AVANT-PROPOS

C'est en forgeant qu'on devient forgeron...

Ce vieil adage, dont les hommes ont usé depuis la nuit des temps, est mis dans cet ouvrage au service des étudiants de licence et de première année de master de mathématiques et d'informatique, des élèves-ingénieurs et de toute autre personne souhaitant goûter les joies et maîtriser les difficultés de l'algorithmique.

Au cœur de l'informatique, cette science est celle de la conception de méthodes efficaces pour résoudre des problèmes à l'aide d'un ordinateur. Elle définit des outils généraux permettant de répondre aux questions suivantes : Sur quelle propriété mathématique d'un problème peut-on établir une méthode de résolution ? Étant donné un algorithme, résout-il le problème posé ? Lorsque l'on dispose de deux méthodes de résolution différentes, laquelle est préférable ?

L'algorithmique a donné lieu à de nombreux ouvrages remarquables depuis plus de trente ans, sur lesquels se fondent les enseignements dispensés dans les universités et écoles d'ingénieurs, et dont nous donnons une liste non exhaustive à la fin de cet ouvrage. L'expérience des auteurs, enseignants chevronnés dans différentes universités, les a depuis longtemps confrontés aux difficultés de leurs étudiants face à cette matière. Celles-ci semblent de plusieurs ordres :

- il est nécessaire de pouvoir expérimenter les algorithmes sur différents exemples pour en comprendre intuitivement le fonctionnement ;
- apprendre des éléments de cours implique de pouvoir refaire, au besoin, les démonstrations qui y ont été présentées, de les rédiger ;
- les algorithmes manipulent des entités qui évoluent dans le temps, et cela ajoute une dimension inhabituelle aux raisonnements mathématiques nécessaires pour les prouver et analyser leurs performances ;
- l'apprentissage souvent simultané d'un langage de programmation peut parfois ajouter des difficultés de compréhension dues à la syntaxe propre du langage.

C'est pour répondre à ces difficultés que les auteurs ont formé le projet de cet ouvrage, à partir d'exercices progressifs conçus lors de leurs années de pratique, et utilisés dans le cadre de travaux dirigés, d'examens, ou pour des devoirs de plus grande envergure. L'ouvrage a pour objectif d'aider l'étudiant dans son apprentissage de la conception et de l'analyse d'algorithmes en insistant sur le raisonnement et sa rédaction, en vue d'écrire dans le langage de son choix des programmes efficaces.

Si la plupart des ouvrages de cours d'algorithmique contiennent des énoncés d'exercices, peu sont corrigés. C'est donc l'une des originalités de ce livre que de

proposer une correction entièrement rédigée, rigoureuse et complète de chaque question.

On y trouvera, pour chaque notion, des exercices visant la compréhension du cours, qui permettent d'appliquer un algorithme connu à des données numériques, ou de redémontrer, à partir d'éléments de base dont les définitions sont explicitées, les propriétés de certains algorithmes du cours décomposées en questions progressives. On y trouvera aussi des exercices qui enrichissent des algorithmes classiques de nouvelles fonctionnalités ou qui permettent de les intégrer dans des applications originales.

Concernant la programmation, le parti pris de cet ouvrage est d'employer, pour la rédaction des algorithmes, un pseudo-langage impératif, plutôt qu'un véritable langage de programmation, afin de se concentrer sur ce qui ressort de la conception de l'algorithme, et non sur son implémentation. Ce choix devrait toutefois permettre une implémentation aisée des algorithmes dans des langages comme Pascal, C ou C++, avec sans doute un effort supplémentaire en ce qui concerne les langages objets. Nous indiquons plus loin les conventions d'écriture de ce langage, auxquelles le lecteur pourra se référer pour comprendre, mais aussi pour programmer les algorithmes dans le langage de son choix.

L'ouvrage aborde un panel assez vaste de domaines d'application de l'algorithmique, et devrait couvrir la plupart des cours dispensés actuellement en license et master de mathématiques et informatique et en écoles d'ingénieurs.

On y développe notamment, après un chapitre méthodologique qui traite de la preuve et de l'analyse de la complexité d'un algorithme, les types de données abstraits pour représenter, gérer et manipuler des ensembles dynamiques, et leur implémentation à l'aide de structures de données linéaires ou arborescentes (piles, files, listes, arbres binaires de recherche, arbres équilibrés, tas). On aborde ensuite l'étude des algorithmes de tri. Les chapitres suivants sont consacrés à l'étude de l'algorithmique de base des graphes valués et non valués : connexité, accessibilité, parcours, arbres couvrants et chemins de coût minimum, pour ne citer que les principaux problèmes abordés. Ensuite, deux chapitres consacrés l'un aux algorithmes relatifs à la géométrie plane et l'autre aux algorithmes relatifs aux automates et aux mots achèvent cet ouvrage.

