Licence informatique - L3 Année 2012/2013

Conception d'algorithmes et applications (LI325)

COURS 2

Résumé. Cette deuxième séance est entièrement consacrée aux applications du principe Diviser pour Régner. Nous regarderons plus particulièrement une application à l'algorithmique numérique (un algorithme de multiplication d'entiers - Karatsuba), une application à l'algorithmique des graphes (recherche d'un élément dans un arbre de recherche) et enfin une application à la géométrie algorithmique (trouver deux points les plus proches dans un nuage).

1. Diviser pour Régner (en algorithmique numérique)

Multiplication de deux entiers par la méthode de Karatsuba. Dans cette section, nous évaluerons le coût des algorithmes en nombre de multiplications élémentaires effectuées (multiplications de nombres à un chiffre). On remarque que pour le produit de deux nombres de n chiffres en utilisant la méthode naïve (celle apprise à l'école), on fait n^2 multiplications élémentaires, le coût T en calcul d'une multiplication de deux nombres à n chiffres est donc $T(n) = \Theta(n^2)$.

L'algorithme de Karatsuba (A. Karatsuba, Ofman Yu, Multiplication of multiple numbers by mean of automata, Dokadly Akad. Nauk SSSR 145, no 2, 1962, pp293-294.) est une application du principe " diviser pour régner " qui permet d'améliorer le coût des multiplications des grands nombres. Le principe en est assez simple :

Soient a et b deux nombres positifs écrits en base 2 de $n = 2k$ chiffres, on peut écrire :

$$a = (a_1 \times 2^k + a_0) \quad \text{et} \quad b = (b_1 \times 2^k + b_0),$$

avec a_0, b_0, a_1, b_1 des nombres binaires à k chiffres. On remarque alors que le calcul de ab :

$$(a_1 \times 2^k + a_0)(b_1 \times 2^k + b_0) = a_1b_1 \times 2^{2k} + (a_1b_0 + a_0b_1) \times 2^k + a_0b_0$$

ne nécessite pas les quatre produits a_1b_1, a_1b_0, a_0b_1 et a_0b_0, mais peut en fait être effectué seulement avec les trois produits a_1b_1, a_0b_0 et $(|a_1 - a_0|)(|b_1 - b_0|)$ en regroupant les calculs sous la forme suivante (on note $sgn(x)$ le signe de x) :

$$ab = a_1b_1 \times 2^{2k} + (a_1b_1 + a_0b_0 - sgn(a_1 - a_0)sgn(b_1 - b_0)(|a_1 - a_0|)(|b_1 - b_0|)) \times 2^k + a_0b_0$$

Ce que l'on peut écrire ainsi :

Diviser : On décompose a et b des nombres de $2k$ chiffres en $a = (a_1 \times 2^k + a_0)$ et $b = (b_1 \times 2^k + b_0)$ où a_0, b_0, a_1, b_1 sont des nombres à k chiffres.

Régner : On résout par appel récursif le problème pour a_0b_0, a_1b_1 et $(|a_1 - a_0|)(|b_1 - b_0|)$.

Combiner : On obtient le produit ab en faisant $(a_1 \times 2^k + a_0)(b_1 \times 2^k + b_0) = a_1b_1 \times 2^{2k} + (a_1b_1 + a_0b_0 - sgn(a_1 - a_0)sgn(b_1 - b_0)(|a_1 - a_0|)(|b_1 - b_0|)) \times 2^k + a_0b_0$.

On admet que les opérations $/2^k$ (division entière par une puissance de 2) et $\bmod 2^k$ (reste de la division entière par une puissance de 2) ne nécessitent pas de multiplication. En fait, ces opérations se font en machine en temps constants et sont négligeables. D'où le pseudo-code suivant :

