9

Complexité d'un algorithme

9.1 Compétences exigées

— Objectifs —

La capacité évaluée dans cette partie de la formation est :

- s'interroger sur l'efficacité algorithmique temporelle d'un algorithme.

9.2 Notion de complexité

Il existe souvent plusieurs façons de programmer un algorithme. Si le nombre d'opérations à effectuer est peu important et les données d'entrée de l'algorithme sont de faibles tailles, le choix de la solution importe peu. En revanche, lorsque le nombre d'opérations et la taille des données d'entrée deviennent importants, deux paramètres deviennent déterminants : le temps d'exécution et l'occupation mémoire.
On n'exige donc pas seulement d'un algorithme qu'il résolve un problème, on veut également qu'il soit efficace, c'est-à-dire :

- rapide (en termes de temps d'exécution),
- peu gourmand en ressources (espace de stockage, mémoire utilisée).

On a alors besoin d'outils qui nous permettront d'évaluer la qualité théorique des algorithmes proposés. Le coût de la mémoire étant aujourd'hui relativement faible, on cherche en général à améliorer la complexité en temps plutôt que la complexité en mémoire. Pour cette raison, on s'intéressera donc au temps d'exécution.

— Complexité —

La complexité d'un algorithme en temps donne le nombre d'opérations effectuées lors de l'exécution d'un programme.
On appelle C_i le coût en temps d'une opération i.

La complexité en mémoire donne le nombre d'emplacements mémoires occupés lors de l'exécution d'un programme.

La complexité permet de :

- classer les problèmes selon leur difficulté,
- classer les algorithmes selon leur efficacité,
- comparer les algorithmes correspondant à un problème donné sans avoir à les implémenter, c'est-à-dire à les traduire dans un langage particulier.

— Remarque —

Attention de ne pas confondre la complexité d'un algorithme et la complexité d'un problème :
la complexité d'un problème correspond à la complexité de l'algorithme le plus efficace résolvant ce problème.

9.3 Calcul de la complexité

9.3.1 Les opérations

L'efficacité d'un algorithme ne se mesure pas en secondes, par exemple, car cela impliquerait justement de les implémenter mais de plus, ces mesures ne seraient pas significatives car dépendantes de la machine utilisée.

On utilise donc des unités de temps abstraites correspondant au nombre d'opérations effectuées. Chaque opération (addition, multiplication, comparaison, incrémentation, affectation, ...) consomme alors une unité de temps. Le nombre d'opérations est égal à la somme de toutes les opérations.

Exemple du calcul de *factorielle (n)* :

```
 1: VARIABLES
 2: i, n, fact : int
 3: SORTIES
 4: AFFICHER factorielle
 5: DEBUT_ALGORITHME
 6: |  INITIALISATION
 7: |  fact ← 1                          Initialisation : 1
 8: |  POUR i ALLANT_DE 2 A n           Itérations : n − 1
 9: |  |  DEBUT_POUR
10: |  |  fact ← fact ∗ i               multiplication + affectation : 2
11: |  |  FIN_POUR
12: |  AFFICHER fact                  affichage : 1
13: FIN_ALGORITHME
```

Algorithme 34 : Complexité 1

Il y a aussi des opérations cachées :

- i est initialisé à 2 en début de boucle : 1 opération,
- pour chaque boucle, i est incrémenté, affecté, puis comparé à n : 3 opérations.

Il y a donc au total :
$1 + 1 + 5 \times (n - 1) + 1 = 5n - 2$ opérations.

9.3.2 Complexité dans les différents cas

Il est fréquent que la complexité en temps soit améliorée au prix d'une augmentation de la complexité en espace, et vice-versa. La complexité dépend notamment :

- de la puissance de la machine sur laquelle l'algorithme est exécuté,
- du langage et compilateur (ou interpréteur) utilisé pour coder l'algorithme,
- du style du programmeur.

On distingue la complexité dans le pire des cas, la complexité dans le meilleur des cas, et la complexité en moyenne. En effet, pour un même algorithme, suivant les données à manipuler, le résultat sera déterminé plus ou moins rapidement.

- La complexité au meilleur est le plus petit nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée. C'est une borne inférieure de la complexité de l'algorithme sur un jeu de données de taille n.

