Licence informatique - L2 Année 2013/2014

Conceptions algorithmiques (L2)
TD Algorithmes *diviser pour régner*
Séances 2,3,4

Rappel du théorème maître

Théorème 1. *(Résolution de récurrences diviser pour régner). Soient $a \geq 1$ et $b > 1$ deux constantes, soient f une fonction à valeurs dans $\mathbb{R}^+$ et T une fonction de $\mathbb{N}^*$ dans $\mathbb{R}^+$ vérifiant, pour tout n suffisamment grand, l'encadrement suivant : $aT(\lfloor n/b \rfloor)+f(n) \leq T(n) \leq aT(\lceil n/b \rceil)+f(n)$. Alors T peut être bornée asymptotiquement comme suit :*

1. *Si $f(n) = O(n^{(\log_b a)-\varepsilon})$ pour une certaine constante $\varepsilon > 0$, alors $T(n) = \Theta(n^{\log_b a})$.*
2. *Si $f(n) = \Theta(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a} \log n)$.*
3. *Si $f(n) = \Omega(n^{(\log_b a)+\varepsilon})$ pour une certaine constante $\varepsilon > 0$, et si on a asymptotiquement pour une constante $c < 1$, $af(n/b) < cf(n)$, alors $T(n) = \Theta(f(n))$.*

1 Notations Asymptotiques

Exercice 1.1. Rappeler les définitions des notations O, Ω, Θ pour les fonctions à valeur dans $\mathbb{R}_+^*$.

Solution:

$f(n) = O(g(n)) \Leftrightarrow$ il existe des constantes stt positives c et n_0 telles que $0 \leq f(n) < c.g(n)$ pour tout $n \geq n_0$ $\Leftrightarrow \limsup_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| < \infty$

$f(n) = \Omega(g(n)) \Leftrightarrow$ il existe des constantes stt positives c et n_0 telles que $0 \leq c.g(n) \leq f(n)$ pour tout $n \geq n_0$ $\Leftrightarrow \liminf_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| > 0$

$f(n) = \Theta(g(n)) \Leftrightarrow$ il existe des constantes stt positives c_1, c_2 et un n_0 tels que $0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n)$ pour tout $n \geq n_0$

Exercice 1.2. Le but de cet exercice est de tester votre compréhension de la notion de complexité dans le pire des cas.

1. Si je prouve que la complexité dans le pire des cas d'un algorithme est en $O(n^2)$, est-il possible qu'il soit en $O(n)$ sur *certaines* données ?

2. Si je prouve que la complexité dans le pire des cas d'un algorithme est en $O(n^2)$, est-il possible qu'il soit en $O(n)$ sur *toutes* les données ?

3. Si je prouve que la complexité dans le pire des cas d'un algorithme est en $\Theta(n^2)$, est-il possible qu'il soit en $O(n)$ sur *certaines* données ?

4. Si je prouve que la complexité dans le pire des cas d'un algorithme est en $\Theta(n^2)$, est-il possible qu'il soit en $O(n)$ sur *toutes* les données ?

Solution:

1. OUI
2. OUI
3. OUI
4. NON

Exercice 1.3. Sur une échelle croissante, classer les fonctions suivantes selon leur comportement asymptotique : c'est-à-dire $g(n)$ suit $f(n)$ si $f(n) = O(g(n))$.

$f_1(n) := 2\,n$	$f_2(n) := 2^n$	$f_3(n) := \log(n)$	$f_4(n) := \frac{n^3}{3}$
$f_5(n) := n!$	$f_6(n) := \log(n)^2$	$f_7(n) := n^n$	$f_8(n) := n^2$
$f_9(n) := n + \log(n)$	$f_{10}(n) := \sqrt{n}$	$f_{11}(n) := \log(n^2)$	$f_{12}(n) := e^n$
$f_{13}(n) := n$	$f_{14}(n) := \sqrt{\log(n)}$	$f_{15}(n) := 2^{\log_2(n)}$	$f_{16}(n) := n\,\log(n)$

Solution: On classe d'abord les fonctions en trois catégories : les fonctions polylogarithmiques ($\log^k n$), les fonctions polynomiales (n^k) et les fonctions exponentielles (a^n). Pour les autres fonctions, on essaie quand même de les faire rentrer dans une de ces trois catégories. Par exemple pour $n \geq 2$, on a $2^n \leq n! \leq n^n$ donc $n!$ est plus qu'exponentielle.

$$f_{14} < f_3 \equiv f_{11} < f_6 < f_{10} < f_{13} = f_{15} \equiv f_9 \equiv f_1 < f_{16} < f_8 < f_4 < f_2 < f_{12} < f_5 < f_7$$

$$\sqrt{\log(n)} < \log(n) \equiv \log(n^2) < \log(n)^2 < \sqrt{n} < n = 2^{\log_2(n)} \equiv n + \log(n) \equiv 2\,n$$
$$< n\,\log(n) < n^2 < \frac{n^3}{3} < 2^n < e^n < n! < n^n$$

Remarque :

$$\log_a n = \frac{\log_b n}{\log_b a}$$
$$n\,! \sim \sqrt{2\pi n}\,\left(\frac{n}{e}\right)^n$$

2 Fonctions définies par récurrence

Exercice 2.1. Soit un algorithme dont la complexité $T(n)$ est donnée par la relation de récurrence :

$$\begin{aligned} T(1) &= 1, \\ T(n) &= 3\,T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n^2. \end{aligned}$$

a. Calculer $T(n)$ en résolvant la récurrence.

b. Déterminer $T(n)$ à l'aide du théorème maître.