Algorithme 1 : KARATSUBA

Entrées : deux entiers positifs.
Sorties : un entier.

```
1  KARATSUBA(a, b)
2  n := max(⌊log2 a⌋, ⌊log2 b⌋) + 1;
3  si n ≤ 1 alors
4  |  retourner ab;
5  sinon
6  |  k = ⌊n/2⌋;
7  |  a0 = a mod 2^k;
8  |  a1 = a/2^k;
9  |  b0 = b mod 2^k;
10 |  b1 = b/2^k;
11 |  x := KARATSUBA(a0, b0);
12 |  y := KARATSUBA(a1, b1);
13 |  z := KARATSUBA(((|a1 − a0|), (|b1 − b0|));
14 |  retourner y × 2^2k + (x + y − sgn(a1 − a0)sgn(b1 − b0)z) × 2^k + x;
```

Analyse de l'algorithme KARATSUBA

Preuve de Terminaison. Par induction sur $n = \max(\lfloor \log_2 a \rfloor, \lfloor \log_2 b \rfloor) + 1$:
- Si $n = 1$ alors, on ne fait qu'une multiplication, donc l'algorithme est fini.
- Supposons que pour tout $n < n_0$, l'algorithme KARATSUBA(a, b) se termine, alors pour tous a et b avec $n_0 = \max(\lfloor \log_2 a \rfloor, \lfloor \log_2 b \rfloor) + 1$, KARATSUBA$(a, b)$ se termine aussi car il appelle recursivement trois copies de KARATSUBA sur des instances plus petites (qui par hypothèse d'induction se terminent).

Preuve de Validité. Par induction sur $k = \max(\lfloor \log_2 a \rfloor, \lfloor \log_2 b \rfloor) + 1$:
- Si $k = 1$ alors, KARATSUBA(a, b) renvoie ab.
- Supposons que pour tout $k < n_0$, l'algorithme KARATSUBA(a, b) renvoie ab, alors pour tous a et b avec $n_0 = \max(\lfloor \log_2 a \rfloor, \lfloor \log_2 b \rfloor) + 1$:

$$\text{KARATSUBA}(a, b) = y \times 2^{2k} + (x + y - sgn(a_1 - a_0)sgn(b_1 - b_0)z) \times 2^k + x,$$

qui vaut ab car par récurrence $x = a_0 b_0$, $y = a_1 b_1$ et $z = (|a_1 - a_0|)(|b_1 - b_0|)$.

Analyse de la Complexité en nombre de multiplications élémentaires :
Notons $T(n)$ le nombre de multiplications élémentaires nécessaires pour multiplier deux nombres de n chiffres. Le principe Diviser pour Régner permet de donner la formule de récurrence pour T suivante :

$$T(1) = 1 \quad \text{et} \quad T(n) = 3T(n/2).$$

On en déduit que $T(n) = \Theta(n^{\log_2 3})$. Or $\log_2 3 = \frac{\ln(3)}{\ln(2)} \approx 1.585$, ce qui bien mieux que le n^2 de la multiplication naïve.

2. Diviser pour Régner (en algorithmique des graphes)

Arbres binaires de recherche. L'ensemble $\mathcal{AB}$ des *arbres binaires étiquetés* sur $\mathbb{N}$ est défini inductivement comme suit :
Base : $\emptyset \in \mathcal{AB}$
Induction : Si $G \in \mathcal{AB}$, $D \in \mathcal{AB}$ et $n \in \mathbb{N}$ alors $(n, G, D) \in \mathcal{AB}$
Si (n, G, D) est un arbre binaire étiqueté alors G est appelé le *sous-arbre gauche* et D le *sous-arbre droit* de (n, G, D). L'étiquette n de (n, G, D) est appelée la *clef de la racine* de (n, G, D). Plus généralement, les nombres présents dans un arbre binaire sont appelés des *clefs*.
Un *arbre binaire de recherche* est un arbre binaire étiqueté dont les étiquettes vérifient la propriété suivante : Pour tout noeud x, la clef de x est supérieure à toutes les clefs du sous-arbre gauche de x et inférieure à toutes les clefs du sous-arbre droit de x.

