Chapitre 6
Notion de complexité algorithmique

1. Introduction

Déterminer la *complexité* [1] d'un algorithme, c'est évaluer les ressources nécessaires à son exécution (essentiellement la quantité de mémoire requise) et le temps de calcul à prévoir. Ces deux notions dépendent de nombreux paramètres matériels qui sortent du domaine de l'algorithmique : nous ne pouvons attribuer une valeur absolue ni à la quantité de mémoire requise ni au temps d'exécution d'un algorithme donné. En revanche, il est souvent possible d'évaluer l'*ordre de grandeur* de ces deux quantités de manière à identifier l'algorithme le plus efficace au sein d'un ensemble d'algorithmes résolvant le même problème.

Prenons un exemple concret : la détermination du nombre de diviseurs d'un entier naturel n. Une première solution consiste à essayer si chacun des entiers compris entre 1 et n est un diviseur de n. Ceci conduit à définir la fonction `diviseurs1` de la figure 1.

```
def diviseurs1(n):
    d = 0
    k = 1
    while k <= n:
        if n % k == 0:
            d += 1
        k += 1
    return d
```

```
def diviseurs2(n):
    d = 0
    k = 1
    while k * k < n:
        if n % k == 0:
            d += 2
        k += 1
    if k * k == n:
        d += 1
    return d
```

FIGURE 1 – Deux fonctions calculant le nombre de diviseurs d'un entier n.

Mais on peut aussi se faire la réflexion suivante : si $d \in [\![1, n]\!]$ divise n alors $d' = \dfrac{n}{d}$ aussi ; de plus, $d \leqslant \sqrt{n} \iff d' \geqslant \sqrt{n}$. Autrement dit, il suffit de rechercher les diviseurs de n qui sont inférieurs ou égaux à $\sqrt{n}$ pour en connaitre le nombre total : c'est le principe utilisé par la fonction `diviseurs2` (on notera le traitement particulier réservé aux carrés parfaits).

Comment comparer ces deux versions ? Si on se focalise sur les deux boucles conditionnelles de ces algorithmes on constate que dans les deux cas on effectue deux additions, une division euclidienne et un test. Chacune de ces opérations est effectuée n fois dans le premier cas, $\sqrt{n}$ fois [2] dans le second. Nous ne connaissons pas le temps τ_1 nécessaire à la réalisation de ces différents calculs, mais on peut légitimement penser que le temps total d'execution de `diviseurs1` n'est pas éloigné de $\tau_1 n + \tau_2$, le temps τ_2 étant le temps requis par les autres opérations. De même, le temps requis par la fonction `diviseurs2` est de l'ordre de $\tau_1' \sqrt{n} + \tau_2'$.
Les temps τ_2 et τ_2' sont négligeable pour de grandes valeurs de n ; de plus les valeurs de τ_1 et τ_1' importent peu ; elle dépendent de conditions matérielles qui nous échappent. Nous n'allons retenir que le taux de croissance de chacune de ces deux fonctions : proportionnel à n pour la première (on dira plus loin qu'il s'agit d'un algorithme de coût *linéaire*), proportionnel à $\sqrt{n}$ pour la seconde. Autrement dit :

- multiplier n par 100 multiplie le temps d'exécution de `diviseurs1` par 100 ;
- multiplier n par 100 multiplie le temps d'exécution de `diviseurs2` par 10.

Ainsi, connaissant le temps d'exécution pour une valeur donnée de n il est possible d'évaluer l'ordre de grandeur du temps d'exécution pour de plus grandes valeurs.

1. On dit aussi le *coût*.
2. tres exactement $\lceil \sqrt{n} \rceil - 1$ fois.

2. Évaluation de la complexité algorithmique

2.1 Instructions élémentaires

Pour réaliser l'évaluation de la complexité algorithmique, il est nécessaire de préciser un modèle de la technologie employée ; en ce qui nous concerne, il s'agira d'une machine à processeur unique pour laquelle les instructions seront exécutées l'une après l'autre, sans opération simultanées. Il faudra aussi préciser les instructions élémentaires disponibles ainsi que leurs coûts. Ceci est particulièrement important lorsqu'on utilise un langage de programmation tel que Python pour illustrer un cours d'algorithmique car ce langage possède de nombreuses instructions de haut niveau qu'il serait irréaliste de considérer comme ayant un coût constant : il existe par exemple une fonction `sort` qui permet de trier un tableau en une instruction, mais il serait illusoire de croire que son temps d'exécution est indépendant de la taille du tableau.