Remarque :
On a

$$a^{\log_x b} = b^{\log_x a} \quad \text{et} \quad \sum_{i=0}^{k} x^i = \frac{x^{k+1}-1}{x-1}.$$

Solution:

a.

$$\begin{aligned}
T(n) &= 3\,T\left(\frac{n}{2}\right) + n^2 \\
&= 3\left(3\,T\left(\frac{n}{4}\right) + \frac{n^2}{4}\right) + n^2 \\
&= 3\left(3\left(3\,T\left(\frac{n}{8}\right) + \frac{n^2}{16}\right) + \frac{n^2}{4}\right) + n^2 \\
&\vdots \\
&= \sum_{i=0}^{\log_2 n} 3^i \frac{n^2}{4^i} = n^2 \sum_{i=0}^{\log_2 n} \left(\frac{3}{4}\right)^i \\
&= -4n^2 \left(\frac{3}{4}\right)^{\log_2 n+1} + 4n^2 \\
&= -3n^{\log_2 3} + 4n^2
\end{aligned}$$

Remarque :
Des fois, on s'en sort mieux en posant $n = 2^N$ pour avoir $T(N-1)$ (pas dans cet exemple).

b. $a = 3$, $b = 2$, $f(n) = n^2 = \Omega(n^2)$, cas 3, $\varepsilon = 2 - \log_2 3$, $c = 3/4 + 0.1$ par ex. $\Rightarrow T(n) = \Theta(n^2)$.

Exercice 2.2. Le but de cet exercice est d'utiliser le théorème maître pour donner des ordres de grandeur asymptotique pour des fonctions définies par récurrence.

1. En utilisant le théorème maître, donner un ordre de grandeur asymptotique pour $T(n)$:

a.

$$T(1) = 1, \quad T(n) = 2\,T\left(\frac{n}{2}\right) + n^2;$$

b.

$$T(1) = 0, \quad T(n) = 2\,T\left(\frac{n}{2}\right) + n;$$

c.

$$T(1) = 1, \quad T(n) = 2\,T\left(\frac{n}{2}\right) + \sqrt{n};$$

d.

$$T(1) = 0, \quad T(n) = T\left(\frac{n}{2}\right) + \Theta(1).$$

2. Essayer de donner des exemples d'algorithmes dont le coût est calculé par l'une de ces récurrences.

Solution:

1. a. $a = b = 2$, $f(n) = n^2 = \Omega(n^2)$, cas 3, $\varepsilon = 1$, $c = 3/4$ par exemple. $\Rightarrow T(n) = \Theta(n^2)$.

b. $a = b = 2$, $f(n) = n = \Theta(n)$, cas 2. $\Rightarrow T(n) = \Theta(n \log(n))$.

c. $a = b = 2$, $f(n) = \sqrt{n} = O(n^{\frac{1}{2}})$, cas 1, $\varepsilon = \frac{1}{2}$. $\Rightarrow T(n) = \Theta(n)$.

d. $a = 1$, $b = 2$, $f(n) = 1 = \Theta(n^0)$, cas 2. $\Rightarrow T(n) = \Theta(\log(n))$.

2. a. Les point rapprochés, méthode naïve.

b. Tri fusion, Quicksort, ...

c. ...

d. Recherche dichotomique.

Exercice 2.3. Le but de cet exercice est de choisir l'algorithme de type *diviser pour régner* le plus rapide pour un même problème.

1. Pour résoudre un problème manipulant un nombre important de données, on propose deux algorithmes :

a. un algorithme qui résout un problème de taille n en le divisant en 2 sous-problèmes de taille $n/2$ et qui combine les solutions en temps quadratique,

b. un algorithme qui résout un problème de taille n en le divisant en 4 sous-problèmes de taille $n/2$ et qui combine les solutions en temps $O(\sqrt{n})$.

Lequel faut-il choisir ?

2. Même question avec les algorithmes suivants :

a. un algorithme qui résout un problème de taille n en le réduisant à un sous-problème de taille $n/2$ et qui combine les solutions en temps $\Omega(\sqrt{n})$,

b. un algorithme qui résout un problème de taille n en le divisant en 2 sous-problèmes de taille $n/2$ et qui combine les solutions en temps constant.

Solution:

1. a. $T(n) = 2\,T\left(\frac{n}{2}\right) + n^2 \Rightarrow T(n) = \Theta(n^2)$.

b. $a = 4$, $b = 2$, $f(n) = O(\sqrt{n})$, cas 1. $\varepsilon = \frac{3}{2}$, $\Rightarrow T(n) = \Theta(n^2)$.

Les 2 algos sont équivalents.

2. a. $a = 1$, $b = 2$, $f(n) = \Omega(n^{\frac{1}{2}})$, cas 3. $\varepsilon = \frac{1}{2}$, $c = 1/\sqrt{2}$ par exemple. $\Rightarrow T(n) = \Theta(\sqrt{n})$.

b. $a = b = 2$, $f(n) = \Theta(1)$, cas 1. $\varepsilon = 1$, $\Rightarrow T(n) = \Theta(n)$.

Le premier algo est meilleur.

Exercice 2.4. Considérons maintenant un algorithme dont le nombre d'opérations sur une entrée de taille n est donné par la relation de récurrence :

$$T(1) = 0 \quad \text{et} \quad T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2\,n}{3}\right) + n.$$

1. Construire un arbre représentant les appels récursifs de l'algorithme. En déduire la complexité de $T(n)$.
2. Vérifier le résultat par récurrence (méthode par substitution).
3. Cette récurrence peut-elle être résolue en utilisant le théorème maître ?