Le problème que l'on se pose est : étant donnés un nombre x et un arbre binaire de recherche B, x est-il une clef présente dans B ? On va suivre pour cela le principe diviser pour régner suivant :
Si B est vide, on renvoie NON sinon
Diviser : On divise B en trois parties : la racine, le sous-arbre gauche, le sous-arbre droit.
Régner : Si x est égal à la clef de la racine de B, on renvoie OUI sinon
Si x est inférieur à la clef de la racine de B, on cherche si x est une clef du sous-arbre gauche sinon on cherche x dans le sous-arbre droit.
Ceci peut s'écrire en pseudo-code de la manière suivante :

Algorithme 2 : RECHERCHE(x, B)

Entrées : Une valeur x et un arbre binaire de recherche B.
Sorties : Un booléen indiquant si x est une clef de B.

```
1 RECHERCHE(x, B)
2 si B = ∅ alors
3 |  retourner NON;
4 si x = clefracine(B) alors
5 |  retourner OUI;
6 si x < clefracine(B) alors
7 |  retourner RECHERCHE(x,sous-arbre gauche(B));
8 sinon
9 |  retourner RECHERCHE(x,sous-arbre droit(B));
```

Analyse de l'algorithme RECHERCHE

Preuve de Terminaison :
Immédiat, étant donné que l'on fait des appels sur des arbres de plus en plus petits et que si l'arbre est vide, on s'arrête.

Preuve de Validité :
Sans difficulté par induction.

Analyse de la Complexité en nombre de comparaisons :
Notons $T(h)$ le nombre de comparaisons pour trouver x dans un arbre binaire de profondeur h, on a $T(0) = 0$ et $T(h) \leq T(h-1) + 1$, en effet dans le pire des cas, on appelle récursivement RECHERCHE sur le plus profond des sous-arbres. Donc $T(h) = O(h)$. Remarquer que si l'arbre binaire de recherche est "équilibré", il peut contenir de l'ordre de 2^h clefs. RECHERCHE est donc un algorithme de recherche en temps logarithmique (si l'arbre est équilibré) et non linéaire tel que la présentation pourrait le laisser penser.

3. DIVISER POUR RÉGNER (EN GÉOMÉTRIE ALGORITHMIQUE)

Recherche des deux points les plus rapprochés dans un nuage de points dans le plan. Le problème consiste à trouver les deux points les plus proches dans un nuage de n points ($n > 1$ évidement). Il peut y avoir plusieurs points ayant les mêmes coordonnées dans ce nuage.

Bien sûr par les plus proches, l'on entend deux points dont la distance euclidienne est minimale. On rappelle à ce titre que la distance entre $p_1 = (x_1, y_1)$ et $p_2 = (x_2, y_2)$ vaut $d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

Un algorithme naïf consisterait à calculer la distance entre toutes les paires possibles de points dans L et d'en extraire le minimum. Mais il y a C_n^2 possibilités, ce qui donne un algorithme en $\Theta(n^2)$.

On va montrer qu'il existe un algorithme de type diviser pour régner qui permet de trouver les deux points les plus rapprochés en $O(n \ln(n))$.

En voici l'idée :

Diviser : Partager le tableau T en deux tableaux T_G et T_D avec $|T_D| = \lceil |T|/2 \rceil$ et $|T_G| = \lfloor |T|/2 \rfloor$. et tous les points de T_G ont une abscisse plus petite ou égale à ceux de T_D (Les points de T_G sont à gauches ou sur une droite d tandis que des points de T_D à droite ou sur d).

Régner : On résoud récursivement le problème des points rapprochés sur T_G et T_D tant qu'ils ont au moins 4 éléments, sinon on résoud naïvement.

Fusionner : Soit (A_G, B_G) (resp. (A_D, B_D)) les deux points les plus proches dans T_G (resp. dans T_D) et soit (C, D) les deux points les plus proches avec C dans T_G et D dans T_D. On retourne parmis ces trois paires, celle qui a la distance la plus petite.

Il nous faut donc pour fusionner, un algorithme spécifique pour calculer si la distance minimale entre T_G et T_D est plus petite que celles dans T_G et T_D. On va voir qu'il en existe un efficace.

Structure de données : On va utiliser deux tableaux TX et TY, le premier contient les points classés par ordre croissant de leur abscisse (plus précisément, par ordre lexicographique sur les coordonnées (x, y)). Le deuxième par ordre croissant sur les ordonnées (plus précisément, par ordre lexicographique sur les coordonnées inversées (y, x)). Donc, on stoque deux fois trop de données (chaque point est présent dans les deux tableaux), mais cette structure va nous permettre d'obtenir un algorithme en $O(n \ln(n))$.

