outils mathématiques qui permettent de démontrer que vos algorithmes font ce qui est attendu d'eux et qu'ils le font de manière efficace. Cette partie vous propose de découvrir les fondamentaux de ces activités. Les parties suivantes du livre s'appuieront sur ces bases.

Le chapitre 1 propose une présentation générale des algorithmes et de leur rôle dans les systèmes informatiques modernes. Il définit le concept d'algorithme en donnant quelques exemples. Il rappelle en outre que les algorithmes constituent un domaine technologique à part entière, au même titre que le matériel, les interfaces graphiques (GUI), la programmation orientée objet ou les réseaux informatiques.

Dans le chapitre 2, nous découvrons nos premiers algorithmes, lesquels répondent au besoin de trier une séquence de n nombres. Les programmes sont écrits dans un pseudo-langage informatique nommé pseudocode. Ce langage ne correspond pas directement à un langage de programmation réel, mais le pseudocode présente la structure détaillée de l'algorithme de manière suffisamment claire pour que vous puissiez le transposer dans le langage de programmation de votre choix. Les deux algorithmes de tri que nous étudierons dans ce chapitre sont le tri par insertion, qui utilise une stratégie incrémentale, et le tri fusion, qui se fonde sur une technique récursive baptisée *diviser-pour-régner*. Pour chacun des deux algorithmes, le temps requis pour l'exécution croît avec la valeur de n, mais le taux de croissance n'est pas identique. Nous déterminerons ces temps d'exécution dans ce même chapitre et introduirons une notation « asymptotique », très pratique pour les exprimer.

Le chapitre 3 définit la notation asymptotique de façon plus précise, notation qui nous servira à borner par au-dessus et en dessous la croissance des fonctions, qui servent le plus souvent à décrire le temps d'exécution des algorithmes. Le chapitre commence par définir de manière informelle les notations asymptotiques les plus utilisées en donnant un exemple d'application. Il définit ensuite formellement cinq notations asymptotiques et présente des conventions sur la façon de les assembler. Le reste du chapitre consiste essentiellement en une présentation de quelques notations mathématiques afin de s'assurer que celles que auxquelles vous êtes habitué concordent avec celles adoptées dans la suite du livre. Cette présentation n'a pas pour but de vous enseigner de nouveaux concepts mathématiques.

Le chapitre 4 va plus loin dans la méthode diviser-pour-régner, évoquée au chapitre 2. Il fournit deux exemples supplémentaires d'algorithmes diviser-pour-régner pour la multiplication de deux matrices carrées, dont la surprenante méthode de Strassen. Le chapitre donne des méthodes pour la résolution des récurrences, qui sont très utiles pour décrire les durées d'exécution des algorithmes récursifs. Dans la méthode de substitution, vous devinez une réponse puis vous prouvez qu'elle est correcte. Les arbres de récursivité constituent un moyen de générer une supposition. Le chapitre 4 présente également la puissante technique du « théorème maître » [1], que vous pouvez souvent utiliser pour résoudre les récurrences issues des algorithmes de type diviser-pour-régner. Le chapitre fournit la preuve d'un théorème fondamental dont dépend le théorème maître, mais n'hésitez pas à utiliser ce dernier sans vous sentir obligé de plonger dans sa démonstration. Le chapitre se termine sur quelques sujets avancés.

Le chapitre 5 aborde l'analyse probabiliste et les algorithmes probabilistes. L'analyse probabiliste sert d'abord à déterminer le temps d'exécution d'un algorithme dans le cas où, en raison d'une distribution probabiliste intrinsèque, ce temps varierait pour différentes entrées bien qu'elles soient de même taille. Dans certains cas, vous pourrez supposer que les entrées obéissent à une distribution probabiliste connue, de sorte que vous pourrez faire la moyenne des temps d'exécution sur l'ensemble des entrées possibles. Dans d'autres cas, la distribution probabiliste ne viendra pas des entrées, mais de décisions aléatoires prises au cours de l'exécution de l'algorithme. Un algorithme dont le comportement dépend non seulement de l'entrée, mais aussi de valeurs produites par un générateur de nombres aléatoires est un algorithme probabiliste. Vous pouvez recourir à un tel algorithme pour garantir une distribution probabiliste sur les entrées, et ainsi vous assurer qu'aucune entrée en particulier n'entraîne systématiquement de mauvaises performances, voire pour borner les taux d'erreur de ceux des algorithmes autorisés à produire, dans certaines limites, des résultats erronés.

Les annexes A à D regroupent de nombreux sujets mathématiques fondamentaux qui vous faciliteront la lecture du livre. Vous avez certainement déjà rencontré une bonne partie de ces notions (bien que les définitions et conventions de notation spécifiques que nous utilisons ici puissent, à l'occasion, différer de celles que vous connaissez). Considérez ces annexes comme une sorte de référence. En revanche, la plupart des concepts présentés dans cette partie I sont probablement inédits pour vous. Tous les chapitres de cette première partie et toutes les annexes ont été rédigés dans un esprit didactique.

1. Ou théorème sur les récurrences de partition.

1 Rôle des algorithmes en informatique

Qu'est-ce qu'un algorithme ? En quoi l'étude des algorithmes est-elle utile ? Quel est le rôle des algorithmes par rapport aux autres technologies informatiques ? Ce chapitre a pour objectif de répondre à ces questions.

1.1 Algorithmes

Commençons par une définition informelle du terme ***algorithme*** : c'est une procédure calculatoire clairement définie qui attend en ***entrée*** une ou plusieurs valeurs, et génère en ***sortie*** une ou plusieurs valeurs, l'opération devant être réalisée en temps fini. Un algorithme est donc une suite d'étapes de traitement qui transforment des données d'entrée en données de sortie.

On peut aussi considérer un algorithme comme un outil permettant de résoudre un ***problème informatique*** correctement spécifié. L'énoncé du problème indique, en termes généraux, la relation désirée entre l'entrée et la sortie pour des instances d'un problème de taille arbitraire. L'algorithme décrit une procédure de traitement permettant d'obtenir cette relation entre entrée et sortie pour toutes les instances du problème.

Supposons, par exemple, qu'il faille trier une suite de nombres dans l'ordre croissant. Ce problème revient fréquemment en pratique. Il offre de multiples possibilités pour l'introduction de nombreuses techniques de conception et d'outils d'analyse standard. Nous définissons formellement le ***problème de tri*** ainsi :

Entrée : Une suite de n nombres $\langle a_1, a_2, \cdots, a_n \rangle$.

Sortie : Une permutation (réorganisation) $\langle a'_1, a'_2, \cdots, a'_n \rangle$ de la suite donnée en entrée, telle que $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Ainsi, à partir de la suite $\langle 31, 41, 59, 26, 41, 58 \rangle$, un algorithme de tri correct produit en sortie la suite $\langle 26, 31, 41, 41, 58, 59 \rangle$. Une telle suite donnée en entrée est appelée une ***instance*** du problème de tri. En général, une ***instance de problème*** [1] consiste en l'entrée (satisfaisant aux contraintes, quelles qu'elles soient, imposées dans l'énoncé du problème) requise pour le calcul d'une solution au problème.

1. Lorsque le contexte du problème est connu, il arrive que ses instances soient elles-mêmes appelées « problèmes ».

Le tri est une opération majeure en informatique car de nombreux programmes l'exploitent en tant que phase intermédiaire, ce qui explique que l'on dispose d'un grand nombre d'algorithmes de tri. L'algorithme optimal pour une application spécifique dépend, entre autres facteurs, du nombre d'éléments à trier, de la façon dont les éléments sont plus ou moins triés au départ, des restrictions éventuelles concernant les valeurs des éléments, de l'architecture de l'ordinateur, ainsi que du type de périphérique de stockage à utiliser : circuits mémoire, disques durs ou bandes magnétiques.

Un algorithme pour un problème informatique est dit ***correct*** si, pour chaque instance de problème fournie en entrée, il ***s'arrête*** (se termine en un temps limité) en produisant la solution correcte. On dit que cet algorithme correct ***résout*** le traitement demandé. Un algorithme incorrect peut, pour certaines instances d'entrées, ne jamais se terminer, ou bien se terminer en fournissant une réponse erronée. Contrairement à ce que vous pourriez en conclure, un algorithme même incorrect peut s'avérer utile dans certains cas, à condition que vous ayez le contrôle de son taux d'erreur. Nous en verrons un exemple au chapitre 31, quand nous étudierons des algorithmes pour déterminer de grands nombres premiers. En général, cependant, nous nous intéresserons seulement aux algorithmes corrects.

Un algorithme peut être spécifié en langage humain ou en langage informatique, mais peut aussi être matérialisé par un système physique. L'unique obligation est que la spécification fournisse une description précise de la procédure de traitement à suivre.

Types de problèmes susceptibles d'être résolus par des algorithmes

Le tri n'est absolument pas le seul problème informatique pour lequel ont été inventés des algorithmes. (Vous l'avez sans doute deviné au vu de la taille de ce livre.) Les applications concrètes des algorithmes sont innombrables, entre autres :

- Le projet du génome humain a bien progressé vers ses objectifs qui consistent à identifier les 30 000 gènes de l'ADN humain, de déterminer les séquences d'environ 3 milliards de paires de bases chimiques qui constituent l'ADN humain, de stocker ces informations dans des bases de données et de développer des outils d'analyse de données. Chacune de ces étapes exige des algorithmes très élaborés. Les solutions aux divers problèmes sous-jacents sortent du cadre de ce livre, mais les concepts traités dans cet ouvrage sont utilisés pour résoudre ces problèmes de biologie, permettant ainsi aux scientifiques de faire leur travail tout en utilisant les ressources avec efficacité. Nous verrons, dans le chapitre 14, l'importante technique de résolution de plusieurs de ces problèmes de biologie qui se nomme la ***programmation dynamique***. Elle convient en particulier aux problèmes de recherche de similarité entre séquences d'ADN. Cela permet de gagner du temps, aussi bien du temps humain que du temps machine, et de l'argent, puisque les techniques de laboratoire peuvent ainsi extraire des données un plus grand volume d'informations.
- Le réseau Internet permet à des personnes réparties tout autour du monde de consulter et de récupérer rapidement d'énormes volumes de données. Les sites Internet reposent

sur des algorithmes intelligents qui leur permettent de gérer et de manipuler ces grands volumes. Parmi les problèmes dont la résolution fait un usage essentiel d'algorithmes, citons la recherche de routes optimales pour les transferts de données (ce genre de technique sera présentée au chapitre 22) ou l'utilisation d'un moteur de recherche pour trouver rapidement les pages contenant tel ou tel type de données (les techniques concernées seront vues aux chapitres 11 et 32).

