

DM 1

Informatique commune, 2ème année

I Approche exhaustive

1. (/1) Écrire une fonction `distance(i, j)` renvoyant la distance entre les points M_i et M_j . On utilisera la fonction `sqrt` après l'avoir importée.

Solution :

```
1  from math import sqrt
2
3  def distance(i, j):
4      return sqrt((coords_x[i] - coords_x[j])**2 + (coords_y[i] - coords_y[j])**2)
```

2. (/1.5) Rappeler sommairement comment sont stockés les flottants en mémoire. Quelle conséquence cela peut-il avoir sur le calcul de la distance ? On ignorera par la suite les problèmes d'approximation.

Solution : En Python, un flottant x est stocké dans la mémoire RAM avec 64 bits en utilisant la norme IEEE754, qui met x sous la forme scientifique $x = \pm 1, m_2 \times 2^e$ et utilise :

- 1 bit pour le signe
- 12 bits pour l'exposant e (stocké en valeur absolue avec un biais)
- 52 bits pour la mantisse m (chiffres après la virgule)

Conséquences : Les éventuels chiffres après la virgule au delà du 52ème sont oubliés. Les calculs sur les flottants peuvent donc induire des erreurs d'approximation.

3. (/1.5) Écrire une fonction `plus_proche` qui renvoie, à l'aide d'une recherche exhaustive, le couple d'entiers des indices i et j des deux points les plus proches du nuage de points.

Solution :

```
1  def plus_proche():
2      i_min, j_min = 0, 1
3      for i in range(len(coords_x)):
4          for j in range(i): # pour que i soit différent de j
5              if distance(i, j) < distance(i_min, j_min):
6                  i_min, j_min = i, j
7      return i_min, j_min
```

4. (/0.5) Donner, en la justifiant sommairement, la complexité de la fonction précédente en fonction de n .

Solution : `distance` est en complexité $O(1)$ (indépendant de n). Donc les deux boucles `for` imbriquées de `plus_proche` donnent une complexité $O(n^2)$.

II Quelques outils pour s'améliorer

On souhaite maintenant obtenir la distance entre les deux points les plus proches avec une meilleure complexité. Pour cela nous allons décrire un algorithme utilisant une méthode de type diviser pour régner. Cette partie introduit des fonctions utiles pour la mise en œuvre de cet algorithme.

On se donne la fonction suivante :

```
1  def tri(liste):
2      n = len(liste)
3      for i in range(n):
4          pos = i
5          while pos > 0 and liste[pos] < liste[pos-1]:
6              liste[pos], liste[pos-1] = liste[pos-1], liste[pos]
7              pos -= 1
```

5. (/1.5) Que renvoie cette fonction ? Que fait-elle ? Le démontrer soigneusement en exhibant un invariant de boucle.

Solution : Comme il n'y a pas de `return`, cette fonction renvoie `None`. La fonction trie la liste en argument en utilisant le tri par insertion. On peut montrer que la liste est bien triée à la fin en utilisant l'invariant de boucle suivant :

H_n : au début l'itération i de la boucle `for`, les i premiers éléments de la liste sont triés par ordre croissant.

- H_0 est vraie car il y a alors 0 élément.
- Supposons H_i et considérons l'itération i du `for`. Les i premiers éléments de `liste` sont donc triés. Le `while` va échanger `liste[i]` avec son prédécesseur jusqu'à le mettre à sa bonne place. Les $i + 1$ premiers éléments de `liste` vont donc être triés à l'itération suivante. Donc H_{i+1} est vraie.

Par principe de récurrence, H_i est vraie pour tout $i \in \{0, \dots, n\}$. Donc d'après H_n , quand la boucle `for` se termine, la liste entière est triée.

6. (/1) Donner, en la démontrant, la complexité de la fonction `tri` en fonction de la taille de la liste donnée en paramètre.

Solution : On passe au plus i fois dans la boucle `while` puisque `pos` commence à i , diminue de 1 à chaque itération, et que la boucle s'arrête quand `pos` devient nul. Donc le `while` est en $O(i)$. Comme ce `while` est répété pour $i = 0, 1, \dots, n - 1$, la complexité totale est :

$$\sum_{i=0}^{n-1} O(i) = O\left(\sum_{i=0}^{n-1} i\right) = O\left(\frac{n(n-1)}{2}\right) = \boxed{O(n^2)}$$

7. (/0.5) On souhaite trier une liste contenant des indices de points suivant l'ordre des abscisses croissantes. Que faudrait-il changer à la fonction `tri` ci-dessus pour qu'elle réalise cette opération ?

Solution : Il faudrait remplacer `liste[pos]` par `coords_x[liste[pos]]` dans le `while`.