Dans chaque section, les exercices sont présentés dans un ordre de difficulté croissante, sauf nécessité de regroupement thématique, et souvent les questions d'un exercice respectent une certaine progressivité dans la difficulté.

D'autres algorithmes auraient mérité de figurer dans ce livre, plus proches de l'algorithmique numérique, ou de la recherche opérationnelle. Avec quelques encouragements, les auteurs pourraient envisager une suite...

Prérequis

Le livre nécessite les connaissances de mathématiques du niveau des deux premières années de licence, en particulier une bonne maîtrise du raisonnement par récurrence, ainsi que la connaissance de base d'un langage de programmation. Chacun des chapitres demande des prérequis, explicités dans les exercices lorsque cela semble nécessaire.

Il est recommandé de traiter les exercices en ayant d'autre part étudié un cours oral ou un livre de référence. Toutefois, de nombreux exercices peuvent être traités en se référant uniquement aux définitions et résultats de base rappelés en début de sous-chapitre. Cet en-tête de chapitre n'a pas pour objectif d'être un abrégé de cours. Les définitions présentées sont loin d'être exhaustives, mais sont celles jugées indispensables pour traiter les exercices avec le plus d'autonomie possible.

Conventions relatives à la présentation des algorithmes

Les algorithmes se présentent sous la forme de segments de code, de fonctions ou de procédures. Les types des paramètres, des fonctions et des variables sont toujours explicités. Par défaut, les passages de paramètres se font par valeur. Un texte accompagne l'algorithme lorsqu'un paramètre est modifié, et que le passage doit se faire par référence.

Comme dans tous les langages de programmation impératifs, les structures de contrôle suivantes sont employées :

- **si** condition **alors** instruction **sinon** instruction **finsi** ;
- **tant que** condition **faire** instruction **fintantque** ;
- **répéter** instruction **jusqu'à** condition **finrépéter** ;
- **pour** variable de valeur initiale **à** valeur finale **faire** instruction **finpour** ;
- **pour tout** élément d'un ensemble **faire** instruction **finpour**.

Les tableaux sont indexés par des entiers, et un élément d'indice i d'un tableau T s'écrit $T[i]$, de même les éléments d'un tableau M à deux dimensions se notent $M[i, j]$.

La principale spécificité du pseudo-langage est le traitement des pointeurs. D'abord, nous nous affranchissons des problèmes liés à l'allocation dynamique de mémoire (les objets manipulés sont supposés alloués). Ensuite, pour ne pas alourdir la présentation des algorithmes, lorsqu'une variable x désigne un pointeur sur un enregistrement à plusieurs champs, par exemple un enregistrement comportant des champs nommés a et b, alors on notera $x.a$ la variable correspondant au champ a de l'enregistrement pointé par x. La même notation sera employée lorsque x représente l'enregistrement lui-même. Un pointeur qui ne pointe sur aucun enregistrement a la valeur *nul*.

Conventions relatives à l'index

Les entrées de l'index font référence à des notions de base traitées dans l'ouvrage. Lorsqu'une notion fait l'objet d'une définition explicite, ou d'un exercice qui lui est spécifiquement consacré, le numéro de page référencé apparaît en gras. Lorsqu'il s'agit d'un algorithme, le numéro de page apparaît en italique.

Remerciements

Les auteurs se souviennent avec tendresse du vide de leurs premiers regards qu'une explication miraculeuse a un jour allumé d'une petite flamme.

Ils saluent leurs maîtres, qui leur ont inculqué une forme de pensée archaïque consistant à chercher en toute circonstance la maîtrise absolue de la situation, par l'intermédiaire d'une preuve au pire longue et laborieuse, au mieux élégante et belle.

Ils arrosent d'une larme émue les bancs des facultés, usés par ceux qui ont, depuis vingt ans, à leur insu, participé à la conception de cet ouvrage.

Ils remercient chaleureusement les collègues dont ils ont sans vergogne pillé les archives.

Troisième édition

La troisième édition introduit dans la plupart des chapitres des exercices nouveaux qui se caractérisent par une plus grande originalité, une plus grande difficulté et par leur intérêt pédagogique pour appréhender en profondeur les fondements théoriques des concepts algorithmiques présentés dans le chapitre.

