Corrigé

# Examen d'algorithmique

EPITA ING1 2014 S1; A. DURET-LUTZ

Durée : 1h30

janvier 2012

## La paire la plus proche (17 points)

On considère un ensemble de points du plan euclidien, numérotés de 0 à $n-1$. Chaque point $p$ est représenté par une paire de nombres flottants (ses coordonnées) et ces $n$ points sont rangés dans un tableau $P$ de taille $n$. Ainsi $P[i].x$ est l'abscisse du $i^e$ point, tandis que $P[i].y$ est son ordonnée.

Notre but est de trouver la plus petite distance existant entre deux points de cet ensemble c'est-à-dire le valeur de $\min_{i \neq j} \sqrt{(P[i].x - P[j].x)^2 + (P[i].y - P[j].y)^2}$.

Voici un algorithme naïf calculant cette plus petite distance en considérant toutes les paires de points :

```
1  MINDIST1(P, n)
2     d ← ∞
3     for i ← 0 to n − 1
4        for j ← 0 to n − 1
5           if i ≠ j then
6              d ← min(d, √((P[i].x − P[j].x)² + (P[i].y − P[j].y)²))
7     return d
```

1. **(1pt)** Combien de fois la ligne 6 est-elle exécutée ? Donnez une réponse précise en fonction de $n$.

*Réponse :*

il y a $n^2$ paires $(i, j)$ dont $n$ paires de la forme $(i, i)$ a retirer. Donc la ligne 6 est exécutée $n^2 - n = n(n-1)$ fois.

2. **(1pt)** Donnez la complexité de MINDIST1. (En fonction de $n$ toujours.)

*Réponse :*

$\Theta(n^2)$

3. **(1pt)** Comment pourrait-on très simplement diviser par 2 le nombre d'exécutions de la ligne 6 ?

*Réponse :*

Puisque la distance de $P[i]$ à $P[j]$ est la même que celle de $P[j]$ à $P[i]$, il est inutile de tester ces deux distances. On peut se restreindre au cas où $i < j$ (la restriction inverse, $i > j$, serait correcte aussi).

```
1  MINDIST(P, n)
2     d ← ∞
3     for i ← 0 to n − 1
4        for j ← i + 1 to n − 1
5           d ← min(d, √((P[i].x − P[j].x)² + (P[i].y − P[j].y)²))
6     return d
```

Cela fait $n(n-1)/2$ appels à min.

4. **(1pt)** Comment pourrait-on faire pour que la fonction racine carrée ne soit appelée qu'une seule fois dans tout l'algorithme ?

*Réponse :*

On ne stocke que des distances au carré, et on fait la racine une fois pour toute à la fin.

```
1  MINDIST(P, n)
2      d ← ∞
3      for i ← 0 to n − 1
4          for j ← i + 1 to n − 1
5              d ← min(d, (P[i].x − P[j].x)² + (P[i].y − P[j].y)²)
6      return √d
```

5. **(1pt)** Quelle est l'influence des deux dernières optimisations sur la classe de complexité de l'algorithme ?

*Réponse :*

Aucune. La complexité est toujours dans $\Theta(n^2)$ même si le la troisième variante de l'algorithme est plus rapide que les deux précédentes.

Considérons maintenant un algorithme "diviser pour régner". (A) Tout d'abord le plan est découpé par une ligne verticale en deux régions : la moitié des points à gauche et l'autre moitié à droite de la frontière. (B) On calcule ensuite récursivement la plus petite distance dans chacune de ces deux régions : cela nous donne deux distances $d_l$ (*left*) et $d_r$ (*right*), mais il se peut que nous ayons manqué la plus petite distance si elle est à cheval sur la frontière comme dans la FIG. 1. (C) On compare donc $d_l$ et $d_r$ avec les distances des couples à cheval sur la frontière : une étape délicate à réaliser efficacement.

