# I Approche exhaustive

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** :

   ```
   from math import sqrt

   def distance(i, j):
       return sqrt((coords_x[i] - coords_x[j])**2 + (coords_y[i] - coords_y[j])**2)
   ```

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.

   **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. 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** :

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

4. 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 $\boxed{O(n^2)}$.

# 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 :

```
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
```

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

**Solution** : Cette fonction tri 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 $i \in \{0, ..., n\}$. Donc d'après $H_n$, quand la boucle `for` se termine, la liste entière est triée.

6. 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 $i = 0, 1, ..., n-1$, la complexité totale est :

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

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 ?

**Solution** : Il faudrait remplacer `liste[pos]` par `coords_x[liste[pos]]`.

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.

**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. 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.

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

```
def sous_cluster(cl, x_min, x_max):
    c = [[], []]
    for i in cl[0]:
        if x_min <= coords_x[i] <= x_max:
            c[0].append(i)
    for i in cl[1]:
        if x_min <= coords_x[i] <= x_max:
            c[1].append(i)
    return c
```

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).

**Solution** :

```
def mediane(cl):
    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. 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** :

```
def gauche(cl):
    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. Justifier que l'on peut se contenter de chercher les points M1 et M2 de l'étape 5 de l'algorithme dans l'ensemble des points dont l'abscisse appartient à $I_0 = [x_0 d_0, x_0 + d_0]$.

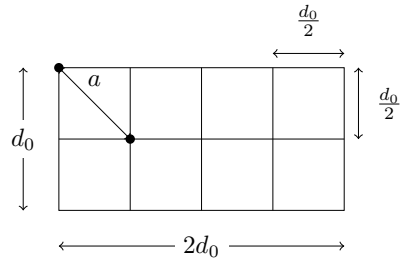
**Solution** : Notons $x_1$ et $x_2$ les abscisses de $M_1$ et $M_2$. Supposons par exemple que $M_1$ est dans $G$ (donc $x_1 \leq x_0$), que $M_2$ est dans $D$ mais que $x_2 > x_0 + d_0$. Alors la distance entre $M_1$ et $M_2$ serait supérieure à $x_2 - x_1 > x_0 + d_0 - x_0 = d_0$. $M_1$ et $M_2$ ne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

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 I0. Cette fonction doit avoir une complexité linéaire en la taille du cluster.

```
def bande_centrale(cl, d0):
    x0 = mediane(cl)
    return sous_cluster(cl, x0-d0, x0+d0)
```

14. Montrer que deux points M1 et M2 (de l'étape 5 de l'algorithme) situés à une distance inférieure à d0 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 2d0 × d0 contient au plus 8 points.

**Solution** : Soient $y_1$ et $y_2$ les ordonnées de M1 et M2. 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 :

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

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 ordonnées comprisent entre $y_1$ et $y_2$ étant forcément dans $R$, on a donc bien montré ce qui était demandé.

15. En déduire une fonction fusion(cl, d0) qui prend en entrée un cluster de points dont toutes les abscisses sont dans un intervalle [x0 − d0, x0 + d0], et renvoie la distance minimale entre deux points du cluster si elle est inférieure à d0, ou d0 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 $\mathrm{O}(6 \times n) = \boxed{\mathrm{O}(n)}$.

```
def fusion(cl, d0):
    d = d0
    for i in range(len(cl[1])):
        for j in range(i, min(i + 6, len(cl[1]))):
            d = min(distance(i, j), d)
    return d
```

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

**Solution** :

```
def distance_minimale(cl):
    if len(cl) <= 3:
        return plus_proche(cl)
    d0 = min(distance_minimale(gauche(cl)), distance_minimale(droite(cl)))
    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 :

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

**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 :

$$\boxed{C(n) = 2C(n/2) + \mathrm{O}(n)}$$