Solution:

1. On déduit une complexité $O(n \log_3 n)$.
2. On veut montrer la propriété $P(n) : T(n) = O(n \log_3 n)$, c-a-d, $\exists c,\ T(n) \leq c.n \log_3 n$. On suppose que $P(k)$ est vraie pour tout $k < n$. On a donc :

$$\begin{aligned} T(n) &\leq c \cdot \frac{n}{3} \log_3 \frac{n}{3} + c \cdot \frac{2n}{3} \log_3 \frac{2n}{3} + n \\ &= c \cdot \frac{n}{3} \log_3 n + c \cdot \frac{2n}{3} \log_3 n - c \cdot \frac{n}{3} + c \cdot \frac{2n}{3} \log_3 2 - c \cdot \frac{2n}{3} + n \\ &= c \cdot n \cdot \log_3 n - c \cdot n + c \cdot \frac{2n}{3} \log_3 2 + n \\ &= c \cdot n \cdot \log_3 n + c \cdot (\frac{2n}{3} \log_3 2 - n) + n \end{aligned}$$

   Pour que $P(n)$ soit vraie, il faut que $c \cdot (\frac{2n}{3} \log_3 2 - n) + n < 0$, c-a-d :

$$c \geq \frac{-1}{\frac{2}{3} \log_3 2 - 1} \sim 1,7.$$

   On peut par exemple choisir $c = 3$ et on vérifie que

$$3 \cdot \frac{n}{3} \log_3 \frac{n}{3} + 3 \cdot \frac{2n}{3} \log_3 \frac{2n}{3} + n = 3n \log_3 n + 2n(\log_3 2 - 1) \leq 3n \log_3 n.$$

3. Le théorème maître ne permet pas de résoudre cette récurrence.

3 Recherche d'éléments, de suite d'éléments

Exercice 3.1. Donnez un algorithme de recherche d'un élément dans un tableau **trié** (dans l'ordre croissant) :

- **a.** en utilisant une recherche séquentielle,
- **b.** en utilisant le principe *diviser pour régner* .

Donnez la complexité (en nombre de comparaisons) dans le pire cas de chacun de ces deux algorithmes en fonction de la taille n du tableau.

Solution:

1.
```
Recherche1(t, e)
    i <- 1;
    tantque i≤longueur(t) et t[i]<e faire
      i <- i+1;
```

```
    fintantque
    si i=longueur(t)+1 OU t[i]≠e alors
      retourne FAUX
    finsi
    retourne i;
  fin
```

Complexité : $O(n)$

2.
```
Recherche2(t, g, d ,e)
  si g=d alors
    si t[g]=e alors retourne g;
    sinon retourne FAUX
    finsi;
  finsi;
  si e≤t[⌊(g+d)/2⌋] alors
    retourne Recherche2(t, g, ⌊(g+d)/2⌋, e);
  sinon
    retourne Recherche2(t, ⌊(g+d)/2⌋+1, d, e);
  finsi
fin
```

Complexité : $T(n) = T\left(\frac{n}{2}\right) + O(1)$ cf. exo **??**.

Exercice 3.2. On cherche le k-ième plus petit élément dans un tableau **non trié**.

a. Proposer un algorithme utilisant le principe *diviser pour régner* en s'inspirant de celui du tri rapide.

b. Faire la trace de l'algorithme sur le tableau

2	4	1	6	3	5

pour $k = 4$.

c. Donnez la complexité (en nombre de comparaisons) dans le pire cas de cet algorithme en fonction de la taille n du tableau.

d. Que se passe-t-il si, à chaque étape, le tableau considéré est coupé en 2 sous-tableaux de tailles équivalentes ?

Solution:

a.
```
RechercheKième(t, g, d, k)
    si g=d alors retourne t[g] finsi;

    p <- t[g];   #on choisit le 1er élément comme pivot.
    j <- g;      #dernière entrée <=p.
    pour i de g+1 à d faire
      si t[i]≤p alors
        swap(t[i],t[j+1]);
```

```
            j <- j+1;
          finsi;
        fin pour;
        swap(t[g],t[j]);

        si k=j alors
          retourne t[j];
        sinon
          si k<j alors
           retourne RechercheKième(t, g, j, e)
          sinon
            retourne RechercheKième(t, j+1, d, e)
          finsi
        finsi
    fin
```

b.

		i					
2	4	1	6	3	5		p=2,
j=1							
		i					
2	4	1	6	3	5		p=2,
j							
			i	i	i		
2	1	4	6	3	5	⟶ swap ⟶	1 2 | 4 6 3 5
	j=2						
		i					
4	6	3	5				p=4,
j=3							
			i	i			
4	6	3	5			⟶ swap ⟶	3 4 6 5 ⟶ j=k ⟶ fin.
	j=4						j=4

c. Pire cas : tableau trié et $k = lg(tableau)$: $T(n) = T(n-1) + O(n)$.
La complexité est $O(n^2)$.

d. $T(n) = T\left(\frac{n}{2}\right) + O(n)$.
$a = 1$, $b = 2$, $f(n) = n = \Omega(n)$, cas 3 du th.maître, $\varepsilon = 1$, $c = 3/4$ par exemple.
$\Rightarrow T(n) = \Theta(n)$.

Exercice 3.3. On dispose d'un tableau d'entiers *relatifs* de taille n. On cherche à déterminer la suite d'entrées consécutives du tableau dont la somme est maximale. Par exemple, pour le tableau $T = [-1, 9, -3, 12, -5, 4]$, la solution est 18 (somme des éléments de $T[2..4] = [9, -3, 12]$).

a. Proposer un algorithme élémentaire pour résoudre ce problème. Donner sa complexité.

b. Donner un (meilleur) algorithme de type *diviser pour régner* . Quelle est sa complexité ?