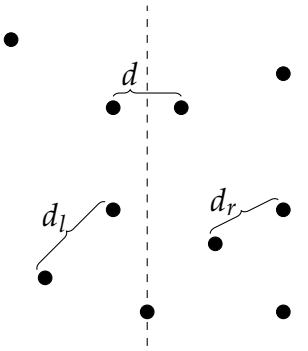


FIGURE 1 – Cas où la plus petite distance est à cheval sur la séparation.

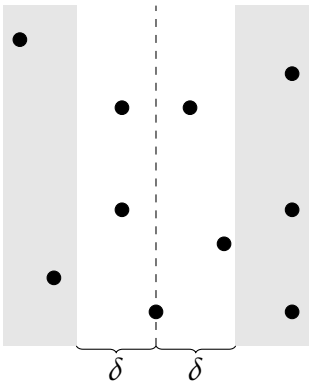


FIGURE 2 – On élimine les points qui sont à une distance supérieure à $\delta = \min(d_l, d_r)$.

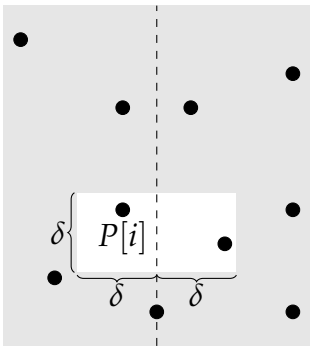


FIGURE 3 – Les points à comparer avec $P[i]$ sont dans un rectangle de taille $(2\delta, \delta)$.

Reprenons ces trois étapes en détail :

(A) Le découpage du tableau $P$ en deux parties est simple si les points on été préalablement triés par abscisses croissantes : il suffit alors de placer la frontière à l'abscisse du point médian : tous les points $P[0], P[1], \ldots P[\lfloor n/2 \rfloor]$ sont à gauche, et tous les points $P[\lfloor n/2 \rfloor + 1], \ldots, P[n-1]$ sont à droite.

(B) Les appels récursifs qui calculeront des distances sur ces deux moitiés de tableaux devront les diviser à leur tour : on va donc devoir passer en paramètre les indices $l$ et $r$ des bornes gauche et droite du sous-tableau considéré, comme ce que nous avons fait dans QUICKSORT en cours. La récursion se termine lorsqu'on demande de calculer la plus petite distance d'un tableau ne contenant qu'un point.

(C) Une fois que l'on connaît $d_l$ et $d_r$, on sait que la plus petite distance est au plus $\delta = \min(d_l, d_r)$ : elle peut être plus petite si deux points de part et d'autre de la frontière sont plus proches. Pour vérifier on pourrait calculer les distances de tous les points de gauche à tous les points de droite, mais ce serait inefficace. En fait, la valeur de $\delta$ nous permet déjà d'éliminer tous les points du tableau qui sont à

une distance de la frontière supérieure à $\delta$. Nous ne devons donc considérer que les points sur fond blanc de la FIG. 2. Prenons maintenant un point $P[i]$ de cette bande blanche : s'il existe un point tel que sa distance à $P[i]$ est inférieure à $\delta$, alors il existe forcément un rectangle de taille $(2\delta, \delta)$ qui entoure ces deux points et est centré sur la frontière (FIG. 3). Combien de points ce rectangle peut-il contenir ? Le carré de gauche ne peut pas contenir plus de 4 points (un à chaque coin) autrement la distance $d_l$ serait plus petite que ce que nous avons calculée. De même le carré de droite ne peut contenir plus de 4 points.[1] Cela nous fait donc un maximum de 8 points dans le rectangle autour de $P[i]$. Revenons maintenant à la bande blanche de la FIG. 2 : si nous trions tous les points de cette bande blanche par rapport à leurs ordonnées, ils est maintenant clair que pour chaque point nous ne devons que calculer les distances par rapport aux 7 points qui le suivent (dans l'ordre des ordonnées).

