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

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

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

Question 1. Majoration. Démontrez les propriétés suivantes

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

Correction. Pour que $g(n) \in O(f(n))$, il faut qu'il existe un réel positif c et un entier n_0 tel que $\forall n \geq n_0$, $g(n) \leq c.f(n)$.

- $g(n) = 6n + 12 \in O(n)$ car $6n \leq 7n$ quand $n \geq 12$ ($c = 7$ et $n_0 = 12$). (*On aurait pu prendre d'autres valeurs pour c (8, 9, etc) et pour n_0 (13, 14, etc). Il suffit de trouver un couple de valeurs de c et n_0 qui fonctionne.*)
- $g(n) = 3n \in O(n^2)$ car $3n < 3n^2$ quand $n \geq 2$ ($c = 3$ et $n_0 = 2$).
- $g(n) = 10^{1000000}n \in O(n)$ car $10^{1000000}n \leq (10^{1000000} + 1)n$ quand $n \geq 1$ ($c = 10^{1000000} + 1$ et $n_0 = 1$).
- $g(n) = 5n^2 + 10n \in O(n^2)$ car $5n^2 + 10n \leq 1000n^2$ quand $n \geq 1000$ ($c = 1000$ et $n_0 = 1000$).
- Pour montrer que $g(n) = n^3 + 1000n^2 + n + 8 \notin O(n^2)$, il suffit de montrer qu'il est faux que "il existe un réel positif c et un entier n_0 tel que $\forall n \geq n_0$, $n^3 + 1000n^2 + n + 8 \leq c.n^2$" c'est-à-dire que pour tout réel positif c et pour tout entier n_0, $\exists n \geq n_0$ tel que $n^3 + 1000n^2 + n + 8 > c.n^2$. Or, quand $n > 0$, $n^3 + 1000n^2 + n + 8 > c.n^2$ équivaut à $n > c - 1000 - \frac{1}{n} - \frac{8}{n^2}$ et alors $n > c$. Donc, quels que soient c et n_0, il existe un entier n plus grand que n_0 et c et qui vérifie $n^3 + 1000n^2 + n + 8 > c.n^2$.

Question 2. Ordre exact. Démontrez les propriétés suivantes

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

Correction. Pour que $g(n) \in \Theta(f(n))$, il faut qu'il existe deux réels positifs c_1 et c_2 et un entier n_0 tel que $\forall n \geq n_0$, $c_1.f(n) \leq g(n) \leq c_2.f(n)$.

- $g(n) = 5n^2 + 10 \in \Theta(n^2)$ car $5n^2 \leq 5n^2 + 10 \leq 6n^2$ quand $n \geq 4$ ($c_1 = 5$, $c_2 = 6$ et $n_0 = 4$).
- $g(n) = n^2 + 1000000n \in \Theta(n^2)$ car $n^2 \leq n^2 + 1000000n \leq 2n^2$ quand $n \geq 10^9$ ($c_1 = 1$, $c_2 = 2$ et $n_0 = 10^9$).
- $g(n) = 4n^2 + nlog(n) \in \Theta(n^2)$ car $4n^2 \leq 4n^2 + nlog(n) \leq 5n^2$ quand $n \geq 1000$ ($c_1 = 4$, $c_2 = 5$ et $n_0 = 1000$).
- Pour montrer que $g(n) = 3n + 8 \notin \Theta(n^2)$, il suffit de montrer qu'il est faux que "il existe un réel positif c_2 et un entier n_0 tel que $\forall n \geq n_0$, $c_2.n^2 \leq 3n + 8$" c'est-à-dire que pour tout réel positif c_2 et pour tout entier n_0, $\exists n \geq n_0$ tel que $c_2.n^2 > 3n + 8$. Or, l'inéquation $c_2.n^2 - 3n - 8 > 0$ a pour discriminant $9 + 32c_2$, qui est strictement positif donc il existe un entier n supérieur à la plus grande de ses racines qui vérifie l'inéquation.

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).

Correction.

```
En entree : deux entiers x et p strictement positifs
En sortie : x^p

puissance(k, p) :
(1) res = 1
(2) Tant que p > 0
(3)      res = res * x
(4)      p = p - 1
(5) retourne res
```

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

Correction.. La boucle "`Tantque`" est exécutée p fois mais p est codé sur $t_p = \lfloor log_2(p) \rfloor + 1$ bits donc la boucle "`Tantque`" est exécutée $2^{t_p - 1}$ fois. L'algorithme a une complexité au moins exponentielle en t_p, la taille d'une des données d'entrée. On ne peut donc pas dire que la complexité est linéaire, même si elle est linéaire en fonction de la valeur de p, valeur qui n'est pas la **taille** de la donnée en entrée p.