Solution:

a. 1. `MaxSum1(t)`

```
max <- 0;
pour i de 1 à longueur(t) faire
  pour j de i+1 à longueur(t) faire
    S <- 0;
    pour k de i à j faire
      S <- S+t[k];
      si S>max alors
        max <- S;
  finsi
fin pour
  fin pour
fin pour
retourne max;
fin
```

Complexité : $O(n^3)$

2. `MaxSum2(t)`

```
max <- 0;
pour i de 1 à longueur(t) faire
  S <- 0;
  pour j de i+1 à longueur(t) faire
    S <- S+t[j];
    si S>max alors
      max <- S;
finsi
  fin pour
fin pour
retourne max;
fin
```

Complexité : $O(n^2)$

b. `MaxAcheval(t, g, d)`

```
int max <- 0;
int S <- 0;
pour i de ⌊(g+d)/2⌋ + 1 à d faire
  S<- S+t[i];
  si S>max alors
    max <- S;
  finsi
fin pour
S <-max;
pour i de ⌊(g+d)/2⌋ à g pas de -1 faire
  S <- S+t[i];
  si S>max alors
    max <- S;
  fin si
```

```
        fin pour
        retourne max;
    fin

    MaxSum3(t, g, d) {
        si g>d alors retourne 0 finsi;
        si g=d alors
          si t[g]<0 alors
            retourne 0;
          sinon
            retourne t[g];
          finsi
        finsi
        m1 <- MaxSum3(t,g, ⌊(g+d)/2⌋);
        m2 <- MaxSum3(t,⌊(g+d)/2⌋ + 1,d);
        m3 <- MaxAcheval(t,g,d);

        retourne max(m1,m2,m3);
    fin
```

Complexité : $T(n) = 2\,T\left(\frac{n}{2}\right) + O(n)$.
$a = b = 2$, $f(n) = n = \Theta(n)$, cas 2. du th. maître. $\Rightarrow T(n) = \Theta(n \log(n))$.

Exercice 3.4. Un élément x est majoritaire dans un ensemble E de n éléments si E contient strictement plus de $n/2$ occurrences de x. La seule opération dont nous disposons est la comparaison (égalité ou non) de deux éléments.

Le but de l'exercice est d'étudier le problème suivant : étant donné un ensemble E, représenté par un tableau, existe-t-il un élément majoritaire, et si oui quel est-il ?

1. Algorithme naïf

a. Écrivez un algorithme NombreOccurrences qui, étant donné un élément x et deux indices i et j, calcule le nombre d'occurrences de x entre $E[i]$ et $E[j]$. Quelle est sa complexité (en nombre de comparaisons) ?

b. Écrivez un algorithme Majoritaire qui vérifie si un tableau E contient un élément majoritaire, et si oui le retourne. Quelle est sa complexité ?

2. Algorithme de type *diviser pour régner*
Pour calculer l'élément majoritaire de l'ensemble E (s'il existe), on découpe l'ensemble E en deux sous-ensembles E_1 et E_2 de même taille, et on calcule récursivement dans chaque sous-ensemble l'élément majoritaire. Pour qu'un élément x soit majoritaire dans E, il suffit que l'une des conditions suivantes soit vérifiée :
– x est majoritaire dans E_1 et dans E_2,
– x est majoritaire dans E_1 et non dans E_2, mais suffisamment présent dans E_2,
– x est majoritaire dans E_2 et non dans E_1, mais suffisamment présent dans E_1.
Écrivez un algorithme utilisant cette approche *diviser pour régner* . Analysez sa complexité.

3. Pour améliorer l'algorithme précédent, on propose de construire un algorithme PseudoMajoritaire tel que :
– soit l'algorithme garantit que E ne possède pas d'élément majoritaire,

– soit il rend un couple (x, p) tel que $p > n/2$, x est un élément apparaissant au plus p fois dans E, et tout élément $y \neq x$ de E apparaît au plus $n - p$ fois dans E.

Donnez un algorithme récursif vérifiant ces conditions. Donnez sa complexité. Déduisez-en un algorithme de recherche d'un élément majoritaire, et donnez sa complexité.

Solution:

Rmq. 1 pour simplifier, on peut supposer que $n = 2^k$.

Rmq. 2 pour la question 2, comme indice : renvoyer un couple (booléen, élément).

4 Tris

Exercice 4.1 (Tri fusion)**.** Le but de l'exercice est de trier un tableau en utilisant l'approche *diviser pour régner* : on commence par couper le tableau en deux, on trie chaque moitié avec notre algorithme, puis on fusionne les deux moitiés.

1. Écrire un algorithme FUSION(A, B) permettant de fusionner deux tableaux A et B déjà triés de tailles n_A et n_B en un seul tableau trié T. Quelle est sa complexité ?

2. À l'aide de l'algorithme décrit ci-dessus, écrire l'algorithme du tri fusion.

3. Quelle est la complexité du tri fusion ?

Solution:

1.
```
FUSIONNER(A : tableau de nombres, B : tableau de nombres)

  i_A := 0
  i_B := 0
  i_R := 0

  n_A := longueur de A
  n_B := longueur de B

  Initialiser un tableau R de taille n_A + n_B

  TANT QUE i_A < n_A ou i_B < n_B
  FAIRE
    SI i_B >= n_B A[i_A]<B[i_B]
    ALORS
      R[i_R] = A[i_A]
      i_A ++
    SINON
      R[i_R] = B[i_B]
      i_B ++
    FIN SI
    i_R ++
```

```
FIN

TANT QUE i_A < n_A
FAIRE
  R[i_R] = A[i_A]
  i_A ++
  i_R ++
FIN

TANT QUE i_B < n_B
FAIRE
  R[i_R] = B[i_B]
  i_B ++
  i_R ++
FIN

RENVOYER R
```

La complexité est linéaire, $O(n)$

2.
```
TRIFUSION(T)

A = TRIFUSION(1ere moitie de T)
B = TRIFUSION(2eme moitie de T)
RENVOYER(FUSIONNER(A,B))
```

3. $C(n) = n + 2C(n/2)$
Une application immediate du theoreme maitre (2eme cas) donne $C(n) = O(n \log n)$

5 Algèbre et arithmétique

Exercice 5.1 (Multiplication de deux polynômes)**.** On représente un polynôme $p(x) = \sum_{i=0}^{m-1} a_i x^i$ par la liste de ses coefficients $[a_0, a_1, \ldots, a_{m-1}]$. Nous nous intéressons au problème de la multiplication de deux polynômes : étant donnés deux polynômes $p(x)$ et $q(x)$ de degré au plus $n-1$, calculer $pq(x) = p(x)q(x) = \sum_{i=0}^{2n-2} c_i x^i$.