Tout d'abord, voici le pseudo-code qui permet de diviser les deux tableaux TX et TY :

Algorithme 3 : SeparePoints

Entrées : les tableaux de points TX , TY.
Sorties : les tableaux scindés T_GX , T_DX , T_GY , T_DY et l'abscisse de la droite d qui sépare les points.

```
1  SeparePoints (TX, TY)
2  n := longueur(TX);
3  T_GX := TX[1..⌊n/2⌋];
4  T_DX := TX[⌊n/2⌋ + 1..n];
5  T_GY := CreerTableau[1..⌊n/2⌋];
6  T_DY := CreerTableau[⌊n/2⌋ + 1..n];
7  g := 1; d := 1;
8  med := T_GX[⌊n/2⌋];
9  pour i allant de 1 à n faire
10     si TY[i]_x < med_x ou (TY[i]_x = med_x et TY[i]_y <= med_y et g <= ⌊n/2⌋) alors
11         T_GY[g] := TY[i] ; g := g + 1 ;
12     sinon
13         T_DY[d] := TY[i]; d := d + 1;
14 retourner (med_x, T_GX, T_DX, T_GY, T_DY)
```

Analyse du pseudo-code SeparePoints

Preuve de Terminaison :
Il n'y a qu'une boucle définie.

Preuve de Validité :
Exercice facile.

Analyse de la Complexité en nombre de comparaisons :
On fait au plus deux comparaisons à chaque passage dans la boucle **Pour** de la ligne 9. On a donc un algorithme linéaire en la taille de l'entrée.

Soit δ_G (resp. δ_D) la plus petite distance entre deux points de T_G (resp. T_D), notons δ le minimum de δ_G et δ_D. Pour fusionner, il nous faut déterminer s'il existe une paire de points dont l'un est dans T_G et l'autre dans T_D, et dont la distance est strictement inférieure à δ. Il est facile de voir que si une telle paire existe, alors les deux points se trouvent dans la bande verticale centrée en $x = med_x$ et de largeur 2δ.

Nous allons donc devoir construire un tableau TY', trié selon les ordonnées (nous verrons en suite pourquoi ce tri), et obtenu à partir de TY en ne gardant que les points d'abscisse x vérifiant $med_x - \delta \leq x \leq med_x + \delta$. En voici le pseudo-code :

Algorithme 4 : BandeVerticale

Entrées : le tableau de points TY , la distance δ et l'abscisse med_x de la droite d.
Sorties : le tableau TY'.

```
1 BandeVerticale(TY, δ, med_x)
2 n := longueur(TX);
3 j := 1;
4 pour i allant de 1 à n faire
5     si TY[i]_x ≥ med_x − δ et TY[i]_x ≤ med_x + δ alors
6         TY'[j] := TY[i];
7         j := j + 1;
8 retourner TY'
```

Analyse du pseudo-code BandeVerticale

Preuve de Terminaison :
Il n'y a qu'une boucle définie.

Preuve de Validité :
Immédiat.

Analyse de la Complexité en nombre de comparaisons :
On fait au plus deux comparaisons à chaque passage dans la boucle **Pour** de la ligne 4. On a donc un algorithme linéaire en la taille de l'entrée.

Nous sommes presque en mesure de donner un pseudo-code efficace pour déterminer s'il existe une paire de points dont l'un est dans T_G et l'autre dans T_D, et dont la distance est strictement inférieure à δ. Les deux remarques restant à faire sont :

- S'il existe deux points $p_1 = (x_1, y_1)$ et $p_2 = (x_2, y_2)$ de TY' tels que $dist(p_1, p_2) < \delta$ et $y_1 < y_2$, alors p_2 se trouve dans le rectangle $R = \{(x, y); med_x - \delta \leq x \leq med_x + \delta \text{ et } y_1 \leq y \leq y_1 + \delta\}$.
- R ne peut contenir au plus que les 7 points qui suivent p_1 dans le tableau TY'.