- Le commerce électronique, qui permet de négocier et d'échanger, de manière électronique, des biens et des services, exige que soit préservée la confidentialité des données personnelles telles que les numéros de carte de crédit, les mots de passe et les identifiants bancaires. La cryptographie à clé publique et les signatures numériques (traitées au chapitre 31), qui font partie des technologies fondamentales employées dans ce contexte, s'appuient sur des algorithmes numériques et sur la théorie des nombres.
- Dans l'industrie et le commerce, il faut souvent optimiser l'affectation de ressources limitées. Une compagnie pétrolière voudra savoir où forer ses puits de façon à maximiser les profits escomptés. Un candidat aux élections voudra savoir dans quels supports publicitaires il doit investir pour maximiser ses chances d'être élu. Une compagnie aérienne voudra réaliser l'affectation des équipages aux vols de telle façon que les coûts soient minimisés, et les vols assurés tout en respectant la législation. Un fournisseur de services Internet voudra savoir où déployer des compléments de ressources pour servir ses clients plus efficacement. Ce sont là des exemples de problèmes susceptibles d'être résolus en les modélisant en programmation linéaire, comme nous le verrons dans le chapitre 29.

Bien que certains détails de ces exemples sortent du cadre de cet ouvrage, nous montrerons plusieurs techniques fondamentales applicables à ces catégories de problèmes. Nous verrons aussi comment résoudre un certain nombre de problèmes spécifiques, parmi lesquels :

- Vous disposez d'une carte routière sur laquelle sont indiquées toutes les distances entre chaque paire d'intersections voisines et vous souhaitez déterminer le trajet le plus court entre deux intersections. Le nombre d'itinéraires possibles peut être énorme, même si l'on n'autorise pas les routes à se recroiser. Comment trouver le trajet le plus court ? Vous pouvez commencer par modéliser la carte (qui, elle-même, modélise les routes réelles) sous forme d'un graphe (sujet traité dans la partie VI et dans l'annexe B). Vous cherchez ensuite, dans ce graphe, à déterminer le chemin le plus court entre deux sommets. Le chapitre 22 montrera comment résoudre ce problème de manière efficace.
- Soit un schéma mécanique défini en relation avec un catalogue de pièces détachées, où chaque pièce peut être constituée d'instances d'autres pièces. Il faut trier la liste des pièces de telle façon que chaque pièce apparaisse avant toutes les pièces qui l'utilisent. Si le schéma se compose de n pièces, il existe $n!$ ordres possibles, où $n!$ désigne la fonction factorielle. Comme la fonction factorielle croît encore plus vite qu'une fonction exponentielle, vous ne pouvez pas générer chaque ordre possible puis vérifier que, au sein de cet ordre, chaque pièce apparaît bien avant celles qui y font référence

(sauf si le nombre de pièces est très petit). Ce problème relève du tri topologique. Le chapitre 20 montrera comment le résoudre efficacement.

- Un médecin doit évaluer la nature cancéreuse ou pas d'un tissu sur une image. Il dispose des clichés de nombreuses autres tumeurs, certaines malignes et d'autres pas. Une tumeur maligne est susceptible d'être plus semblable à d'autres tumeurs malignes qu'à une tumeur bénigne, et de même pour une tumeur bénigne. En utilisant un algorithme de regroupement, comme au chapitre 33, le médecin peut déterminer le résultat le plus probable.
- Vous avez besoin de compresser un fichier de texte volumineux pour qu'il occupe moins d'espace. Il existe de nombreuses méthodes pour y parvenir, notamment la compression LZW, qui recherche les séquences de caractères répétitives. Le chapitre 15 étudie une approche différente, le « codage Huffman », qui code les caractères par séquences de bits de différentes longueurs, les caractères les plus fréquents étant codés par les séquences de bits les plus courtes.

Ces listes de domaines applicatifs sont loin d'être exhaustives (le poids de ce livre en témoigne), mais elle suffisent à souligner deux caractéristiques que l'on retrouve dans nombre de problèmes algorithmiques intéressants :

1. Pour chaque problème, il existe plusieurs solutions envisageables, mais la plupart d'entre elles ne le résolvent pas. Trouver une solution qui convienne, voire la « meilleure solution », sans devoir examiner explicitement chaque solution possible, peut représenter un défi.
2. Ces problèmes ont des applications concrètes. L'exemple le plus évident parmi ceux mentionnés plus haut est la recherche du plus court chemin. Une entreprise de transport par route ou par voie ferrée a intérêt à trouver les trajets les plus courts, pour réduire les coûts en main-d'œuvre et en énergie. Un nœud de routage Internet doit déterminer le chemin le plus court à travers le réseau pour minimiser le délai d'acheminement des messages. Pour aller, par exemple, de Brest à Belfort, il sera utile de profiter du guidage d'itinéraire produit par une application de navigation GPS.

Un problème résolu par des algorithmes ne possède pas toujours un ensemble facilement identifié de solutions candidates. Supposez, par exemple, que vous ayez un ensemble de valeurs numériques représentant les échantillons d'un signal recueillis à fréquence fixe. La transformée de Fourier discrète de ces échantillons convertit le domaine temporel en domaine fréquentiel. Autrement dit, elle produit une approximation du signal sous forme d'une somme pondérée de sinusoïdes, représentant l'intensité des diverses fréquences qui, une fois additionnées, constituent une approximation du signal échantillonné. Outre qu'elles sont au cœur des techniques de traitement du signal, les transformées de Fourier discrètes ont des applications dans la compression de données et dans la multiplication des grands polynômes et grands entiers. Le chapitre 30 présente un algorithme efficace pour résoudre ce problème, la transformée de Fourier rapide (ou FFT). Le même chapitre esquisse le schéma d'un circuit matériel de traitement FFT.

Structures de données

Cet ouvrage présente plusieurs ***structures de données***. Une structure de données est un moyen de stocker et d'organiser des données pour faciliter l'accès à ces données et leur modification. Le choix de la ou des structures de données les mieux adaptées à chaque problème constitue une étape cruciale dans la conception des algorithmes. Il n'existe aucune structure de données répondant à tous les besoins. Vous devez donc connaître les forces et les limites de plusieurs de ces structures.

Technique

Vous pouvez considérer cet ouvrage comme un « livre de recettes » pour algorithmes, mais vous risquez tôt ou tard de tomber sur un problème pour lequel il n'existe pas d'algorithme publié (c'est le cas de beaucoup d'exercices et problèmes du livre). Cet ouvrage vous enseignera des techniques de conception et d'analyse d'algorithme, afin que vous puissiez créer des algorithmes de votre cru, puis prouver qu'ils fournissent la bonne réponse et analyser leur efficacité. Les différents aspects de la résolution des problèmes algorithmiques sont traités dans différents chapitres. Certains chapitres sont dédiés à des problèmes spécifiques, par exemple la recherche de médianes et de statistiques d'ordre au chapitre 9, le calcul d'arbres couvrants minimaux au chapitre 21 et la détermination d'un flot maximal dans un réseau au chapitre 24. D'autres chapitres présentent des techniques, par exemple la technique diviser-pour-régner aux chapitres 2 et 4, la programmation dynamique au chapitre 14 et l'analyse amortie au chapitre 16.

Problèmes difficiles

Une bonne partie de ce livre concerne les algorithmes efficaces. La mesure habituelle de l'efficacité est la vitesse, c'est-à-dire la durée que requiert un algorithme pour produire ses résultats. Il existe cependant des problèmes pour lesquels on ne connaît aucun algorithme qui achève son traitement dans un délai raisonnable. Le chapitre 34 étudie un sous-ensemble intéressant de ces problèmes, connus sous l'appellation de problèmes NP-complets.

En quoi les problèmes NP-complets sont-ils intéressants ? Tout d'abord, nous n'avons encore jamais trouvé d'algorithme efficace pour un problème NP-complet, mais personne n'a, à ce jour, prouvé qu'un tel algorithme ne peut pas exister. Autrement dit, on ne sait pas s'il existe des algorithmes efficaces pour les problèmes NP-complets. Deuxièmement, l'ensemble des problèmes NP-complets possède la propriété remarquable suivante : s'il s'avère qu'il existe un algorithme efficace pour l'un quelconque des problèmes de cette famille, alors il existe des algorithmes efficaces pour tous. Cette relation entre les problèmes NP-complets rend d'autant plus frustrante l'absence de solution efficace. Enfin, plusieurs problèmes NP-complets ressemblent, sans être identiques, à des problèmes pour lesquels nous connaissons des algorithmes efficaces. Les spécialistes sont intrigués par le fait qu'un

petit changement dans l'énoncé du problème peut entraîner un changement majeur au niveau de l'efficacité du meilleur algorithme connu.

Vous devez avoir quelques connaissances concernant les problèmes NP-complets étant donné que, chose surprenante, certains d'entre eux apparaissent souvent dans les applications concrètes. Si l'on vous demande de concocter un algorithme efficace pour un problème NP-complet, vous risquez de perdre beaucoup de temps à chercher pour rien. En revanche, si vous arrivez à démontrer que c'est un problème NP-complet, vous pourrez alors réorienter vos efforts et développer un algorithme approximatif efficace qui fournira une solution acceptable sans être nécessairement optimale.

À titre d'exemple concret, prenons le cas d'une entreprise de livraison ayant un entrepôt central. Chaque jour, chaque camion est chargé à l'entrepôt puis part faire sa tournée de livraison. En fin de journée, le camion retourne à l'entrepôt, prêt au chargement du lendemain. Pour réduire les coûts, l'entreprise veut choisir un ordre de livraison pour limiter le plus possible la distance parcourue par chaque camion. Ce problème, qui n'est autre que le fameux « problème du voyageur de commerce », est un problème NP-complet.[2] Il n'y a pas d'algorithme efficace connu. Sous certaines hypothèses, cependant, il existe des algorithmes efficaces qui calculent des distances proches de la distance minimale. Le chapitre 35 présente ce genre d'algorithmes, dits ***algorithmes d'approximation***.

