

Chapitre ITC3

Complexité des algorithmes

Temps d'exécution d'un programme

La rapidité d'un programme est essentielle :

- dans un programme en temps réel, les calculs doivent être assez rapides pour suivre le rythme de l'affichage.
Dans le même ordre d'idées, si le calcul de la météo pour demain prend plus d'1 jour, c'est embêtant...
- un programme de simulation prendra un temps d'exécution qui dépendra de la précision voulue
- le programme de factorisation d'un grand nombre est très long, et la cryptographie RSA se base sur ce fait.

Espace mémoire d'un programme

Si un programme stocke trop de résultats intermédiaires, il ne peut plus se satisfaire de la mémoire vive et doit utiliser la mémoire de masse, qui est beaucoup moins performante.

Exemple de l'échange de 2 variables

Complexité
des
algorithmes

La complexité
en opérations

Études de la
complexité de
quelques
algorithmes

Les différentes
complexités

Méthode 1

1 $z=x$
2 $x=y$
3 $y=z$

Méthode 2

1 $x=y-x$
2 $y=y-x$
3 $x=y+x$

Comparaison des 2 méthodes

La première méthode utilise moins d'opérations, la seconde utilise moins de mémoire

Section 1

La complexité en opérations

Temps ou opérations ?

Le temps d'exécution d'un programme dépend de la machine où on l'exécute. On compte donc plutôt les opérations élémentaires effectuées :

- nombre d'affectations
- nombre d'additions, de soustractions
- nombre d'appels à des fonctions
- ...

Complexité exacte ou ordre de grandeur ?

Complexité
des
algorithmes

La complexité
en opérations

Études de la
complexité de
quelques
algorithmes

Les différentes
complexités

Loi de Moore

La loi de Moore indique que la puissance des ordinateurs est multipliée par 2 tous les 1 an et demi.

Conséquence

- Si un programme est deux fois plus rapide qu'un autre, c'est bien, mais le gain est faible : dans 1 an et demi, on aura gagné ce même facteur 2 ; ou bien on peut mettre 2 ordinateurs en parallèle.
- Par contre, l'évolution de la complexité en fonction d'un paramètre est importante. Si en doublant la précision d'un calcul, le temps de calcul est multiplié par 2, cela veut dire que dans 1 an et demi on aura 2 fois plus de précision ; mais s'il est multiplié par 50, il faudra attendre...

Paramètre n

Le plus souvent, il y a dans l'algorithme un paramètre important qu'on veut prendre le plus grand possible : le **paramètre de complexité**. On le notera n dans la suite.

Le but de l'étude de la complexité est d'établir le nombre de chaque opération élémentaire en fonction de n .

Exemple : Tri d'une liste

Le paramètre de complexité est la taille de la liste.

Domination et négligeabilité

Soient f et g deux fonctions d'une variable n strictement positives. On dit que :

- f est négligeable devant g (noté $f = o(g)$) si
$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$
- f est dominée par g (noté $f = O(g)$) si $\frac{f(n)}{g(n)}$ est bornée lorsque n tend vers l'infini
- f est de l'ordre de g (noté $f = \Theta(g)$) si f domine g et g domine f

Exemple : Domination

- $f : n \mapsto 2n$ est dominée par (et de l'ordre de) $g : n \mapsto n$
- $f : n \mapsto 2n$ est dominée par $g : n \mapsto n^2$
- $f : n \mapsto 3n^2 + 2n$ est dominée par (et de l'ordre de) $g : n \mapsto n^2$
- $f : n \mapsto \log(n)$ est dominée par $g : n \mapsto n$
- $f : n \mapsto n + \log(n)$ est dominée par (et de l'ordre de) $g : n \mapsto n$

On cherche à exprimer la complexité comme dominée par une fonction simple

Quelques cas usuels (non exhaustifs) :

- **complexité constante** en $O(1)$: la complexité ne dépend pas de la taille.
Exemple : échange de 2 variables
- **complexité sous-linéaire ou logarithmique** en $O(\log(n))$: la complexité augmente lentement.
Exemple : recherche par dichotomie
- **complexité linéaire** en $O(n)$: la complexité est proportionnelle à n .
Exemple : recherche du maximum dans une liste

On cherche à exprimer la complexité comme dominée par une fonction simple

- **complexité quasi-linéaire** en $O(n \log(n))$
Exemple : tri fusion
- **complexité quadratique** en $O(n^2)$: la complexité commence à augmenter très vite
Exemple : tri à bulles
- **complexité polynômiale** en $O(n^k)$
Exemple : pivot de Gauss en $O(n^3)$
- **complexité exponentielle** en $O(c^n)$: algorithme inutilisable

Temps de calcul nécessaire

Complexité
des
algorithmes

La complexité
en opérations

Études de la
complexité de
quelques
algorithmes

Les différentes
complexités

Ordre de grandeur du temps nécessaire à l'exécution d'un algorithme (source : Wikipédia)

n	5	10	20	50	250	1000	10000	1000000
$O(1)$	10ns	10ns	10ns	10ns	10ns	10ns	10ns	10ns
$O(\log(n))$	10ns	10ns	10ns	20ns	30ns	30ns	40ns	60ns
$O(n)$	50ns	100ns	200ns	500ns	2,5μs	10μs	100μs	10ms
$O(n \log(n))$	40ns	100ns	260ns	850ns	6μs	30μs	400μs	60ms
$O(n^2)$	250ns	1μs	4μs	25μs	625μs	10ms	1s	2,8h
$O(n^3)$	1,25μs	10μs	80μs	1,25ms	156ms	10s	2,7h	316ans
$O(2^n)$	320ns	10μs	10ms	130j	10^{59} ans	...	...	...
$O(n!)$	1,2μs	36ms	770ans	10^{48} ans	...	...	...	...