La première étape consiste donc à préciser quelles sont les instructions élémentaires, c'est-à-dire celle qui seront considérées comme ayant un coût constant, indépendant de leurs paramètres. Parmi celles-ci figurent en général :

- les opérations arithmétiques (addition, soustraction, multiplication, division, modulo, partie entière, ...)
- la comparaisons de données (relation d'égalité, d'infériorité, ...)
- le transferts de données (lecture et écriture dans un emplacement mémoire)
- les instructions de contrôle (branchement conditionnel et inconditionnel, appel à une fonction auxiliaire, ...)

Attention : il est parfois nécessaire de préciser la portée de certaines de ces instructions. En arithmétique par exemple, il est impératif que les données représentant les nombres soient codées sur un nombre *fixe* de bits. C'est le cas en général des nombres flottants (le type *float*) et des entiers relatifs (le type *int*) représentés usuellement sur 64 bits [3], mais dans certains langages existe aussi un type *entier long* dans lequel les entiers ne sont pas limités en taille. C'est le cas en Python, où coexistaient jusqu'à la version 3.0 du langage une classe *int* et une classe *long*. Ces deux classes ont depuis fusionné, le passage du type *int* au type *long* étant désormais transparent pour l'utilisateur. *Cependant, dans un but de simplification nous considérerons désormais toute opération arithmétique comme étant de coût constant.*
Dans le cas des nombres entiers, l'exponentiation peut aussi être source de discussion : s'agit-t'il d'une opération de coût constant ? En général on répond à cette question par la négative : le calcul de n^k nécessite un nombre d'opérations élémentaires (essentiellement des multiplications) qui dépend de k. Cependant, certains processeurs possèdent une instruction permettant de décaler de k bits vers la gauche la représentation binaire d'un entier, autrement dit de calculer 2^k en coût constant.

Les comparaisons entre nombres (du moment que ceux-ci sont codés sur un nombre fixe de bits) seront aussi considérées comme des opérations à coût constant, de même que la comparaison entre deux caractères. En revanche, la comparaison entre deux chaînes de caractères ne pourra être considérée comme une opération élémentaire, même s'il est possible de la réaliser en une seule instruction Python. Il en sera de même des opérations d'affectation : lire ou modifier le contenu d'un case d'un tableau est une opération élémentaire, mais ce n'est plus le cas s'il s'agit de recopier tout ou partie d'un tableau dans un autre, même si la technique du *slicing* en Python permet de réaliser très simplement ce type d'opération.

2.2 Notations mathématiques

Une fois précisé la notion d'opération élémentaire, il convient de définir ce qu'on appelle la *taille* de l'entrée. Cette notion dépend du problème étudié : pour de nombreux problèmes, il peut s'agir du nombre d'éléments constituant les paramètres de l'algorithme (par exemple le nombre d'éléments du tableau dans le cas d'un algorithme de tri) ; dans le cas d'algorithmes de nature arithmétique (le calcul de n^k par exemple) il peut s'agir d'un entier passé en paramètre, voire du nombre de bits nécessaire à la représentation de ce dernier. Enfin, il peut être approprié de décrire la taille de l'entrée à l'aide de deux entiers (le nombre de sommets et le nombre d'arêtes dans le cas d'un algorithme portant sur les graphes).
Une fois la taille n de l'entrée définie, il reste à évaluer en fonction de celle-ci le nombre $f(n)$ d'opérations élémentaires requises par l'algorithme. Mais même s'il est parfois possible d'en déterminer le nombre exact, on se contentera le plus souvent d'en donner l'ordre de grandeur à l'aide des notations de Landau.

3. Voir le chapitre 4.

La notation la plus fréquemment utilisée est le « grand O » :

$$f(n) = \mathrm{O}(\alpha_n) \iff \exists \mathrm{B} > 0 \;\middle|\; f(n) \leqslant \mathrm{B}\alpha_n.$$

Cette notation indique que dans le pire des cas, la croissance de $f(n)$ ne dépassera pas celle de la suite (α_n). L'usage de cette notation exprime l'objectif qu'on se donne le plus souvent : déterminer le temps d'exécution dans le cas le plus défavorable. On notera qu'un usage abusif est souvent fait de cette notation, en sous-entendant qu'il existe des configurations de l'entrée pour lesquelles $f(n)$ est effectivement proportionnel à α_n. On dira par exemple que la complexité de la fonction `diviseurs2` est un $\mathrm{O}(\sqrt{n})$ mais jamais qu'elle est un $\mathrm{O}(n)$, même si mathématiquement cette assertion est vraie puisque $f(n) = \mathrm{O}(\sqrt{n}) \Longrightarrow f(n) = \mathrm{O}(n)$.