Autres modèles informatiques

Pendant de nombreuses années, on pouvait s'appuyer sur le fait que les vitesses d'horloge des processeurs augmentaient de façon régulière. Les limitations physiques présentent, cependant, une barrière fondamentale à l'augmentation continue des vitesses d'horloge : comme la densité de courant croît de façon superlinéaire avec la vitesse d'horloge, les circuits risquent d'être détruits par surchauffe à partir d'un certain niveau de vitesse. Pour pouvoir effectuer plus de calculs par seconde, les puces sont désormais conçues de façon à intégrer plusieurs « cœurs » de traitement au lieu d'un seul. On peut comparer un ordinateur multicœur à plusieurs ordinateurs séquentiels sur une même puce. Autrement dit, il s'agit d'un type « d'ordinateur parallèle ». Pour obtenir les meilleures performances des ordinateurs multicœur, il faut concevoir les algorithmes dans une approche de parallélisme. Le chapitre 26 présente un modèle pour les algorithmes « parallélisables » qui tirent profit des cœurs de traitement multiples. Ce modèle présente des avantages tant du point de vue théorique que pratique, et de nombreuses plateformes modernes de programmation parallèle adoptent quelque chose de similaire.

La plupart des exemples présentés dans ce livre supposent que la totalité des données d'entrée est disponible lorsqu'un algorithme commence à fonctionner. La majorité des

2. Pour être précis, seuls les problèmes de décision, c'est-à-dire ceux dont la réponse est « oui ou non », peuvent être NP-complets. La version décisionnelle du problème du voyageur de commerce demande s'il existe un ordre de passage dont la distance totale ne dépasse pas un plafond donné.

travaux sur la conception d'algorithmes reposent sur la même hypothèse. Cependant, dans le monde réel, pour de nombreux exemples importants, les données d'entrée arrivent au fil du temps et l'algorithme doit décider de la marche à suivre sans savoir quelles données arriveront à l'avenir. Dans un centre de données, les tâches arrivent et partent constamment, et un algorithme de planification doit décider quand et où exécuter quelle tâche, sans savoir quelles tâches arriveront à l'avenir. Sur Internet, le trafic doit être acheminé en fonction de l'état actuel, sans savoir son état futur. Dans les hôpitaux, les services d'urgence prennent des décisions de triage des patients sans savoir quand d'autres patients arriveront et quels traitements ils requerront. Les algorithmes qui reçoivent leurs données au fil du temps, plutôt que d'avoir toutes les données à disposition dès le départ, sont des ***algorithmes incrémentaux*** (*online algorithms*). Le chapitre 27 les étudie.

Exercices

1.1-1

Donnez un exemple concret tiré de votre expérience personelle qui exige un tri, puis un exemple de recherche de plus court chemin entre deux points.

1.1-2

À part la vitesse, qu'est-ce qui doit être pris en compte pour mesurer l'efficacité dans un contexte réel ?

1.1-3

Sélectionnez une structure de données que vous connaissez déjà et étudiez-en les avantages et les inconvénients.

1.1-4

En quoi le problème du chemin minimal et celui du voyageur de commerce, mentionnés plus haut, se ressemblent-ils ? En quoi sont-ils différents ?

1.1-5

Proposez un problème concret pour lequel seule conviendra la solution optimale. Trouvez ensuite un problème pour lequel une solution « approchée » conviendra.

1.1-6

Décrivez un problème du monde réel dans lequel les données d'entrée sont parfois toutes disponibles au moment où vous en avez besoin, mais parfois n'arrivent que progressivement.

1.2 Les algorithmes en tant que technologie

Si la puissance des ordinateurs était devenue infinie et si le coût des composants des circuits mémoire était devenu négligreable, devriez-vous encore vous soucier d'étudier les algorithmes ? Absolument, ne serait-ce que pour montrer que votre solution ne reboucle pas indéfiniment et se termine en donnant la réponse correcte.

Si les ordinateurs étaient infiniment rapides, n'importe quelle méthode correcte de résolution de problème ferait l'affaire. Vous voudriez sans doute que votre solution entre dans le cadre d'une bonne méthodologie d'ingénierie (par exemple, que votre implémentation soit bien conçue et bien documentée), mais vous privilégierez le plus souvent la méthode la plus simple à mettre en œuvre.

Si rapides que puissent être les ordinateurs, ils ne sont pas infiniment rapides. Le temps machine est une ressource limitée, donc précieuse. Vous connaissez l'adage « Le temps, c'est de l'argent ». En fait, le temps est encore plus précieux que l'argent : on peut regagner de l'argent après en avoir perdu, mais le temps passé ne se rachète pas. De même, si bon marché que puissent être les composants mémoire, ils ne sont ni gratuits, ni disponibles en quantité illimitée. Vous devez donc rechercher des algorithmes efficaces en temps de traitement et en espace mémoire.

Efficacité

Il arrive souvent que des algorithmes différents conçus pour résoudre le même problème se distinguent fortement en matière d'efficacité. Ces différences peuvent être bien plus importantes que celles liées aux capacités du matériel et à la qualité du logiciel.

À titre d'exemple, nous verrons deux algorithmes de tri dans le chapitre 2. Le premier, appelé ***tri par insertion*** (*insertion sort*), prend un temps approximativement égal à $c_1 n^2$ pour trier n éléments, c_1 étant une constante indépendante de n. La durée du tri est donc à peu près proportionnelle à n^2. Le second, appelé ***tri fusion*** (*merge sort*), prend un temps approximativement égal à $c_2 n \cdot \lg n$, où $\lg n$ désigne $\log_2 n$ et c_2 est une autre constante indépendante, elle aussi, de n. Le tri par insertion a généralement un facteur constant inférieur à celui du tri fusion, de sorte que $c_1 < c_2$. Nous verrons que les facteurs constants peuvent être beaucoup moins significatifs, au niveau de la durée d'exécution, que la dépendance par rapport à la taille de l'entrée n. Choisissons d'écrire la durée d'exécution du tri par insertion sous la forme $c_1 n \cdot n$ et celle du tri fusion sous la forme $c_2 n \cdot \lg n$. Nous voyons alors que, tandis que le tri par insertion a un facteur de n dans sa durée, le tri fusion a un facteur de $\lg n$, ce qui est très inférieur. Par exemple, quand $n = 1\,000$, $\lg n$ vaut environ 10, et quand n vaut un million, $\lg n$ ne vaut qu'environ 20. Le tri par insertion est généralement plus rapide que le tri fusion pour de petits volumes de données mais, dès que le nombre n d'éléments à trier devient suffisamment grand, l'avantage du tri fusion ($\lg n$ comparé à n) fait plus que

compenser la différence entre les facteurs constants. Même si c_1 est très inférieur à c_2, il y a toujours un palier au-delà duquel le tri fusion devient plus rapide.

Pour un exemple concret, comparons un ordinateur rapide (ordinateur A) exécutant un tri par insertion et un ordinateur lent (ordinateur B) exécutant un tri fusion. Chacune des deux machines doit trier un tableau de dix millions de nombres. (Cela peut paraître énorme, mais si ces nombres sont des entiers sur huit octets, les données d'entrée occupent environ 80 Mo, ce qui est très peu même pour la mémoire d'un ordinateur portable de début de gamme actuel.) Supposez que l'ordinateur A soit capable d'exécuter 10 milliards d'instructions par seconde (à l'heure actuelle, aucun ordinateur non parallélisé n'en est capable) alors que l'ordinateur B ne réussit à exécuter que dix millions d'instructions par seconde (bien moins que les ordinateurs courants actuels). Ainsi, l'ordinateur A est 1 000 fois plus rapide que l'ordinateur B en puissance de calcul brute. Pour rendre la différence encore plus sensible, supposez que ce soit le meilleur programmeur du monde qui rédige le code du tri par insertion directement en langage machine (langage assembleur) pour l'ordinateur A et que le code résultant requiert $2\,n^2$ instructions pour trier n nombres. Supposez, en outre, que le tri fusion soit programmé par un programmeur médiocre utilisant un langage de haut niveau avec un compilateur peu performant, de sorte que le code binaire résultant représente $50\,n \lg n$ instructions. Pour le tri par insertion de 10 millions de nombres, l'ordinateur A réclame :

$$\frac{2 \cdot (10^7)^2 \text{ instructions}}{10^{10} \text{ instructions/seconde}} = 20\,000 \text{ secondes (+ de 5 heures } 1/2) \,,$$

alors que l'ordinateur B demande pour son tri fusion :

$$\frac{50 \cdot 10^7 \lg 10^7 \text{ instructions}}{10^7 \text{ instructions/seconde}} \approx 1\,163 \text{ secondes (- de 20 minutes)} \,.$$

Avec un algorithme dont le temps d'exécution croît plus lentement, même si le compilateur est médiocre, la machine B travaille 17 fois plus vite que la machine A ! L'avantage du tri fusion est encore plus net avec 100 millions de nombres : là où le tri par insertion demande plus de 23 jours, le tri fusion prend moins de quatre heures. Bien que le chiffre de 100 millions semble énorme, rappelons que, chaque heure, plus de 200 millions de recherches sont lancées sur le Web, plus de 100 millions de courriels sont émis chaque minute, et certaines des plus petites galaxies connues (les naines ultra-compactes) réunissent de l'ordre de 100 millions d'étoiles. En général, plus la taille du problème augmente, plus l'avantage relatif du tri fusion augmente aussi.

Algorithmes et autres technologies

L'exemple précédent montre que les algorithmes doivent être considérés comme une ***technologie***, au même titre que le matériel informatique. Les performances globales d'un système dépendent autant des algorithmes que du matériel. Comme toute technologie informatique, les algorithmes ne cessent de progresser.

Vous pourriez vous demander si les algorithmes gardent autant d'importance sur les puissants ordinateurs modernes, compte tenu de l'état d'avancement d'autres technologies telles que les suivantes :

- les architectures sophistiquées et les technologies de fabrication innovantes ;
- les interfaces utilisateur graphiques (GUI) conviviales et ergonomiques ;
- les systèmes orientés objet ;
- les technologies intégrées du Web ;
- les technologies réseau à haut débit (filaire et sans fil) ;
- les techniques d'apprentissage machine ou automatique ;
- les appareils et équipements connectés nomades.

La réponse est oui. Même si certaines applications n'exigent pas explicitement d'algorithmes au niveau de l'application elle-même (c'est le cas, par exemple, de certaines applications simples utilisées sur le Web), la plupart en dépendent. Prenez le cas d'un service Web pour établir des itinéraires. Sa mise en œuvre exige des matériels rapides, une interface graphique, des technologies réseau sophistiquées à large échelle, voire une approche orientée objets. À cela s'ajoutent des algorithmes pour certains traitements tels que la recherche d'itinéraire (utilisant vraisemblablement un algorithme de plus court chemin), le tracé de cartes et l'interpolation d'adresses.

