Introduction à l'algorithmique et à la programmation

Jean-Éric Pin

Cette section de l'encyclopédie est consacrée à trois outils fondamentaux de l'informatique : l'algorithmique, les modèles de machine et les langages de programmation.

1 Algorithmique

Le mot *algorithme* est dérivé du nom du mathématicien persan Al Khuwârizmî (780-850), dont l'un des ouvrages est aussi à l'origine du mot *algèbre*, mais la notion d'algorithme est beaucoup plus ancienne. Un algorithme est la description de la suite des opérations élémentaires qui permettent de résoudre un problème donné. Le choix des opérations élémentaires dépend du problème et de la précision souhaitée pour la description de sa résolution.

On dispose de plusieurs critères pour comparer l'efficacité des algorithmes. Les deux principaux paramètres sont le *temps d'exécution* et la *mémoire requise* pour mettre en œuvre l'algorithme. Calculer la valeur exacte de ces paramètres présente peu d'intérêt, car ils sont tributaires de beaucoup trop de variables : vitesse du processeur, charge de l'ordinateur, vitesse d'accès au disque, etc. Il est beaucoup plus pertinent de déterminer leur évolution en fonction de la taille des données en résolvant par exemple la question suivante : qu'advient-il du temps de calcul (ou de la mémoire requise), lorsque la taille des données est multipliée par 10, 100 ou 1000 ?

En pratique, on se contente souvent de valeurs asymptotiques, ce qui conduit à utiliser la notation mathématique $O(f)$, où f est une fonction numérique. La *complexité d'un algorithme* est en $O(f(n))$ si son temps de calcul pour des données de taille n est borné par $cf(n)$, où c est une constante, lorsque n tend vers l'infini. La *complexité spatiale* est définie de façon analogue, mais mesure la mémoire utilisée par l'algorithme. Dernière subtilité : on distingue la complexité dans le *pire des cas* qui donne la complexité garantie quelle que soit la distribution des données et la *complexité en moyenne* qui donne le comportement moyen de l'algorithme.

Le meilleur exemple pour illustrer ces notions un peu abstraites, qui est aussi l'un des plus étudiés, est le *problème du tri*, qui consiste à trier un tableau d'éléments suivant un ordre déterminé. Il est facile de concevoir des algorithmes en $O(n^2)$, à la fois en moyenne et dans le pire des cas. C'est le cas du *tri par insertion*, un algorithme que chacun utilise intuitivement pour ranger un jeu de cartes. Il consiste à insérer les cartes une à une parmi les cartes déjà triées.

Il n'est pas si simple d'obtenir un algorithme plus performant. Nous en décrirons sommairement deux. L'algorithme de *tri rapide* (ou *QuickSort*) a été inventé en 1960 par Hoare, à qui l'on doit également le premier compilateur commercial pour le langage ALGOL 60. Le tri rapide repose sur la notion de partitionnement. On choisit un élément du tableau, appelé *pivot*, puis on permute tous les éléments de façon à ce que tous ceux qui lui sont inférieurs soient à sa gauche et tous ceux qui lui sont supérieurs soient à sa droite. À l'issue de ce processus, le pivot est à sa place définitive. On applique alors récursivement la même procédure sur les sous-tableaux à la gauche et à la droite du pivot. La complexité du tri rapide est en $O(n \log n)$, mais en $O(n^2)$ dans le pire des cas.

Le *tri par fusion* a été proposé en 1945 par John von Neumann, un mathématicien qui a également conçu l'architecture des premiers ordinateurs modernes. Son algorithme repose sur le fait que n comparaisons suffisent à fusionner deux tableaux déjà triés dont la somme des tailles est n. Par exemple, pour fusionner les tableaux

1	5	7	12	18	et	3	10	11

il suffit de comparer leurs premiers éléments, 1 et 3. Le minimum est 1, ce qui fournit le premier élément du tableau trié. Il suffit ensuite d'it́erer le procédé avec les tableaux

5	7	12	18	et	3	10	11

jusqu'à épuisement des éléments.

