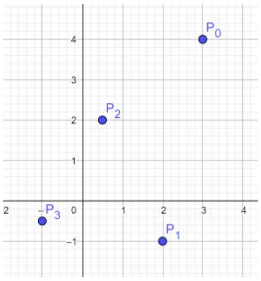
Méthode « diviser pour régner »
Paire des points les plus proches

1. Description du problème

On dispose d'un nuage de *n* points du plan (*n* ≥ 2), repérés par leurs coordonnées. On veut trouver les deux points les plus proches.

Exemple
points = [(3, 4), (2, -1), (0.5, 2), (-1, -0.5)]



Les deux points les plus proches sont
points[2] et points[3], nommés P_2 et P_3 sur le dessin.

2. Algorithme naïf

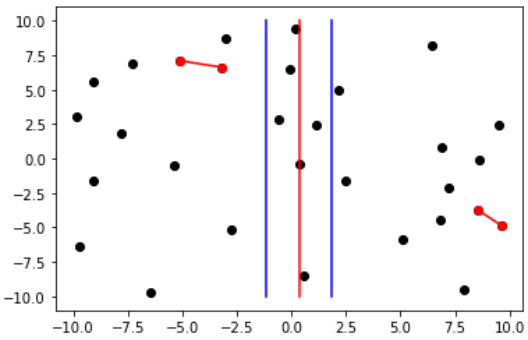
L'algorithme naïf consiste à calculer toutes les distances possibles, en examinant toutes les paires de points. Avec *n* points, on peut former $\frac{n(n-1)}{2}$ paires. Le calcul de distance étant en temps constant pour chaque paire de points, cet algorithme est à complexité quadratique.

3. Algorithme « diviser pour régner »

Soit *P* un ensemble de *n* points. On note d_P la distance minimale entre deux points de *P*.

3.1. Idée

On partage *P* verticalement en deux sous-ensembles *A* et *B de même taille (ligne rouge).*



Alors la plus petite distance entre deux points de P est atteinte, soit entre deux points de A, soit entre deux points de B, soit entre un point de A et un point de B.
On calcule récursivement la distance minimale d_A entre deux points de A (en rouge) et la distance minimale d_B entre deux points de B (en rouge aussi). On note δ le minimum entre d_A et d_B.
Il reste à calculer la distance minimale entre un point de A et un point de B, ce qui laisse beaucoup de combinaisons, mais on réduit les possibilités en écartant toutes les distances supérieures à δ. On s'intéresse uniquement aux points situés dans la bande verticale de largeur 2δ délimitée par les lignes bleues.

3.2. Description de l'algorithme

1ère étape (préliminaire)
On crée un tableau *Px* contenant les points de *P* triés dans l'ordre croissant des abscisses et *Py* contenant les points de *P* triés dans l'ordre croissant des ordonnées, avec un algorithme de tri en $O(n \log_2(n))$.

2e étape (diviser)
Si $n \leq 3$, on effectue une recherche naïve et on renvoie le résultat.
Sinon, on partage *P* à l'aide d'une droite verticale en deux sous-ensembles *A* et *B* de même taille (à un élément près), *A* contenant les points de plus petites abscisses et *B* les points de plus grandes abscisses.

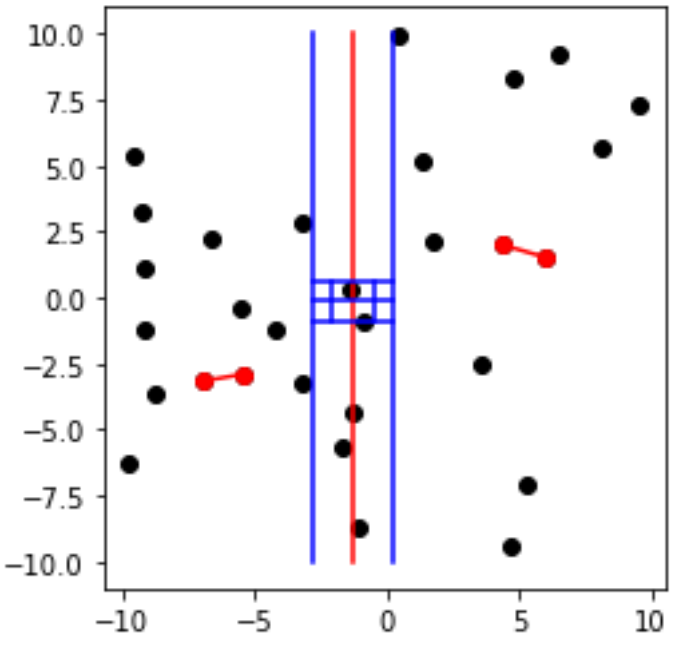
Dans notre implémentation en Python, on construira directement *Ax* et *Bx* en partageant *Px* en deux, puis *Ay* et *By* à partir de la liste *Py*.