D'un usage beaucoup moins fréquent, la notation Ω exprime une minoration du meilleur des cas :

$$f(n) = \Omega(\alpha_n) \iff \exists \mathrm{A} > 0 \;\middle|\; \mathrm{A}\alpha_n \leqslant f(n).$$

L'expérience montre cependant que pour de nombreux algorithmes le cas « moyen » est beaucoup plus souvent proche du cas le plus défavorable que du cas le plus favorable. En outre, on souhaite en général avoir la certitude de voir s'exécuter un algorithme en un temps raisonnable, ce que ne peut exprimer cette notation.

Enfin, lorsque le pire et le meilleur des cas ont même ordre de grandeur, on utilise la notation Θ :

$$f(n) = \Theta(\alpha_n) \iff f(n) = \mathrm{O}(\alpha_n) \text{ et } f(n) = \Omega(\alpha_n) \iff \exists \mathrm{A}, \mathrm{B} > 0 \;\middle|\; \mathrm{A}\alpha_n \leqslant f(n) \leqslant \mathrm{B}\alpha_n.$$

Cette notation exprime le fait que quelle que soit le configuration de l'entrée, le temps d'exécution de l'algorithme sera *grosso-modo* proportionnel à α_n.

Ordre de grandeur et temps d'exécution

Nous l'avons dit, la détermination de la complexité algorithmique ne permet pas d'en déduire le temps d'exécution mais seulement de comparer entre eux deux algorithmes résolvant le même problème. Cependant, il importe de prendre conscience des différences d'échelle considérables qui existent entre les ordres de grandeurs usuels que l'on rencontre. En s'appuyant sur une base de 10^9 opérations par seconde, le tableau de la figure 2 est à cet égard significatif. Il indique en fonction de la taille n de l'entrée (10^2, 10^3, …) et du nombre d'opérations requis par un algorithme ($\log n$, n, …) le temps d'exécution de ce dernier.

	$\log n$	n	$n \log n$	n^2	n^3	2^n
10^2	7 ns	100 ns	0,7 µs	10 µs	1 ms	$4 \cdot 10^{13}$ années
10^3	10 ns	1 µs	10 µs	1 ms	1 s	10^{292} années
10^4	13 ns	10 µs	133 µs	100 ms	17 s	
10^5	17 ns	100 µs	2 ms	10 s	11,6 jours	
10^6	20 ns	1 ms	20 ms	17 mn	32 années	

Figure 2 – Temps nécessaire à l'exécution d'un algorithme en fonction de sa complexité.

La lecture de ce tableau est édifiante : on comprend que les algorithmes ayant une complexité supérieure à une complexité quadratique soient en général considérées comme inutilisables en pratique (sauf pour de petites voire très petites valeurs de n).

$\mathrm{O}(\log n)$	logarithmique
$\mathrm{O}(n)$	linéaire
$\mathrm{O}(n \log n)$	quasi-linéaire
$\mathrm{O}(n^2)$	quadratique
$\mathrm{O}(n^k) \quad (k \geqslant 2)$	polynomiale
$\mathrm{O}(k^n) \quad (k > 1)$	exponentielle

Figure 3 – Qualifications usuelles des complexités.

Exercice 1 Pour vous familiariser avec ces notions, évaluez pour chacune des fonctions suivantes le temps d'exécution en fonction de n :

```
def f1(n):
    x = 0
    for i in range(n):
        for j in range(n):
            x += 1
    return x
```

```
def f2(n):
    x = 0
    for i in range(n):
        for j in range(i):
            x += 1
    return x
```

```
def f3(n):
    x = 0
    for i in range(n):
        j = 0
        while j * j < i:
            x += 1
            j += 1
    return x
```

```
def f4(n):
    x, i = 0, n
    while i > 1:
        x += 1
        i //= 2
    return x
```

```
def f5(n):
    x, i = 0, n
    while i > 1:
        for j in range(n):
            x += 1
        i //= 2
    return x
```

```
def f6(n):
    x, i = 0, n
    while i > 1:
        for j in range(i):
            x += 1
        i //= 2
    return x
```

2.3 Différents types de complexité

Certains algorithmes ont un temps d'exécution qui dépend non seulement de la taille des données mais de ces données elles-mêmes. Dans ce cas on distingue plusieurs types de complexités :

- la complexité *dans le pire des cas* : c'est un majorant du temps d'exécution possible pour toutes les entrées possibles d'une même taille. On l'exprime en général à l'aide de la notation O.
- la complexité *dans le meilleur des cas* : c'est un minorant du temps d'exécution possible pour toutes les entrées possibles d'une même taille. On l'exprime en général à l'aide de la notation Ω. Cependant cette notion n'est que rarement utilisée car souvent peu pertinente au regard des complexités dans le pire des cas et en moyenne.
- la complexité *en moyenne* : c'est une évaluation du temps d'exécution moyen portant sur toutes les entrées possible d'une même taille supposées équiprobables.