Le tri par fusion consiste à fusionner les éléments un à un, puis deux à deux, quatre à quatre, jusqu'à ce que tous les éléments soient dans le même tableau trié. Sa complexité est en $O(n \log n)$ dans le pire des cas, mais nécessite l'emploi d'un tableau auxiliaire.

Le tableau ci-dessous permet de mieux comprendre les ordres de grandeur des complexités les plus usuelles :

Taille	10	100	1000
$\log_2 n$	3,32	6,64	9,96
n	10	100	1000
$n \log_2 n$	33,2	664	9965
n^2	100	1000	10^6
n^3	1000	10^6	10^9
2^n	1024	$1,26\ 10^{30}$	$> 10^{301}$

On voit en particulier que passer d'un algorithme de tri en $O(n^2)$ à un algorithme en $O(n \log n)$ permet de trier plus de cent fois plus vite un tableau à un million d'entrées. Gagner un facteur 100 sur un algorithme permet concrètement d'attendre trois secondes au lieu de cinq minutes, une différence appréciable quand on est au téléphone... Quant au nombre 10^{301} figurant dans le tableau, il est largement supérieur au nombre d'atomes dans l'univers, estimé à 10^{80}...

Les algorithmes usuels peuvent être classés en quelques grandes classes de complexité :

– les algorithmes *en temps constant*, dont la complexité est en $O(1)$, c'est-à-dire indépendante de la taille des données. Certains algorithmes de recherche en table, utilisant des techniques de hachage, ont une complexité moyenne constante ;
– les algorithmes *sous-linéaires*, dont la complexité est en général en $O(\log n)$. C'est le cas par exemple de nombreux algorithmes sur les arbres ;
– les algorithmes *linéaires*, dont la complexité est en général en $O(n)$ ou en $O(n \log n)$ sont considérés comme rapides. Les plus connus sont les algorithmes optimaux de tri et les algorithmes de parcours de graphes ;
– les algorithmes dont la complexité se situe entre *quadratique* (en $O(n^2)$) et *cubique* (en $O(n^3)$) sont plus lents, mais encore acceptables dans certains cas. C'est le cas de la multiplication des matrices et de nombreux algorithmes sur les graphes. Par contre, un algorithme cubique est trop lent pour être utilisé en génomique ;
– les algorithmes polynomiaux en $O(n^k)$ pour $k > 3$ ou exponentiels (notamment ceux de complexité $O(2^n)$) sont impraticables dès que la taille des données dépasse quelques centaines.

On ne peut même pas compter sur la loi de Moore pour améliorer sensiblement un algorithme exponentiel. Cette loi, qui prédit le doublement de la puissance des ordinateurs tous les 18 mois, prévoit bien pourtant une progression de la taille des problèmes traitables par un algorithme exponentiel. Mais cette progression n'est que d'une unité tous les 18 mois, une avancée très insuffisante...

Pourtant, même lorsqu'aucun algorithme polynomial n'est connu, tout n'est pas encore perdu. Une heuristique bien choisie permet parfois de résoudre efficacement tel cas particulier d'un problème donné. On utilise aussi très fréquemment des algorithmes qui, sans donner la solution optimale, fournissent une solution approchée. C'est le cas notamment pour les problèmes de découpage (moquette, tissu, métal, etc.). Une autre technique très utilisée consiste à trouver la solution, mais seulement avec une certaine probabilité. Par exemple, bien que l'on connaisse depuis 2002 un algorithme polynomial pour tester si un nombre entier est premier, les cryptographes lui préfèrent des algorithmes plus rapides, mais qui ne donnent qu'une réponse partielle. Ces algorithmes détectent les nombres composés avec une erreur que l'on peut borner a priori, mais ne peuvent garantir la primalité. Enfin, on peut chercher à utiliser un algorithme parallèle afin d'utiliser simultanément la puissance de plusieurs processeurs. On peut même aller au delà, et réquisitionner, via Internet, des milliers d'ordinateurs du monde entier.