1 PREUVE ET COMPLEXITÉ

Un algorithme $(\mathcal{O}, \mathcal{V}, \mathcal{S})$ est formé d'un ensemble fini $\mathcal{O}$ d'opérations liées par une structure de contrôle $\mathcal{S}$ et dont les opérandes portent sur un ensemble fini $\mathcal{V}$ de variables. Un algorithme résout le problème P si pour tout énoncé I de P (stocké dans le sous-ensemble des variables d'entrée de $\mathcal{V}$), d'une part la suite des opérations exécutées est finie (condition de terminaison) et si d'autre part, lors de la terminaison, le sous-ensemble des variables de sortie de $\mathcal{V}$ contient le résultat associé à l'énoncé I (condition de validité). Prouver un algorithme, c'est démontrer mathématiquement que la propriété précédente est satisfaite. Une mesure de complexité d'un algorithme est une estimation de son temps de calcul, mémoire utilisée ou de toute autre unité significative. On s'intéresse le plus souvent à la recherche d'une estimation par excès à une constante positive multiplicative près du cas le plus défavorable sur le sous-ensemble des énoncés de taille fixée n (complexité dite « pire-cas »). On obtient ainsi le taux asymptotique de croissance de la mesure indépendamment de la machine sur laquelle l'algorithme est exécuté et l'on peut alors comparer les complexités pire-cas de plusieurs algorithmes résolvant un même problème.

Ce chapitre propose d'exercer cette démarche d'analyse sur des algorithmes de base, itératifs et récursifs, en demandant le développement détaillé des preuves à l'aide de questions progressives. Afin de définir rigoureusement la notion de mesure de complexité, nous introduisons les trois notations de Landau relatives aux ensembles de fonctions $O(g)$, $\Omega(g)$ et $\Theta(g)$, lorsque g est une fonction de $\mathbb{N}$ dans $\mathbb{N}$.

Définition 1.1 Borne supérieure asymptotique

$$O(g(n)) = \{f : \mathbb{N} \to \mathbb{N} \mid \exists k > 0 \text{ et } n_0 \geq 0 \text{ tels que } \forall n \geq n_0 \;\; 0 \leq f(n) \leq k\, g(n)\}$$

Si une fonction $f(n) \in O(g(n))$, on dit que $g(n)$ est une *borne supérieure asymptotique* pour $f(n)$.

On note abusivement : $f(n) = O(g(n))$.

Définition 1.2 Borne inférieure asymptotique

$$\Omega(g(n)) = \{f : \mathbb{N} \to \mathbb{N} \mid \exists k > 0 \text{ et } n_0 \geq 0 \text{ tels que } \forall n \geq n_0 \;\; 0 \leq k\, g(n) \leq f(n)\}$$

Si une fonction $f(n) \in \Omega(g(n))$, on dit que $g(n)$ est une *borne inférieure asymptotique* pour $f(n)$.
On note abusivement : $f(n) = \Omega(g(n))$.

Définition 1.3 Borne asymptotique

$$\Theta(g(n)) = \{f : \mathbb{N} \to \mathbb{N} \mid \exists k_1 > 0,\; k_2 > 0 \text{ et } n_0 \geq 0 \text{ tels que } \forall n \geq n_0 \;\; 0 \leq k_1\, g(n) \leq f(n) \leq k_2\, g(n)\}$$

Si une fonction $f(n) \in \Theta(g(n))$, on dit que $g(n)$ est une *borne asymptotique* pour $f(n)$.
On note abusivement : $f(n) = \Theta(g(n))$.

Exercice 1.1 *Généralités sur les notations asymptotiques*

1. Démontrer que

$$n^2 \in O(10^{-5} n^3)$$
$$25n^4 - 19n^3 + 13n^2 \in O(n^4)$$
$$2^{n+100} \in O(2^n)$$

2. Donner les relations d'inclusion entre les ensembles suivants : $O(n \log n)$, $O(2^n)$, $O(\log n)$, $O(1)$, $O(n^2)$, $O(n^3)$ et $O(n)$.

3. Soit un ordinateur pour lequel toute instruction possède une durée de 10^{-6} secondes. On exécute un algorithme qui utilise, pour une donnée de taille n, $f(n)$ instructions, $f(n)$ étant l'une des fonctions précédentes ($\log n$, $n \log n$, n, n^2, n^3 et 2^n). Remplir un tableau qui donne, en fonction de la taille $n = 10$, 20, 30 et 60, et de la fonction $f(n)$, la durée d'exécution de l'algorithme.

Soient f, g, S et $T : \mathbb{N} \to \mathbb{N}$.