Voici l'algorithme correspondant :

```
 1  MINDISTREC(P, l, r)
 2     if r ≤ l then
 3        return +∞
 4     else
 5        m ← l + ⌊(r − l)/2⌋
 6        d_l ← MINDISTREC(P, l, m)
 7        d_r ← MINDISTREC(P, m + 1, r)
 8        δ ← min(d_l, d_r)
 9        k ← 0
10        for i ← l to r
11           if |P[i].x − P[m].x| < δ then
12              Q[k] ← P[i]
13              k ← k + 1
14        R ← sort Q (array of size k) according to y-coordinates
15        for i ← 0 to k − 1
16           for j ← i + 1 to min(i + 7, k − 1)
17              δ ← min(δ, √((R[i].x − R[j].x)² + (R[i].y − R[j].y)²))
18        return δ
19
20  MINDIST2(P, n)
21     P ← sort P according to x-coordinates
22     return MINDISTREC(P, 0, n − 1)
```

6. **(0.5pt)** Quel algorithme de tri proposez-vous d'utiliser aux lignes 21 et 14 ?

*Réponse :*

Un tri introspectif par exemple. (D'autres choix de tri comparatifs étaient possible.)

7. **(1pt)** Quelle est la complexité de la ligne 21 en fonction de $n$ pour l'algorithme de tri que vous avez choisi.

*Réponse :*

$\Theta(n \log n)$

8. **(1pt)** Quelle valeur faut-il retourner ligne 3 ? **Répondez directement sur l'algorithme.**

9. **(1pt)** Si on note $n = r + 1 - l$, quelle valeur maximale peut prendre $k$ (la taille de $Q$) lorsque la ligne 14 est exécutée ?

1. Dans ce scénario deux des points de gauche sont superposés avec deux des points de droite, ce qui est possible si le tableau $P$ comporte des doublons.

*Réponse :*

Au pire tous les points sont dans la bande blanche et $k = n$.

10. **(1pt)** En déduire la complexité de la ligne 14.

*Réponse :*

$O(n \log n)$

11. **(1.5pt)** Donnez une définition récursive de la complexité de MINDISTREC en fonction de la taille $n = r + 1 - l$ du tableau manipulé. Pour simplifier, ignorez les parties entières et supposez que les deux appels récursifs lignes 6 et 7 se font sur des tableaux de même taille.

*Réponse :*

$T(n) = 2T(n/2) + O(n \log n)$

12. **(3pts)** Quelle est la solution de l'équation précédente. (Indice : si le théorème général ne s'applique pas, appliquez la méthode par réécritures successives sans vous décourager—simplifiez les log et cherchez à utiliser la formule des $\sum kx^k$.)

*Réponse :*

$a = b = 2$ on doit comparer $O(n \log n)$ avec $n^{\log_2(2)} = n$. Malheureusement le théorème ne s'applique pas, car on ne peut pas trouver de $\varepsilon > 0$ tel que $n \log n = \Omega(n^{1+\varepsilon})$. On dit que $n \log n$ n'est pas "polynomialement plus grand que" $n$.
Fonctionnons donc par réécritures successives.

$$\begin{aligned} T(n) &= 2T(n/2) + O(n \log n) \\ &\leq cn \log n + 2T(n/2) \\ &\leq cn \log n + cn \log(n/2) + 4T(n/4) \\ &\leq cn \log n + cn \log(n/2) + \cdots + cn \log(n/2^i) + 2^i T(n/2^i) \end{aligned}$$

Les réécritures s'arrêtent lorsque $n/2^i = 1$ soit $i = \log n$. Alors $T(1) = \Theta(1)$.

$$\begin{aligned} &\leq \Theta(1) + \sum_{i=1}^{\log n} cn \log(n/2^i) \\ &\leq \Theta(1) + cn \sum_{i=1}^{\log n} (\log(n) - i) \\ &\leq \Theta(1) + cn \log(n) \log(n) - cn \sum_{i=1}^{\log n} i \\ &\leq \Theta(1) + cn \log(n) \log(n) - cn \frac{(1 + \log n)(\log n)}{2} \\ &\leq \Theta(n \log^2 n) \\ T(n) &= O(n \log^2 n) \end{aligned}$$

L'algorithme précédent n'est pas optimal à cause du tri ligne 14. Une méthode plus efficace consiste à garder dans un tableau une copie de tous les points "pré-triés" par rapport aux ordonnées, et d'utiliser ce tableau lors du filtrage des points lignes 9–13.