Ceci nous permet d'écrire le pseudo-code suivant :

Algorithme 5 : DePartEtDAutre

Entrées : le tableau de points TY' , la distance δ.
Sorties : Un quadruplet (b, P_1, P_2, δ) où b est un booléen qui vaut faux s'il n'y a pas de points dans TY' à distance inférieur à δ et sinon P_1 et P_2 sont les coordonnées de deux points les plus proches dans TY' et δ leur distance.

```
1  DePartEtDAutre(TY', δ)
2  n :=longueur(TY');
3  δ_min := δ;
4  res := (FAUX, (0,0), (0,0), 0);
5  pour i allant de 1 à n faire
6      j := i + 1;
7      tant que j ≤ n et j ≤ i + 7 faire
8          d := dist(TY'[i], TY'[j]);
9          si d < δ_min alors
10             δ_min := d;
11             res := (Vrai, TY'[i], TY'[j], δ_min);
12         j := j + 1;
13 retourner res
```

Analyse du pseudo-code DePartEtDAutre

Preuve de Terminaison :
Il y a une boucle définie ligne 5 et emboîté dedans un **tant que** ligne 7, mais celui-ci n'ai jamais itéré plus de 7 fois à cause du compteur j. Donc l'algorithme se termine.

Preuve de Validité :
Immédiat si l'on est d'accord avec les remarques.

Analyse de la Complexité en nombre de comparaisons :
On parcourt TY' une fois, et pour chaque point, on fait au plus 7 comparaisons. On a donc un algorithme linéaire en la taille de l'entrée.

Nous pouvons finalement ecrire un pseudo-code pour la recherche de deux points les plus rapprochés (voir Algorithme 6 sur la page suivante).

Analyse du pseudo-code PlusProches

Preuve de Terminaison :
Tant que la taille des tableaux est supérieur à 4, elle diminue de moitié à chaque appel recursif et dès qu'elle est inférieur à 4, l'algorithme s'arête en lançant PlusProcheNaïf qui n'est pas recursif. Donc l'algorithme se termine.

Preuve de Validité :
Immédiat.

Analyse de la Complexité en nombre de comparaisons :
Le nombre de comparaisons effectuées par l'algorithme suit la relation de récurence $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$. Le $O(n)$ vient du fait que DiviserPoints, BandeVerticale et DePartEtDAutre se font tous les 3 en $O(n)$. Les $T(\lfloor n/2 \rfloor)$ et $T(\lceil n/2 \rceil)$ décrivent seulement les appels recursifs ligne 6 et 7. Donc, on a une complexité en nombre de comparaisons qui est en $O(n \ln(n))$.

Algorithme 6 : PlusProches

Entrées : les tableaux de points TX, TY.

Sorties : Deux points à distance minimum et δ la distance minimum.

```
1  PlusProches(TX, TY)
2  si longueur(TX) < 4 alors
3  |  retourner PlusProcheNaïf(TX, TY);
4  sinon
5  |  (med_x, T_G X, T_D X, T_G Y, T_D Y) := DiviserPoints(TX, TY);
6  |  (p_G 0, p_G 1, δ_1) := PlusProches(T_G X, T_G Y);
7  |  (p_D 0, p_D 1, δ_2) := PlusProches(T_D X, T_D Y);
8  |  si δ_1 < δ_2 alors
9  |  |  (p_0, p_1, δ) := (p_G 0, p_G 1, δ_1);
10 |  sinon
11 |  |  (p_0, p_1, δ) := (p_D 0, p_D 1, δ_2);
12 |  TY' := BandeVerticale(TY, δ, med_x);
13 |  (b, p'_0, p'_1, δ') := DePartEtDautre(TY', δ);
14 |  si b = VRAI alors
15 |  |  (p_0, p_1, δ) := (p'_0, p'_1, δ');
16 |  retourner (p_0, p_1, δ);
```

Le prétraitement qui consiste à créer les tableaux triés TX et TY se fait aussi en $O(n \ln(n))$ comparaisons avec un algorithme de tri comme tri fusion par exemple.