Section 2

Études de la complexité de quelques algorithmes

Algorithme : Calcul de la somme des n premiers entiers

Entrée : un entier n

Sortie : *somme*

$somme \leftarrow 0$

pour i **variant de** 1 à n **faire** :

$somme \leftarrow somme + i$

fin pour

Complexité en fonction de n

La boucle est parcourue n fois. On a donc :

- n additions
- $n + 1$ affectations

La complexité est donc en $O(n)$.

On veut maintenant obtenir la liste des sommes des entiers entre 1 et n

c'est-à-dire : $[1, 1+2, 1+2+3, \dots, 1+2+3+\dots+n]$

Algorithme : Première méthode

Entrée : un entier n

Sortie : *liste*

liste $\leftarrow$ liste vide

pour i **variant de** 1 à n **faire** :

somme $\leftarrow 0$

pour j **variant de** 1 à i **faire** :

somme $\leftarrow$ *somme* + j

fin pour

 ajouter *somme* à la fin de *liste*

fin pour

Étude de la complexité par rapport à n

La grande boucle est parcourue n fois, pour i allant de 1 à n . À chaque fois, on observe :

- 1 affectation au départ
- i additions et i affectations dans la boucle
- 1 insertion dans un tableau

On a donc au total :

- $1 + \sum_{i=1}^n (1 + i) = 1 + n + \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{3}{2}n + 1$ affectations, soit $O(n^2)$ affectations
- $\sum_{i=1}^n i = \frac{n(n+1)}{2}$ additions soit $O(n^2)$ additions
- n insertions dans une liste (soit $O(n)$).

On veut obtenir la liste des sommes des entiers entre 1 et n
c'est-à-dire : $[1, 1+2, 1+2+3, \dots, 1+2+3+\dots+n]$

Algorithme : Seconde méthode

Entrée : un entier n

Sortie : liste

liste ← liste vide

somme ← 0

pour i variant de 1 à n **faire** :

somme ← *somme* + i

 ajouter *somme* à la fin de *liste*

fin pour

Étude de la complexité par rapport à n

La boucle est parcourue n fois, pour i allant de 1 à n . On a donc

- $n + 2$ affectations soit $O(n)$
- n additions soit $O(n)$
- n insertions dans une liste soit $O(n)$

Notebook

La seconde méthode sera plus performantes pour de grands n .

Algorithme : Calcul normal de x^n

Entrée : un réel x , un entier positif n

Sortie : *resultat*

resultat $\leftarrow 1$

pour i variant de 1 à n faire

$\text{resultat} \leftarrow \text{resultat} * x$

fin pour

Étude de la complexité par rapport à n

C'est assez facile : il y a $n + 1$ affectations et n multiplications (opérations assez coûteuses en temps).

Principe (résumé) de l'exponentiation rapide

Si on veut calculer par exemple x^6 , on commence par décomposer 6 en base 2, ou (de manière équivalente) en somme de puissances de 2 : on trouve $6 = 2 + 4$ donc $x^6 = x^2 \cdot x^4$. Il suffit donc de calculer x^2 , puis $x^4 = (x^2)^2$, et de les multiplier, ce qui donne 3 multiplications au lieu de 6.

Algorithme : Exponentiation rapide de x^n

Entrée : un réel x , un entier positif n

Sortie : *resultat*

```
puissance, resultat  $\leftarrow$   $x, 1$ 
tant que  $n > 0$  faire
    si  $n$  est impair faire
        resultat  $\leftarrow$  resultat * puissance
    fin si
    puissance  $\leftarrow$  puissance * puissance
     $n \leftarrow n/2$  (division entière)
fin tant que
```

Algorithme : Exponentiation rapide de x^n

Entrée : un réel x , un entier positif n

Sortie : *resultat*

```
puissance, resultat  $\leftarrow$   $x, 1$ 
tant que  $n > 0$  faire
    si  $n$  est impair faire
         $resultat \leftarrow resultat * puissance$ 
    fin si
     $puissance \leftarrow puissance * puissance$ 
     $n \leftarrow n/2$  (division entière)
fin tant que
```

Étude de la complexité par rapport à n Notebook

À chaque tour de boucle, n est divisé par 2 donc le nombre de tours de boucles sera majoré par $\log_2(n)$. À chaque tour il y a 1 ou 2 multiplications, donc on aura au maximum $2 \log_2(n)$ multiplications, d'où une complexité en $O(\log(n))$ (idem pour les divisions).