4. Montrer que si $f(n) \in O(g(n))$, alors $g(n) \in \Omega(f(n))$.

5. Montrer que si $f(n) \in O(g(n))$, alors $f(n) + g(n) \in O(g(n))$.

6. Montrer que $f(n) + g(n) \in \Theta(\max(f(n), g(n)))$.

7. Montrer que $O(f(n) + g(n)) = O(\max(f(n), g(n)))$.

On suppose que $S(n) \in O(f(n))$ et $T(n) \in O(g(n))$.

8. Montrer que si $f(n) \in O(g(n))$, alors $S(n) + T(n) \in O(g(n))$.

9. Montrer que $S(n)T(n) \in O(f(n)g(n))$.

Solution

1. Pour la première relation, il suffit donc de trouver deux entiers strictement positifs k et n_0 tels que

$$n^2 \leq k\,10^{-5}\,n^3 \quad \forall n \geq n_0$$

Si l'on prend $k = 10^5$ et $n_0 = 1$, il est évident que

$$n^2 \leq k\,10^{-5}\,n^3 = n^3 \quad \forall n \geq n_0$$

donc $n^2 \in O(10^{-5}n^3)$.

Pour la seconde relation, remarquons tout d'abord que

$$25n^4 - 19n^3 + 13n^2 \leq 25n^4 + 19n^3 + 13n^2 \leq (25 + 19 + 13)n^4 \quad \forall n \geq 1$$

On en déduit que $25n^4 - 19n^3 + 13n^2 \in O(n^4)$.

Il suffit en effet de choisir $k = 25 + 19 + 13 = 57$ et $n_0 = 1$.

Finalement, soit $k = 2^{100}$ et $n_0 = 0$, il est évident que

$$2^{n+100} \leq k2^n \quad \forall n \geq n_0$$

2. Il est évident que $O(1) \subset O(n) \subset O(n^2) \subset O(n^3)$, puisque

$$1 \leq n \leq n^2 \leq n^3 \quad \forall n \geq 1$$

On a également $O(1) \subset O(\log n) \subset O(n) \subset O(n \log n)$, puisque

$$1 \leq \log n \quad \forall n \geq 3$$

et

$$\log n \leq n \quad \forall n \geq 1$$

et pour la même raison, $O(n \log n) \subset O(n^2)$.

On montre facilement que $O(n^3) \subset O(2^n)$. En effet,

$$n^3 \leq 2^n \;\Leftrightarrow\; \log n^3 \leq \log 2^n \;\Leftrightarrow\; 3 \log n \leq n \log(2)$$

qui est vrai à partir d'une certaine valeur de n puisque $\log n \in O(n)$.

L'ordre d'inclusion est donc finalement :

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n)$$

Un algorithme ayant une complexité logarithmique sera donc meilleur (en terme de complexité) qu'un algorithme linéaire, lui-même meilleur qu'un algorithme polynomial, à son tour meilleur qu'un algorithme exponentiel.

3. Le tableau suivant est calculé en utilisant un logarithme népérien :

$f(n)$	10	20	30	60
$\log n$	$2{,}30 \times 10^{-6}$	$3{,}00 \times 10^{-6}$	$3{,}40 \times 10^{-6}$	$4{,}09 \times 10^{-6}$
n	10^{-5}	2×10^{-5}	3×10^{-5}	6×10^{-5}
$n \log n$	$2{,}30 \times 10^{-5}$	$5{,}99 \times 10^{-5}$	$1{,}02 \times 10^{-4}$	$2{,}46 \times 10^{-4}$
n^2	10^{-4}	4×10^{-4}	9×10^{-4}	$3{,}60 \times 10^{-3}$
n^3	10^{-3}	8×10^{-3}	$2{,}70 \times 10^{-2}$	$2{,}16 \times 10^{-1}$
2^n	$1{,}02 \times 10^{-3}$	1,05	$1{,}07 \times 10^{3}$	$1{,}15 \times 10^{12}$

Il est intéressant de remarquer que $1,15 \times 10^{12}$ s $\approx$ 36 558 ans.

4. Si $f(n) \in O(g(n))$, alors il existe $k > 0$ et $n_0 \geq 0$ tels que

$$f(n) \leq kg(n) \quad \forall n \geq n_0$$

On déduit de l'expression précédente que

$$\frac{1}{k} f(n) \leq g(n) \quad \forall n \geq n_0$$

Donc $g(n)$ est en $\Omega(f(n))$.