2 Modèles de machine

La définition précise d'un algorithme et de sa complexité est un problème qui a occupé les mathématiciens — et plus particulièrement les logiciens — avant même l'invention des ordinateurs.

Le modèle abstrait créé par Alan Turing, connu désormais sous le nom de *machine de Turing* a permis de résoudre ce problème de façon satisfaisante. Ces machines modélisent en effet tout ce qui est calculable et en particulier, tout ce qui est réalisable par un langage de programmation.

Les efforts des logiciens ont permis de démontrer l'existence de *problèmes indécidables*. L'un des plus célèbres continue de ruiner les espoirs de bien des apprentis-programmeurs : il ne peut exister de « superprogramme » capable de détecter si un programme donné comporte ou non une boucle infinie.

Les machines de Turing permettent aussi de définir rigoureusement les classes de complexité. Les plus célèbres sont les classes P et NP. La classe P (resp. NP) désigne l'ensemble des problèmes résolubles en temps polynomial par une machine de Turing déterministe (resp. non déterministe). Proposé en 1979, le problème $P = NP$ est le premier des sept « problèmes du millénaire » pour la résolution desquels l'institut Clay offre un prix d'un million de dollars. Sa signification intuitive est la suivante : si la solution d'un problème peut être *vérifiée* rapidement, peut-on aussi la *trouver* rapidement ? Les amateurs de Su-Do-Ku[1] savent bien qu'il est beaucoup plus facile de vérifier une solution que de la trouver, et on pense donc généralement que P est strictement inclus dans NP. Mais personne n'est parvenu à ce jour à le démontrer...

Si les machines de Turing constituent le modèle de machine le plus général, les *automates finis* en constituent le modèle le plus simple. Contrairement aux machines de Turing, qui possèdent une mémoire potentiellement non bornée, un automate fini ne dispose que d'une mémoire finie. C'est un modèle très proche des circuits digitaux, au point que l'on peut directement implanter certains types d'automates dans le silicium.

Les automates finis constituent un ingrédient de base de la quasi-totalité des modèles de machines qui ont été proposés à ce jour. Contrairement à une opinion assez répandue, les problèmes que l'on peut formuler en termes d'automates ne sont pas pour autant faciles à résoudre. D'une part parce que la théorie des automates elle-même offre une quantité de problèmes non encore résolus, et ensuite parce que la complexité des algorithmes sur les automates peut être rédhibitoire dans certains cas. En particulier, la vérification des systèmes et des protocoles, qui est l'une des applications les plus connues des automates finis, se heurte fréquemment à des phénomènes d'*explosion combinatoire*.

La recherche d'un modèle abstrait de machine a rejoint les préoccupations de linguistes, tels que Noam Chomsky, qui cherchaient une définition mathématique de la notion de grammaire. De ce fait, les linguistes ont fortement influencé la terminologie du domaine. Si l'on dit couramment que les ordinateurs représentent l'information par des suites de 0 et de 1 (ou *bits*), il est préférable, pour une définition abstraite, de ne pas privilégier la représentation binaire. Un *alphabet* désigne simplement un ensemble fini, dont les éléments sont appelés des *lettres*. L'alphabet $\{0, 1\}$ n'est donc qu'un exemple parmi d'autres, même si c'est le plus fréquemment utilisé. Un *mot* est une suite finie de lettres, que l'on note par simple juxtaposition. Ainsi 0211002 est un mot sur l'alphabet $\{0, 1, 2\}$. Un *langage* est un ensemble de mots. L'intérêt de ce formalisme est qu'il permet de représenter de façon abstraite des données de nature très diverse, sans avoir à ce préoccuper des détails de cette représentation. Une grammaire est donnée par un ensemble fini de règles. La définition formelle en sera donnée au chapitre ££*Modèles de machines*££, mais nous l'illustrerons ici par un exemple très simple : la grammaire constituée des deux règles $S \to 2$ et $S \to 0S1$ engendre le langage $\{2, 021, 00211, 0002111, \dots\}$ formé des mots de la forme 0^n21^n, où n est un entier positif ou nul. Par exemple le mot 0002111 est obtenu en appliquant trois fois la deuxième règle, et une fois la première règle, suivant le schéma :

