Université d'Aix-Marseille - Master Informatique 1^{ere} année

UE Complexité - TD 1 - Rappels d'algorithmique et d'analyse de la complexité

Préambule : cette première série d'exercices porte sur des notions relatives à l'algorithmique et à l'analyse de la complexité qui sont normalement maîtrisées. Lorsque vous avez à écrire des algorithmes, un pseudo-langage algorithmique convient, mais parfois, pour l'analyse de leur complexité, certaines précisions doivent être apportées au code de sorte à s'assurer que l'évaluation de la complexité est correcte (dans certains langages de programmation, des opérations complexes et coûteuses en temps peuvent être masquées par la simplicité de leur expression...). Aussi, une façon efficace de conserver une certaine rigueur peut consister à écrire les algorithmes dans un langage proche du langage C par exemple, même si cela ne sera pas exigé ici.

Exercice 1. Analyse de la complexité : notations O et Θ.

Question 1. Majoration. Démontrez les propriétés suivantes

- $g(n) = 6n + 12 \in \mathrm{O}(n)$
- $g(n) = 3n \in \mathrm{O}(n^2)$
- $g(n) = 10^{1000000}n \in \mathrm{O}(n)$
- $g(n) = 5n^2 + 10n \in \mathrm{O}(n^2)$
- $g(n) = n^3 + 1000n^2 + n + 8 \notin \mathrm{O}(n^2)$

Question 2. Ordre exact. Démontrez les propriétés suivantes

- $g(n) = 5n^2 + 10n \in \Theta(n^2)$
- $g(n) = n^2 + 1000000n \in \Theta(n^2)$
- $g(n) = 4n^2 + n.\log(n) \in \Theta(n^2)$
- $g(n) = 3n + 8 \notin \Theta(n^2)$

Exercice 2. Calcul de puissances

Question 1. Donnez un algorithme itératif (sans récursivité donc) qui étant donnés deux entiers strictement positifs x et p, réalise le calcul de la valeur de x^p en un temps linéaire en la valeur de p (le recours à des primitives de calcul d'exponentielles est bien sûr interdit).

Question 2. Peut-on affirmer que l'algorithme proposé dans la question 1 est de complexité linéaire ? Justifiez votre réponse.

Question 3. On suppose maintenant que l'entier p est une puissance de 2, c'est-à-dire qu'il existe un entier k tel que $p = 2^k$. On remarque que $x^p = x^{p/2}.x^{p/2}$. Sur la base de cette observation, on peut déduire une nouvelle méthode pour calculer x^p. Il suffit de calculer $x^{p/2}$, puis de calculer son carré. Par exemple, si $x = 5$ et $p = 8$, on a $x^p = 5^8 = 5^4.5^4 = (5^4)^2 = (5^2.5^2)^2 = ((5^2)^2)^2 = ((5^1.5^1)^2)^2 = (((5^1)^2)^2)^2 = (((5)^2)^2)^2 = ((25)^2)^2 = (625)^2 = 390625$. On a ainsi eu uniquement 3 multiplications à calculer à la place de 8 (ou 7) selon la méthode de base de l'exercice 1. En exploitant cette approche, donnez un algorithme itératif réalisant le calcul de x^p. Évaluez sa complexité.

Exercice 3. Produits et puissances de matrices carrées

On s'intéresse au produit de matrices carrées (d'entiers) d'ordre n. On suppose que si A représente une matrice carrée $n \times n$, alors la notation $A[i,j]$ représente l'élément (le coefficient) de A situé à l'intersection de la i^{eme} ligne et de la j^{eme} colonne de la matrice. Selon cette hypothèse, le produit de deux matrices A et B est une matrice carrée P d'ordre n vérifiant, $\forall i, 1 \leq i \leq n, \forall j, 1 \leq j \leq n$, $P[i,j] = \Sigma_{k=1}^{n}\ A[i,k].B[k,j]$.

Question 1. En supposant que les matrices sont représentées par des tableaux d'entiers à 2 dimensions, donnez un algorithme qui réalise le calcul du produit de 2 matrices carrées d'ordre n. Évaluez sa complexité.

Question 2. Peut-on affimer que l'algorithme proposé dans la question 1 est de complexité linéaire, quadratique, cubique? Justifiez votre réponse.

Question 3. Donnez un algorithme itératif qui, étant donnés une matrice carrées A d'ordre n et un entier p strictement positif, réalise le calcul de A^p. Évaluez sa complexité.

Question 4. On suppose maintenant que l'entier p est une puissance de 2, c'est-à-dire qu'il existe un entier k tel que $p = 2^k$. En faisant les mêmes constats que dans l'exercice 2 sur les calculs de puissance, et du fait de l'associativité du produit de matrices, on peut remarquer que $A^p = A^{p/2}.A^{p/2}$, et par conséquent, pour calculer A^p, il suffit de calculer $A^{p/2}$, puis de calculer son carré $(A^{p/2})^2$. Donnez un algorithme itératif qui réalise le calcul de A^p selon cette approche. Évaluez sa complexité.