Cela donne l'algorithme suivant :

```
1  MINDISTREC'(P, Q, l, r)
2      if r ≤ l then
```

```
3         return +∞
4       else
5         m ← l + ⌊(r − l)/2⌋
6         d_l ← MinDistRec(P, Q, l, m)
7         d_r ← MinDistRec(P, Q, m + 1, r)
8         δ ← min(d_l, d_r)
9         k ← 0
10        for i ← 0 to n − 1
11          if |Q[i].x − P[m].x| < δ then
12            R[k] ← Q[i]
13            k ← k + 1
14        for i ← 0 to k − 1
15          for j ← i + 1 to min(i + 7, k − 1)
16            δ ← min(δ, √((R[i].x − R[j].x)² + (R[i].y − R[j].y)²))
17        return δ
18
19  MinDist3(P, n)
20      P ← sort P according to x-coordinates
21      Q ← sort P according to y-coordinates
22      return MinDistRec'(P, Q, 0, n − 1)
```

13. **(2pts)** La complexité $T(n)$ de l'algorithme MinDistRec' présenté ci-dessus satisfait l'équation $T(n) = 2T(n/2) + \Theta(n)$. Quelle est la solution de cette equation ?

*Réponse :*

$\Theta(n \log n)$ c'est la même équation que le tri fusion.

14. **(1pt)** Déduisez-en la complexité de MinDist3.

*Réponse :*

Les deux tris se font en $\Theta(n \log n)$, l'appel à MinDistRec' aussi. La complexité totale est donc $\Theta(n \log n)$.

# La plus longue sous-séquence commune (9 points)

Le problème de la PLSSC est celui-ci :

Entrée : deux chaînes, p.ex. `"loutre"` et `"troupe"`.
Sortie : une plus longue sous-séquence commune, `"oue"` ou `"tre"`.

Une *sous-séquence* d'une chaîne $S$ est une suite de lettres de $S$ qui ne sont pas forcément consécutives dans $S$ mais doivent apparaître dans le même ordre. Une sous-séquence peut être vide.

## Approche bovine (4.5 points)

1. **(1pt)** Si une chaîne possède $n$ lettres (supposées toutes différentes), combien peut-on construire de sous-séquences différentes ?

*Réponse :*

Pour chaque lettres on doit décider si on la met dans la sous-séquence construite. Ce la fait donc $2^n$ choix possible pour construire une sous-séquence, donc $2^n$ sous-séquences.

2. **(1pt)** Parmi ces sous-séquences combien en existe-t-il de taille $k$, pour $0 \leq k \leq n$.

*Réponse :*

On doit choisir $k$ lettres parmi $n$, il y a $C_n^k$ combinaisons possibles.

3. **(1pt)** Étant donné une chaîne $S$ de taille $k$, et une chaîne $C$ de taille $m$, quelle est la complexité de tester si $S$ est une sous-séquence de $C$ ? (Donnez une formule en fonction de $k$ et $m$.)

*Réponse :*

Une façon simple de faire est de parcourir $S$ en se décalant dans $C$ pour y chercher les lettres de $S$ au fur et à mesure.

```
1   TESTSUBSEQ(C, m, S, k)
2       j ← −1
3       for i ← 0 to k − 1
4           do
5               j ← j + 1
6               if k − i > m − j then return false
7           until C[j] = S[i]
8       return true
```

L'indice $i$ parcourt la chaîne $S$ tandis que l'indice $j$ parcourt la chaîne $C$. Pour chaque nouvelle lettre de $S[i]$, on avance $j$ jusqu'à trouver une position telle que $C[j] = S[i]$.
Évidement, il est inutile de continuer si le nombre de lettres restant dans $S$ ($m - j$) est inférieur au nombre de lettres restant dans $C$ ($k - i$). C'est le rôle de la ligne 5.
Dans ces conditions la ligne 5 s'exécute au pire $m + k$ fois et l'algorithme est en $O(m + k)$.
Cependant il est clair que si $k$ est plus grand que $m$, l'algorithme s'arrête en $\Theta(1)$ dès son premier passage ligne 6. Dans les autres cas, si $k \leq m$, on a $O(m + k) = O(m)$.
On peut donc conclure que la complexité de cet algorithme est $O(m)$.

