Chapitre 2
Complexité des algorithmes

L'exécution d'un programme nécessite l'utilisation des ressources de l'ordinateur : temps de calcul pour exécuter les opérations, et occupation de la mémoire pour contenir et manipuler le programme et ses données.

L'objet de l'analyse de la complexité est de quantifier les deux grandeurs physiques «temps d'exécution» et «place mémoire», dans le but de comparer entre eux différents algorithmes qui résolvent le même problème.

Il s'agit d'abord de déterminer quelle mesure utiliser pour calculer ces deux quantités : pour un programme donné sur une machine donnée, on peut par exemple exprimer la complexité en temps (resp. en place) par le nombre de cycles machine (resp. mots mémoire) utilisés lors de l'exécution du programme, en comptant

• pour le temps : le nombre d'opérations effectuées par le programme et le temps nécessaire pour chaque opération,

• pour la place : le nombre d'instructions et le nombre de données du programme, avec le nombre de mots mémoire nécessaires pour stocker chacune d'entre elles, ainsi que le nombre de mots mémoire supplémentaires pour la manipulation des données. Ce type d'analyse conduit à des énoncés comme :

> «*L'algorithme A implémenté par le programme P sur l'ordinateur O, et exécuté sur la donnée D utilise k secondes de calcul et j bits de mémoire.*»

Un résultat de ce genre peut être une source d'information intéressante, mais le but de l'analyse de la complexité des algorithmes est d'établir des résultats plus généraux permettant d'estimer l'efficacité intrinsèque de la méthode utilisée par un algorithme, indépendamment de la machine, du langage de programmation, du compilateur et de tous les détails d'implémentation.

Le type d'énoncé que l'on souhaite produire est :

> «*Sur toute machine, et quel que soit le langage de programmation, l'algorithme A1 est meilleur que l'algorithme A2 pour les données de grande taille.*»

ou encore,

> *«L'algorithme A est optimal en nombre de comparaisons pour résoudre le problème Q.»*

On précise plus loin ce que l'on entend par «meilleur», «taille d'une donnée», «optimal»...

1. Complexité d'un algorithme

On cherche à déterminer une mesure qui rende compte de la complexité intrinsèque des algorithmes, indépendamment de l'implémentation, et permette ainsi de comparer entre eux des algorithmes.

On va pour l'instant se consacrer à l'étude de la complexité en temps, et on reviendra à la fin du paragraphe sur la complexité en place.

1.1. Mesure de la complexité en temps

Pour certains problèmes, on peut mettre en évidence une ou plusieurs *opérations* qui sont *fondamentales* au sens où le temps d'exécution d'un algorithme résolvant ce problème est toujours proportionnel au nombre de ces opérations. Il est alors possible de comparer des algorithmes traitant ce problème selon cette mesure simplifiée.

Donnons quelques exemples d'opérations fondamentales :

- pour la recherche d'un élément dans une liste en mémoire centrale : le nombre de comparaisons entre cet élément et les entrées de la liste;
- pour la recherche d'un élément sur un disque : le nombre d'accès à la mémoire secondaire;
- pour trier une liste d'éléments : on peut considérer deux opérations fondamentales : le nombre de comparaisons entre deux éléments et le nombre de déplacements d'éléments;
- pour multiplier deux matrices de nombres : le nombre de multiplications et le nombre d'additions.

Remarquons que si l'on choisit plusieurs opérations fondamentales, on doit les décompter séparément puis, si besoin est, on les affecte chacune d'un poids qui tient compte des temps d'exécution différents.

Soulignons deux points importants.

a) En faisant varier le nombre d'opérations fondamentales, on fait varier le degré de précision de l'analyse, et aussi son degré d'abstraction, i.e. d'indépendance par rapport à l'implémentation. A la limite si l'on veut faire une «microanalyse» très

précise du temps d'exécution du programme, il suffit de décider que toutes les opérations du programme sont fondamentales.

b) On a fait l'hypothèse que le temps d'exécution est proportionnel à la mesure choisie. Cependant il se peut qu'après avoir analysé quelques algorithmes en fonction d'une certaine opération choisie comme fondamentale pour le problème, on découvre des algorithmes résolvant le même problème – ou un sous-problème – par des méthodes si différentes qu'ils ne font aucune opération de ce type. Dans ce cas, il faut se restreindre à la «classe d'algorithmes» pour laquelle la mesure choisie est significative. Les algorithmes qui utilisent d'autres techniques nécessitant le choix d'autres opérations fondamentales sont étudiés séparément, et ne peuvent pas être comparés à ceux de la classe précédente (on verra un exemple de cette situation pour des algorithmes de tri).