3e étape (régner)
On calcule d_A et d_B par appel récursif et on note δ le minimum de ces deux valeurs.
On stocke les deux points correspondants à cette distance minimale.

4e étape (combiner)
4a Soit *x* l'abscisse du point médian *Px*[*n*//2] et *Q* l'ensemble des points dont l'abscisse appartient à l'intervalle]*x* - δ , *x* + δ[(bande délimitée par les lignes bleues).
On crée le tableau *Qy* contenant les points de *Q* triés par ordonnées croissantes.
En Python, on a *Qy* = [*p* for *p* in *Py* if x - *δ* < *p*[0] < *x* + δ].
4b Pour chaque point *p* de *Qy*, on calcule les distances entre *p* et les 7 points qui le suivent et on renvoie le minimum, entre la plus petite de toutes ces distances et δ, ainsi que les deux points correspondants.

Explication de l'étape 4b
Pourquoi ne considère-t-on que les points qui suivent *p* et pas ceux qui précèdent *p* ?
Pour chaque point *p* de *Qy*, on ne considère que les points situés au-dessus afin de ne pas considérer chaque paire deux fois : (*p*, *q*) et (*q*, *p*).

Pourquoi se contente-t-on de 7 points ?
Pour chaque point *p* de *Qy*, on considère le rectangle *R* s'appuyant sur les lignes bleues, dont le côté inférieur passe par *p* et de hauteur δ (de largeur 2δ). S'il existe un point de *Qy* à une distance inférieure à δ de *p*, alors il est forcément dans *R*.



On divise *R* en huit carrés de côté δ/2 comme sur le schéma. Alors, il ne peut y avoir au plus qu'un point par carré. En effet, s'il y avait deux points (ou plus) dans un carré, alors ces points seraient du même côté de la ligne rouge, donc tous les deux dans *A* ou dans *B*, et à une distance inférieure à $\delta/\sqrt{2}$, ce qui est impossible puisque δ est la distance minimale entre deux points de *A* ou deux points de *B*.

Il y a donc au plus 8 points dans *R*, le point *p* et 7 autres.

Il suffit de tester les 7 points qui suivent *p* dans *Qy* car les points suivants sont forcément en dehors de *R*.

3.3. Complexité

La complexité de cet algorithme est en $O(n \log_2 n)$.
L'idée de la preuve est qu'il y a $O(n)$ opérations par niveau d'appels et de l'ordre de $O(\log_2 n)$ niveaux d'appels.

Avec un milliard de points, l'algorithme naïf en $O(n^2)$ demande de l'ordre de 10^{18} opérations. Avec des ordinateurs effectuant 10^9 opérations par secondes, il faut de l'ordre de 10^9 secondes, soit environ 30 ans.

L'algorithme « diviser pour régner » demande de l'ordre de $10^9 \times \log_2(10^9) \approx 30 \times 10^9$ opérations, ce qui se fait en 30 secondes.

Dans les années 70, le meilleur algorithme connu pour ce problème était en $O(n^2)$, et les ordinateurs étaient 1000 fois plus lents qu'aujourd'hui (1 Mhz). Il aurait fallu 30 000 ans pour traiter ce problème.

4. Implémentation

On suppose que tous les points ont une abscisse différente. Ce n'est pas obligatoire, mais le programme est un peu plus simple dans ce cas.

4.1. Ouvrir le script distance_points.py.
4.2. Écrite la fonction distance renvoyant la distance entre deux points.
4.3. Écrire la fonction d'entête **def** plus_proches(points: list) -> (float, tuple, tuple): prenant en argument une liste de points définis par leurs coordonnées $[(x_1, y_1), \ldots, (x_n, y_n)]$ et renvoyant la distance minimale et les coordonnées des deux points les plus proches en utilisant l'algorithme naïf.
4.4. Exécuter le script. Si le doctest ne renvoie pas de message d'erreur, alors vos fonctions passent les tests.
4.5. Compléter la fonction plus_proches_dpr. Montrer votre travail au professeur après chacune des trois étapes.
4.6. Exécuter les fonctions test_plus_proches_dpr et graphique_ppdpr pour tester votre code.