Qui plus est, même une application qui n'incorpore pas d'algorithmes au niveau de son code source dépend fortement de certains algorithmes. L'application requiert-elle du matériel performant ? La conception de ce matériel a nécessité des algorithmes. Comporte-t-elle une interface graphique ? Toute interface graphique repose sur des algorithmes. Si l'application fonctionne en réseau, elle dépend de la fonction de routage qui s'appuie fondamentalement sur des algorithmes. Si l'application a été écrite dans un autre langage que du code machine, alors elle a été traduite par un compilateur, un interpréteur ou un assembleur, toutes ces belles créations qui font un usage intensif des algorithmes. Les algorithmes sont nécessaires à la plupart des technologies employées dans les ordinateurs modernes.

L'apprentissage automatique (*machine learning*) peut être considéré comme une technique consistant à effectuer des tâches algorithmiques sans avoir demandé au préalable de spécifier explicitement les algorithmes exploités. Il s'agit de déduire des modèles à partir des données pour produire automatiquement une solution. À première vue, l'apprentissage automatique, puisqu'il automatise le processus de conception algorithmique, peut sembler rendre obsolète l'apprentissage des algorithmes par l'humain. Bien au contraire : l'apprentissage automatique est lui-même une collection d'algorithmes, sous un nouveau nom. En outre, il semble actuellement que ses succès concernent principalement des problèmes pour lesquels nous, en tant qu'humains, ne comprenons pas vraiment quel est le bon algorithme. Les exemples les plus marquants sont la vision par ordinateur et la traduction automatique des langues. Pour les problèmes algorithmiques que les humains comprennent bien, comme la plupart des problèmes abordés dans ce livre, un algorithme éprouvé et conçu pour résoudre un problème spécifique est, en général, plus efficace que l'approche par apprentissage automatique.

La science des données (*data science*) est un domaine interdisciplinaire dont l'objectif est d'extraire des connaissances et des concepts à partir de données structurées ou non structurées. Elle se fonde sur les statistiques, la science informatique et des méthodes d'optimisation. La conception et l'analyse d'algorithmes y sont fondamentales. Les techniques de base de la science des données, qui se recoupent largement avec celles de l'apprentissage machine, exploitent un grand nombre des algorithmes présentés dans cet ouvrage.

Les performances sans cesse accrues des ordinateurs nous permettent de leur confier le traitement de problèmes de plus en plus complexes. Comme nous l'avons constaté dans la comparaison entre le tri par insertion et le tri fusion, c'est avec les problèmes de grande taille que les différences d'efficacité entre algorithmes deviennent les plus flagrantes.

La possession de solides bases en algorithmique est ce qui caractérise un programmeur vraiment compétent. Les technologies informatiques modernes permettent de réussir certaines tâches, même sans bien s'y connaître en algorithmique, mais une bonne formation en algorithmique permet d'aller beaucoup plus loin.

Exercices

1.2-1

Donnez un exemple d'application exigeant du contenu algorithmique au niveau applicatif, puis discutez des fonctions de l'algorithme concerné.

1.2-2

Supposons que sur un ordinateur en particulier, pour un nombre n d'éléments à trier, le tri par insertion demande $8\,n^2$ étapes alors que le tri fusion en demande $64\,n \lg n$. Quelles sont les valeurs de n pour lesquelles le tri par insertion est meilleur que le tri fusion ?

1.2-3

Quelle est la valeur minimale n pour laquelle, sur la même machine, un algorithme dont le temps d'exécution est $100\,n^2$ s'exécute plus vite qu'un algorithme dont le temps d'exécution est 2^n ?

Problèmes

1-1 Comparaison de temps d'exécution

Dans le tableau suivant, déterminez pour chaque fonction $f(n)$ et pour chaque durée t la taille maximale n d'un problème susceptible d'être résolu dans le temps t, en supposant que l'algorithme mette $f(n)$ microsecondes pour traiter le problème.

	1 sec	1 min	1 h	1 jour	1 mois	1 an	1 siècle
$\lg n$	.	.	.	.	.	.	.
$\sqrt{n}$	.	.	.	.	.	.	.
n	.	.	.	.	.	.	.
$n \lg n$	.	.	.	.	.	.	.
n^2	.	.	.	.	.	.	.
n^3	.	.	.	.	.	.	.
2^n	.	.	.	.	.	.	.
$n!$	.	.	.	.	.	.	.

Notes du chapitre

Vous trouverez un grand nombre d'excellents textes abordant le thème général des algorithmes, dont les ouvrages de Aho, Hopcroft et Ullman [5, 6], Dasgupta, Papadimitriou et Vazirani [107], Edmonds [133], Erickson [135], Goodrich et Tamassia [195, 196], Kleinberg et Tardos [257], Knuth [259, 260, 261, 262, 263], Levitin [298], Louridas [305], Mehlhorn et Sanders [325], Mitzenmacher et Upfal [331], Neapolitan [342], Roughgarden [385, 386, 387, 388], Sanders, Mehlhorn, Dietzfelbinger et Dementiev [393], Sedgewick et Wayne [402], Skiena [414], Soltys-Kulinicz [419], Wilf [455] et Williamson et Shmoys [459]. Certains des aspects plus concrets de la conception des algorithmes sont traités par Bentley [49, 50, 51], Bhargava [54], Kochenderfer et Wheeler [268], et McGeoch [321]. Une présentation du champ de recherche des algorithmes est donnée par Atallah et Blanton [27, 28] et Mehta et Sahhi [326]. Des descriptions moins techniques se trouvent dans les livres de Christian et Griffiths [92], Cormen [104], Erwig [136], MacCormick [307] et Vöcking et al. [448]. Enfin, vous trouverez des présentations d'algorithmes utilisés en biologie informatique dans les manuels de Jones et Pevzner [240], Elloumi et Zomaya [134], et Marchisio [315].

2 Premiers pas

Ce chapitre vous propose de découvrir la démarche que nous adopterons tout au long de ce livre pour guider notre réflexion sur la conception et l'analyse des algorithmes. Vous pouvez le lire indépendamment de la suite, tout en sachant qu'il contient plusieurs références à des notions abordées dans les chapitres 3 et 4. (Il contient aussi plusieurs calculs que l'annexe A montre comment résoudre.)

Nous commencerons par étudier l'algorithme du tri par insertion, ce qui nous permettra de résoudre la problématique de tri exprimée dans le chapitre 1. Nous représenterons le code source de nos algorithmes dans le format qui porte le nom de ***pseudocode***, compréhensible par tout lecteur à partir du moment où il a déjà pratiqué un peu de programmation informatique. Nous montrerons comment l'algorithme trie correctement et nous étudierons son temps d'exécution. L'analyse permettra d'introduire une notation mettant en évidence la façon dont le temps d'exécution croît en fonction du nombre d'éléments à trier. Nous présenterons ensuite le concept diviser-pour-régner appliqué à la conception d'algorithme. Cette technique nous servira à développer un second algorithme, celui du tri fusion. Nous terminerons par l'analyse du temps d'exécution de ce tri.

2.1 Tri par insertion

Notre premier algorithme, le tri par insertion (*insertion sort*), résout la ***problématique du tri*** vue dans le chapitre 1 de la façon suivante :

Entrée Une suite de n nombres $\langle a_1, a_2, \ldots, a_n \rangle$.

Sortie Une permutation (réorganisation) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ de la suite de données d'entrée, de façon que $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Les nombres à trier sont aussi désignés en tant que ***clés***. En théorie, il s'agit de trier une suite, mais les données d'entrée se présentent sous forme d'un tableau de n éléments. Le besoin de trier des nombres est souvent lié au fait que ces nombres constituent des clés qui contrôlent l'accès à d'autres données, que nous pouvons appeler données ***satellites***. Chaque paire d'entités constituée d'une clé et d'un champ de données satellites se nomme un ***enregistrement*** (*record*). Prenons l'exemple d'une feuille de calcul. Cette feuille contient une liste d'étudiants avec de nombreuses données (âge, moyenne générale, nombre de cours suivis, etc.), donc des enregistrements. Chacune de ces données peut être utilisée comme

clé, mais lorsque vous demandez au tableur de faire un tri, il change la position de tous les champs de chaque enregistrement (les données satellites) en même temps que celle de la clé. Lorsque nous décrivons un algorithme de tri, nous nous concentrons sur les clés, mais il est important de se rappeler que ces clés sont généralement couplées à des données satellites.

Dans ce livre, nous ferons généralement incarner les algorithmes par des lignes d'instructions utilisant un langage symbolique qui rappelle les langages très usités que sont le C, le C++, Java, Python [1] ou JavaScript. (Que les lecteurs pratiquant un autre langage nous pardonnent de ne pas en citer d'autres.) Si vous connaissez déjà l'un de ces langages, vous n'aurez aucun mal à étudier nos algorithmes. Ce qui fait diverger notre pseudocode d'un code source véritable c'est que, en pseudocode, nous utilisons la tournure de langage qui nous semble la plus claire et la plus concise pour décrire l'algorithme ; ne soyez donc pas surpris de voir apparaître un mélange de mots français et de mots clés typiques d'un langage informatique tels que **for** et **while**. Autre différence entre pseudocode et code effectif : le pseudocode ne se soucie pas, en principe, des problèmes d'ingénierie logicielle tels que le niveau d'abstraction des données, la modularité, la gestion des erreurs, etc. Cela permet au pseudocode de rester focalisé sur l'essence de l'algorithme concerné.

Nous commencerons par le ***tri par insertion***, qui est un algorithme efficace quand il s'agit de trier un petit nombre d'éléments. Ce tri s'inspire de la manière dont on tient habituellement les cartes à jouer à la main. Au début, la main qui sert à tenir les cartes est vide, et les cartes empilées sur la table forment une pioche. On prend une première carte avec la main droite et on la tient dans la main gauche. On prélève alors une par une d'autres cartes avec la main droite en les insérant au bon endroit dans la main gauche. Pour décider où placer chaque carte, on compare, comme le montre la figure 2.1, sa valeur à celle des cartes déjà dans son jeu, en commençant par la carte la plus à droite dans sa main gauche. Par exemple, si la carte que l'on vient de tirer a une valeur inférieure à toutes celles dans son jeu, on la place tout à fait à gauche. Ainsi, les cartes de la main sont toujours triées, et proviennent une à une du dessus de la pioche.