1.2. Calcul de la complexité

Après avoir déterminé les opérations fondamentales, il s'agit de compter le nombre d'opérations de chaque type. Il n'existe pas de système complet de règles permettant de compter le nombre d'opérations en fonction de la syntaxe des algorithmes mais l'on peut faire quelques remarques.

a) Lorsque les opérations sont dans une séquence d'instructions, leurs nombres s'ajoutent.

b) Pour les branchements conditionnels, il est en général difficile de déterminer quelle branche de la condition est exécutée, et donc quelles sont les opérations à compter. Cependant, on peut majorer ce nombre d'opérations : par exemple, si $P(X)$ est le nombre d'opérations fondamentales de la construction X :

$$P\ (\textbf{if}\ C\ \textbf{then}\ I_1\ \textbf{else}\ I_2) \ \leq\ P(C) + \max\left(P(I_1),\ P(I_2)\right)$$

c) Pour les boucles, le nombre d'opérations dans la boucle est $\sum P(i)$, où i est la variable de contrôle de la boucle, et $P(i)$ le nombre d'opérations fondamentales lors de l'exécution de la $i^{ième}$ itération.

Pour évaluer les bornes de i dans la somme précédente, il faut connaître le nombre d'itérations du corps de la boucle. Ce nombre peut être défini dans l'algorithme (cas d'une «boucle **for**»), sinon il doit être calculé à partir de l'algorithme et ce calcul peut s'avérer difficile; dans ce cas on peut se contenter quelquefois d'un majorant.

d) Pour les appels de procédure ou de fonction :

• s'il n'y a pas de procédures ou de fonctions récursives, on peut toujours trouver une façon d'ordonner les procédures et fonctions de telle sorte que chacune d'entre elles n'appelle que des procédures ou fonctions dont le nombre d'opérations fondamentales a déjà été évalué;

• pour les procédures ou fonctions récursives, compter le nombre d'opérations fondamentales donne en général lieu à la résolution de relations de récurrence. En effet, le nombre $T(n)$ d'opérations dans l'appel de la procédure avec un argument de taille n s'écrit, selon la récursion, en fonction de divers $T(k)$, pour $k < n$. Par exemple, l'algorithme ci-dessous calcule la factorielle d'un entier positif.

```
function fact (n : integer) : integer;
begin
    if n = 0 then fact := 1 else fact := n * fact (n - 1)
end fact;
```

Si l'on choisit comme opération fondamentale la multiplication de deux entiers, on obtient clairement que le nombre $T(n)$ d'opérations fondamentales vérifie $T(0) = 0$ et $T(n) = T(n-1) + 1$, pour $n \geq 1$, d'où par une résolution directe de cette récurrence très simple, $T(n) = n$.

(On trouvera en annexe un exposé de certaines méthodes de résolution de relations de récurrence.)

Pour illustrer les indications ci-dessus sur la manière de déterminer la complexité d'un algorithme (on dit qu'on analyse l'algorithme), on va traiter un exemple.

Reprenons l'algorithme de recherche séquentielle d'un élément dans une liste (on a indiqué des numéros de ligne pour faciliter la compréhension de l'analyse).

```
        var L : array [1..n] of element;
            X : element;
            j : integer;
        begin
(1)         j := 1;
(2)         while (j ≤ n) and (L[j] ≠ X)
(3)               do j := j + 1;
(4)         if j > n then j := 0
        end;
```

Dans cet algorithme, les éléments significatifs pour analyser la complexité en nombre d'opérations sont les suivants :

- le nombre d'itérations,
- le nombre d'opérations par itération.

On peut remarquer que l'instruction $j := j + 1$ de la ligne (3) est dépendante de la programmation : elle disparaît si on programme l'algorithme différemment, avec une boucle **for**. Il en est de même de la comparaison $j \leq n$ de la ligne (2). On voit bien qu'il ne faut pas prendre en compte ces opérations pour l'évaluation de l'algorithme.

De plus ces instructions sont dépendantes de la structure de données choisie pour représenter la liste d'éléments (cf. chapitre 5). Avec une liste chaînée on aurait d'autres instructions : manipulations de pointeurs, test si un pointeur est égal à nil.

Les opérations significatives dans cet algorithme sont donc les comparaisons de X avec les éléments de la liste. Il y en a une par itération.

Le nombre d'itérations est égal à n si X n'est pas dans la liste, et à j, rang de la première occurrence de X, si X est dans la liste.

L'analyse se fait en établissant des *invariants de boucle*, c'est-à-dire des propriétés qui sont vraies à chaque itération, et des *conditions d'arrêt.*

a) Invariant de boucle :

(i) – au début de la première itération (ligne (2)) on a $j = 1$,
– au début de la $k^{ième}$ itération on a : $j = k$ et $\forall i,\ 1 \leq i < k,\ L[i] \neq X$.

b) Conditions d'arrêt :

(ii) si au début de la $k^{ième}$ itération de la boucle on a : $k \leq n$ et $L[k] = X$, alors on va s'arrêter avec $j = k$;
(iii) si on a $k = n + 1$, alors on va s'arrêter avec $j = 0$.