8. (/1) Indiquer le nom d'un autre algorithme de tri plus efficace dans le pire des cas, ainsi que sa complexité. On ne demande pas de le programmer.

Solution : Le tri fusion a une meilleure complexité dans le pire cas : $\boxed{O(n \log(n))}$.

On admettra que l'on dispose de deux listes de n entiers `liste_x` (resp. `liste_y`) contenant les indices des points du nuage triés par abscisses croissantes (resp. par ordonnées croissantes). On supposera désormais que deux points quelconques ont des abscisses et des ordonnées distinctes.

Dans toute la suite, un sous-ensemble de points sera décrit par un cluster. Un cluster est une matrice de deux lignes contenant chacune les mêmes numéros correspondant aux numéros des points dans le sous-ensemble considéré. Dans la première ligne, les points sont triés par abscisses croissantes ; dans la seconde, ils sont triés par ordonnées croissantes. La figure 1 donne la représentation de deux clusters.

Pour être efficace, notre algorithme ne doit pas re-trier les listes des indices de points à chaque étape. Nous allons donc définir une fonction qui permet d'extraire des indices d'un cluster et former ainsi un nouveau cluster plus petit.

9. (/2) Écrire une fonction `sous_cluster(c1, x_min, x_max)` qui prend en arguments un cluster `c1` et deux flottants `x_min` et `x_max`, et renvoie le sous-cluster des points dont l'abscisse est comprise entre `x_min` et `x_max` (au sens large). Cette fonction doit avoir une complexité linéaire en la taille du cluster.

Solution : On parcourt une fois le cluster, ce qui donne bien une complexité linéaire :

```
1 def sous_cluster(cl, x_min, x_max):  
2     c = [], []  
3     for i in cl[0]:  
4         if x_min <= coords_x[i] <= x_max:  
5             c[0].append(i)  
6     for i in cl[1]:  
7         if x_min <= coords_x[i] <= x_max:  
8             c[1].append(i)  
9     return c
```

10. (/1.5) Écrire une fonction `mediane(cl)` qui prend en entrée un cluster `cl` contenant au moins 2 points et renvoie une abscisse médiane, c'est-à-dire que la moitié (au moins) des points a une abscisse inférieure ou égale à cette valeur, et la moitié (au moins) des points a une abscisse supérieure ou égale à cette valeur. Cette fonction doit avoir une complexité en $O(1)$.

Solution :

```
1 def mediane(cl):  
2     return coords_x[cl[0][len(cl[0])//2]]
```

III Méthode sophistiquée

Le fonctionnement de l'algorithme utilisant une méthode de type diviser pour régner est illustré par la figure 2 :

- Si le cluster contient deux ou trois points, on calcule la distance minimale en calculant toutes les distances possibles.
 - Sinon, on sépare le cluster en deux parties G et D qu'on supposera de tailles égales (éventuellement à un point près) suivant la médiane des abscisses, qu'on notera x_0 .
 - Les deux points les plus proches sont soit tous les deux dans G , soit tous les deux dans D , soit un dans G et un dans D .
 - On calcule récursivement le couple le plus proche dans G et le couple le plus proche dans D . On note d_0 la plus petite des deux distances obtenues.
 - On cherche s'il existe une paire de points (M_1, M_2) telle que M_1 est dans G , M_2 dans D , et $d(M_1, M_2) < d_0$.
 - Si on en trouve une (ou plusieurs), on renvoie la plus petite de ces distances. Sinon, on renvoie d_0 .
11. (/1) Écrire une fonction `gauche(cl)` qui prend en argument un cluster `cl` contenant au moins deux points et renvoie le cluster constitué uniquement de la moitié (éventuellement arrondie à l'entier supérieur) des points les plus à gauche du cluster `cl`.

Solution :

```
1 def gauche(cl):  
2     return sous_cluster(cl, 0, median(cl))
```

On suppose qu'on dispose d'une fonction `droite(cl)` qui renvoie le cluster contenant tous les autres points du cluster `cl` n'appartenant pas au cluster renvoyé par la fonction `gauche(cl)`.

12. (/1) Justifier que l'on peut se contenter de chercher les points M_1 et M_2 de l'étape 5 de l'algorithme dans l'ensemble des points dont l'abscisse appartient à $I_0 = [x_0 d_0, x_0 + d_0]$.

Solution : Notons x_1 et x_2 les abscisses de M_1 et M_2 . Supposons par exemple que M_1 est dans G (donc $x_1 \leq x_0$), que M_2 est dans D mais que $x_2 > x_0 + d_0$. Alors la distance entre M_1 et M_2 serait supérieure à $x_2 - x_1 > x_0 + d_0 - x_0 = d_0$. M_1 et M_2 ne peuvent donc pas être les deux points les plus proches (puisque'il y a deux autres points à distance d_0).