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.

Correction. Dans le cas où $p = 2^k$, on peut soit prendre p comme donnée d'entrée, soit prendre k, puisque la valeur de k suffit à déterminer la valeur de p. Nous allons donc proposer deux variantes de solutions.

Première variante :

```
En entree : deux entiers x et p = 2^k strictement positifs, k entier
En sortie : x^p

puissance2(x, p) :
(1) res = x
(2) Tant que p > 1
(3)      res = res * res
(4)      p = p / 2
(5) retourne res
```

Calculons la complexité en fonction de la taille t_x de x et de la taille t_p de p. En ligne (1), "res = x" se fait en $\Theta(t_x)$ car l'affectation de `x` à `res` se fait bit à bit (précisément : octet par octet ou mot machine par mot machine). En ligne (2), le test "p > 1" peut se faire en $\Theta(t_p)$ car on compare deux entiers bit

à bit[1]. La ligne (3) s'exécute en $\Theta(taille(res)^2)$ et donc lors de la dernière étape en $\Theta(taille((x^p)^2)) = \Theta((p.t_x)^2) = \Theta(p^2.t_x^2) = \Theta((2^{t_p})^2.t_x^2) = \Theta(2^{2.t_p}.t_x^2)$. (en enlevant le +1 pour alléger). En ligne (4), "$p = p/2$" peut se faire en $\Theta(t_p)$ (on décale les bits vers la droite). La boucle "`Tantque`" est répétée $\lfloor log_2(p) \rfloor$ fois. La complexité de l'algorithme est donc en $\Theta(t_x + log_2(p)(2^{2.t_p}.t_x^2 + t_p)) = \Theta(t_x + t_p(2^{2.t_p}.t_x^2 + t_p)) = \Theta(t_p.2^{2.t_p}.t_x^2)$.

Deuxième variante : dans cette deuxième variante, nous allons supposer, contrairement à la première solution, que les données sont constituées de x et maintenant de k (et non plus de p) sachant que la valeur de p est déterminée en fait en fonction de celle de k, puisque $p = 2^k$. L'algorithme n'est donc plus le même et bien sûr l'analyse de la complexité n'est plus la même.

```
En entree : deux entiers x et k strictement positifs
En sortie : x^(2^k)

puissance2(x, k) :
(1) res = x
(2) Tant que k > 0
(3)      res = res * res
(4)      k = k - 1
(5) retourne res
```

Calculons très précisément la complexité en fonction de la taille t_n de n et de la taille t_k de k. En ligne (1), "res = x" se fait en $\Theta(t_x)$. En ligne (2), le test "k > 0" peut se faire en $\Theta(t_k)$ car on compare deux entiers bit à bit (pour le mêmes raison qu'avec p dans la première variante). La ligne (3) s'exécute en $\Theta(taille(res)^2)$ et donc dans le pire des cas en $\Theta(taille((x^{2^k}))^2) = \Theta((2^k.t_x)^2) = \Theta((2^{2^{t_k}}.t_x)^2)$. En ligne (4), "$k = k - 1$" peut se faire en $\Theta(t_k)$ (depuis le bit des unités, on passe les bits de 0 à 1 jusqu'à trouver un bit à 1, qu'on passe à 0). La boucle "`Tantque`" est répétée k fois. La complexité de l'algorithme est donc en $\Theta(t_x + k(t_k + (2^{2^{t_k}}.t_x)^2 + t_k)) = \Theta(t_x + 2^{t_k}(2t_k + (2^{2^{t_k}})^2)) = \Theta(t_x + 2^{t_k}(2^{2^{t_k}}.t_x)^2) = \Theta(2^{t_k}(2^{2^{t_k}}.t_x)^2)$. NB : on a affaire à une double exponentielle de t_k mais cela ne veut évidemment pas dire que le temps d'exécution est plus long qu'en question 2, au contraire : le nombre de tours de boucle est en $log_2(p)$.

1. On peut aussi imaginer qu'on fait une comparaison avec 1, ce qui consiste à vérifier si p possède au moins deux bits, ce qui se fait en temps constant mais ne change pas l'évaluation globale de la 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.