Notre pseudocode de tri par insertion se présente sous forme d'une procédure nommée TRI-INSERTION. Elle attend deux paramètres en entrée : un tableau A qui contient une séquence de données à trier et le nombre n d'éléments du tableau. Les valeurs occupent les positions de $A[1]$ à $A[n]$, ce qui correspond à une taille de tableau $A[1\,..\,n]$. Lorsque la procédure TRI-INSERTION se termine, le tableau contient la séquence de données triée.

Invariants de boucle et validité du tri par insertion

La figure 2.2 illustre le fonctionnement de cet algorithme sur un tableau $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. L'indice i indique la « carte courante » en cours d'insertion dans la main. Au début de chaque tour de la boucle **for** indicée par i, le ***sous-tableau*** composé des éléments $A[1\,..\,i-1]$

1. Si vous ne connaissez que le langage Python, considérez ces tableaux comme des listes.

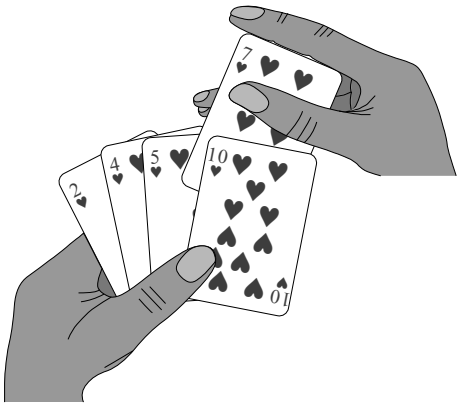

FIGURE 2.1 Tri d'une main de cartes à jouer, via un tri par insertion.

```
TRI-INSERTION(A, n)
1  for i = 2 to n
2      clé = A[i]
3      // Insère A[i] dans le sous-tableau trié A[1 . . i − 1].
4      j = i − 1
5      while j > 0 et A[j] > clé
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = clé
```

(c'est-à-dire, de $A[1]$ à $A[i-1]$) correspond aux cartes qui sont déjà dans la main. L'autre sous-tableau $A[i+1\,..\,n]$ (les éléments de $A[i+1]$ à $A[n]$) incarne la pioche des cartes encore sur la table. En fait, les éléments $A[1\,..\,i-1]$ sont ceux qui occupaient *initialement* les positions 1 à $i-1$, mais ils ont été triés entre temps. Nous utiliserons ces propriétés de $A[1\,..\,i-1]$ pour définir de manière formelle un ***invariant de boucle*** :

> Au début de chaque itération de la boucle **for** des lignes 1 à 8, le sous-tableau $A[1\,..\,i-1]$ se compose des éléments qui occupaient initialement les positions $A[1\,..\,i-1]$ mais ces éléments sont maintenant triés.

Les invariants de boucle nous aident à comprendre pourquoi un algorithme est correct. Nous devons montrer trois choses en cas d'utilisation d'un invariant de boucle :

Initialisation : L'invariant est vrai avant la première itération de la boucle.

Conservation : (également appelée « hérédité ») S'il est vrai avant une itération de la boucle, il le reste avant l'itération suivante.

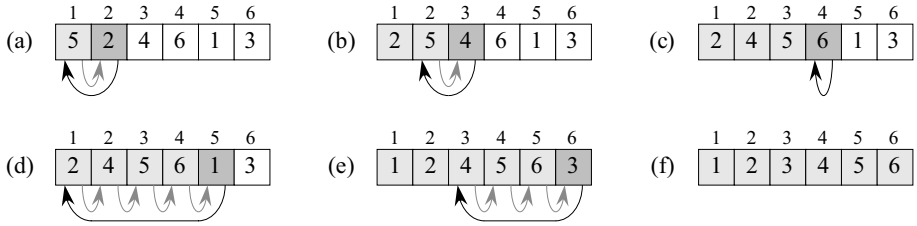

FIGURE 2.2 Fonctionnement de TRI-INSERTION(A, n) avec A contenant au départ la séquence $\langle 5, 2, 4, 6, 1, 3\rangle$ et $n = 6$. Les indices apparaissent au-dessus des cases, et les valeurs du tableau dans les cases. (a)-(e) Itérations de la boucle **for** des lignes 1 à 8. À chaque itération, la case en gris moyen la plus à droite (a2, b3, c4, d5, e6) contient la clé lue dans $A[i]$ qui est comparée aux valeurs de ses voisines de gauche (test en ligne 5). Les flèches courtes locales montrent les déplacements des valeurs d'une position vers la droite (ligne 6), alors que les flèches longues noires (qui en enjambent des courtes à rebours) indiquent vers où est déplacée la clé en ligne 8. (f) Le tableau trié final.

Terminaison : Une fois la boucle terminée, l'invariant fournit, en même temps que la raison de la sortie de boucle, une propriété utile qui aide à montrer la validité de l'algorithme.

Si les deux premières propriétés sont vérifiées, alors l'invariant est vrai avant chaque itération de la boucle. (Bien évidemment, vous êtes libre d'utiliser des faits établis autres que l'invariant de boucle pour prouver que l'invariant reste vrai avant chaque itération.) Un invariant de boucle est une sorte d'induction mathématique dans laquelle vous prouvez qu'une propriété est respectée en prouvant un cas de base suivi d'une étape inductive. Dans notre cas, montrer que l'invariant est vrai pour la valeur initiale correspond au cas de base, et montrer qu'il le reste d'une itération à la suivante correspond à la phase inductive.

La troisième propriété est peut-être la plus importante, puisque nous utilisons l'invariant de boucle pour prouver la validité de l'algorithme. En principe, on utilise cet invariant avec la condition ayant forcé la boucle à se terminer. Dans une récurrence mathématique, la phase inductive se répète en général indéfiniment ; dans un invariant de boucle, on arrête « l'induction » quand la boucle se termine.

Voyons comment ces propriétés s'appliquent au tri par insertion.

Initialisation : Commençons par montrer que l'invariant est bien vérifié avant la première itération de la boucle[2], quand $i = 2$. Le sous-tableau $A[1\,..\,i-1]$ se résume au seul élément $A[1]$ qui est, en fait, l'élément originel de $A[1]$. Par ailleurs, un sous-tableau ne

2. Quand la boucle est une boucle **for**, le moment où nous contrôlons l'invariant avant la première itération se situe juste après l'affectation de la valeur initiale à la variable servant de compteur de boucle et juste avant le premier test dans l'en-tête de boucle. Dans le cas de TRI-INSERTION, cet instant se situe après affectation de la valeur 2 à la variable i mais avant le premier test $i \leq n$.

contenant qu'un élément est évidemment trié (c'est une trivialité), ce qui montre bien que l'invariant est vérifié avant la première itération de la boucle.

Conservation : Nous passons ensuite à la deuxième propriété : montrer que chaque itération conserve l'invariant. De manière informelle, le corps de la boucle **for** fonctionne en déplaçant $A[i-1]$, $A[i-2]$, $A[i-3]$, etc. d'une position vers la droite jusqu'à ce qu'on trouve la bonne position pour $A[i]$ (lignes 4 à 7), auquel cas on insère la valeur de $A[i]$ (ligne 8). Le sous-tableau $A[1\,..\,i]$ se compose alors des éléments situés initialement dans $A[1\,..\,i]$, mais en ordre trié. L'***incrémentation*** de i (en ajoutant 1) pour l'itération suivante de la boucle **for** préserve l'invariant de boucle.

Un traitement plus formel de la deuxième propriété nous obligerait à formuler et montrer un invariant pour la boucle **while** des lignes 5 à 7. Pour l'instant, ne nous noyons pas dans un tel formalisme. Nous nous appuierons uniquement sur notre analyse informelle pour montrer que la deuxième propriété est vérifiée pour la boucle extérieure.

Terminaison : Enfin, voyons ce qui se passe quand la boucle se termine. La variable de boucle i commence à la valeur 2 et augmente de 1 par tour tant que cette valeur ne dépasse pas la valeur n stipulée en ligne 1, donc la condition de sortie de boucle **for** est $i = n + 1$. En substituant $n + 1$ à i dans la formulation de l'invariant de boucle, le sous-tableau $A[1\,..\,n]$ se compose des éléments qui appartenaient originellement à $A[1\,..\,n]$ mais qui ont été triés depuis. Par conséquent, tout le tableau étant trié, l'algorithme est correct.

Nous aurons recours, à plusieurs reprises dans ce livre, à cette méthode basée sur les invariants de boucle pour justifier la validité des algorithmes.

Conventions concernant les blocs de pseudocode

Nous adopterons les conventions suivantes pour le pseudocode :

- La distance par rapport à la marge gauche (l'indentation) détermine le niveau d'imbrication dans une structure de bloc. Par exemple, le corps de la boucle **for** qui commence en ligne 1 se compose des lignes 2 à 8, et le corps de la boucle **while** qui commence en ligne 5 contient les lignes 6 à 7 mais pas la ligne 8. Notre style d'indentation s'applique également aux instructions **if-else**[3]. La convention consistant[4] à rendre significatif le nombre d'espaces de début de ligne (l'indentation) évite de devoir ajouter des marqueurs de blocs tels que des accolades ou des couples de mots clés **begin** et

3. Dans un bloc **if-else**, nous indentons **else** au même niveau que le **if** correspondant. La première instruction d'une branche **else** est placée sur la même ligne que ce **else**. Pour les tests multiples, nous employons **elseif** pour les autres tests que le premier.

4. N.d.T. : comme en Python.

end pour délimiter les niveaux de blocs. Cela réduit sensiblement l'encombrement tout en préservant, voire en améliorant, la clarté[5].