- La complexité au pire est le plus grand nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée. Comme il s'agit d'un maximum, l'algorithme finira donc toujours avant d'avoir effectué ce nombre maximum d'opérations.
  Cette complexité peut cependant ne pas refléter le comportement " usuel " de l'algorithme, le pire cas ne pouvant se produire que rarement.
- La complexité en moyenne est plus difficile à calculer. Il ne s'agit pas comme on pourrait le penser de la moyenne des complexités au mieux et au pire. Concrètement, on se donne la probabilité d'apparition de chacun des jeux de données.
  Cette complexité en moyenne reflète le comportement " général " de l'algorithme si les cas extrêmes sont rares ou si la complexité varie peu en fonction des données.
  La complexité en pratique sur un jeu de données particulier peut être nettement plus importante que la complexité en moyenne : dans ce cas la complexité en moyenne ne donnera pas une bonne indication du comportement de l'algorithme.

Généralement, on s'intéresse au cas le plus défavorable, à savoir, la complexité dans le pire des cas, notée $\mathcal{O}(f)$, car cela correspond à la performance de l'algorithme lorsqu'il prend le plus de temps : il ne prendra jamais plus de temps que ce qu'on a estimé.
La complexité en moyenne est également souvent utilisée.

9.3.3 Simplifications : Règles de calcul

Partons d'un algorithme effectuant $4n^3 + 5n^2 - 2n + 8$ opérations.
Les processeurs actuels effectuent plusieurs millions d'opérations par seconde : ainsi, qu'une affectation requière 2, ou bien 4 unités de temps, ne modifie pas fondamentalement les choses.
On remplace donc les constantes multiplicatives par des 1, ce qui donne : $n^3 + n^2 - 2n + 8$.
Un nombre constant d'instructions est négligeable par rapport à la croissance de la taille des données, ce qui permet d'annuler les constantes additives. Cela donne : $n^3 + n^2 - 2n$.
Pour de grandes valeurs de n, c'est bien sûr le terme de plus haut degré qui est prépondérant.
Il reste alors : n^3.

9.3.4 Classes de complexités

La notation $\mathcal{O}(f)$ décrit le comportement asymptotique d'une fonction, c'est à dire pour des grandes valeurs.
Les complexités d'algorithmes (dans le pire des cas, que l'on notera $\mathcal{O}$) les plus courantes sont :

Complexité	temps d'éxécution
$\mathcal{O}(1)$	temps constant
$\mathcal{O}(\log(n))$	logarithmique
$\mathcal{O}(n)$	linéaire
$\mathcal{O}(n * log(n))$	linéarithmique
$\mathcal{O}(n^p)$	polynomiales
$\mathcal{O}(p^n)$	exponentielle

TABLE 9.1 – Complexités d'algorithmes

Un algorithme logarithmique est considérablement plus efficace qu'un algorithme linéaire (lui-même plus efficace qu'un algorithme quadratique ou pire exponentiel). Cette différence devient fondamentale si la taille des données est importante : le résultat peut se compter en millisecondes pour une méthode et en heures pour une autre !

Avec 10^9 instructions par seconde, on aurait les temps d'exécution :

Complexité	temps d'exécution en fonction du nombre d'opérations					
n	5	10	15	20	100	1000
$\log n$	3.10^{-9} s	4.10^{-9} s	4.10^{-9} s	5.10^{-9} s	7.10^{-9} s	10^{-8} s
$2n$	10^{-8} s	2.10^{-8} s	3.10^{-8} s	4.10^{-8} s	2.10^{-7} s	2.10^{-6} s
$n \log n$	$1{,}2.10^{-8}$ s	3.10^{-8} s	6.10^{-8} s	10^{-7} s	7.10^{-7} s	10^{-5} s
n^2	$2{,}5.10^{-8}$ s	10^{-7} s	$2{,}3.10^{-7}$ s	4.10^{-7} s	10^{-5} s	10^{-3} s
n^5	3.10^{-6} s	10^{-4} s	$7{,}6.10^{-4}$ s	3.10^{-3} s	10 s	10^6 s 11 jours
2^n	$3{,}2.10^{-8}$ s	10^{-6} s	$3{,}3.10^{-5}$ s	10^{-3} s	$1{,}2.10^{21}$ s 4.10^{13} ans	10^{292} s 3.10^{284} ans
$n!$	$1{,}2.10^{-7}$ s	4.10^{-3} s	$1{,}4.10^{3}$ s 23 minutes	$2{,}4.10^{9}$ s 77 ans	10^{147} s 3.10^{141} ans	10^{500} s
n^n	3.10^{-6} s	10 s	$4{,}4.10^{8}$ s 13 ans	10^{17} s 3.10^{9} ans	10^{191} s 3.10^{183} ans	10^{300} s

