DM 1 d'informatique : à rendre la semaine du 03/01/22

Points proches dans le plan

Pour ce sujet, **le langage de programmation utilisé sera Python**. Vous pourrez utiliser les fonctions **Python** de manipulation de listes ou de matrices suivantes :

- Création d'une liste de taille n remplie avec la valeur x : `li = [x] * n`.
- Obtention de la taille d'une liste `li` : `len(li)`.
- Si `li` est une liste de n éléments, on peut accéder au k^e élément (pour $0 \leq k < $ `len(li)`) avec `li[k]`. On peut définir sa valeur avec `li[k] = x`.
- Un élément `x` peut être ajouté dans une liste `li` à l'aide de `li.append(x)`. On considèrera qu'il s'agit d'une opération élémentaire.
- Les matrices sont des listes de listes, chaque sous-liste étant considérée comme une ligne de la matrice. Si `mat` est une matrice, elle possède `len(mat)` lignes et `len(mat[0])` colonnes.
- Création d'une matrice de n lignes et p colonnes, dont toutes les cases contiennent x : `mat = [[x for j in range(p)] for i in range(n)]`.
- On accède à (resp. modifie) l'élément de `mat` dans la i^e ligne et j^e colonne avec `mat[i][j]` (resp. `mat[i][j] = x`).

À moins de les redéfinir explicitement, l'utilisation de toute autre fonction sur les listes (`sort`, `index`, `max`, etc.) est interdite. On rappelle enfin qu'une fonction qui s'arrête sans avoir rencontré l'instruction `return` renvoie `None`.

Ce problème, pouvant par exemple survenir dans le domaine de la navigation maritime, vise à déterminer, dans un nuage de points du plan, la paire de points les plus proches. Il est constitué de trois parties dépendantes.

Formellement, on suppose qu'on dispose de n points dans le plan $(M_0, M_1, \ldots, M_{n-1})$ dans un ordre quelconque pour le moment. Ils seront représentés en Python par deux listes de flottants de taille n : `coords_x` et `coords_y`, donnant respectivement les abscisses et les ordonnées des points. On dira ainsi que M_i est le point d'indice i, qu'il a pour abscisse $x_i =$ `coords_x[i]` et pour ordonnée $y_i =$`coords_y[i]`. On supposera que `coords_x` et `coords_y` sont des variables globales, qu'on ne modifiera jamais au cours de l'exécution de l'algorithme.

1 Approche exhaustive

On utilise la distance euclidienne définie par $d(M_i, M_j) = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}$.

► **Question 1** Écrire une fonction `distance(i, j)` qui renvoie la distance entre les points M_i et M_j. On utilisera la fonction `sqrt` après l'avoir importée.

► **Question 2** 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.

► **Question 3** 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.

► **Question 4** Donner, en la justifiant sommairement, la complexité de la fonction précédente en fonction de n.

2 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 :

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

► **Question 5** Que renvoie cette fonction ? Que fait-elle ? Le démontrer soigneusement en exhibant un invariant de boucle.

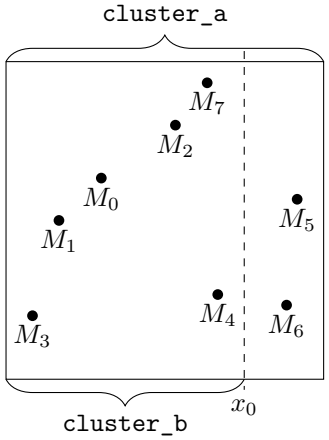
► **Question 6** Donner, en la démontrant, la complexité de la fonction `tri` en fonction de la taille de la liste donnée en paramètre.

► **Question 7** 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 ?

► **Question 8** 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.

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.




```
liste_x = [3, 1, 0, 2, 7, 4, 6, 5]
liste_y = [3, 6, 4, 1, 5, 0, 2, 7]

cluster_a = [liste_x, liste_y]

cluster_b = [ [3, 1, 0, 2, 7, 4],
              [3, 4, 1, 0, 2, 7] ]
```

FIGURE 1 – Représentation en Python 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.

► **Question 9** Écrire une fonction `sous_cluster(cl, x_min, x_max)` qui prend en arguments un cluster `cl` 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*.

► **Question 10** 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 $O(1)$.

3 Méthode sophistiquée

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

1. Si le cluster contient deux ou trois points, on calcule la distance minimale en calculant toutes les distances possibles.
2. 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.
3. 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.
4. 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.
5. 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 $d(M_1, M_2) < d_0$.
6. Si on en trouve une (ou plusieurs), on renvoie la plus petite de ces distances. Sinon, on renvoie d_0.

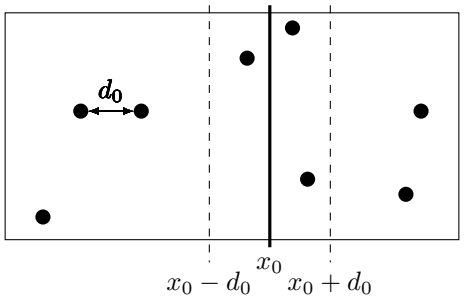


FIGURE 2 – Illustration du diviser pour régner

► **Question 11** 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`.

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)`.

► **Question 12** 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 $I_0 = [x_0 - d_0, x_0 + d_0]$.

► **Question 13** Écrire une fonction `bande_centrale(cl, d0)` qui prend en argument un cluster `cl` et un réel `d0`, et renvoie le cluster des points dont l'abscisse est dans I_0. Cette fonction doit avoir une complexité linéaire en la taille du cluster.

► **Question 14** 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.

► **Question 15** En déduire une fonction `fusion(cl, 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.

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

► **Question 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 :

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

► **Question 18** En déduire, en la démontrant, la complexité $C(n)$. On pourra se limiter au cas où n est une puissance de 2.