5. Si $f(n) \in O(g(n))$, alors il existe $k > 0$ et $n_0 \geq 0$ tels que

$$f(n) \leq kg(n) \quad \forall n \geq n_0$$

On déduit de l'expression précédente que

$$f(n) + g(n) \leq (k+1)g(n) \quad \forall n \geq n_0$$

Donc $f(n) + g(n)$ est en $O(g(n))$.

6. De façon évidente,

$$\max(f(n), g(n)) \leq f(n) + g(n) \leq 2 \max(f(n), g(n)) \quad \forall n \geq 0$$

$f(n) + g(n)$ est donc en $\Theta(\max(f(n), g(n)))$.

7. Pour montrer l'égalité des deux ensembles $O(f(n) + g(n))$ et $O(\max(f(n), g(n)))$, il faut montrer la double inclusion.

$O(f(n) + g(n)) \subset O(\max(f(n), g(n)))$:

Pour toute fonction $h(n) \in O(f(n) + g(n))$, il existe $k > 0$ et $n_0 \geq 0$ tels que

$$h(n) \leq k(f(n) + g(n)) \leq 2k \max(f(n), g(n)) \quad \forall n \geq n_0$$

Donc $h(n)$ est donc en $O(\max(f(n), g(n)))$.

$O(\max(f(n), g(n))) \subset O(f(n) + g(n))$:

Pour toute fonction $h(n) \in O(\max(f(n), g(n)))$, il existe $k > 0$ et $n_0 \geq 0$ tels que

$$h(n) \leq k \max(f(n), g(n)) \leq k(f(n) + g(n)) \quad \forall n \geq n_0$$

Donc $h(n)$ est en $O(f(n) + g(n))$.

8. Si $S(n) \in O(f(n))$ et $T(n) \in O(g(n))$, alors il existe $k > 0$, $k' > 0$, $n_0 \geq 0$ et $n'_0 \geq 0$ tels que

$$S(n) \leq kf(n) \quad \forall n \geq n_0$$

$$T(n) \leq k'g(n) \quad \forall n \geq n'_0$$

Si $f(n) \in O(g(n))$, alors il existe $k'' > 0$ et $n''_0 \geq 0$ tels que

$$f(n) \leq k''g(n) \quad \forall n \geq n_0$$

On définit alors $K = kk'' + k'$ et $m_0 = \max(n_0, n'_0, n''_0)$. De façon évidente :

$$S(n) + T(n) \leq kf(n) + k'g(n) \leq (kk'' + k')g(n) = Kg(n) \quad \forall n \geq m_0$$

Donc $S(n) + T(n)$ est en $O(g(n))$.

Si dans un algorithme, on a une première partie en $O(f(n))$ suivie (séquentiellement) d'une seconde partie en $O(g(n))$ et que $f(n) \in O(g(n))$, alors l'algorithme est globalement en $O(g(n))$.

9. De la même façon, si on définit $K = kk'$ et $m_0 = \max(n_0, n'_0)$, on a de façon évidente :

$$S(n)T(n) \leq Kf(n)g(n) \quad \forall n \geq m_0$$

Donc $S(n)T(n)$ est en $O(f(n)g(n))$.

Si dans un algorithme, on a une première partie en $O(f(n))$ dans laquelle est imbriquée une seconde partie en $O(g(n))$, alors l'algorithme est globalement en $O(f(n)g(n))$.

Exercice 1.2 Somme des éléments d'un tableau

Soit A un tableau de n entiers ($n \geq 1$).

1. Écrire un algorithme itératif qui calcule la somme des éléments de A et prouver cet algorithme.

2. Déterminer la complexité de cet algorithme.

3. Écrire un algorithme récursif qui calcule la somme des éléments de A et prouver cet algorithme.

4. Déterminer la complexité de cet algorithme récursif.

Solution

1. L'algorithme calculant la somme des n éléments du tableau A est le suivant :

```
Somme : entier
Somme ← 0
pour i ← 1 à n faire
   Somme ← Somme + A[i]
fin pour
```

Pour prouver cet algorithme il faut démontrer deux choses : la terminaison et la validité de l'algorithme.

La terminaison de l'algorithme est triviale puisque la boucle est exécutée n fois et que le corps de la boucle n'est constitué que d'une somme et d'une affectation, opérations qui s'exécutent en un temps fini.