On démontre les propriétés précédentes par récurrence sur k, nombre d'itérations.

Pour $k = 1$:

- La propriété (i) est vraie.
- Pour (ii) : $1 \leq n$ et $L[1] = X$, donc on sort de la boucle, on exécute le **if-then** et j reste inchangé (car $j \leq n$) : $j = k = 1$.
- Pour (iii) : si $k = n + 1$ alors $n = 0$; la liste est vide.
  On va exécuter le **if-then.**
  Comme $j > n$ et $n = 0$, on exécute $j := 0$.

Supposons les propriétés vraies pour k. On les démontre pour $k + 1$:

- Démontrons la propriété (i) à la $(k+1)^{ième}$ itération : on avait $j = k$ au début de la $k^{ième}$ itération (hypothèse de récurrence). On a eu $k \leq n$ et $L[k] \neq X$ puisqu'on a continué la boucle.

Donc, on a exécuté $j := j + 1$. Donc, on a maintenant $j = k + 1$.

Or, on avait $L[i] \neq X,\ 1 \leq i < k$ par hypothèse de récurrence, de plus $L[k] \neq X$ donc $L[i] \neq X,\ 1 \leq i < k + 1$.

- Les propriétés (ii) et (iii) se démontrent de la même façon que pour le cas où $k = 1$.

Les propriétés (ii) et (i) impliquent que j est l'indice de la première occurrence de X dans L.

Les propriétés (iii) et (i) impliquent que si $j = 0$ à la fin de l'algorithme, il n'y a pas d'occurrence de X dans L.

On a donc démontré qu'il y a au plus n itérations, et qu'il y en a j, si j est l'indice de la première occurrence de X. Comme il y a une comparaison par itération, si $j \leq n$, on sait maintenant que cet algorithme effectue : j comparaisons si j est l'indice de la première occurrence de X, n comparaisons si X n'est pas dans la liste.

Ce premier exemple d'analyse d'algorithme met en évidence trois points essentiels :

- le choix de (ou des) l'opération(s) que l'on prend en compte doit être établi avant toute analyse et précisé dans le résultat,
- la complexité dépend de la taille des données (ici n),
- la complexité dépend, pour une taille fixée, des différentes données possibles.

Ici pour les données de taille n, la complexité varie de 1 à n. Ceci dépend des données X et L : les données où X apparaît au rang i de L correspondent à une complexité i en nombre de comparaisons; celles où X n'apparaît pas dans L, à une complexité n.

1.3. Autres critères d'évaluation

La complexité en temps n'est pas le seul critère d'évaluation d'un algorithme. D'autres critères entrent en ligne de compte, comme la place mémoire, la simplicité de l'algorithme, ou l'adéquation à certaines données.

Il se peut en effet que l'on soit intéressé principalement par la complexité en place, c'est-à-dire la quantité de mémoire nécessaire à l'exécution d'un algorithme. De même que pour la complexité en temps, on peut définir différentes mesures qui rendent compte de la complexité intrinsèque d'un algorithme avec une certaine structuration des données, indépendamment de l'implémentation. Très souvent, un algorithme plus rapide utilisera plus de place; dans le cas extrême où un algorithme est très efficace, mais utilise plus de place qu'il n'y en a en mémoire centrale, étant donné qu'il est au moins un million de fois plus lent d'accéder à une information rangée en mémoire secondaire, il vaut évidemment mieux utiliser un algorithme moins «rapide» pouvant être traité uniquement en mémoire centrale. Il est donc important de choisir un algorithme avec un bon *compromis espace-temps.*

D'autre part, la simplicité et la clarté d'un algorithme peuvent aussi être un critère de choix : un algorithme efficace mais complexe sera probablement plus difficile, donc plus long, à implanter correctement qu'un algorithme moins subtil; il faut tenir

compte de ce «temps humain», en particulier si l'algorithme doit être utilisé peu de fois.

De plus, certains algorithmes, peu performants en général, sont très efficaces sur certaines formes de données (par exemple, le tri insertion sur des listes presque triées, cf. chapitre 14).

Et enfin, pour des problèmes particuliers, d'autres critères peuvent être prépondérants : par exemple, la précision pour un algorithme numérique ou la stabilité pour un algorithme de tri (cf. chapitres 14 et 16). Il faut donc savoir faire des compromis lorsqu'on choisit un algorithme et prendre en compte le contexte dans lequel le programme sera développé et utilisé.

2. Complexité en moyenne et au pire

Il est clair, et l'exemple de la recherche séquentielle l'a bien mis en évidence, que le temps d'exécution d'un algorithme dépend de la donnée sur laquelle il opère.