Principe de la dichotomie

Soit f une fonction continue définie sur un intervalle $[a : b]$ et telle que $f(a)$ et $f(b)$ soient de signes opposés ; alors f s'annule sur cet intervalle. Soit x_0 la racine ; on veut déterminer sa valeur à ε près.

L'idée de la dichotomie est de partager l'intervalle en deux. Soit $m = \frac{a+b}{2}$. Si $f(m)$ est du signe de $f(a)$, alors la racine est entre m et b , sinon elle est entre a et m .

Le paramètre de complexité est ici ε , ou plutôt prenons $p = \frac{1}{\varepsilon}$ car le but est d'avoir ε le plus petit possible, et dans ce cas p tend vers l'infini, ce qui est le comportement classique d'un paramètre de complexité.

Recherche d'une racine par dichotomie

```
1 def racine(f,a,b,epsilon):  
2     g,d=a,b  
3     while (d-g)>epsilon:  
4         m=(d+g)/2  
5         if f(m)*f(g)>0:  
6             g=m  
7         else:  
8             d=m  
9     return (g,d)
```

Étude de la complexité par rapport à $p = \frac{1}{\varepsilon}$ Notebook

On voit facilement que $d - g$ est divisé par 2 à chaque étape. La boucle tournera donc i fois avec i le plus petit entier tel que $\frac{b-a}{2^i} < \varepsilon$ soit $i \approx \log_2 \left(\frac{b-a}{\varepsilon} \right)$. Le nombre de comparaisons et d'affectations sera donc en $O(\log(\varepsilon))$ ou $O(\log(p))$. Mais l'opération la plus coûteuse ici est certainement l'évaluation de la fonction f en g et en m ; il y en a 2 par boucle, donc on est encore en $O(\log(p))$.

Section 3

Les différentes complexités

Recherche d'un élément dans une liste

Complexité
des
algorithmes

La complexité
en opérations

Études de la
complexité de
quelques
algorithmes

Les différentes
complexités

Algorithme : Recherche si un élément est présent

Entrée : une liste L , un élément a

Sortie : Vrai ou Faux

```
pour  $i$  allant de 0 à  $\text{longueur}(L) - 1$  faire  
    si  $L[i] = a$  faire  
        retourner Vrai  
    fin si  
fin pour  
retourner Faux
```

Ce algorithme effectue :

Recherche d'un élément dans une liste

Complexité
des
algorithmes

La complexité
en opérations

Études de la
complexité de
quelques
algorithmes

Les différentes
complexités

Algorithme : Recherche si un élément est présent

Entrée : une liste L , un élément a

Sortie : Vrai ou Faux

```
pour  $i$  allant de 0 à  $\text{longueur}(L) - 1$  faire  
    si  $L[i] = a$  faire  
        retourner Vrai  
    fin si  
fin pour  
retourner Faux
```

Ce algorithme effectue :

- dans le meilleur cas, 1 seule comparaison

Recherche d'un élément dans une liste

Complexité
des
algorithmes

La complexité
en opérations

Études de la
complexité de
quelques
algorithmes

Les différentes
complexités

Algorithme : Recherche si un élément est présent

Entrée : une liste L , un élément a

Sortie : Vrai ou Faux

```
pour  $i$  allant de 0 à  $\text{longueur}(L) - 1$  faire
    si  $L[i] = a$  faire
        retourner Vrai
    fin si
fin pour
retourner Faux
```

Ce algorithme effectue :

- dans le meilleur cas, 1 seule comparaison
- dans le pire des cas, $\text{longueur}(L)$ comparaisons

Recherche d'un élément dans une liste

Complexité
des
algorithmes

La complexité
en opérations

Études de la
complexité de
quelques
algorithmes

Les différentes
complexités

Algorithme : Recherche si un élément est présent

Entrée : une liste L , un élément a

Sortie : Vrai ou Faux

```
pour  $i$  allant de 0 à  $\text{longueur}(L) - 1$  faire  
    si  $L[i] = a$  faire  
        retourner Vrai  
    fin si  
fin pour  
retourner Faux
```

Ce algorithme effectue :

- dans le meilleur cas, 1 seule comparaison
- dans le pire des cas, $\text{longueur}(L)$ comparaisons
- en moyenne, $\frac{1}{2}\text{longueur}(L)$ comparaisons

Recherche du maximum d'une liste

Complexité
des
algorithmes

La complexité
en opérations

Études de la
complexité de
quelques
algorithmes

Les différentes
complexités

Recherche du maximum

```
1 def maxi(L):  
2     max=L[0]  
3     for i in range(len(L)):  
4         if L[i]>max:  
5             max=L[i]  
6     return max
```

Complexité en fonction de $n = \text{len}(L)$:

- meilleur des cas : n comparaisons, 1 affectation
- pire des cas : n comparaisons, n affectations
- en moyenne : n comparaisons, $O(\log(n))$ affectations (plus difficile à montrer)