1. Préliminaires : énumérez les opérations basiques (de coût 1) qui serviront à calculer la complexité de vos algorithmes. Vérifiez que l'addition de deux polynômes de degré $n-1$ se fait en $\Theta(n)$.

2. Méthode directe.
Donnez un algorithme calculant directement chaque coefficient c_i. Quelle est sa complexité ?

3. On découpe chaque polynôme en deux parties de tailles égales :

$$p(x) = p_1(x) + x^{n/2} p_2(x),$$
$$q(x) = q_1(x) + x^{n/2} q_2(x),$$

et on utilise l'identité

$$pq = p_1 q_1 + x^{n/2}(p_1 q_2 + p_2 q_1) + x^n p_2 q_2.$$

Donnez un algorithme récursif, utilisant le principe *diviser pour régner*, qui calcule le produit de deux polynômes à l'aide de l'identité précédente. Si $T(n)$ est le coût de la multiplication de deux polynômes de taille n, quelle est la relation de récurrence vérifiée par $T(n)$? Quelle est la complexité de votre algorithme ? Comparez-la à la complexité de la méthode directe.

4. On utilise maintenant la relation

$$p_1q_2 + p_2q_1 = (p_1 + p_2)(q_1 + q_2) - p_1q_1 - p_2q_2.$$

Donnez un algorithme calculant le produit pq à l'aide de cette relation, et explicitez le nombre de multiplications et d'additions de polynômes de degré $n/2$ utilisées. Quelle est la complexité de votre algorithme ?

5. Pensez-vous que cet algorithme a une complexité optimale ?

Solution:

1. Nous comptons comme opération élémentaire (de coût 1) les additions et les multiplications entre les coefficients (scalaires).
Si $p = \sum_{i=0}^{n-1} a_i x^i$ et $q = \sum_{i=0}^{n-1} b_i x^i$ alors $p + q = \sum_{i=0}^{n-1} (a_i + b_i)x^i$. On effectue donc n additions $a_i + b_i$ pour $0 \leq i \leq n-1$. L'addition de deux polynômes de degré $n-1$ se fait donc en $\Theta(n)$.

2. Toujours si $p = \sum_{i=0}^{n-1} a_i x^i$ et $q = \sum_{i=0}^{n-1} b_i x^i$ alors $pq = \sum_{i=0}^{2n-2} c_i x^i$ avec $c_i = \sum_{j=0}^{i} a_j b_{i-j}$. L'algorithme consiste donc à calculer les $2n-1$ coefficients c_i.

```
Entrées : 2 polynômes p et q
Sortie : 1 polynôme pq
Mult(p,q)
    c_i = 0 pour i allant de 0 à 2n − 1
    pour i de 0 à n − 1
        pour j de 0 à n − 1
            c_{i+j} = c_{i+j} + a_i * b_j
        fin pour
    fin pour
fin
```

Par conséquent $T(n) = \Theta(n^2)$.

3.
```
Entrées : 2 polynômes p et q
Sortie : 1 polynôme pq
Mult(p,q)
    k := max(deg(p), deg(q))
     si k ≤ 1 alors
        retourne pq
    sinon
        m := n/2
        p_1 := p mod x^m
        p_2 := p/x^m
        q_1 := q mod x^m
        q_2 := q/x^m
        r_1 := Mult(p_1, q_1)
        r_2 := Mult(p_2, q_2)
```

$r_3 := \texttt{Mult}(p_1, q_2)$
$r_4 := \texttt{Mult}(p_2, q_1)$
retourne $r_1 + (r_3 + r_4)x^m + r_2 x^n$

Soit $T(n)$ le nombre d'opérations élémentaires nécessaires pour calculer le produit de 2 polynômes de taille n. Dans l'algorithme, on fait 4 multiplications de polynômes de taille $n/2$. La reconstruction nécessite d'additionner des polynômes donc est en $O(n)$. Par conséquent

$$T(n) = 4T(n/2) + O(n).$$

Le théorème maître donne $T(n) = \Theta(n^2)$ avec $a = 4$, $b = 2$ et $\log_b a = 2$ et $\varepsilon = 1$. On n'a rien gagné par rapport à l'approche directe.

4. `Entrées : 2 polynômes` p `et` q
`Sortie : 1 polynôme` pq
`MultKaratsuba(p,q)`
$k := \max(\deg(p), \deg(q))$
si $k \leq 1$ **alors**
retourne pq
sinon
$m := n/2$
$p_1 := p \mod x^m$
$p_2 := p/x^m$
$q_1 := q \mod x^m$
$q_2 := q/x^m$
$r_1 := \texttt{MultKaratsuba}(p_1, q_1)$
$r_2 := \texttt{MultKaratsuba}(p_2, q_2)$
$r_3 := \texttt{MultKaratsuba}(p_1 + p_2, q_1 + q_2)$
retourne $r_1 + (r_3 - r_1 - r_2)x^m + r_2 x^n$

On ne fait maintenant plus que 3 multiplications de polynômes de taille $n/2$. Là encore la division et la reconstruction se font en $O(n)$. Par conséquent $T(n) = 3T(n/2) + O(n)$. Pour $a = 3$, $b = 2$, $\log_b a = \log_2 3 \approx 1.585$ et $\varepsilon = 1/2$, le théorème maître donne $T(n) = \Theta(n^{\log_2 3})$.

5. Non, l'algorithme n'est pas optimal. On peut multiplier deux polynômes en $O(n \log n)$ grâce à la FFT (voir le cours AAA).