Correction.

```
En entree : deux matrices A et B de cote n
En sortie : A x B

produit(A, B) :
(1) Pour i = 1 a n
(2)    Pour j = 1 a n
(3)        P[i,j] = 0
(4)        Pour k = 1 a n
(5)            P[i,j] = P[i,j] + A[i,k] . B[k,j]
(6) retourne P
```

Sous l'hypothèse où les opérations sur les entiers se font en temps constant, le nombre d'opérations en temps constant est :

- le test et l'incrémentation de la boucle sur i est fait n fois.
- le test et l'incrémentation de la boucle sur j est fait n^2 fois.
- l'affectation de la ligne (3) est faite n^2 fois.
- le test et l'incrémentation de la boucle sur k ainsi que la ligne (5) sont fait n^3 fois.

Le temps d'exécution est donc en $\Theta(n^3)$ mais ...

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

Correction. (suite du "mais...")... on doit exprimer la complexité en fonction de la taille des données d'entrée. Si on suppose que les valeurs dans la matrice sont bornées par une constante, la taille des données d'entrées est $t = 2.n^2.v$ où v est la taille constante pour stocker une valeur. Le temps d'exécution est donc en $\Theta(t^{\frac{3}{2}})$.

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.

Correction.

```
En entree : une matrice A de cote n et un entier p
En sortie : A^p

puissance(A, p) :
(1) P = A
(2) Pour i = 1 a p - 1
(3)     P = produit(A, P)
(4) retourne P
```

Complexité : la ligne (1) est en $\Theta(n^2)$ (transferts de $n2$ valeurs de A vers P), la ligne (3) est répétée p - 1 fois et sa complexité est en $\Theta(n^3)$. En tout, on est donc en $\Theta(p.n^3)$ soit en $\Theta(2^{t_p}.n^{\frac{3}{2}t_A})$, où t_p est la taille de p et t_A la taille de A.

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.

Correction. Nous procédons ici comme dans la deuxième variante de la correction de la question 3 de l'exercice 2, en supposant que les données sont constituées d'une matrice carré A de taille $n \times n$ et de k (et non plus de p) sachant que la valeur de p est déterminée en fonction de celle de k, puisque $p = 2^k$.

```
En entree : une matrice A de cote n et un entier k
En sortie : A^(2^k)

puissance2(A, k) :
(1) P = A
(2) Pour i = 1 a k
(3)     P = produit(P, P)
(4) retourne P
```

Complexité : la ligne (1) est en $\Theta(n^2)$ (transferts de $n2$ valeurs de A vers P), la ligne (3) est répétée k fois et sa complexité est en $\Theta(n^3)$. En tout, on est donc en $\Theta(k.n^3)$ soit en $\Theta(2^{t_k}.n^{\frac{3}{2}t_A})$, où t_k est la taille de k et t_A la taille de A. NB : Cependant, comme $k = log_2(p)$, on est beaucoup plus rapide qu'avant.

Exercice 4. Pour en finir, 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]`).

Correction. Dans le pire des cas, le tableau est trié et la boucle `while` et donc le test "$t[p] < t[i]$" est exécutée $i + 1$ fois. Cette boucle est exécutée pour les valeurs de i de 1 à $n - 1$ donc le test est exécutée $\Sigma_{i=1}^{i=n-1}(i + 1) = \frac{n(n-1)}{2} + n - 1 = \frac{n^2}{2} - \frac{n}{2} + n - 1 = \frac{n^2}{2} + \frac{n}{2} - 1$ fois. On est donc en $\Theta(n^2)$.

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`).

Correction. Dans le pire des cas, le tableau est trié en sens inverse et la boucle for, et donc l'affectation "$t[j + 1] = t[j]$", est exécutée i fois. Cette boucle est exécutée pour les valeurs de i de 1 à n - 1 donc l'affectation est exécutée $\Sigma_{i=1}^{i=n-1} i = \frac{n(n-1)}{2}$ fois. On est en $\Theta(n^2)$.

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

Correction. Le pire des cas n'est pas le même pour le nombre de tests et le nombre d'affectations. Cependant, dans ces deux cas, la complexité globale est en $\Theta(n^2)$. Dans tous les autres cas, le nombre de tests et de comparaisons ne peut pas être supérieur à du $\Theta(n^2)$. Donc, dans le(s) pire(s) des cas globalement, on est en $\Theta(n^2)$.

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

Correction. À vous de travailler !!! Pour les TP notamment...