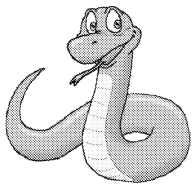
IPT : Cours 8



Etude de la complexité temporelle

– 2h –

MPSI : Prytanée National Militaire

Pascal Delahaye

12 janvier 2017

Le temps d'exécution d'un algorithme est un des éléments fondammentaux qui permet d'évaluer sa qualité.
Voici deux algorithmes de calcul de a^{2^n}.

Python

```
def calc1(a,n)                      def calc2(a,n)
    res = 1                             res = a
    for i in range(1,2**n+1):           for i in range(1,n+1):
        res = res*a                         res = res**2
    print(res)                          print(res)
```

1. Calculer le nombre de multiplications $c_1(n)$ et $c_2(n)$ effectués dans chacun des 2 algorithmes.
2. Lequel selon vous est le plus rapide ?

1 Cadre théorique

Bien entendu, le temps de calcul d'un algorithme dépend aussi de la puissance de l'ordinateur utilisé. Cependant, ce temps est souvent proportionnel à une opération élémentaire qui est exécutée de façon répétitive par l'algorithme. Dans l'exemple précédent, nous pouvons ainsi considérer que le temps d'exécution de nos deux algorithmes est proportionnel au nombre de multiplications effectuées. Compter le nombre de multiplications donne donc une idée assez précise de la différence de temps d'exécution de chacun des algorithmes.

Le temps d'exécution dépend aussi de la taille des arguments. Dans notre exemple précédent, plus n est grand et plus le temps d'exécution est important.

L'étude de la complexité temporelle s'effectue donc ainsi :

1. Taille des arguments :
   On commence par définir le nombre entier n qui donne une idée de la taille des arguments.

2. Opération significative :
   Puis, on doit choisir une opération dont la répétition doit donner une idée du temps d'exécution de l'algorithme. Pour que ce temps soit proportionnel au nombre d'opérations effectuées, on veillera dans la mesure du possible à ce que le temps d'exécution de cette *opération significative* puisse être considérée indépendant du moment où celle-ci est exécutée.

3. Calcul de la complexité :
   Il s'agit alors de calculer $c(n)$ qui représente le nombre d'opérations significatives nécessaires pour effectuer l'algorithme.

Lorsque la valeur de n est petite, les algorithmes s'effectuent souvent en quelques millisecondes. Il nous importe peu alors de savoir qu'un programme soit 10 fois plus lent qu'un autre. En revanche lorsque n devient grand, les différences de temps d'exécution peuvent s'évérer très importantes pour l'utilisateur. Pensez par exemple à un programme de recherche sur internet...

Par conséquent, nous nous intéresserons à $c(n)$ uniquement pour des valeurs importantes de n. Ceci nous permettra en particulier de négliger certaines valeurs qui pourraient compliquer le calcul. Plutôt que la valeur exacte de $c(n)$, nous chercherons plutôt à déterminer $c(n)$ sous la forme d'un "grand O" ou d'un équivalent.

Définition 1 : **"Grand O" et "Equivalent"**
Soit (α_n) une suite strictement positive pour de grande valeurs de n.
$c(n)$ étant une valeur positive, nous dirons que :

- $c(n) = O(\alpha_n)$ lorsque la suite de terme général $\dfrac{c(n)}{\alpha_n}$ est majorée.
- $c(n) \sim \alpha_n$ lorsque la suite de terme général $\dfrac{c(n)}{\alpha_n}$ tend vers 1 en $+\infty$.