La plupart du temps on se contentera d'analyser la complexité dans le pire des cas.

Exemple. Considérons les algorithmes de recherche dans une liste de longueur n ; nous en avons écrit deux dans le chapitre précédent (voir figure 4).

– L'algorithme de recherche séquentielle effectue dans le meilleur des cas une seule comparaison (lorsque le premier élément testé est l'élément recherché) et dans le pire des cas n comparaisons (par exemple lorsque l'élément recherché ne se trouve pas dans la liste). On dira que le coût dans le meilleur des cas est constant, ce qu'on traduira par un coût en $\Theta(1)$, et linéaire dans le pire des cas, c'est-à-dire une complexité en $\Theta(n)$.

Dans tous les cas la complexité est donc un $O(n)$.

– Lorsque la liste est triée, l'algorithme de recherche dichotomique effectue lui aussi une seule comparaison dans le meilleur des cas (lorsque l'élément recherché se trouve au milieu de la liste) et dans le pire des cas un nombre de comparaison proportionnel à $\log n$. En effet, si $C(n)$ désigne la complexité dans le pire des cas on dispose de la relation : $C(n) = C(n/2) + \Theta(1)$. Ce type de relation est souvent étudié dans le cas particulier des puissances de 2 en posant $u_p = C(2^p)$ car alors on dispose de la relation $u_p = u_{p-1} + \Theta(1)$ qui conduit à $u_p = \Theta(p)$, soit $C(n) = \Theta(\log n)$ lorsque $n = 2^p$.

Dans tous les cas, la complexité est donc un $O(\log n)$.

```python
def cherche(x, l):
    for y in l:
        if y == x:
            return True
    return False
```

```python
def cherche_dicho(x, l):
    i, j = 0, len(l)
    while i < j:
        k = (i + j) // 2
        if l[k] == x:
            return True
        elif l[k] > x:
            j = k
        else:
            i = k + 1
    return False
```

FIGURE 4 – Les algorithmes de recherche linéaire et de recherche dichotomique.

Pour évoquer la notion de complexité en moyenne, il est nécessaire de disposer d'une hypothèse sur la distribution des données. Dans le cas de l'algorithme de recherche séquentielle, nous allons supposer que les éléments du tableau sont des entiers distribués de façon équiprobable entre 1 et $k \in \mathbb{N}^*$ (au sens large).
Nous disposons donc de k^n tableaux différents ; parmi ceux-ci, $(k-1)^n$ ne contiennent pas l'élément que l'on cherche et dans ce cas l'algorithme procède exactement à n comparaisons.
Dans le cas contraire, l'entier recherché est dans le tableau, et sa première occurrence est dans la i^{e} case avec la probabilité $\dfrac{(k-1)^{i-1}}{k^i}$. L'algorithme réalise alors i comparaisons. La complexité moyenne est donc égale à :

$$\mathrm{C}(n) = \frac{(k-1)^n}{k^n} \times n + \sum_{i=1}^{n} \frac{(k-1)^{i-1}}{k^i} \times i.$$

À l'aide de la formule : $\displaystyle\sum_{i=1}^{n} i x^{i-1} = \frac{1 + (nx - n - 1)x^n}{(1-x)^2}$ (valable pour $x \neq 1$) cette expression se simplifie en :

$$\mathrm{C}(n) = n\left(1 - \frac{1}{k}\right)^n + k\left(1 - \left(1 + \frac{n}{k}\right)\left(1 - \frac{1}{k}\right)^n\right) = k\left(1 - \left(1 - \frac{1}{k}\right)^n\right).$$

Lorsque k est petit devant n nous avons $\mathrm{C}(n) \approx k$ et il est légitime de considérer que la complexité moyenne est constante ; lorsque n est petit devant k, $\mathrm{C}(n) \approx n$ et la complexité moyenne rejoint la complexité dans le pire des cas.

2.4 Complexité spatiale

De la même façon qu'on définit la complexité temporelle d'un algorithme pour évaluer ses performances en temps de calcul, on peut définir sa *complexité spatiale* pour évaluer sa consommation en espace mémoire. Le principe est le même sauf qu'ici on cherche à évaluer l'ordre de grandeur du volume en mémoire utilisé : il ne s'agit pas d'évaluer précisément combien d'octets sont consommés par un algorithme mais de préciser son taux de croissance en fonction de la taille n de l'entrée. Cependant, on notera que la complexité spatiale est bien moins que la complexité temporelle un frein à l'utilisation d'un algorithme : on dispose aujourd'hui le plus souvent d'une quantité pléthorique de mémoire vive, ce qui rend moins important la détermination de la complexité spatiale.