$$S \to 0S1 \to 00S11 \to 000S111 \to 0002111$$

Chomsky a proposé quatre types de grammaires. Le type 0 est le plus puissant et engendre exactement les langages acceptés par des machines de Turing, appelés aussi *langages récursivement énumérables*. Le type 1 définit les langages qui sont reconnus par des machines de Turing dont on a restreint la mémoire (elle doit être linéairement bornée en fonction de la longueur de l'entrée). Le type 2 correspond aux langages acceptés par des *automates à pile*, qui, comme leur nom l'indique, sont des automates finis enrichis par un mécanisme de pile. Enfin le type 3 correspond aux langages acceptés par des automates finis. Ainsi, l'approche par les grammaires rejoint celle basée sur les machines.

3 Programmation

L'algorithmique permet de préparer la résolution d'un problème en détaillant les opérations élémentaires à accomplir. Il reste ensuite à traduire ces opérations en instructions pour l'ordinateur. C'est le rôle des langages de programmation. Il en existe déjà des centaines, avec de nombreuses variations, et ils font l'objet de recherches très actives.

Le point de vue adopté par le programmeur détermine le style, ou *paradigme* de la programmation. On parle ainsi de langage de programmation de haut ou de bas niveau, orienté objet ou non, de typage statique ou dynamique, impératif ou fonctionnel, etc.. Nous n'évoquerons ici que quatre classes particulièrement importantes : les langages impératifs, fonctionnels et orientés objet et la programmation logique par contraintes.

Un *programme impératif* est constitué d'une suite d'instructions élémentaires qui permettent de modifier l'*état mémoire* du programme. Ce style de programmation a été retenu pour la conception de la quasi-totalité des microprocesseurs. Dans ce cas, l'état d'un programme est décrit par le contenu de la mémoire centrale de l'ordinateur et les instructions élémentaires constituent le *langage machine* spécifique à chaque processeur. Un *langage assembleur* est

[1] Un jeu de logique fréquemment proposé dans les journaux.

une traduction d'un langage machine plus lisible pour les humains. Les langages de programmation impératifs utilisent des instructions de plus haut niveau : assignation de variables, branchements, mécanismes de répétition.

Un objectif souvent cité par les programmeurs est de pouvoir écrire un programme sans avoir à se soucier du modèle de microprocesseur ou du système d'exploitation sur lequel il sera exécuté, ce que l'on résume souvent par la formule « *s'abstraire de la machine* ». L'utilisation de structures de données adéquates (en commençant par les tableaux) va dans ce sens. La notion de sous-programme (ou *procédure*) permet également de mieux structurer les programmes tout en conservant une bonne lisibilité. Mais une procédure reste du domaine de la programmation impérative, car elle ne fait que modifier l'état mémoire. La *programmation fonctionnelle* propose une autre approche. Comme son nom l'indique, on y utilise des fonctions, au sens mathématique du terme, comme la fonction $x \to x^2 + 1$. Ces fonctions produisent un résultat unique et peuvent intervenir à tous les niveaux : comme paramètres ou comme résultat d'autres fonctions, comme composantes d'une structure de données, etc. Mieux, le programme tout entier est conçu comme une fonction que l'on décrit en s'appuyant sur des techniques issues de la logique, parmi lesquelles la récursion joue un rôle essentiel. Comme les fonctions sont des valeurs manipulables, on peut facilement composer des calculs en programme fonctionnelle.