- Les boucles **while**, **for** et **repeat-until**, ainsi que le bloc conditionnel **if-else**, ont la même signification qu'en C, C++, Java, Python et JavaScript[6]. Dans ce livre, le compteur de boucle conserve sa valeur après la sortie de la boucle, contrairement à certaines situations se rencontrant en C++ et en Java. Juste en sortant d'une boucle **for**, la valeur du compteur de boucle est la valeur immédiatement supérieure à la borne de la boucle[7]. Nous avons utilisé cette propriété dans notre démonstration de la correction du tri par insertion. L'en-tête de la boucle **for**, en ligne 1, est **for** $i = 2$ **to** n. Quand la boucle se termine, $i = n + 1$. Nous utilisons le mot clé **to** (incrémentation) quand une boucle **for** incrémente son compteur dans chaque itération, et nous utilisons le mot clé **downto** (décrémentation) quand la boucle **for** le décrémente (réduit sa valeur d'une unité). Lorsque la valeur du compteur de boucle change d'un montant différent de l'unité (supérieur ou inférieur à 1), le montant du changement est spécifié après le mot clé facultatif **by**.
- Le symbole « // » indique que le reste de la ligne est un commentaire.
- Les variables comme i, j ou *clé* sont locales à la procédure donnée. Nous n'utiliserons pas de variables globales, sauf indication explicite.
- Pour accéder aux éléments d'un tableau, on indique le nom du tableau suivi de l'indice entre crochets droits. Ainsi, $A[i]$ représente le i-ème élément du tableau A.
  De nombreux langages de programmation imposent de faire commencer à zéro l'indexation dans les tableaux (0 est l'index du premier élément). Nous avons cependant décidé d'opter pour une approche plus intuitive pour le commun des mortels puisque les gens commencent à compter à 1 et non à 0. C'est pourquoi la plupart des tableaux présentés dans ce livre utilisent l'indexation en base 1 (mais pas tous). Lorsque la situation n'est pas claire, nous indiquerons explicitement les bornes du tableau. Si vous exploitez un algorithme que nous avons spécifié en utilisant l'indexation commençant à 1, mais que vous écrivez dans un langage de programmation qui impose l'indexation commençant à 0 (par exemple le C, C++, Java, Python ou JavaScript), vous saurez faire l'ajustement requis. Soit vous ôtez 1 de chaque indice, soit vous agrandissez le tableau d'une unité et ignorez la position 0.

---

5. Chacune des procédures et fonctions en pseudocode dans ce livre tient sur une seule page, afin que vous n'ayez pas à deviner l'alignement vertical des colonnes pour déterminer les niveaux d'indentation dans du code réparti sur plusieurs pages.
6. La plupart des langages structurés ont des constructions équivalentes, sachant que la syntaxe exacte peut différer. Python ne connaît pas les boucles **repeat** et ses boucles **for** fonctionnent un peu différemment de celles de ce livre.
7. En Python, le compteur de boucle conserve sa valeur en sortie de boucle. Cette valeur est celle en vigueur au cours du dernier tour de boucle, et non celle qui a provoqué la sortie de boucle. En effet, les boucles **for** de Python servent à balayer des listes, qui peuvent contenir d'autres types de données que des valeurs numériques.

La notation $[..]$ indique un sous-tableau. $A[1..i]$ représente donc le sous-tableau de A constitué des éléments $A[1], A[2], \ldots, A[i]$. Cette même notation[8] sert à décrire les bornes d'un tableau comme nous l'avons fait en présentant le tableau $A[1..n]$.

- Les données composées sont le plus souvent organisées en ***objets***, constitués d'***attributs***. On accède à un attribut spécifique au moyen de la syntaxe en trois parties en vigueur dans la plupart des langages de programmation orientés objet : on cite le nom de l'objet, suivi d'un point séparateur, suivi du nom de l'attribut. Par exemple, pour un objet $monobj$ qui possède un attribut nommé $monatt$, on écrit $monobj.monatt$.
  Une variable représentant un tableau ou un objet est considérée comme un pointeur vers les données représentant le tableau ou l'objet. Pour tous les attributs f d'un objet x, faire $y = x$ entraîne que $y.f = x.f$. Par ailleurs, si l'on écrit ensuite $x.f = 3$, on a ensuite non seulement $x.f = 3$ mais aussi $y.f = 3$. Autrement dit, x et y pointent vers le même objet après l'affectation $y = x$. Cette façon de considérer les tableaux et les objets est en ligne avec les conventions des langages de programmation modernes.
  Notre notation avec attributs peut « se répercuter en cascade ». Supposez, par exemple, que l'attribut f soit lui-même un pointeur vers un certain type d'objet ayant un attribut g. La notation $x.f.g$ est alors implicitement parenthésée sous la forme $(x.f).g$. Autrement dit, si nous avions assigné $y = x.f$, $x.f.g$ est la même chose que $y.g$.
  Il arrive qu'un pointeur ne fasse référence à aucun objet. Nous lui donnons dans ce cas la valeur spéciale NIL.
  Les paramètres sont transmis aux procédures ***par valeur*** : la procédure appelée reçoit une copie des paramètres. Si elle modifie la valeur d'un de ses paramètres d'entrée, le changement n'est *pas* visible pour la procédure appelante. Lorsque les paramètres sont des objets, le pointeur vers les données représentant l'objet est copié, mais pas les attributs de l'objet. Par exemple, si x est un paramètre d'une procédure appelée, l'affectation $x = y$ à l'intérieur de la procédure appelée n'est pas visible pour la procédure appelante. En revanche, l'affectation $x.f = 3$ est visible si la procédure appelante dispose d'un pointeur sur le même objet que x. Les tableaux sont transmis par pointeur, de sorte que c'est un pointeur vers le tableau qui est passé, et non le tableau complet. Les modifications apportées aux éléments que contient le tableau sont visibles pour la procédure appelante. La plupart des langages actuels fonctionnent ainsi.
- Une instruction **return** transfère immédiatement le contrôle au point d'appel dans la procédure appelante. La plupart des instructions **return** renvoient une valeur à l'appelant. Notre pseudocode diffère de nombreux langages par le fait qu'une même

8. Si vous êtes habitué à la syntaxe du langage Python, n'oubliez pas que dans ce livre, un sous-tableau noté $A[i..j]$ contient l'élément $A[j]$ alors qu'en Python, le dernier élément de $A[i..j]$ est $A[j-1]$. D'autre part, Python accepte les indices négatifs (partant de la fin de la liste). Nous n'utilisons pas les indices négatifs dans ce livre.

instruction **return** peut renvoyer plusieurs valeurs (sans demander la création d'un objet conteneur à renvoyer).[9]

- Les opérateurs booléens « et » et « ou » sont ***court-circuitants***. Cela signifie que, quand on évalue l'expression « x et y », on commence par évaluer la valeur booléenne de x. Si x vaut FALSE, alors l'expression globale ne peut pas être égale à TRUE et il est donc inutile d'évaluer l'autre membre, y. Si, en revanche, x vaut TRUE, alors il faut évaluer y pour déterminer la valeur de l'expression globale. De même, dans l'expression « x ou y », on n'évalue y que si x vaut FALSE. Les opérateurs court-circuitants permettent d'écrire des expressions booléennes du genre « $x \neq$ NIL et $x.f = y$ » sans que nous ayons à craindre le souci qui surviendrait si nous tentions de faire évaluer $x.f$ alors que x vaut NIL.
- Le mot clé **error** indique qu'une erreur est survenue, parce que les conditions pour la procédure à appeler étaient erronées, et la procédure s'achève immédiatement. La procédure appelante étant chargée de traiter l'erreur, nous ne spécifions pas l'action à entreprendre.

Exercices

2.1-1

À l'aide de la figure 2.2, illustrez l'action de TRI-INSERTION sur un tableau contenant au départ $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

2.1-2

La procédure TABLEAU-SOMME présentée ci-après calcule la somme de n nombres trouvés dans un tableau $A[1 \,..\, n]$. Définissez un invariant de boucle approprié et servez-vous des étapes d'initialisation, de conservation et de terminaison pour prouver que la procédure renvoie bien la somme des nombres trouvés dans le tableau.

```
TABLEAU-SOMME(A, n)
1  somme = 0
2  for i = 1 to n
3      somme = somme + A[i]
4  return somme
```

2.1-3

Récrivez la procédure TRI-INSERTION pour trier dans l'ordre monotone décroissant plutôt que dans l'ordre monotone croissant.

9. La notation de tuple dans Python permet de renvoyer plusieurs valeurs sans devoir créer un objet à partir d'une classe définie dans ce but.

2.1-4

Considérez le ***problème de la recherche*** :

Entrée Une suite de n nombres $A = \langle a_1, a_2, \ldots, a_n \rangle$ stockés dans un tableau $A[1 \,..\, n]$ et une valeur x.

Sortie Un indice i tel que $x = A[i]$, ou bien la valeur spéciale NIL si x ne figure pas dans A.

Écrivez du pseudocode pour une ***recherche linéaire*** parcourant le tableau du début à la fin en cherchant x. Au moyen d'un invariant de boucle, montrez la validité de l'algorithme. Vérifiez que votre invariant possède bien les trois propriétés requises.

2.1-5

On considère le problème consistant à additionner deux entiers a et b en représentation binaire, stockés sur n bits dans deux tableaux de n éléments $A[0 \,..\, n-1]$ et $B[0 \,..\, n-1]$. Chaque élément vaut soit 0, soit 1, avec $a = \sum_{i=0}^{n-1} A[i] \cdot 2^i$ et $b = \sum_{i=0}^{n-1} B[i] \cdot 2^i$. La somme de chaque couple d'entiers doit être stockée sous forme binaire dans un tableau $C[0 \,..\, n]$ de $n+1$ éléments pour lequel $c = \sum_{i=0}^{n} C[i] \cdot 2^i$. Écrivez en pseudocode une procédure SOMME-ENTIERS-BIN qui attend en entrée deux tableaux A et B et une longueur n pour renvoyer en sortie un tableau de sommes C.

2.2 Analyse des algorithmes

Analyser un algorithme consiste à prévoir les ressources nécessaires à son exécution. Ces ressources sont d'abord la quantité d'espace mémoire, les performances des liaisons réseau et la consommation d'énergie [10]. Néanmoins, l'objectif est très souvent de mesurer le temps de calcul. Si vous analysez plusieurs algorithmes susceptibles de résoudre un problème, vous pourrez identifier le plus efficace d'entre eux. Il peut exister plus d'un algorithme convenable, mais l'analyse permet généralement d'écarter au passage plusieurs algorithmes clairement inférieurs.

Avant de pouvoir analyser un algorithme, il faut disposer d'un modèle technologique sur lequel il sera exécuté, notamment au niveau des ressources de cette technologie et des coûts associés. Dans la majeure partie de ce livre, on supposera que l'on dispose d'un processeur unique implémenté par un modèle de calcul via une ***machine à registres à accès aléatoire*** (*Random Access Machine*, RAM) exploitant un espace machine de type RAM. Nous supposons bien sûr que les algorithmes seront implémentés sous forme de programmes informatiques. Dans le modèle RAM, les instructions sont exécutées en séquence, l'une après l'autre, sans opérations simultanées. Ce modèle RAM suppose que toutes les instructions

10. (N.d.T. : la puissance du processeur également.)

ont un temps d'exécution identique et constant, et que toutes les opérations d'accès aux données (lecture ou écriture de la valeur d'une variable) consomment le même laps de temps. Autrement dit, dans ce modèle, toutes les durées d'exécution sont constantes, y compris les accès aux tableaux par indices [11].