Pour démontrer la validité, nous allons considérer la propriété suivante : « À la fin de l'itération i, la variable *Somme* contient la somme des i premiers éléments du tableau A ». Notons $Somme_i$ la valeur de la variable *Somme* à la fin de l'itération i et $Somme_0 = 0$ sa valeur initiale. On effectue alors un raisonnement par récurrence sur i. La propriété est trivialement vraie pour $i = 1$ puisqu'à l'issue de la première itération *Somme* (initialisée à 0) contient la valeur $A[1]$:

$$Somme_1 = Somme_0 + A[1] = A[1]$$

Maintenant, si on suppose que la propriété est vraie pour une itération i :

$$Somme_i = \sum_{j=1}^{i} A[j]$$

À la fin de l'itération $i + 1$, *Somme* contiendra la valeur qu'elle contenait à la fin de l'itération i, donc la somme des i premiers éléments, plus $A[i + 1]$ qui lui a été ajoutée à l'itération $i + 1$. *Somme* sera donc bien la somme des $i + 1$ premiers éléments de A :

$$Somme_{i+1} = Somme_i + A[i + 1] = \sum_{j=1}^{i} A[j] + A[i + 1] = \sum_{j=1}^{i+1} A[j]$$

Vraie initialement, la propriété reste vraie à chaque nouvelle itération. Cette propriété est donc un invariant de l'algorithme. On parle d'*invariant de boucle*. L'algorithme se terminant, à la fin de l'itération n, il aura donc bien calculé la somme des n éléments du tableau A.

2. On s'intéresse au nombre $a(n)$ d'additions effectuées par l'algorithme. De façon évidente, $a(n) = n$. On en déduit que l'algorithme est en $O(n)$. Il est à noter, que l'inégalité $a(n) \leq n$ suffisait à aboutir à cette conclusion. On peut par ailleurs déduire de l'inégalité $a(n) \geq n$ que l'algorithme est en $\Omega(n)$ et donc finalement conclure qu'il est en $\Theta(n)$.

3. La fonction récursive calculant la somme des éléments du tableau A est la suivante :

```
fonction Somme_Rec(A : tableau ; n : entier) : entier
si n ≤ 0
  alors
    retourner(0)
  sinon
    retourner(Somme_Rec(A, n − 1)+A[n])
fin si
```

On démontre tout d'abord la terminaison, par récurrence sur i. De façon évidente l'appel de la fonction *Somme_Rec* avec le paramètre 0 se termine immédiatement. Si on suppose que l'appel de la fonction *Somme_Rec* avec un paramètre n se termine, l'appel de la fonction *Somme_Rec* avec le paramètre $n + 1$, appelle la fonction *Somme_Rec* avec le paramètre n, qui par hypothèse se termine, ajoute à son résultat la valeur de $A[n]$ et se termine. Donc quelle que soit la valeur du paramètre n, l'appel de la fonction *Somme_Rec* se termine.

On note $Somme_n$ la valeur retournée par l'appel de la fonction *Somme_Rec* avec le paramètre n. Cette fonction récursive reproduit la relation mathématique de récurrence suivante :

$$\begin{cases} Somme_0 = 0 \\ Somme_n = Somme_{n-1} + A[n] & \text{si } n > 0 \end{cases}$$

On montre facilement par récurrence (exactement de la même façon que cela a été fait dans la question précédente), que la solution de cette expression récursive est :

$$Somme_n = \sum_{i=1}^{n} A[i]$$

Dès l'instant où la fonction se termine (ce que nous avons prouvé), elle calcule bien la somme des éléments du tableau A.

4. À nouveau, on appelle $a(n)$ le nombre d'additions effectuées par la fonction *Somme_Rec* appelée avec un paramètre n. $a(n)$ est solution de l'équation de récurrence :

$$a(n) = \begin{cases} 0 & \text{si } n = 0 \\ a(n-1) + 1 & \text{si } n > 0 \end{cases}$$

dont la solution (évidente) est $a(n) = n$. On en déduit, à nouveau, que l'algorithme est à la fois en $O(n)$ et en $\Omega(n)$, donc globalement en $\Theta(n)$.

Exercice 1.3 Fonctions F et G mystères

1. Soit la fonction récursive F d'un paramètre entier n suivante :

```
fonction F(n : entier) : entier
si n = 0
   alors
      retourner(2)
   sinon
      retourner(F(n − 1) ∗ F(n − 1))
fin si
```

Que calcule cette fonction ? Le prouver.

2. Déterminer la complexité de la fonction F. Comment améliorer cette complexité ?

3. Soit la fonction itérative G suivante :

```
fonction G(n : entier) : entier
R : entier
R ← 2
pour i ← 1 à n faire
   R ← R ∗ R
fin pour
retourner(R)
```

Que calcule cette fonction ? Le prouver.