Alors que les fonctions sont au cœur de la programmation fonctionnelle, ce sont les *relations* qui sont à la base de la *programmation logique*, dont le langage le plus connu est PROLOG. La *programmation logique par contraintes* fournit un cadre unifié pour la résolution de problèmes définis par des équations, des contraintes et des formules logiques. Elle est née du rapprochement de la programmation logique, des techniques de propagation de contraintes développées en intelligence artificielle et des méthodes d'optimisation conçues en recherche opérationnelle. Il est très facile par exemple d'écrire un programme de résolution de Su-Do-Ku dans un tel langage.

La *programmation orientée objet* privilégie un autre point de vue. Un programme y est vu comme un ensemble de composantes, les *objets*, qui rassemblent au sein d'une même structure de données, appelée *classe*, des opérations et des données. Un objet est un regroupement de données, les *variables d'instance*, et d'opérations sur ces données, les *méthodes d'instance*, respectant l'interface décrite dans la classe. On dit alors que l'objet est une *instance* de la classe.

L'un des grands avantages de cette approche est sa modularité. En effet, le paradigme objet sépare clairement l'*interface* (la partie visible d'un objet) de l'*implantation* (la façon dont les objets sont réellement représentés en mémoire). Comme les objets n'utilisent que les propriétés définies par l'interface, on peut modifier leur représentation interne sans avoir à rebâtir le programme.

En pratique, un langage de programmation peut reposer simultanément sur plusieurs paradigmes. De plus, certains langages ont évolué vers un dialecte à objets, comme C vers C++ et Objective C, ou Caml vers Objective Caml. Par ailleurs, de nombreux langages, qu'il soient impératifs, fonctionnels ou à objet proposent des extensions de programmation par contrainte.

Quelque soit le paradigme, un langage de programmation peut être soit directement interprété par l'ordinateur, soit traduit en langage machine par un *compilateur* avant d'être exécuté. Utiliser des compilateurs répond au souci des programmeurs de créer des programmes indépendants de l'architecture des processeurs.

4 Florilège

Nous terminerons cette introduction par un florilège de quelques problèmes classiques qui servent de test de performance pour les langages de programmation. Ainsi le calcul de la fonction *factorielle* $n! = 1 \times 2 \times \cdots \times n$ et de la *suite de Fibonacci*, définie par la récurrence

$$F_0 = 0,\ F_1 = 1 \text{ et, pour } n \geqslant 2,$$
$$F_n = F_{n-1} + F_{n-2},$$

servent souvent de modèles de programme récursif. La récursivité à plusieurs variables est souvent illustrée par le problème des *tours de Hanoi*. Ce problème a été imaginé, puis exploité comme casse-tête par le mathématicien français E. Lucas en 1883. Le jeu est composé de trois axes et d'anneaux enfilés sur ces axes par taille décroissante.

FIG. 1.1 – Tours de Hanoi

Le jeu consiste à déplacer tous les anneaux sur l'un des axes libres en un minimum de mouvements.

On ne déplace qu'un anneau à la fois et il ne peut être placé que sur un anneau plus grand que lui. On démontre que pour n anneaux, le nombre minimum de déplacements est de $2^n - 1$.

La programmation par contrainte excelle dans la résolution du *problème des n dames*, qui consiste à placer n dames sur un échiquier $n \times n$ sans qu'aucune d'entre elles ne soit en prise. Rappelons que, suivant les règles des échecs, une dame peut capturer toute pièce située sur sa ligne, sa colonne ou l'une de ses deux diagonales.

5 Présentation de la section

Dans cette section, quatre chapitres et demi sont consacrés à l'algorithmique, trois et demi à la modélisation de la notion de machine et cinq à la programmation.

Le chapitre ££*Structures de données*££ présente les structures de données de base sur lesquelles s'appuient la plupart des programmes : tableaux, structures linéaires (listes, piles et files), arbres et graphes.

Le chapitre ££*Techniques algorithmiques et méthodes de programmation*££ décrit les principaux types d'algorithmes (déterministes, probabilistes, randomisés, par contrat, sans branchement), puis présente quelques techniques classiques (algorithmes gloutons, programmation dynamique, diviser pour régner, récursivité, utilisation de sentinelles, algorithmes en place) illustrées par des algorithmes classiques (parcours d'arbres, parcours de graphe, tris).