À strictement parler, il faudrait définir précisément les instructions de ce modèle RAM et leurs coûts. Cependant, cela serait pénible et n'apporterait pas grand chose en matière de conception et d'analyse d'algorithme. Attention, toutefois, à ne pas enfreindre allègrement le modèle RAM. Par exemple, que se passerait-il si ce modèle RAM disposait d'une instruction de tri ? On pourrait alors trier en une seule instruction, ce qui est irréaliste, vu qu'un ordinateur ne peut pas disposer de ce genre d'instruction très complexe dans son alphabet de traitements élémentaires. Nous devons donc nous laisser guider par le fonctionnement des ordinateurs concrets. Le modèle RAM contient les instructions que l'on trouve dans la plupart des ordinateurs :

- instructions arithmétiques (addition, soustraction, multiplication, division, modulo, minimaux et maximaux) ;
- instructions de transfert de données (lecture, écriture, copie) ;
- instructions de contrôle (branchements conditionnels et inconditionnels, appel et sortie de sous-programme).

Les types de données du modèle RAM sont le type numérique entier, le type numérique flottant (à virgule flottante, pour le stockage des nombres réels) et le type caractère (un type entier de représentation). Dans la réalité, un ordinateur ne prévoit généralement pas de type de données distinct pour les deux valeurs booléennes TRUE (vrai) et FALSE (faux). Au lieu de cela, il teste si une valeur entière est égale à zéro (FALSE) ou pas (TRUE), comme en langage C. Bien que nous ne nous préoccupions généralement pas de la précision des valeurs en virgule flottante dans ce livre (de nombreuses valeurs ne peuvent pas être représentées exactement en virgule flottante), la précision est cruciale pour la plupart des applications. Nous supposons également que chaque mot de données est limité en nombre de bits. Ainsi, quand nous travaillerons avec des entrées de taille n, nous supposerons en principe que les entiers sont représentés par $c(\lfloor \log_2 n \rfloor + 1)$ bits pour une certaine constante $c \geq 1$, avec $\lfloor \log_2 n \rfloor$ le plus grand entier inférieur ou égal à $\log_2 n$. Nous imposons que $c \geq 1$ de façon que chaque mot puisse contenir la valeur n, ce qui nous permettra d'indexer les divers éléments de l'entrée ; nous imposons aussi à c d'être une constante de façon que la taille

11. Nous supposons que chaque élément d'un tableau occupe le même nombre d'octets et que les éléments d'un tableau sont physiquement stockés dans des emplacements mémoire adjacents. Par exemple, si le tableau $A[1 \,..\, n]$ est implanté en mémoire à partir de l'adresse 1000 et si chaque élément occupe quatre octets, alors l'élément $A[i]$ est situé à partir de l'adresse $1000 + 4(i - 1)$. En général, trouver l'adresse mémoire d'un élément de tableau ne demande qu'une soustraction (et aucune dans la convention d'index de tableaux en base 0), une multiplication (souvent sous forme d'un décalage de bits dès que la taille d'élément est une puissance entière de 2) et une addition. Dans les blocs d'instructions qui balayent un tableau en séquence, les compilateurs à optimisation savent produire l'adresse de chaque élément avec une seule addition (en ajoutant la taille unitaire d'élément à l'adresse de l'élément précédent).

du mot n'augmente pas de manière arbitraire. (Si la taille du mot pouvait croître ainsi, on pourrait stocker un grand volume de données dans un seul mot mémoire et manipuler ce volume de données en temps constant, ce qui est un scénario manifestement irréaliste.)

Les ordinateurs réels contiennent des instructions qui ne sont pas prises en compte dans notre approche ; ces instructions supplémentaires constituent une zone d'ombre de notre modèle RAM. Par exemple, est-ce que l'exponentiation est une instruction à délai constant ? En général, non ; elle exige plusieurs instructions pour calculer x^n quand x et n sont des entiers et réclame un temps logarithmique en n (voir l'équation (31.34) dans le chapitre 31). Vous devez garantir que la représentation binaire du résultat ne déborde pas de la taille d'un mot machine. En revanche, si x est une puissance exacte de 2, l'exponentiation peut, en général, être considérée comme à délai constant. La plupart des processeurs disposent d'une instruction de décalage vers la gauche (*shift left*) qui décale en temps constant les bits d'un entier de n positions vers la gauche. Pour la plupart des ordinateurs, décaler les bits d'un entier d'une position vers la gauche revient à le multiplier par 2. Décaler les bits de n positions vers la gauche revient donc à multiplier par 2^n. Par conséquent, ce genre de machine peut calculer 2^n en une unique instruction en temps constant, et ce en décalant l'entier 1 de n positions vers la gauche, du moment que n est inférieur au nombre de bits d'un mot machine. Nous ferons tous nos efforts pour éviter ce genre de situation et nous considérerons le calcul de 2^n comme une opération en temps constant à condition que le résultat ne déborde pas de la taille du mot machine.

Notre modèle mémoire RAM ne tient pas compte des différents types de mémoire qui sont en vigueur dans les ordinateurs modernes. Il ne distingue donc ni les caches, ni le mécanisme de mémoire virtuelle. Seuls quelques problèmes (notamment dans la section 11.5 de ce livre) examinent les effets de cette stratification mémoire. D'autres modèles de calcul essaient de prendre en compte ces effets, qui sont parfois significatifs en pratique, mais ces modèles sont nettement plus complexes que le modèle RAM, et leur utilisation risque d'être délicate. De plus, les analyses basées sur le modèle RAM donnent généralement d'excellentes prévisions quant aux performances obtenues sur des machines réelles.

L'analyse d'un algorithme avec le modèle RAM est en général facile, mais elle peut parfois constituer un défi. On devra parfois faire appel à des outils mathématiques tels que l'algèbre combinatoire et la théorie des probabilités ; en outre, il faudra être capable de jongler avec l'algèbre et de repérer les termes les plus significatifs dans une formule. Sachant que le comportement d'un algorithme peut varier pour chaque entrée possible, il faut se doter d'un moyen de résumer ce comportement en quelques formules simples et faciles à comprendre.

Analyse du tri par insertion

Combien de temps dure l'exécution de la procédure TRI-INSERTION ? Une façon de le savoir serait de l'exécuter sur votre ordinateur en chronométrant la durée d'exécution. Bien sûr, vous devrez d'abord avoir rédigé le code source dans un vrai langage de programmation

et avoir réussi sa compilation, puisque vous ne pouvez pas exécuter notre pseudocode. Ce que ce chronométrage vous apprendrait, c'est le temps requis sur votre ordinateur pour une entrée donnée, avec l'exécutable spécifique que vous avez créé grâce à un compilateur ou un interpréteur spécifique, en fonction des bibliothèques utilisées et en considérant les tâches de fond qui s'exécutent sur cet ordinateur en même temps que votre test (par exemple la surveillance des informations entrantes du réseau). Vous pourriez même obtenir un résultat différent en répétant l'exécution du même programme sur le même ordinateur avec les mêmes paramètres d'entrée. Vous ne pouvez donc absolument pas tirer de conclusion généralisable quant au temps d'exécution du tri par insertion sur une autre machine, avec d'autres paramètres ou un autre langage de programmation. Il nous faut un moyen de prédire la durée d'exécution d'un tri par insertion à partir d'un jeu de données d'entrée.

Au lieu de chronométrer une ou plusieurs exécutions, nous pouvons déterminer le temps nécessaire en analysant l'algorithme lui-même. Nous examinons combien de fois il exécute chaque ligne de pseudocode et combien de temps chaque ligne de pseudocode requiert pour s'exécuter. Nous allons commencer par élaborer une formule précise mais compliquée pour le temps d'exécution. Ensuite, nous distillerons la partie importante de la formule en utilisant une notation pratique qui nous aidera à comparer les temps d'exécution de plusieurs algorithmes répondant au même problème.

Comment procéder pour analyser le tri par insertion ? Nous devons d'abord convenir que le temps d'exécution dépend des données en entrée. Vous ne serez pas surpris de lire que trier un millier d'éléments prend plus de temps qu'en trier trois. De plus, trier deux tableaux de même taille va demander un délai différent selon que leur contenu est déjà plus ou moins trié ou pas du tout. Même si la durée d'exécution dépend des particularités des données d'entrée, nous nous concentrerons sur la caractéristique qui s'est avérée avoir le plus d'impact, à savoir la taille de ces données d'entrée. Nous décrirons donc la durée d'exécution d'un programme comme fonction de la taille de son entrée. Pour ce faire, nous devons définir plus précisément les termes « temps d'exécution » et « taille de l'entrée ». Nous devons également préciser si nous nous intéressons à la durée d'exécution dans le pire cas, le plus favorable ou un autre.

Savoir ce que recouvre la notion de ***taille de l'entrée*** dépend du problème étudié. Pour de nombreux problèmes, tels que le tri ou le calcul de transformées de Fourier discrètes, la notion la plus naturelle est le *nombre d'éléments constituant l'entrée*, par exemple le nombre d'éléments n du tableau à trier. Pour d'autres problèmes, comme la multiplication de deux entiers, la meilleure mesure de la taille de l'entrée est le *nombre total de bits* nécessaires pour représenter la donnée en binaire. Parfois, il est plus approprié d'exprimer la taille de l'entrée par deux nombres. Par exemple, si l'entrée d'un algorithme est un graphe, on pourra décrire la taille de l'entrée par le nombre de sommets et le nombre d'arêtes ou d'arcs [12]. Pour chaque problème, nous indiquerons la mesure utilisée pour exprimer la taille de l'entrée.

12. Dans un graphe orienté, le terme « arête » est remplacé par le terme « arc ».

Le ***temps d'exécution*** d'un algorithme pour une entrée est la somme des temps d'exécution des opérations élémentaires et d'accès aux données des instructions. Le calcul des composantes de temps des instructions doit le moins possible dépendre des particularités du matériel tout en restant conforme au modèle RAM présenté plus haut. Pour le moment, nous allons considérer que l'exécution d'une ligne de pseudocode demande toujours le même temps. Deux lignes différentes peuvent demander des temps différents, mais chaque exécution de la k-ème ligne prendra toujours un temps constant c_k. Ce point de vue est compatible avec le modèle RAM et reflète la manière dont le pseudocode serait implémenté sur la plupart des ordinateurs existants [13].