4. Déterminer la complexité de la fonction G.

Solution

1. On calcule les premières valeurs calculées par la fonction F :

$$F(0) = 2$$
$$F(1) = 2 * 2 = 2^2$$
$$F(2) = 2^2 * 2^2 = 2^4$$
$$F(3) = 2^4 * 2^4 = 2^8$$
$$F(4) = 2^8 * 2^8 = 2^{16}$$

De façon intuitive, on obtient l'expression générale de $F(n)$:

$$F(n) = 2^{2^n}$$

Afin de le prouver, il faut démontrer la terminaison et la validité de l'algorithme.

La terminaison de la fonction F se démontre par récurrence sur n. De façon évidente, l'appel de la fonction F avec un paramètre 0 se termine immédiatement. Si on suppose que l'appel de la fonction F avec un paramètre n se termine, l'appel de la fonction F avec le paramètre $n+1$ appelle deux fois la fonction F avec le paramètre n, ces deux appels se terminant par hypothèse, multiplie les deux résultats (identiques) obtenus et se termine.

Si on note F_n la valeur retournée par l'appel de la fonction F avec le paramètre n, la fonction récursive F reproduit la relation mathématique suivante :

$$\begin{cases} F_0 = 0 \\ F_n = F_{n-1} * F_{n-1} = F_{n-1}^2 & \text{si } n > 0 \end{cases}$$

On montre alors par récurrence sur n que la solution de cette relation de récurrence est :

$$F_n = 2^{2^n}$$

Cette relation est bien satisfaite pour $n = 0$. Si on la suppose satisfaite pour n, $F_n = 2^{2^n}$, on calcule F_{n+1} à partir de la relation de récurrence :

$$F_{n+1} = F_n^2 = (2^{2^n})^2 = 2^{2^{n+1}}$$

On montre donc bien par récurrence que, pour tout $n \geq 0$, $F_n = 2^{2^n}$.

2. On s'intéresse au nombre $m(n)$ de multiplications effectuées. $m(n)$ est solution de l'équation de récurrence :

$$m(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 + 2 * m(n-1) & \text{si } n > 0 \end{cases}$$

On montre très facilement par récurrence que $m(n) = \sum_{i=0}^{n-1} 2^i = 2^n - 1$. On en déduit que l'algorithme est en $\Theta(2^n)$ et possède donc une complexité exponentielle.

Pour améliorer l'algorithme, il suffit bien évidemment d'éviter le double appel récursif ; la fonction F s'écrit alors :

```
fonction F(n : entier) : entier
si n = 0
   alors
      retourner(2)
   sinon
      retourner(F(n − 1)²)
fin si
```

On calcule alors la complexité par rapport au nombre $c(n)$ d'appels de la fonction « carré ». $c(n)$ est solution de l'équation de récurrence :

$$c(n) = \begin{cases} 0 & \text{si } n = 0 \\ c(n-1) + 1 & \text{si } n > 0 \end{cases}$$

De façon évidente, $c(n) = n$. On en déduit que le nouvel algorithme est en $\Theta(n)$.

3. La fonction G calcule la même valeur que la fonction F. Pour le démontrer il faut prouver la terminaison et la validité de l'algorithme.

La terminaison est triviale puisque l'algorithme effectue n itérations.

Pour prouver la validité, il faut déterminer un invariant de boucle. On se propose de prouver que la propriété suivante en est un : « À la fin de l'itération i, $R = 2^{2^i}$ ».

Cette propriété est vérifiée à la fin de la première itération, puisque R (initialisé à 2) vaut alors $4 = 2^{2^1}$. Si on suppose que la propriété est vraie pour i, à la fin de l'itération $i + 1$, $R = 2^{2^i} * 2^{2^i} = 2^{2^{i+1}}$. La propriété est donc bien un invariant de boucle. Elle est donc vraie à la terminaison de l'algorithme. Celui-ci calcule donc bien 2^{2^n}.

4. Si $m(n)$ est à nouveau le nombre de multiplications effectuées par l'algorithme, de façon évidente, $m(n) = n$. On en déduit que l'algorithme est en $\Theta(n)$.

Exercice 1.4 Produit matriciel

On considère deux matrices carrées (d'entiers) d'ordre n, A et B. Le produit de A par B est une matrice carrée C définie par :

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \times B_{k,j}$$

1. Donner un algorithme calculant le produit de deux matrices représentées sous forme d'un tableau à deux dimensions. Calculer la complexité de cet algorithme. Doit-on préciser dans quels cas (pire cas, meilleur des cas, cas moyen) cette complexité est obtenue ?