TABLE 9.2 – Temps d'exécution d'algorithmes

9.4 Exemple : Exponentiation rapide

9.4.1 Méthode naïve

9.4.1.1 Algorithmique

Calculer a^n nécessite a priori $n-1$ multiplications selon l'algorithme suivant :

```
1:  VARIABLES
2:  a : float
3:  n : int
4:  ENTRÉES
5:  un réel a et une puissance n
6:  INITIALISATION
7:  Resultat ← a
8:  DEBUT_ALGORITHME
9:      POUR k ALLANT_DE 2 A n
10:         DEBUT_POUR
11:         Resultat ← Resultat × a
12:         FIN_POUR
13:     AFFICHER Resultat
14: FIN_ALGORITHME
```

Algorithme 35 : Exponentiation naïve

9.4.1.2 Avec *Python*

Programme *puiss-naive.py* :

```python
# -*- coding: utf8 -*-

def puissance_naive(a,n):
    res=a
    for k in range(1,n):
        res*=a
    return res

# print(puissance_naive(2,100000))
from timeit import Timer
duree = Timer("puissance_naive(2,1000000)",\
"from __main__ import puissance_naive")
print("durée = %2.3f" % (duree.timeit(1)))
```

9.4.2 Algorithme de Hörner

9.4.2.1 Conventions

Dans la suite, nous confondrons polynôme et fonction polynôme.

9.4.2.2 Principe

Prenons l'exemple de $P(x) = 3x^5 - 2x^4 + 7x^3 + 2x^2 + 5x - 3$. Le calcul classique nécessite 5 additions et 15 multiplications.

On peut faire pas mal d'économies de calcul en suivant le schéma suivant :

$$\begin{aligned} P(x) &= \underbrace{a_n x^n + \cdots + a_2 x^2 + a_1 x}_{\text{on met } x \text{ en facteur}} + a_0 \\ &= \left(\underbrace{a_n x^{n-1} + \cdots + a_2 x}_{\text{on met } x \text{ en facteur}} + a_1 \right) x + a_0 \\ &= \cdots \\ &= (\cdots(((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})x + \cdots)\, x + a_0 \end{aligned}$$

Ici cela donne $P(x) = (((((3x) - 2)x + 7)x + 2)x + 5)x - 3$ c'est-à-dire 5 multiplications et 5 additions. En fait il y a au maximum 2 fois le degré de P opérations (voire moins avec les zéros).

9.4.2.3 L'algorithme

On peut essayer de faire mieux.

Voyons une première méthode qui utilise l'algorithme de HÖRNER étudié précédemment et la décomposition en base 2 de l'exposant.

Par exemple, $11 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$. On peut donc associer à cette écriture un polynôme P tel que 11 soit égal à $P(2)$:

$$P(X) = 1 \times X^3 + 0 \times X^2 + 1 \times X^1 + 1 \times X^0 = ((X + 0)X + 1)X + 1$$

Donc

$$\begin{aligned} a^{P(2)} &= a^{1+2(1+2(2\times 1+0))} \\ &= a \cdot a^{2(1+2(2\times 1+0))} \\ &= a \cdot \left(a^{1+2(2\times 1+0)}\right)^2 = a^1 \cdot \left(a^1 \cdot \left(a^0 \cdot \left(a^1\right)^2\right)^2\right)^2 \\ &= \left(\left(\left(a^1\right)^2 a^0\right)^2 a^1\right)^2 a^1 \end{aligned}$$

On en déduit l'algorithme :

```
VARIABLES
a : float
n : int
ENTRÉES
un réel a et un entier n
INITIALISATION
C ← décomposition de n en base 2
res ← a
DEBUT_ALGORITHME
    POUR j ALLANT_DE 1 A taille de C − 1
        DEBUT_POUR
        res ← res × res
        SI le j-ème chiffre de la décomposition en base 2 est 1 ALORS
            DEBUT_SI
            res ← a × res
            FIN_SI
        AFFICHER res
        FIN_POUR
FIN_ALGORITHME
```

Algorithme 36 : Exponentiation « à la HÖRNER »

Pour calculer a^{11} nous n'avons donc plus à effectuer que 6 multiplications ou plutôt 5 si on ne compte pas la multiplication par 1.