• Il faut d'abord définir une mesure de taille sur les données qui reflète la quantité d'information contenue. Par exemple, si l'on additionne ou multiplie des entiers, une mesure significative est le nombre de chiffres des nombres; dans le cas de la recherche (cf. 2^e partie de ce livre) ou du tri (cf. 3^e partie de ce livre), la taille sera souvent le nombre d'éléments manipulés; dans le cas du produit de matrices carrées, on peut prendre comme mesure de la taille, la dimension de la matrice; pour les parcours d'arbres (chapitre 7) ou de graphes (chapitre 8), on peut compter le nombre de nœuds ou le nombre d'arcs...

• Pour certains algorithmes (par exemple addition ou multiplication usuelles sur les nombres entiers), le temps d'exécution ne dépend que de la taille des données; mais la plupart du temps la complexité varie aussi, pour une taille fixée des données, en fonction de la donnée elle-même.

On peut définir plusieurs quantités pour caractériser le comportement d'un algorithme sur l'ensemble D_n des données de taille n.

Notons $coût_A(d)$ la complexité en temps de l'algorithme A sur la donnée d, complexité déterminée selon les méthodes décrites au paragraphe 1.2.

Définitions : On s'intéresse à plusieurs mesures.

a) La complexité ***dans le meilleur des cas***

$$\mathrm{Min}_A(n) = \min\{coût_A(d);\ d \in D_n\}$$

b) La complexité ***dans le pire des cas***

$$\mathrm{Max}_A(n) = \max\{coût_A(d);\ d \in D_n\}$$

c) La complexité ***en moyenne***

$$\text{Moy}_A(n) = \sum_{d \in D_n} p(d) \cdot coût_A(d)$$

où $p(d)$ est la probabilité que l'on ait la donnée d en entrée de l'algorithme.

(On omet l'indice A lorsque l'algorithme en cause est évident.)

Ces définitions entraînent plusieurs remarques.

1) Les complexités dans le meilleur et dans le pire des cas donnent des indications sur les bornes extrêmales de la complexité de l'algorithme sur les données de taille n. Leur détermination nécessite en général la construction de données particulières, qui forcent l'algorithme à se comporter de façon extrêmale.

La complexité dans le pire des cas, qui donne une borne supérieure du temps d'exécution, est particulièrement utile car elle permet de donner une estimation de la taille maximale des données qui pourront être traitées par l'algorithme.

2) Les cas extrêmaux ne sont pas les plus fréquents et dans la pratique on aimerait savoir quel comportement attendre «en général» de l'algorithme, d'où l'introduction de la complexité en moyenne.

Si toutes les données sont équiprobables alors la complexité en moyenne s'exprime simplement en fonction du nombre $|D_n|$ de données de taille n :

$$\text{Moy}_A(n) = \frac{1}{|D_n|} \cdot \sum_{d \in D_n} coût_A(d)$$

Mais en général, les données n'ont pas toutes la même probabilité et la définition de la complexité en moyenne nécessite l'introduction d'un modèle probabiliste lié au problème.

Souvent (cf. l'exemple, qui suit, de la recherche séquentielle, et l'exemple de la recherche du maximum au chapitre 3) on partitionne l'ensemble D_n en regroupant les données de taille n de même coût, et on évalue la probabilité $p(D_{n,k})$ de chaque classe $D_{n,k}$ de la partition. La complexité en moyenne devient alors

$$\text{Moy}_A(n) = \sum_{D_{n,k} \subseteq D_n} p(D_{n,k}) \cdot coût_A(D_{n,k})$$

où $coût_A(D_{n,k})$ représente le coût d'une donnée quelconque de $D_{n,k}$.

En pratique, la complexité en moyenne est souvent beaucoup plus difficile à déterminer que la complexité dans le pire des cas, d'une part parce que l'analyse devient mathématiquement difficile, et d'autre part parce qu'il n'est pas toujours facile de déterminer un modèle de probabilités adéquat au problème.

3) Clairement, il existe entre la complexité en moyenne et les complexités extrêmales la relation suivante :

$$\mathrm{Min}_A(n) \leq \mathrm{Moy}_A(n) \leq \mathrm{Max}_A(n)$$

Si le comportement de l'algorithme ne dépend que de la taille des données (comme dans l'exemple de la multiplication de matrices, donné plus loin), alors ces trois quantités sont confondues. Mais en général, ce n'est pas le cas et l'on ne sait même pas si le coût moyen est plus proche du coût minimal ou du coût maximal (sauf si l'on sait déterminer les fréquences relatives des données qui correspondent à un coût minimal et de celles qui correspondent à un coût maximal).

Remarquons enfin que ce n'est pas parce qu'un algorithme est meilleur qu'un autre en moyenne, qu'il est meilleur dans le pire des cas.