Passons à l'analyse de la procédure TRI-INSERTION. Comme annoncé, nous commençons par élaborer une formule précise qui utilise la taille des données d'entrée et tous les coûts des instructions c_k. Le résultat étant peu pratique, nous changerons pour une notation plus concise et plus facile à manipuler. Cette notation simplifiée nous aidera à comparer les performances des algorithmes, notamment en relation avec la croissance de la taille des données d'entrée.

TRI-INSERTION(A, n)		*Coût*	*Temps*
1	**for** $i = 2$ **to** n	c_1	n
2	$clé = A[i]$	c_2	$n-1$
3	// Insère $A[i]$ dans sous-tableau trié $A[1\,..\,i-1]$.	0	$n-1$
4	$j = i-1$	c_4	$n-1$
5	**while** $j > 0$ et $A[j] > clé$	c_5	$\sum_{i=2}^{n} t_i$
6	$A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n}(t_i-1)$
7	$j = j-1$	c_7	$\sum_{i=2}^{n}(t_i-1)$
8	$A[j+1] = clé$	c_8	$n-1$

Commençons l'analyse en étudiant le pseudocode de la procédure TRI-INSERTION avec le coût temporel de chaque instruction et le nombre de fois que chaque instruction est exécutée. Pour toutes valeurs $i = 2, 3, \ldots, n$, soit t_i le nombre de fois que le test de la boucle **while**, en ligne 5, est exécuté pour cette valeur de i. Quand une boucle **for** ou **while** se termine normalement (c'est-à-dire, suite à l'échec du test effectué dans la tête de boucle), le test aura été exécuté une fois de plus que le corps de la boucle. Les commentaires étant ignorés par le

13. Il y a lieu ici de noter quelques subtilités. Les étapes de calcul spécifiées en langage naturel sont souvent des variantes d'une procédure qui demande plus qu'un volume de temps constant. Par exemple, lorsque nous verrons dans le chapitre 8 la procédure TRI-BASE (RADIX SORT), quand nous disons « utiliser un tri stable pour trier le tableau A selon la valeur i », cela demande en fait plus qu'un temps constant. Alors que l'instruction d'appel de procédure requiert un temps constant, l'exécution de la procédure peut en requérir plus. Autrement dit, on sépare le processus d'***appel*** de procédure (passage des paramètres, etc.) du processus d'***exécution*** de la procédure.

compilateur, ils ne produisent pas d'instructions exécutables et ne consomment donc pas de temps.

Le temps d'exécution de l'algorithme est la somme des temps d'exécution des instructions. Une instruction qui demande un temps c_k et qui est exécutée m fois compte pour $c_k m$ dans le temps d'exécution total.[14] En général, la durée d'exécution des algorithmes pour un volume d'entrée n s'écrit $T(n)$. Pour calculer cette durée d'exécution pour TRI-INSERTION avec n valeurs en entrée, on additionne les produits des colonnes *Coût* et *Temps* avec la formule suivante :

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1) \\ &\quad + c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1) \; . \end{aligned}$$

Même pour des entrées différentes ayant la même taille, le temps d'exécution d'un algorithme peut dépendre de la *valeur* de chaque entrée. Par exemple, dans TRI-INSERTION, le cas le plus favorable est celui où le tableau est déjà trié. Lors de l'exécution de la ligne 5, la valeur de l'élément $clé$ qui est celle se trouvant au départ dans $A[i]$ est déjà supérieure ou égale à toutes les valeurs de $A[1\,..\,i-1]$. De ce fait, la boucle **while** des lignes 5 à 7 se termine systématiquement dès le premier tour (après le premier test en ligne 5). Donc $t_i = 1$ pour $i = 2, 3, \ldots, n$, et le temps d'exécution du cas optimal est le suivant :

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \; . \end{aligned} \tag{2.1}$$

Nous pouvons exprimer ce temps d'exécution sous la forme $an + b$, sachant que a et b sont des *constantes* dépendantes des coûts d'instructions c_k (avec $a = c_1 + c_2 + c_4 + c_5 + c_8$ et $b = -(c_2 + c_4 + c_5 + c_8)$). Le temps d'exécution est donc une ***fonction linéaire*** en n.

Le pire cas est celui d'un tableau qui a été parfaitement trié, mais dans l'ordre inverse (décroissant). On doit alors comparer chaque élément $A[i]$ à chaque élément du sous-tableau trié $A[1\,..\,i-1]$, et donc $t_i = i$ pour $i = 2, 3, \ldots, n$. (La procédure détermine que $A[j] > clé$ lors de chaque exécution de la ligne 5. La boucle **while** ne se termine que lorsque j atteint 0.) Nous notons que :

$$\begin{aligned} \sum_{i=2}^{n} i &= \left(\sum_{i=1}^{n} i \right) - 1 \\ &= \frac{n(n+1)}{2} - 1 \quad \text{(selon équation (A.2))} \end{aligned}$$

14. Cette caractéristique n'est pas forcément valable pour une ressource comme la mémoire. Une instruction qui référence m mots mémoire et qui est exécutée n fois ne consomme pas nécessairement mn mots mémoire distincts.

selon l'équation (A.2) de l'annexe A et que, selon la même équation :

$$\begin{aligned}\sum_{i=2}^{n}(i-1) &= \sum_{i=1}^{n-1} i \\ &= \frac{n(n-1)}{2} \quad \text{(toujours selon équation (A.2))}\end{aligned}$$

On trouve que, dans le pire cas, le temps d'exécution de TRI-INSERTION est :

$$\begin{aligned}T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned} \tag{2.2}$$

Le temps d'exécution du pire cas peut donc s'exprimer sous la forme $an^2 + bn + c$, avec a, b et c des constantes qui dépendent des coûts d'instructions c_k. Notez que $a = c_5/2 + c_6/2 + c_7/2$, que $b = c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8$ et que $c = -(c_2 + c_4 + c_5 + c_8)$. C'est donc une ***fonction quadratique*** en n.

Généralement, et c'est le cas du tri par insertion, le temps d'exécution d'un algorithme est constant pour une même entrée ; encore que nous verrons plus loin quelques algorithmes « probabilistes » dont le comportement peut varier même pour une entrée fixe.

Analyse du pire cas et du cas moyen

Dans notre analyse du tri par insertion, nous nous sommes intéressés aussi bien au cas le plus favorable, celui où le tableau en entrée est déjà trié, qu'au pire cas, celui où le tableau en entrée est trié en sens inverse. Dans la suite du livre, nous nous focaliserons en général (mais pas toujours) sur le ***temps d'exécution dans le pire cas***, c'est-à-dire le temps d'exécution maximal pour une entrée *quelconque* de taille n. Pourquoi ? Voici nos trois arguments en ce sens.

- Le temps d'exécution le plus défavorable constitue le majorant (borne supérieure) du temps d'exécution *quelle que soit* la configuration des données d'entrée. Connaître cette valeur garantit que le traitement ne durera jamais plus. Vous n'avez plus besoin de vous perdre en conjectures à propos du temps d'exécution en espérant ne jamais être contredit par le réel. Cette sécurité est encore plus indispensable dans les projets de traitement en temps réel dans lesquels les temps de réponse sont critiques.
- Pour certains algorithmes, le pire cas est assez fréquent. Si vous cherchez par exemple une information dans une base de données, et que cette information n'existe pas encore

dans la base, vous êtes souvent dans le pire cas de l'algorithme de recherche. Dans certaines applications, la recherche négative sert de confirmation d'absence [15].

- Souvent, le « cas moyen » est presque aussi mauvais que le pire cas. Supposons que l'on c applique un tri par insertion à un tableau de n nombres tirés au hasard. Combien de temps faut-il pour trouver où insérer l'élément $A[i]$ dans le sous-tableau $A[1 \mathinner{..} i-1]$? En moyenne, la moitié des éléments de $A[1 \mathinner{..} i-1]$ sont inférieurs à $A[i]$ et la moitié lui sont supérieurs. Donc, en moyenne, on teste la moitié du sous-tableau $A[1 \mathinner{..} i-1]$, auquel cas t_i vaut environ $i/2$. Si l'on calcule alors le temps d'exécution global associé au cas moyen, on voit que c'est une fonction quadratique de la taille de l'entrée, tout comme le temps d'exécution.

Dans certains cas précis, nous nous intéressons au temps d'exécution du ***cas moyen*** d'un algorithme. Nous verrons notamment dans ce livre la technique d'***analyse probabiliste*** appliquée à divers algorithmes. La portée de l'analyse du cas moyen est limitée, car il n'est pas toujours évident de savoir ce qu'est une entrée « moyenne » pour un problème particulier. Souvent, nous supposerons que toutes les entrées ayant une taille donnée sont équiprobables. Cette hypothèse n'est pas toujours vérifiée dans la pratique, mais nous pourrons parfois employer un ***algorithme probabiliste*** qui force des choix aléatoires afin de permettre l'utilisation d'une analyse probabiliste et de donner un temps d'exécution ***prévu***. Nous étudierons les algorithmes probabilistes dans le chapitre 5 et dans plusieurs chapitres ultérieurs.

Ordre de grandeur

Afin de faciliter notre analyse de la procédure TRI-INSERTION, nous avions adopté des hypothèses simplificatrices. Nous avions ignoré le coût réel de chaque instruction en employant les constantes c_k pour représenter ces coûts. Pourtant, les durées du meilleur et du pire cas des équations (2.1) et (2.2) sont clairement peu pratiques. Nous avons constaté que ces constantes nous donnaient trop de détails. C'est pourquoi nous avons formulé le temps d'exécution du meilleur cas comme $an + b$, avec a et b deux constantes dépendant des coûts c_k et celui du pire cas comme $an^2 + bn + c$, a, b et c étant des constantes dépendant des coûts c_k. Nous avons donc ignoré non seulement les coûts réels des instructions, mais aussi les coûts abstraits c_k.

Osons une simplification supplémentaire. Ce qui nous intéresse vraiment, c'est le ***taux de croissance*** ou ***ordre de grandeur*** du temps d'exécution. On ne considérera donc que le terme dominant d'une formule (par exemple an^2), puisque les termes d'ordre inférieur sont moins significatifs pour n grand. On ignorera également le coefficient constant du terme dominant, puisque les facteurs constants sont moins importants que l'ordre de grandeur pour ce qui est de la détermination de l'efficacité du calcul pour les entrées volumineuses. Pour le temps du tri par insertion, quand on ignore les termes d'ordre inférieur et le coefficient constant du

15. N.d.T. : elle n'est alors pas une erreur, mais le cas d'usage principal