Le chapitre ££*Algorithmes des graphes et des réseaux*££ illustre l'utilisation d'algorithmes classiques sur les graphes (parcours de graphe, plus court chemin, arbre couvrant, problèmes de flots, etc.) pour des questions concrètes liées aux réseaux (routage sur Internet, réseau de téléphonie mobile, etc.)

Le chapitre ££*Algorithmes algébriques*££ présente un panorama des algorithmes opérant sur des structures algébriques simples (polynômes, entiers, matrices). En particulier, le problème de la factorisation des entiers et la recherche de nombres premiers sont au cœur des recherches en cryptographie moderne.

Le chapitre ££*Algorithmique parallèle*££ est consacré aux algorithmes utilisant plusieurs processeurs travaillant en parallèle. Le chapitre explore les différents modèles formels (variantes de machines P-RAM) et illustre l'utilisation du parallélisme sur quelques exemples simples : calculs en chaîne sur une liste, calcul du maximum, diffusion de messages, produits de matrices.

Le chapitre ££*Automates finis*££ présente le modèle le plus simple de machine. On y décrit les automates déterministes, non déterministes, les langages rationnels et reconnaissables, puis on s'intéresse aux automates déterministes avec sortie (automates séquentiels). On y présente également divers exemples d'applications issus notamment des industries de la langue.

Le chapitre ££*Automates et vérification*££ décrit les applications du modèle des automates finis à la vérification des logiciels, et particulièrement de protocoles. Les automates permettent en effet de décrire les programmes à analyser et les propriétés de leur comportement. Ils donnent aussi une représentation finie de certains ensembles infinis de valeurs.

Le chapitre ££*Modèles de machines*££ est consacré aux principaux modèles formels utilisés en informatique, à l'exception des modèles de machine parallèles, qui sont traités dans le chapitre ££*Algorithmique parallèle*££. Les composantes d'une machine, les machines de Turing, la notion de problème indécidable, les mesures de complexité, les automates à pile et la hiérarchie de Chomsky sont les ingrédients principaux de ce chapitre.

Le chapitre ££*Les compilateurs*££ présente l'état de l'art sur les techniques de compilation. On y décrit les différentes phases de la compilation (analyse syntaxique, représentation intermédiaire, optimisation, génération de code), puis les différentes techniques d'optimisation et on termine par la parallélisation automatique et les problèmes posés par les architectures modernes.

Le chapitre ££*Langages de programmation impératifs*££ présente les mécanismes classiques des langages impératifs (variables, affectation, évaluation, instructions conditionnelles, boucles, sous-programmes, récursivité). On y décrit ensuite sommairement les plus célèbres de ces langages : Fortran, Algol, Pascal, Modula, COBOL, C, etc.

Le chapitre ££*Programmation fonctionnelle*££ décrit les mécanismes de base communs aux langages fonctionnels, les problèmes de typage, d'ordre d'évaluation des expressions et les structures de contrôle. On y évoque les techniques de mises en œuvre (compilation, bibliothèque d'exécution, interopérabilité), puis on décrit brièvement les langages fonctionnels les plus connus (Lisp, Scheme, ML, Miranda, Clean, Haskell et leurs dialectes).

Le chapitre ££*Les langages à objets*££ revient sur la genèse du modèle objet, présente ses concepts fondateurs, les illustre sur le problème des tours de Hanoi, présente les contributions majeures au domaine du génie logiciel et évoque pour conclure les perspectives futures en programmation.

Le chapitre ££*Programmation logique et contraintes*££ présente le cadre et les méthodes de la programmation par contrainte. On y décrit d’abord les différentes procédures de recherche (arbres de recherche, heuristiques, séparation-évaluation, recherche locale, etc.) puis les trois grands modèles d’exécution (résolutions de contraintes, programmation logique avec contraintes et langages concurrents avec contraintes).

Index