Pour terminer ce paragraphe, on traite deux exemples qui illustrent les définitions introduites :

Exemple A : Multiplication de matrices carrées.

Soit $A = (a_{ij})$ et $B = (b_{ij})$ deux matrices $n \times n$ à coefficients dans $\mathbb{R}$; l'algorithme suivant calcule les coefficients (c_{ij}) de la matrice produit $C = A \times B$ selon la formule classique

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

pour i et j compris entre 1 et n.

```
type matrice = array [1..n, 1..n] of integer;
procedure multmat (a, b : matrice; var c : matrice);
var i, j, k : integer;
begin
    for i := 1 to n do
        for j := 1 to n do
            begin c[i, j] := 0;
                for k := 1 to n do
                    c[i, j] := c[i, j] + a[i, k] * b[k, j]
            end
end multmat;
```

La complexité de l'algorithme *multmat*, comptée en nombre de multiplications de réels ne dépend que de la taille des matrices :

$$\mathrm{Min}(n) = \mathrm{Moy}(n) = \mathrm{Max}(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} 1 = n^3$$

Exemple B : Recherche séquentielle.

On cherche à déterminer la complexité, en nombre de comparaisons, de l'algorithme de recherche séquentielle d'un élément dans une liste, présenté au paragraphe 1.2. D'après l'analyse faite alors, on a $\text{Max}(n) = n$ et $\text{Min}(n) = 1$.

Pour calculer $\text{Moy}(n)$ on doit se donner des probabilités sur L et X :

- soit q la probabilité que X soit dans L;
- on suppose que si X est dans L, toutes les places sont équiprobables.

On note $D_{n,i}$ pour $1 \leq i \leq n$, l'ensemble des données où X apparaît à la $i^{ième}$ place et $D_{n,0}$ l'ensemble des données où X est absent. D'après les conventions ci-dessus on a :

$$p(D_{n,i}) = q/n \qquad \text{et} \qquad p(D_{n,0}) = 1 - q$$

D'après l'analyse de l'algorithme on a :

$$\textit{coût}(D_{n,i}) = i \qquad \text{et} \qquad \textit{coût}(D_{n,0}) = n$$

On a donc :

$$\begin{aligned}\text{Moy}(n) &= \sum_{i=0}^{n} p(D_{n,i}) \cdot \textit{coût}(D_{n,i}) = (1-q) \cdot n + \sum_{i=1}^{n} i.q/n \\ &= (1-q) \cdot n + (n+1) \cdot q/2\end{aligned}$$

Si on sait que X est dans la liste, on a $q = 1$ et :

$$\text{Moy}(n) = (n+1)/2$$

Si X a une chance sur deux d'être dans la liste, on a $q = 1/2$ et :

$$\text{Moy}(n) = n/2 + (n+1)/4 = (3n+1)/4$$

3. Comparaisons de deux algorithmes; ordre de grandeur

On a déterminé la complexité d'un algorithme comme une fonction de la taille des données; il est très important de connaître la rapidité de croissance de cette fonction lorsque la taille des données croît. En effet, pour traiter un problème de petite taille la méthode employée importe peu, alors que pour un problème de grande taille, les différences de performance entre algorithmes peuvent être énormes.

Souvent, une simple approximation de la fonction de complexité suffit pour savoir si un algorithme est utilisable ou non, ou pour comparer entre eux différents algorithmes.

Par exemple, pour n grand, il est souvent secondaire de savoir si un algorithme fait $n+1$ ou $n+2$ opérations.

Parfois les constantes multiplicatives ont, elles aussi, peu d'importance : supposons que l'on ait à comparer l'algorithme A_1 de complexité $M_1(n) = n^2$ et l'algorithme A_2 de complexité $M_2(n) = 2n$. A_2 est meilleur que A_1 pour presque tous les n $(n > 2)$; de même si $M_1(n) = 3n^2$ et $M_2(n) = 25n$, A_2 est meilleur que A_1 pour $n > 8$. Quelles que soient les constantes multiplicatives k_1 et k_2 telles que $M_1(n) = k_1 \cdot n^2$ et $M_2(n) = k_2 \cdot n$, l'algorithme A_2 est toujours meilleur que A_1 à partir d'un certain n, car la fonction $f(n) = n^2$ croît beaucoup plus vite que la fonction $g(n) = n$ (en effet $\lim_{n\to\infty} g(n)/f(n) = 0$).

On dit alors que l'***ordre de grandeur asymptotique*** de $f(n)$ est strictement plus grand que celui de $g(n)$.

La figure 1 met en évidence la différence de rapidité de croissance de certaines fonctions usuelles : les ordres de grandeur asymptotiques des fonctions $1, \log_2 n, n\log_2 n, n^2, n^3, 2^n$ vont en croissant strictement; ces fonctions forment une ***échelle de comparaison*** (cf. annexe).