On distingue alors différents ordres de complexité temporelle :

Type	Définition
Complexité constante	$c(n) = O(1)$
Complexité logarithmique	$c(n) = O(\ln n)$ ou $c(n) = O(\ln^k n)$ avec $k > 1$
Complexité quasi-linéaire	$c(n) = O(n \ln n)$
Complexité linéaire	$c(n) = O(n)$
Complexité polynomiale	$c(n) = O(n^k)$ avec $k > 1$
Complexité quadratique	$c(n) = O(n^2)$
Complexité exponentielle	$c(n) = O(a^n)$ avec $a > 1$
Complexité hyperexponentielle	$c(n)/a^n \to +\infty, \forall a > 1$ ex : $C(n) = n!$

`Exemple 1.` La bibiothèque NUMPY de Python propose la fonction `sort()` pour trier une liste donnée.
L'aide nous fourni alors les informations suivantes :

> The various sorting algorithms are characterized by their average speed, worst case performance, work space size, and whether they are stable. A stable sort keeps items with the same key in the same relative order. The three available algorithms have the following properties:

kind	speed	worst case	work space	stable
'quicksort'	1	O(n^2)	0	no
'mergesort'	2	O(n*log(n))	~n/2	yes
'heapsort'	3	O(n*log(n))	0	no

Combien distingue-t-on d'algorithmes de tri possibles ?
Quelles sont leurs complexités respectives dans le pire des cas ?

Remarque 1. Dans certains algorithmes, la valeur de $c(n)$ dépend de la donnée de taille n traitée.
On définit alors 3 types de complexité :
- La complexité dans le meilleur des cas
- La complexité moyenne
- La complexité dans le pire des cas

On peut par exemple illustrer cette situation par l'algorithme du tri à bulle.

2 Exemples d'étude de complexité

Exercice : 1
Algorithmes de complexité constante

1. Déterminer la complexité de l'algorithme permettant de choisir un nombre au hasard dans une liste.

Exercice : 2
Algorithmes de complexité logarithmique

1. Déterminer la complexité de l'algorithme permettant de déterminer le nombre de chiffres d'un entier naturel à l'aide de divisions par 10 successives.

2. Evaluer la complexité de l'algorithme de recherche dichotomique de la solution d'une équation $f(x) = 0$.

Exercice : 3
Algorithmes de complexité linéaire

1. Proposez un algorithme de recherche d'une valeur e dans une liste L de longueur n.
   Dans le pire des cas, quelle est la complexité de votre algorithme ?

Exercice : 4
Algorithmes de complexité quasi-linéaire

1. Les tris "rapides" (quicksort) et "fusion" (mergesort) ont une complexité quasi-linéaire (vus en 2ème année)

Exercice : 5
Algorithmes de complexité quadratique

1. Présentation de l'algorithme de tri par sélection et étude de sa complexité.

Exercice : 6

Algorithmes de complexité polynomiale

1. Etudier la complexité de l'algorithme de résolution de systèmes linéaires par la méthode de Gauss.

Exercice : 7

Algorithmes de complexité exponentielle

1. Etudier la complexité du calcul du nème terme de la suite de fibonnacci à l'aide d'un algorithme récursif

Exercice : 8

(∗) Etudier en fonction des valeurs a et b, la complexité de l'algorithme de division euclidienne suivant :

Python

```
def division(a,b):
    r = a
    q = 0
    while r >= b:
         r = r - b
         q = q+1
    print((q,r))
```

Exercice : 9

L'algorithme d'Euclide

(∗∗) Le but de cet exercice est d'évaluer la complexité de la fonction récursive donnant la valeur du PGCD de a et b (avec $a \leq b$) par l'algorithme d'Euclide. On considère le nombre de division euclidienne comme indicateur de la complexité.
On considére a et b strictement positifs avec $b \leq a$ et on prend $n = a$ comme mesure de la taille des données.

1. Montrer que l'on a toujours $a \mod b \leq \frac{a}{2}$. (on traitera deux cas !)

2. Montrer que $c(a) = 2 + c(a \mod b)$.
   En faisant l'hypothèse réaliste que la fonction "c" est croissante, on en déduit que $c(n) \leq\leq c(\frac{n}{2}) + 2$.
   En considérant que c vérifie $c(n) = c(\frac{n}{2})+2$*, on obtient alors une estimation majorée de la complexité souhaitée.*

3. En déduire que $c(n) = O(\log_2 n)$.