2. Modifier l'algorithme précédent lorsque la matrice A est de dimension (m, n) et la matrice B de dimension (n, p). Quelle est alors la complexité de l'algorithme ?

Solution

1. L'algorithme calculant le produit des deux matrices A et B est le suivant :

```
i, j, k : entier
pour i ← 1 à n faire
   pour j ← 1 à n faire
      C[i, j] ← 0
      pour k ← 1 à n faire
         C[i, j] ← C[i, j] + A[i, k] × B[k, j]
      fin pour
   fin pour
fin pour
```

Si l'on s'intéresse au nombre $a(n)$ d'additions (ou de multiplications) effectuées par l'algorithme, on obtient de façon immédiate que $a(n) = n^3$. On en déduit que l'algorithme est en $\Theta(n^3)$. Il n'y a donc ici ni meilleur ni pire des cas.

2. L'algorithme modifié est le suivant :

```
i, j, k : entier
pour i ← 1 à m faire
   pour j ← 1 à p faire
      C[i, j] ← 0
      pour k ← 1 à n faire
         C[i, j] ← C[i, j] + A[i, k] × B[k, j]
      fin pour
   fin pour
fin pour
```

Le nombre d'additions (ou de multiplications) effectuées est cette fois-ci $a(n) = npm$. On en déduit que l'algorithme est en $\Theta(npm)$.

Exercice 1.5 *Complexité en fonction de deux paramètres*

Déterminer la complexité des algorithmes suivants (par rapport au nombre d'itérations effectuées), où m et n sont deux entiers positifs.

```
Algorithme A
i ← 1 ; j ← 1
tant que (i ≤ m) et (j ≤ n) faire
   i ← i + 1
   j ← j + 1
fin tant que
```

```
Algorithme B
i ← 1 ; j ← 1
tant que (i ≤ m) ou (j ≤ n) faire
   i ← i + 1
   j ← j + 1
fin tant que
```

```
Algorithme C
i ← 1 ; j ← 1
tant que j ≤ n faire
   si i ≤ m
      alors
         i ← i + 1
      sinon
         j ← j + 1
   fin si
fin tant que
```

```
Algorithme D
i ← 1 ; j ← 1
tant que j ≤ n faire
   si i ≤ m
      alors
         i ← i + 1
      sinon
         j ← j + 1
         i ← 1
   fin si
fin tant que
```

Solution

Le corps des différents algorithmes ne contient que des opérations en $O(1)$. En comptant le nombre d'itérations effectuées par chacun des algorithmes, on obtient immédiatement que :

- l'algorithme A est en $O(\min(m, n))$;
- l'algorithme B est en $O(\max(m, n))$;
- l'algorithme C est en $O(m + n)$;
- l'algorithme D est en $O(mn)$.

Exercice 1.6 Polynômes

On considère la fonction suivante, qui a deux arguments : un tableau $T[0..n]$ de $n + 1$ nombres réels et un nombre réel x.

```
fonction Calcul(T : tableau ; x : réel) : réel
s ← T[n]
pour i ← n − 1 à 0 faire
   s ← T[i] + x ∗ s
fin pour
retourner(s)
```

1. Donner le déroulement de l'algorithme pour la donnée suivante :

$n = 5 \quad T =$ | 3 | 8 | 0 | 1 | 1 | 2 | et $x = 2$

2. Exprimer le résultat de la fonction *Calcul* en fonction de x et de T. Que calcule cette fonction ? Le prouver.

3. Écrire une fonction récursive réalisant le même calcul. Prouver l'algorithme.

Solution

1. On donne les valeurs de s et de $T[i]$ à l'initialisation et à la fin de chaque itération :

	s	*T[i]*
Init.	2	2
***i* = 4**	5	1
***i* = 3**	11	1
***i* = 2**	22	0
***i* = 1**	52	8
***i* = 0**	107	3

2. On note s_i la valeur de la variable s à la fin de l'itération i et s_n sa valeur initiale. On a :

$$s_n = T[n] \text{ et } s_i = T[i] + x \times s_{i+1}$$

On montre facilement par récurrence (inversée) que :

$$s_i = \sum_{j=i}^{n} T[j] \times x^{j-i}$$

On en déduit que la fonction *Calcul* évalue le polynôme de degré n dont les coefficients sont les valeurs du tableau T, au point x :

$$s_0 = \sum_{j=0}^{n} T[j] \times x^{j}$$

On vient en fait de démontrer la validité de l'algorithme. Il reste à vérifier la terminaison. Celle-ci est bien sûr triviale puisque l'algorithme n'est constitué que d'une boucle de n itérations.