Pour analyser la complexité [1] $M_A(n)$ d'un algorithme A, on s'attache d'abord à déterminer l'ordre de grandeur asymptotique de $M_A(n)$: on cherche dans une échelle de comparaison, éventuellement plus complète que celle qui est formée par les fonctions de la figure 1, une fonction qui a une rapidité de croissance voisine de celle de $M_A(n)$.

Supposons que l'on ait à comparer deux algorithmes A_1 et A_2 de complexités $M_{A_1}(n)$ et $M_{A_2}(n)$[2]. Si l'ordre de grandeur de $M_{A_1}(n)$ est strictement plus grand que l'ordre de grandeur de $M_{A_2}(n)$, alors on peut conclure immédiatement que A_1 est meilleur que A_2 pour n grand. Par contre, si $M_{A_1}(n)$ et $M_{A_2}(n)$ ont même ordre de grandeur asymptotique, il faut faire une analyse plus fine pour pouvoir comparer A_1 et A_2.

Pour comparer les ordres de grandeur asymptotiques des fonctions, on a l'habitude d'utiliser la notion suivante : étant donné deux fonctions f et g de $\mathbb{N}$ dans $\mathbb{R}^+$,

$$f = O(g) \text{ si et seulement si } \exists c \in \mathbb{R}^{+*},\ \exists n_0 \in \mathbb{N} \text{ tel que}$$
$$\forall n > n_0,\ f(n) \leq c.g(n)$$

(1) Il s'agira suivant les cas de $\text{Max}_A(n)$ ou $\text{Moy}_A(n)$ ou $\text{Min}_A(n)$.

(2) Il est clair que pour comparer des algorithmes, on utilise la même mesure de complexité (quelle qu'elle soit) sur chacun d'entre eux.

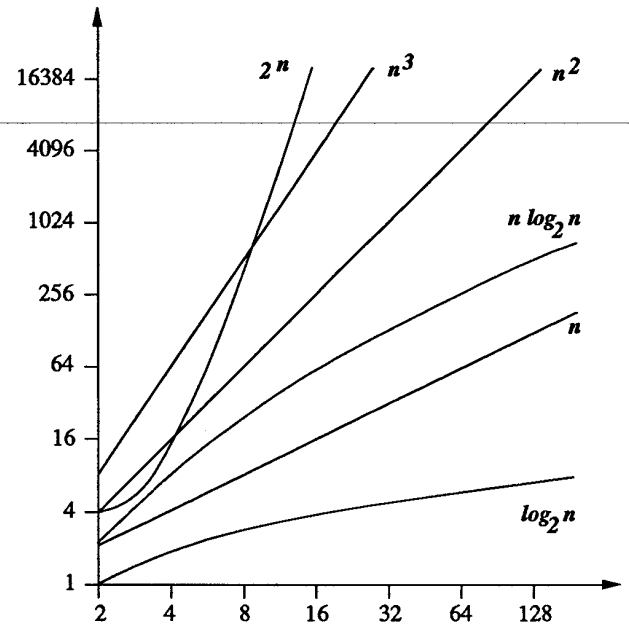

Figure 1. Rapidités de croissance comparées de certaines fonctions usuelles.

Ainsi $f = O(g)$ [3] veut dire que l'ordre de grandeur asymptotique de f est inférieur ou égal à celui de g, on dit aussi que f est ***dominée asymptotiquement*** par g ; par exemple $2n = O(n^2)$, mais aussi $2n = O(n)$ [4].

Cette notion qui donne un majorant de l'ordre de grandeur asymptotique de f, est très utile pour de nombreuses applications, mais elle n'est pas suffisante pour comparer entre elles les performances des différents algorithmes, car il faut connaître les ordres de grandeurs exacts, et non des majorants. Lorsque l'on dit que la complexité $M_A(n)$ d'un algorithme A est en $h(n)$, on veut dire que son ordre de grandeur asymptotique est *exactement* $h(n)$ (i.e. $h(n)$ est le plus petit majorant).

On est donc amené à introduire la notion suivante :

$$f = \Theta(g) \text{ si et seulement si } f = O(g) \text{ et } g = O(f)$$

c'est-à-dire qu'il existe deux réels positifs c et d, et un entier n_0 tels que :

$$\forall n > n_0,\ d \cdot g(n) \leq f(n) \leq c \cdot g(n)$$

On dit que f et g ont ***même ordre de grandeur asymptotique.***

[3] $f = O(g)$ se prononce « f égale grand O de g ». On peut aussi dire « f est en grand O de g ».

[4] Pour une étude plus poussée de cette notion on se reportera à l'annexe «Outils mathématiques».