Exercice 4. Un petit retour sur le tri par insertion

Il existe plusieurs algorithme de tris de tableaux opérant par comparaison. Nous allons revenir ici sur l'un d'eux qui n'est pas réputé pour être le plus efficace, mais son étude est parfois instructive. Il s'agit du *tri par insertion séquentielle* dont nous rappelons ci-dessous une implémentation (très discutable sur le plan méthodologique mais ce n'est pas l'objet ici) en langage C :

```
En entree : un tableau t de n entiers

int i,j,p,x;

...

/* tri du tableau t par insertion sequentielle */
for (i=1; i < n; i=i+1)
{
   /* calcul par recherche sequentielle de la position p d'insertion de t[i] */
   p = 0;
   while ( t[p] < t[i] ) p = p+1;

   /* reorganisation de la section du tableau t allant de t[p] a t[i] */
   x = t[i];
   for(j=i-1; j >= p; j=j-1) t[j+1] = t[j];
   t[p] = x;
}
```

Question 1. Évaluez la complexité de cet algorithme en ne dénombrant que les tests de comparaisons entre éléments du tableau (le test `t[p]<t[i]`).

Question 2. Évaluez la complexité de cet algorithme en ne dénombrant que les transferts d'éléments du tableau (de la forme `x=t[i]` ou `t[j+1]=t[j]` ou `t[p]=x`).

Question 3. Évaluez la complexité de cet algorithme en tenant compte de toutes les opérations comme dans les exercices précédents.

Exercice 5. Pour aller plus loin, les nombres de Fibonacci...

L'objectif ici est d'étudier différents algorithmes de calcul du n^{eme} nombre de Fibonacci dont nous rappelons la relation de récurrence le définissant : $f_0 = 0$ et $f_1 = 1$, et pour tout entier $n > 1$, on a $f_n = f_{n-1} + f_{n-2}$. Il faut noter que parfois, cette suite est définie avec $f_0 = 1$ mais cela ne change rien au propos. Il s'agit ici de concevoir trois algorithmes en d'en étudier la complexité.

Question 1. Donnez un algorithme récursif qui réalise le calcul de f_n pour tout entier n. Cet algorithme reprend bien évidemment le schéma suivant exprimé en français :

f_n a pour valeur :
- 0 si n vaut zéro, 1 si n vaut un (soit n dans chaque cas),
- et $f_{n-1} + f_{n-2}$ pour les valeurs $n > 1$.

Pour en évaluer la complexité, posez une équation de récurrence. Attention, on ne vous demande pas de la résoudre, mais il faut savoir que sa résolution permet de constater que la complexité de l'algorithme décrit ci-dessus est $\Theta(\Phi^n)$ où Φ est le fameux Nombre d'Or, dont la valeur est $\Phi = (1 + \sqrt{5})/2$, soit approximativement 1,62. En d'autres termes, cette approche va se limiter au calcul de f_n sur un ordinateur, fut-il très puisssant, voire "super-puissant", pour seulement de petites valeurs de n. Essayez juste de calculer le temps effectif (en secondes) pour quelques valeurs de n (30, 40 et 50 par exemple) en supposant un temps d'exécution de base de 10^9 actions élémentaires par seconde.

Question 2. Donnez un algorithme "naturellement" itératif qui réalise le calcul de f_n pour tout entier n et dont la complexité est $\Theta(n)$.

Question 3. Peut-on affimer que l'algorithme proposé dans la question 2 est de complexité linéaire ? Justifiez votre réponse.

Question 4. Il est possible de concevoir un algorithme bien plus efficace que les 2 précédents. Il est basés sur une exploitation astucieuse de mise à la puissance d'une matrice 2×2. On donne les propriétés conduisant à cet algorithme :

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \times \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Qui peut se réécrire en :

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \times \begin{pmatrix} f_0 \\ f_1 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}$$

Et on peut constater que :

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \times \begin{pmatrix} f_1 \\ f_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} f_2 \\ f_3 \end{pmatrix}$$

Et également que :

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \times \begin{pmatrix} f_2 \\ f_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \end{pmatrix} = \begin{pmatrix} f_3 \\ f_4 \end{pmatrix}$$

Et plus généralement :

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \times \begin{pmatrix} f_0 \\ f_1 \end{pmatrix} = \begin{pmatrix} f_n \\ f_{n+1} \end{pmatrix}$$

En exploitant cette propriété et les résultats de l'exercice 3, donnez un algorithme plus efficace que ceux donnés en réponse aux questions 1 et 2. Évaluez sa complexité.