9.4.2.4 Avec Python

Programme *puiss-horner.py* :

```python
# -*- coding: utf8 -*-

def base2(n):
    r=n%2
    q=n//2
    R=[r]
    while q>0 :
        r=q%2
        q=q//2
        R=[r]+R
    return R

def puissance_horner(a,n):
    C=base2(n)
    res=a
    for j in range(1,len(C)):
        res*=res
        if C[j]==1: res*=a
    return(res)

# print(puissance_horner(2,100000))
import timeit
duree = timeit.Timer("puissance_horner(2,1000000)",\
"from __main__ import puissance_horner")
print("durée = %.3e" % duree.timeit(1))
```

9.4.3 Variante sans utiliser l'écriture en base 2

9.4.3.1 Algorithme

Voyons comment procéder sur notre exemple habituel :

$$a^{11} = a \cdot a^{10} = a \cdot \left(a^5\right)^2 = a \cdot \left(a \cdot a^4\right)^2 = a \cdot \left(a \cdot \left(a^2\right)^2\right)^2$$

Ainsi, on réduit l'exposant k selon sa parité :

- si k est pair, on écrit $a^k = \left(a^{\frac{k}{2}}\right)^2$;
- sinon, $a^k = a \cdot \left(a^{\frac{k-1}{2}}\right)^2$

```
1: VARIABLES
2: a : float
3: n : int
4: ENTRÉES
5: un réel a et un entier n
6: INITIALISATION
7: res ← 1
8: puissance ← n
9: temp ← a
10: DEBUT_ALGORITHME
11:     TANT_QUE puissance non nulle FAIRE
12:         DEBUT_TANT_QUE
13:         SI puissance est impaire ALORS
14:             DEBUT_SI
15:             res ← temp × res
16:             puissance ← puissance − 1
17:             FIN_SI
18:         puissance ← puissance/2
19:         temp ← temp × temp
20:         FIN_TANT_QUE
21: FIN_ALGORITHME
```

Algorithme 37 : Exponentiation rapide sans utiliser la base 2

9.4.3.2 Avec *Python*

Programme *puiss-rapid.py* :

```python
# -*- coding: utf8 -*-

def expo_rapid(a,n):
    res=1
    Puissance=n
    temp=a
    while Puissance>0:
        if Puissance%2==1 :
            res*=temp
            Puissance+=-1
        Puissance=Puissance/2
        temp*=temp
    return(res)

from timeit import Timer
duree = Timer("expo_rapid(2,1000000)",\
"from __main__ import expo_rapid")
print("durée = %.3e" % duree.timeit(1))
```

9.4.4 Comparaison des performances

Grâce à la méthode `Timer` du module `timeit`, nous pouvons comparer les temps de calcul des différents algorithmes pour calculer $2^{1\,000\,000}$:

```python
from timeit import Timer
duree = Timer("puissance_naive(2,1000000)",\
"from __main__ import puissance_naive")
print("durée = %2.3f" % duree.timeit(1))
```

donne sur un ordinateur récent :

```
durée =  17.896
```

Avec *puiss_rapid* la durée d'exécution est de $4.540e-03$ secondes et avec *puiss_horner* $3.195e-03$ secondes !

En ajoutant un 0 à la puissance, la durée d'exécution de *puiss_naive* est extrêmement longue alors qu'avec *puiss_rapid* la durée d'exécution est de 0.0741 secondes et avec *puiss_horner* 0.03336 secondes !

Nous voyons ici que certains algorithmes sont plus efficaces que d'autres ...
Avec des algorithmes récursifs, la durée d'exécution peut même être encore diminuée : nous verrons cela plus tard ...

— Remarque —

Les écarts sont plus significatifs lorsque les nombres sont grands : si l'on teste avec des nombres petits, il est même probable que les durées d'exécutions se trouvent en ordre inverse.