La notion Θ [5] est plus précise que la notion O. Par exemple, $2n = \Theta(n)$, mais $2n$ n'est pas en $\Theta(n^2)$. Cependant, dans la plupart des ouvrages d'algorithmique, les résultats des analyses sont mis sous la forme $O(f(n))$, alors qu'un décompte précis des opérations fondamentales permet souvent de conclure que la complexité est exactement $\Theta(f(n))$. Par exemple, on dit souvent que le tri par tas (chapitre 15) est en $O(n \log n)$, alors qu'en fait il est en $\Theta(n \log n)$.

Soulignons un point fondamental : les définitions de O et Θ reposent sur l'existence de certaines constantes finies, mais il n'est rien précisé sur la *valeur* de ces constantes. Cela n'a pas d'importance pour obtenir des *résultats asymptotiques* lorsque les fonctions ont des *ordres de grandeur différents* :

par exemple si $f(n) = 2n$ et $g(n) = n^2$, alors $f(n) < g(n)$ pour $n > 2$

si $f(n) = 10000n$ et $g(n) = n^2$, alors $f(n) < g(n)$ pour $n > 10^4$.

Ainsi, si l'ordre de grandeur de f est plus petit que celui de g alors il existe un seuil à partir duquel la valeur de f est c fois plus petite que celle de g, mais on ne sait pas quel est ce seuil.

Par contre, si f et g ont même ordre de grandeur, il devient beaucoup plus difficile de les comparer : la détermination des constantes, et éventuellement des termes d'ordre inférieur nécessite en général des techniques mathématiques beaucoup plus complexes. Il faut bien être conscient de ce que l'obtention de résultats tels que : «l'algorithme A va deux fois plus vite que l'algorithme B sur un ordinateur standard», est en général très difficile, voire impossible.

La notion d'ordre de grandeur de la complexité des algorithmes a une grande importance pratique. Supposons que l'on dispose pour résoudre un problème donné de sept algorithmes dont les complexités dans le cas le pire ont respectivement pour ordre de grandeur 1 (c'est-à-dire une fonction constante, qui ne dépend pas de la taille des données), $\log_2 n$ (c'est-à-dire une fonction logarithmique), n (c'est-à-dire une fonction linéaire), $n.\log_2 n$, n^2 (c'est-à-dire une fonction polynômiale d'ordre 2), n^3 (c'est-à-dire une fonction polynômiale d'ordre 3), 2^n (c'est-à-dire une fonction exponentielle).

Le tableau A donne une estimation du temps d'exécution de chacun de ces algorithmes pour différentes tailles n des données du problème sur un ordinateur pouvant effectuer 10^6 opérations par seconde. Il montre bien que, plus la taille des données est grande, plus les écarts entre les différents temps d'exécution se creusent.

Le tableau B donne une estimation de la taille maximale des données que l'on peut traiter par chacun des algorithmes en un temps d'exécution fixé (et toujours sur un ordinateur effectuant 10^6 opérations par seconde).

[5] $f = \Theta(g)$ se prononce «f égale theta de g».

Tableau A. Temps d'exécution.

Complexité / Taille	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
$n = 10^2$	$\simeq 1\mu s$	6,6 μs	0,1 ms	0,6 ms	10 ms	1 s	4×10^{16} a
$n = 10^3$	$\simeq 1\mu s$	9,9 μs	1 ms	9,9 ms	1 s	16,6 mn	∞
$n = 10^4$	$\simeq 1\mu s$	13,3 μs	10 ms	0,1 s	100 s	11,5 j	∞
$n = 10^5$	$\simeq 1\mu s$	16,6 μs	0,1 s	1,6 s	2,7 h	31,7 a	∞
$n = 10^6$	$\simeq 1\mu s$	19,9 μs	1 s	19,9 s	11,5 j	$31,7 \times 10^3$ a	∞

N.B. On a noté «∞» lorsque la valeur dépasse 10^{100}.

Tableau B. Taille maximum des données.

Complexité / Temps calcul	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
1s	∞	∞	10^6	63×10^3	10^3	100	19
1 mn	∞	∞	6×10^7	28×10^5	77×10^2	390	25
1 h	∞	∞	36×10^8	13×10^7	60×10^3	15×10^2	31
1 jour	∞	∞	86×10^9	27×10^8	29×10^4	44×10^2	36

D'après ces deux tableaux, il est clair que certains algorithmes sont utilisables pour résoudre des problèmes sur ordinateurs, et que d'autres ne sont pas, ou peu utilisables.

Les algorithmes utilisables pour des données de grande taille sont ceux qui s'exécutent en temps :

- constant (c'est le cas de la complexité en moyenne de certaines méthodes de hachage présentées au chapitre 12);
- logarithmique (par exemple la recherche dichotomique présentée au chapitre 9 ou les opérations sur les arbres binaires de recherche présentées au chapitre 10);
- linéaire (par exemple, la recherche séquentielle, vue précédemment, d'un élément dans un tableau non trié);
- $n.\log n$ (par exemple les bons algorithmes de tri présentés au chapitre 15).