13. (/1) Écrire une fonction `bande_centrale(c1, d0)` qui prend en argument un cluster `c1` et un réel `d0`, et renvoie le cluster des points dont l'abscisse est dans `I0`. Cette fonction doit avoir une complexité linéaire en la taille du cluster.

```

1  def bande_centrale(c1, d0):
2      x0 = mediane(c1)
3      return sous_cluster(c1, x0-d0, x0+d0)

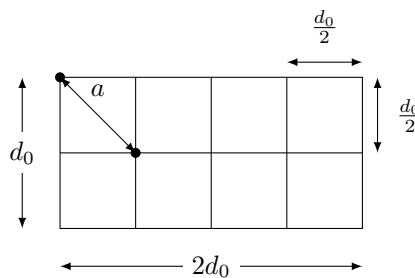
```

14. (/3) Montrer que deux points M_1 et M_2 (de l'étape 5 de l'algorithme) situés à une distance inférieure à d_0 se trouvent, dans la deuxième ligne du cluster (c'est-à-dire la ligne triée par ordonnées croissantes), séparés d'au plus 6 éléments. On pourra montrer par l'absurde qu'un rectangle, à préciser, de dimensions $2d_0 \times d_0$ contient au plus 8 points.

Solution : Soient y_1 et y_2 les ordonnées de M_1 et M_2 . On suppose sans perte de généralité que $y_1 \leq y_2$. On considère le rectangle R dont le sommet le plus en bas à gauche est $(x_0 - d_0, y_1)$ et le sommet le plus en haut à droite est $(x_0 + d_0, y_1 + d_0)$.

Si $d(M_1, M_2) < d_0$ alors M_2 est dans ce rectangle.

Subdivisons ce rectangle en 8 petits rectangles de côté $\frac{d_0}{2}$:



La distance maximum dans un petit rectangle, obtenue pour 2 sommets opposés, est, d'après le théorème de Pythagore :

$$a = \sqrt{\left(\frac{d_0}{2}\right)^2 + \left(\frac{d_0}{2}\right)^2} = \frac{d_0}{\sqrt{2}} < d_0$$

Or, deux sommets dans le même petit rectangle sont aussi du même côté de la droite $x = x_0$ (ils sont tous les deux soit dans G , soit dans D). Comme d_0 est la distance minimum entre deux sommets du même côté, il est donc impossible d'avoir 2 sommets dans le même petit rectangle.

S'il y avait plus de 8 points dans R , il y en aurait 2 dans le même petit rectangle (d'après le principe des tiroirs), ce qui serait une contradiction.

Il y a donc au plus 8 points dans R , et donc 6 points différents de M_1 et M_2 . Les points ayant une ordonnée comprise entre y_1 et y_2 étant forcément dans R , on a donc bien montré ce qui était demandé.

15. (/1.5) En déduire une fonction `fusion(c1, d0)` qui prend en entrée un cluster de points dont toutes les abscisses sont dans un intervalle $[x_0 - d_0, x_0 + d_0]$, et renvoie la distance minimale entre deux points du cluster si elle est inférieure à d_0 , ou d_0 sinon. Cette fonction doit avoir une complexité linéaire en la taille du cluster `cl`. Vous justifierez cette complexité.

Solution : La fonction suivante est en complexité $O(6 \times n) = \boxed{O(n)}$.

```

1  def fusion(c1, d0):
2      d = d0
3      for i in range(len(c1[1])):
4          for j in range(i, min(i + 6, len(c1[1]))):
5              d = min(distance(i, j), d)
6      return d

```

16. Écrire une fonction récursive `distance_minimale(c1)` qui prend en argument un cluster et utilise l'algorithme décrit plus haut pour renvoyer la distance minimale entre deux points du cluster

Solution :

```
1 def distance_minimale(cl):  
2     if len(cl) <= 3:  
3         return plus_proche(cl)  
4     d0 = min(distance_minimale(gauche(cl)), distance_minimale(droite(cl)))  
5     return min(d0, fusion(bande_centrale(cl, d0)))
```

17. Si on note n la taille du cluster cl , et $C(n)$ le nombre d'opérations élémentaires réalisées par la fonction `distance_minimale(cl)`, justifier que l'on a :

$$C(n) = 2C(n/2) + O(n)$$

Solution : `gauche(cl)` et `droite(cl)` sont de taille environ $\frac{n}{2}$ donc la complexité de chacun de ces appels est $C(\frac{n}{2})$. De plus `fusion` est en complexité linéaire, donc :

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