Exercice 5.2 (Algorithme de Strassen pour la multiplication de deux matrices)**.** La multiplication de matrices est très fréquente dans les codes numériques. Le but de cet exercice est de proposer un algorithme rapide pour cette multiplication.

1. Donnez un algorithme direct qui multiplie deux matrices A et B de taille $n \times n$. Donnez sa complexité en nombre de multiplications élémentaires.

2. Notons $C = A\,B$. Dans le but d'utiliser le principe *diviser pour régner* , on divise les matrices A, B et C en quatre sous-matrices $n/2 \times n/2$

$$C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = A\,B.$$

Les sous-matrices de C peuvent être obtenues en effectuant les produits et additions de

matrices $n/2 \times n/2$ suivants :

$$c_{11} = a_{11}b_{11} + a_{12}b_{21}, \qquad c_{12} = a_{11}b_{12} + a_{12}b_{22},$$
$$c_{21} = a_{21}b_{11} + a_{22}b_{21}, \qquad c_{22} = a_{21}b_{12} + a_{22}b_{22}.$$

Notons $T(n)$ le coût de la multiplication de deux matrices de taille n, quelle est la relation de récurrence vérifiée par $T(n)$ dans ce cas ? Donnez la solution de cette relation de récurrence, comparez-la à la complexité de la méthode directe.

3. Strassen (en 1969) a suggéré de calculer les produits intermédiaires suivants :

$$\begin{aligned} m_1 &= (a_{12} - a_{22})\,(b_{21} + b_{22}), & m_4 &= (a_{11} + a_{12})\,b_{22}, \\ m_2 &= (a_{11} + a_{22})\,(b_{11} + b_{22}), & m_5 &= a_{11}\,(b_{12} - b_{22}), \\ m_3 &= (a_{11} - a_{21})\,(b_{11} + b_{12}), & m_6 &= a_{22}\,(b_{21} - b_{11}), \\ & & m_7 &= (a_{21} + a_{22})\,b_{11}, \end{aligned}$$

puis de calculer C comme suit :

$$c_{11} = m_1 + m_2 - m_4 + m_6, \qquad c_{12} = m_4 + m_5,$$
$$c_{21} = m_6 + m_7, \qquad c_{22} = m_2 - m_3 + m_5 - m_7.$$

a. Montrez que C est bien le produit de A par B (attention, la multiplication de deux matrices n'est pas commutative !).

b. Donnez la relation de récurrence vérifiée par $T(n)$. Déduisez-en la complexité de l'algorithme de Strassen.

c. Pensez-vous que cet algorithme a une complexité optimale ?

Solution:

1. Soit $C = AB$ où A et B sont des matrices carrées de taille n. On a $c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$. Par conséquent, le calcul de c_{ij} nécessite n multiplications et $(n-1)$ additions soit au total $2n-1$ opérations élémentaires. Comme il y a n^2 coefficients à calculer, la complexité est en $\Theta(n^3)$.

2. Soit $T(n)$ le coût de la multiplication de deux matrices de taille n. Pour calculer le produit de deux matrices de taille n, on calcule le produit de 8 matrices de taille $n/2$ puis on les recombine (on somme des matrices de taille $n/2$ ce qui se fait en $O(n^2)$. Par conséquent $T(n) = 8T(n/2) + O(n^2)$. Avec $b = 2$, $a = 8$, $\log_b a = \log_2 8 = 3$, $\varepsilon = 1$, le théorème implique que $T(n) = \Theta(n^3)$. Cela n'améliore pas la complexité par rapport à la méthode directe.

3. **a.** Pour montrer que l'on a bien $C = AB$, il suffit de développer et de simplifier ensuite.

b. La séparation se fait en $O(n^2)$ puisque l'on doit sommer des matrices de taille $n/2$. On fait ensuite 7 multiplications de matrices de taille $n/2$ puis on reconstruit en faisant des additions de matrices de taille $n/2$ ce qui est aussi en $O(n^2)$. Par conséquent $T(n) = 7T(n/2) + O(n^2)$. Avec $b = 2$, $a = 7$, $\log_b a = \log_2 7 \approx 2.807$ et $\varepsilon = 1/2$, le théorème maître donne $T(n) = \Theta(n^{\log_2 7})$.

c. Non on peut faire mieux avec l'algorithme de Coppersmith-Winograd qui permet de calculer un produit matriciel en $O(n^{2.376})$. C'est l'algorithme le plus rapide connu à l'heure actuelle même si on conjecture que l'on doit pouvoir multiplier des matrices en $O(n^2)$. Il est trop compliqué pour le cours car il utilise de la théorie des groupes assez avancée. Malgré sa complexité avantageuse, il n'est jamais utilisé en pratique car la constante dans le O est très grande.

Exercice 5.3 (Matrices Toeplitz)**.** Une matrice Toeplitz est une matrice de taille $n \times n$ (a_{ij}) telle que $a_{i,j} = a_{i-1,j-1}$ pour $2 \leq i, j \leq n$.

1. La somme de deux matrices Toeplitz est-elle une matrice Toeplitz ? Et le produit ?

2. Trouver un moyen d'additionner deux matrices Toeplitz en $O(n)$.

3. Comment calculer le produit d'une matrice Toeplitz $n \times n$ par une vecteur de longueur n ? Quelle est la complexité de l'algorithme ?

Solution:

1. Les matrices Toeplitz sont les matrices dont les diagonales sont constantes $(t_{i-j})_{i,j=0}^{n-1}$.

$$\begin{pmatrix} t_0 & t_{-1} & \cdots & t_{1-n} \\ t_1 & t_0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & t_{-1} \\ t_{n-1} & \cdots & t_1 & t_0 \end{pmatrix}$$

Il devrait clair que la somme de deux matrices Toeplitz reste Toeplitz. Par contre, c'est faux pour le produit. En effet

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}.$$

2. On remarque d'après la définition qu'une matrice Toeplitz est caractérisé par sa première ligne et sa première colonnes, c'est-à-dire par $2n - 1$ éléments. Pour additionner deux matrices Toeplitz, il suffit d'additionner les premières lignes et les premières colonnes ce qui se fait en $O(n)$.