Les algorithmes qui prennent un temps polynômial, c'est-à-dire en $\Theta(n^k)$ avec $k > 0$, ne sont vraiment utilisables que pour $k < 2$. Lorsque $2 \leq k \leq 3$, on ne peut traiter que des problèmes de taille moyenne, et lorsque k dépasse 3 on ne peut traiter que des petits problèmes.

Les algorithmes en temps exponentiel, c'est-à-dire en $\Theta(2^n)$ par exemple, sont à peu près inutilisables, sauf pour des problèmes de très petite taille. Ce sont de tels algorithmes que l'on a qualifiés d'inefficaces dans l'introduction.

Tableau C. Evolutions mutuelles du temps et de la taille des données.

Complexité	1	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
Evolution du temps quand la taille est multipliée par 10	t	t+3,32	$10\times t$	$(10+\varepsilon)\times t$	$100\times t$	$1000\times t$	t^{10}
Evolution de la taille quand le temps est multiplié par 10	∞	n^{10}	$10\times n$	$(10-\varepsilon)\times n$	$3{,}16\times n$	$2{,}15\times n$	n+3,32

Le tableau C, enfin, montre comment la taille des données et le temps d'exécution varient en fonction l'un de l'autre. On voit en particulier que si l'on multiplie par 10 la vitesse de calcul de l'ordinateur, on ne modifie quasiment pas la taille maximale des données que l'on peut traiter avec un algorithme exponentiel, alors que l'on multiplie évidemment par 10 la taille des données traitables par un algorithme linéaire. Il est donc toujours d'actualité de rechercher des algorithmes efficaces, même si les progrès technologiques accroissent les performances du matériel !

4. Optimalité

On vient de voir comment comparer des algorithmes. Supposons maintenant que l'on dispose d'un algorithme A pour résoudre un problème donné, il est alors naturel de se demander si l'on peut trouver un algorithme «meilleur» que A; il est donc intéressant de connaître la complexité du meilleur algorithme possible pour traiter un problème.

Soit un problème P, on considère la classe C de tous les algorithmes résolvant P, qui utilisent des opérations d'un certain type, et dont les données sont organisées d'une certaine manière (notons que l'on ne connaît pas nécessairement tous les algorithmes de la classe). On se donne également une mesure de complexité, le nombre d'opérations fondamentales (évalué soit dans le pire des cas, soit en moyenne).

On définit la *complexité optimale* de la classe C, comme la borne inférieure des complexités des algorithmes de la classe. Un algorithme A de C est dit optimal si sa complexité est égale à la complexité optimale de C, que l'on note $M_{\text{opt}}(n)$. Par conséquent, un algorithme A de la classe C est optimal, s'il n'existe pas d'algorithme B dans C qui résolve le problème P en moins d'opérations que A.

Parfois, on ne peut pas déterminer $M_{\text{opt}}(n)$ avec précision, mais on peut connaître *l'ordre de grandeur* de la complexité optimale de la classe. Dans ce cas, un

algorithme A de la classe C est dit optimal si sa complexité est d'ordre de grandeur inférieur ou égal à la complexité de tout algorithme de classe C :

$$\forall B \in C,\ M_A = O(M_B)$$

L'ordre de grandeur de $M_{\text{opt}}(n)$ est alors $\Theta(M_A)$. Il est clair que, dans ce cas, plusieurs algorithmes de même ordre de complexité peuvent être optimaux.

Les techniques de démonstration d'optimalité dépendent énormément du problème et de la classe d'algorithmes étudiés et il n'existe pas de méthode générale pour établir des résultats d'optimalité. Dans ce livre, on en établit plusieurs, pour les problèmes suivants : recherche des deux plus grands éléments d'un tableau, recherche dichotomique, tri par comparaisons. Les techniques utilisées sont présentées dans le chapitre 3.

Il faut savoir que beaucoup de problèmes d'optimalité sont difficiles et qu'un grand nombre n'est pas encore résolu. Prenons l'exemple du produit de deux matrices, réalisé par des algorithmes utilisant les opérations arithmétiques classiques ($+$, $-$, $*$, $/$). On mesure la complexité en prenant la multiplication comme opération fondamentale. On a vu que l'algorithme classique de multiplication de matrices utilise n^3 multiplications; d'autre part, il est établi que la résolution du problème de la multiplication de deux matrices $n \times n$ nécessite au moins n^2 multiplications. Cependant, on ne connaît pas d'algorithme résolvant le problème avec seulement n^2 multiplications, et on ne sait pas s'il en existe. Déterminer la complexité du meilleur algorithme possible pour ce problème est actuellement un problème ouvert.