4. **(2.5pt)** En déduire la complexité d'un algorithme qui énumère toutes les sous-séquences de la première chaîne (de taille $n$) puis teste chacune pour voir si c'est une sous-séquence de la seconde chaîne (de taille $m$). Vous supposerez $n \leq m$.

*Réponse :*

Pour chaque taille $k$, il y a $C_n^k$ sous-séquences qui prennent chacune $\Theta(\min(k, m)) = \Theta(k)$ opérations car $k \leq n \leq m$.
On a donc :

$$T(n) = \sum_{k=0}^{n} C_n^k \Theta(k) = \Theta\left(\sum_{k=0}^{n} k C_n^k\right) = \Theta(n2^{n-1}) = \Theta(n2^n)$$

Note : Voici une façon de montrer que $\sum_{k=0}^{n} kC_n^k = n2^{n-1}$. On part de la série de Taylor $(1+x)^n = \sum_{k=0}^{n} x^k C_n^k$. En dérivant cette équation par rapport à $x$ on obtient $n(1+x)^{n-1} = \sum_{k=0}^{n} kx^{k-1} C_n^k$. On pose maintenant $x = 1$ pour avoir $n2^{n-1} = \sum_{k=0}^{n} kC_n^k$.

## Programmation Dynamique (3.5 points)

Notons $X = x_1 x_2 \cdots x_n$ et $Y = y_1 y_2 \cdots y_m$ les chaînes d'entrée, et $Z = z_1 z_2 \cdots z_k$ une PLSSC. Notons de plus $X_i$ le $i^{\text{e}}$ préfixe de $X$, c.-à-d. $X_i = x_1 x_2 \cdots x_i$. On a :

— $x_n = y_m \implies z_k = x_n = y_m$ et $Z_{k-1}$ est une PLSSC de $X_{n-1}$ et $Y_{m-1}$
— $x_n \neq y_m \wedge z_k \neq x_n \implies Z$ est une PLSSC de $X_{n-1}$ et $Y$
— $x_n \neq y_m \wedge z_k \neq y_m \implies Z$ est une PLSSC de $X$ et $Y_{m-1}$

Le problème a donc une sous-structure optimale.

Définissons récursivement $C[i, j]$, la longueur de la plus longue sous-séquence commune à $X_i$ et $Y_j$ :

$$C[i,j] = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0 \\ C[i-1,j-1]+1 & \text{si } i,j > 0 \text{ et } x_i = y_i \\ \max(C[i,j-1], C[i-1,j]) & \text{si } i,j > 0 \text{ et } x_i \neq y_i \end{cases}$$

1. **(1.5pt)** Quelle est la complexité d'un algorithme de programmation dynamique qui calculerait $C[n,m]$ ?

*Réponse :*

$\Theta(nm)$, la taille du tableau $C$ puisque toutes les cases se calculent en temps constant.

2. **(2pts)** Comment trouver la plus longue sous-séquence commune (et non uniquement sa taille) à partir du tableau $C$ ?

*Réponse :*

On calcule la PLSSC en partant de la fin, à reculons :

```
PLSSC(C, n, m)
    (i, j) ← (n, m)
    s ← ""
    while i > 0 and j > 0 :
        if x_i = y_j then
            s ← x_i · s
            (i, j) ← (i − 1, j − 1)
        else if C[i, j − 1] ≤ C[i − 1, j] then
            j ← j − 1
        else
            i ← i − 1
    return s
```