3. On va considérer des matrices de taille n avec n une puissance de 2. On va décomposer la matrice en blocs de taille $n/2$ ainsi que le vecteur Z

$$TZ = \begin{pmatrix} A & B \\ C & A \end{pmatrix} \times \begin{pmatrix} X \\ Y \end{pmatrix}.$$

On pose alors

$$\begin{aligned} U &= (C + A)X, \\ V &= A(Y - X), \\ W &= (B + A)Y. \end{aligned}$$

On a alors

$$TZ = \begin{pmatrix} W - V \\ U + V \end{pmatrix}.$$

Soit $T(n)$ le coût de la multiplication d'une matrice Toeplitz de taille n par un vecteur de taille n. On fait 3 multiplication de matrices Toeplitz de taille $n/2$ par des vecteurs de taille $n/2$. Lors de la division et de la reconstruction, on additionne des matrices Toeplitz de taille $n/2$ et des vecteurs de taille $n/2$ ce qui se fait en $O(n)$. Par conséquent $T(n) = 3T(n/2) + O(n)$. Avec $b = 2$, $a = 3$, $\log_b a = \log_2 3 \approx 1.585$ et $\varepsilon = 1/2$, le théorème maître donne $T(n) = \Theta(n^{\log_2 3})$.

Remarque : On peut même multiplier une matrice Toeplitz par un vecteur en $O(n \log n)$ via la FFT et multiplier deux matrices Toeplitz en $O(n^2)$.

6 Géométrie algorithmique

Exercice 6.1 (Recherche des deux points les plus rapprochés)**.** Le problème consiste à trouver les deux points les plus rapprochés (au sens de la distance euclidienne classique) dans un ensemble P de n points. Deux points peuvent coïncider, auquel cas leur distance vaut 0. Ce problème a notamment des applications dans les systèmes de contrôle de trafic : un contrôleur du trafic aérien ou maritime peut avoir besoin de savoir quels sont les appareils les plus rapprochés pour détecter des collisions potentielles.

1. Donnez un algorithme naïf qui calcule directement les deux points les plus rapprochés, et analysez sa complexité (on représentera P sous forme de tableau et on analysera la complexité de l'algorithme en nombre de comparaisons).

Nous allons construire un algorithme plus efficace basé sur le principe *diviser pour régner* . Le principe est le suivant :

a. Si $|P| \leq 3$, on détermine les deux points les plus rapprochés par l'algorithme naïf.

b. Si $|P| > 3$, on utilise une droite verticale Δ, d'abscisse l, séparant l'ensemble P en deux sous ensembles (Pgauche et Pdroit) tels que l'ensemble Pgauche contienne la moitié des éléments de P et l'ensemble Pdroit l'autre moitié, tous les points de Pgauche étant situés à gauche de ou sur Δ, et tous les points de Pdroit étant situés à droite de ou sur Δ.

c. On résout le problème sur chacun des deux sous-ensembles Pgauche et Pdroit, les deux points les plus rapprochés sont alors :
- soit les deux points les plus rapprochés de Pgauche,
- soit les deux points les plus rapprochés de Pdroit,
- soit un point de Pgauche et un point de Pdroit.

On supposera qu'initialement l'ensemble P est trié selon des abscisses et ordonnées croissantes ; en d'autres termes l'algorithme prend la forme : PlusProche(PX, PY), où PX est un tableau correspondant à l'ensemble P trié selon les abscisses (et selon les ordonnées pour les points d'abscisses égales) et PY à l'ensemble P trié selon les ordonnées. Les ensembles P, PX et PY contiennent les mêmes points $P[i]$ (seul l'ordre change).

2. Écrivez un algorithme DiviserPoints qui calcule à partir de P les tableaux PgaucheX (resp. PgaucheY) correspondant aux points de l'ensemble Pgauche triés selon les abscisses (resp. les ordonnées), de même pour les tableaux PdroitX et PdroitY. Montrez que cette division peut être effectuée en $O(n)$ comparaisons.

3. Notons δ_g (resp. δ_d) la plus petite distance entre deux points de Pgauche (resp. Pdroit), et $\delta = \min(\delta_g, \delta_d)$. Il faut déterminer s'il existe une paire de points dont l'un est dans Pgauche et l'autre dans Pdroit, et dont la distance est strictement inférieure à δ.

a. Si une telle paire existe, alors les deux points se trouvent dans le tableau PY', trié selon les ordonnées, et obtenu à partir de PY en ne gardant que les points d'abscisse x vérifiant $l - \delta \leq x \leq l + \delta$. Donnez un algorithme qui calcule PY' en $O(n)$ comparaisons.

b. Montrez que si 5 points sont situés à l'intérieur ou sur le bord d'un carré de coté $a > 0$, alors la distance minimale entre 2 quelconques d'entre eux est strictement inférieure à a.

c. Montrez que s'il existe un point $p_g = (x_g, y_g)$ de Pgauche et un point $p_d = (x_d, y_d)$ de Pdroit tels que $\text{dist}(p_g, p_d) < \delta$ et $y_g < y_d$, alors p_d se trouve parmi les 7 points qui suivent p_g dans le tableau PY'.

d. Donnez un algorithme, en $O(n)$ comparaisons, qui vérifie s'il existe une paire de points dans Pgauche et Pdroit dont la distance est strictement inférieurs à δ, et si oui la retourne.

4. En déduire un algorithme recherchant 2 points à distance minimale dans un ensemble de n points.
5. Donnez la relation de récurrence satisfaite par le nombre de comparaisons effectuées par cet algorithme. En déduire sa complexité.

Solution:

1. On supppose que l'on dispose d'une fonction dist qui calcule la distance euclidienne entre deux points (*rappel :* $dist(p_0, p_1) = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$).

```
PlusProchesNaif(P)
Entrée: tableau de points P
Sortie: (P0, P1)
    n ← longueur(P);
    (min,P0,P1) ← (∞,0,0);
    pour i de 1 \'a n-1}
        pour j de i+1 \'a n}
            si dist(P[i],P[j]<min) alors
                (min,P0,P1) ← (dist(P[i],P[j],i,j);
            finsi
        fin pour
    fin pour
    retourne (P0, P1)
fin
```

   Cet algorithme utilise 2 boucles imbriquées sur P, donc sa complexité est $O(n^2)$.

2. On suppose que, pour les points (couples abscisse, ordonnée), on dispose des relations d'ordre $<_x$ et $<_y$ sur les abscisses et les ordonnees. On suppose également, pour cette question, qu'il n'y a pas de points confondus.

```
DiviserPoints(PX,PY)
Entrée: tableaux de points PX, PY
Sortie:(l,PGX,PDX,PGY,PDY)
    n ← longueur(PX);
    PGX ← PX[1..⌊n/2⌋];
    PDX ← PX[⌊n/2⌋ + 1..n];
    PGY ← CreerTableau [1..⌊n/2⌋];
    PDY ← CreerTableau [⌊n/2⌋ + 1..n];
    g,d ← 1;
    med ← PGX[⌊n/2⌋];
    pour i de 1 à n faire
        si PY[i] <x med ou ( PY[i] =x med et PY[i] <y med ) alors
            PGY[g] ← PY[i];
            g ← g+1;
        sinon
            PDY[d] ← PY[i];
            d ← d+1;
        finsi
    fin pour
```

```
    retourne (med.x,PGX,PDX,PGY,PDY);
fin
```

On fait un seul passage sur PY, on a donc $O(n)$ comparaisons.

3. a.
```
BandeVerticale(PY,δ,l)
Entrée: tableau de points PY, distance δ, abscisse
Sortie: PY'
    j ← 1;
    pour i de 1 à n
        si  PY[i] ≥x (l − δ, 0) et PY[i] ≤x (l + δ, 0) alors
            PY'[j] ← PY[i];
            j ← j+1;
        finsi
    fin pour
    retourne PY'
fin
```

On parcourt PY une fois et on fait deux comparaisons à chaque fois, on a bien $O(n)$ comparaisons.

b. Supposons qu'il existe 5 points à l'interieur d'un carré C de côté a et que la distance minimale entre 2 de ces points soit supérieure ou égale à a. Découpons ce carré en 4 cases comme le montre la figure qui suit. On obtient des *petits* carrés de côté $\frac{a}{2}$. La diagonale d'un petit carré est de longueur $\frac{a}{\sqrt{2}}$. 2 points situés dans le même petit carré sont donc à une distance strictement inférieure à a. Pour que la distance minimale entre 2 points du carré C soit supérieure ou égale à a, il faut donc que ces points se trouvent dans des petits carrés différents. On ne peut placer ainsi que quatre points, l'hypothèse de départ est donc absurde. □

c. Soit P_0 un point de PY'. Supposons qu'il existe un point P_1 dans PY' tel dist(P_0,P_1) $< \delta$ et $y_0 < y_1$. P_1 se trouve obligatoirement dans le rectangle R compris entre les ordonnées y_0 et $y_0 + \delta$ (voir figure suivante). Combien peut-il y avoir de points, au maximum, dans ce rectangle ? D'après la question précédente, dans le carré qui se trouve à gauche de Δ, il y en a au maximum 4. Il en va de même pour le carré à droite de Δ (voir figure suivante). Il peut donc y avoir au plus 8 points dans R, P_0 compris, donc si P_1 existe, il se trouve parmis les 7 points qui le suivent dans PY'. □

d.
```
DePartEtDautre(PY',δ)}
Entrée: tableau de points PY', distance δ
Sortie: (booléen, P0, P1)
    δmin ← δ; resultat ← (FAUX,0,0);
    pour i de 1 à longueur(PY')
        j ← i+1;
        tantque j ≤ longueur(PY') et j ≤ i+7
            si dist(PY'[i],PY'[j])< δmin alors
                δmin ← dist(PY'[i],PY'[j]);
                resultat ← (VRAI,i,j);
```

```
                finsi
                j ← j+1;
            fintantque
        fin pour
        retourne resultat;
    fin
```

On parcourt PY' une fois, etr pour chaque point, on fait 7 comparaisons. DePartEtDautre est bien $O(n)$ en nombre de comparaisons.

4.
```
PlusProches(PX,PY)}
Entrée: tableaux de points PX,PY
Sortie: (P0, P1)
    si longueur(PX)≤3 alors
        retourne PlusProcheNaif(PX);
    finsi
    (PGX,PDX,PGY,PDY) ← DiviserPoints(PX,PY);
    (PG0,PG1) ← PlusProches(PGX,PGY);
    (PD0,PD1) ← PlusProches(PDX,PDY);
    δg ← dist(PG0,PG1); δd ← dist(PD0,PD1);
    si δg < δd alors
        δ ← δg; (P0,P1) ← (PG0,PG1);
    sinon
        δ ← δd; (P0,P1) ← (PD0,PD1);
    finsi
    PY' ← BandeVerticale(PY);
    (b,P'0,P'1) ← DePartEtDautre(PY',δ);
    si b=VRAI alors
        (P0,P1) ← (P'0,P'1);
    finsi
    retourne (P0,P1);
```

5. La relation satisfaite par le nombre de comparaisons effectuées est :

$$Tn(n) = 2T(\frac{n}{2}) + O(n)$$

La complexité de cet algorithme est donc $O(n \log n)$