Méthode de programmation
Diviser pour Régner

Cours/Exercice inspiré d'un partage fait sur la liste de diffusion NSI.

I) Description du problème

Vous êtes employé par le service de sécurité d'une grande société. D'après les informateurs de la société, deux personnes, sensées être des espions à la solde d'une société concurrente, doivent se rencontrer sur la grande place d'une ville pour échanger des informations sensibles dérobées à la société dont vous faite partie.

La dernière mission du service est de créer un programme destiné à analyser les images de vidéo-surveillance afin de trouver les personnes qui sont restées en contact étroit le plus longtemps.

Le projet est divisé en plusieurs parties et votre rôle sera de réaliser la programmation du ou des algorithmes qui permettront de trouver les deux personnes les plus proches dans la foule de la place, l'analyse des images et la création d'un tableau de coordonnées de points correspondants aux personnes étant réalisées par un de vos collègue.

Arriverez-vous à démasquer les auteurs de ce méfait ?

II) Lot de données d'entrée

A) Type des données ?

Avant de réaliser directement l'algorithme demandé, vous vous dites qu'il serait bon de vous mettre d'accord avec le collègue qui réalise la partie qui se placera juste avant la votre. En effet, il va vous fournir les coordonnées des personnes, mais sous quelle forme ? Après quelques minutes de discussion, vous vous mettez d'accord sur la forme suivante :

- chaque coordonnée sera sous la forme d'un objet de classe Coord2D comprenant deux propriétés : _x et _y ;
- la totalité des coordonnées se présentera sous la forme d'un simple tableau d'objets de classe Coord2D.

Une fois celui-ci retourné dans son bureau, vous riez sous cape, car vous vous souvenez que durant votre année de terminale NSI, vous aviez développé une classe Coord2D qui correspond à ce sur quoi vous vous êtes mis d'accord.

Vous vous souvenez même que vous aviez ajouté des méthodes d'accesseurs et de mutateurs setX, setY et getX, getY ainsi que développé une méthode permettant de calculer la distance entre deux points et une aussi pour l'afficher proprement (__str__). Cela va vous être particulièrement utile pour ce projet.

Retrouvez votre classe Coord2D et créez un répertoire pour votre projet qui la contiendra. Si vous n'arrivez pas à la retrouver ...

Vous n'avez plus qu'à la recréer …

B) Génération de données aléatoires

Comme tout bon informaticien, vous êtes content de pouvoir réutiliser des choses que vous aviez déjà développées, mais vient maintenant le temps de se mettre un peu au travail, et à un moment, il vous faudra pouvoir tester votre algorithme et vérifier qu'il fonctionne correctement. Il serait donc bon de pouvoir générer des données de manière aléatoire pour vous permettre de le tester.

La meilleure manière serait de créer une fonction qui fournirait un lot de coordonnées aléatoires. Cette fonction prendrait comme paramètres :

- un nombre de coordonnées à obtenir dans le lot renvoyé ;
- un cadre pour la valeur de ces coordonnées qui permettrait de fixer une valeur minimale et maximale à ne pas dépasser.

Vous vous dites donc que l'entête de la définition de cette fonction serait de la forme suivante :

```
def genCoordsAlea(xmin, xmax, ymin, ymax, nb):
    """
    Générateur d'objets Coord2D avec des valeurs aléatoires
    pour _x entre xmin et xmax
    pour _y entre ymin et ymax
    renvoie un tableau de nb éléments Coord2D
    """
```

Codez la fonction genCoordsAlea (vous aurez sans doute besoin de la fonction random, donc n'oubliez pas d'importer le module random).

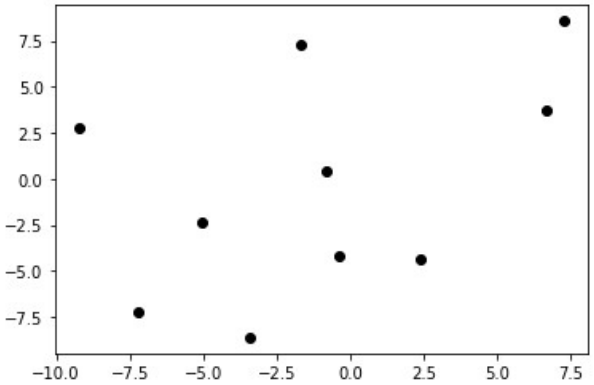
Comme à chaque création de fonction, il faut maintenant la tester. Plutôt que d'afficher simplement dans la console la liste des points, vous vous dites qu'il serait intéressant d'avoir une représentation graphique des coordonnées. Après une rapide recherche sur internet, vous trouvez une bibliothèque, MatPlotLib et une méthode nommé scatter permettant ce genre d'affichage (https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.scatter.html). En vous aidant de la documentation, vous débutez la fonction de test que vous appelez affichCoords :

```
import matplotlib.pyplot as plt

def affichCoords(tabCoord2D):
    """
    fonction d'affichage d'un tableau de coordonnées Coord2D
    """
    coordsX = []
    coordsY = []

    ...

    plt.scatter(coordsX, coordsY, color='black')
    plt.show()
```

Les deux fonctions scatter et show permettent la prise en compte des données et l'affichage des coordonnées sous forme de points. Mais il vous faut encore extraire les abscisses des coordonnées pour les mettre dans le tableau coordsX et les ordonnées dans le tableau coordsY.

Ajoutez les lignes manquantes pour créer les deux tableaux de valeurs coordsX et coordsY.

Testez ensuite les deux fonctions : vous générerez grâce à la fonction genCoordsAlea 10 coordonnées entre [-10 ; +10] pour x et [-10 ; +10] pour y. Vous enverrez ce tableau de 10 coordonnées à la fonction affichCoords. Si tout ce passe correctement, vous verrez un résultat tel que celui-ci :



III) Méthode Naïve

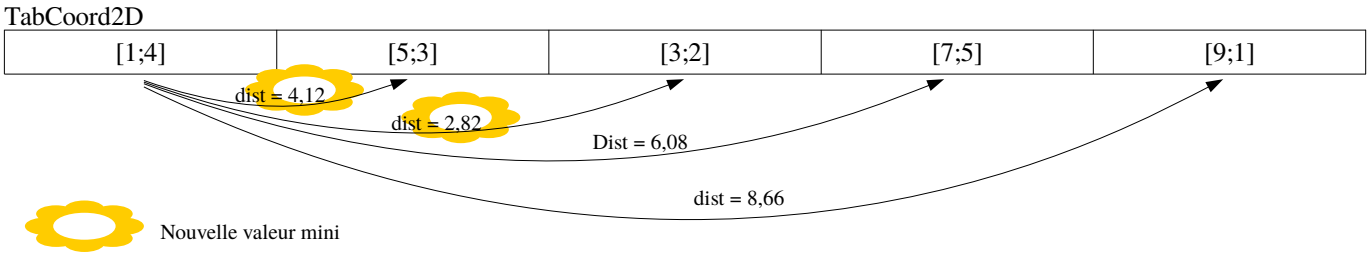
Fort de ce succès, vous vous mettez à réfléchir au fond du problème que vous devez résoudre. Problème que l'on pourrait résumer en :

Trouver les deux points les plus proches parmi un ensemble de points dont vous avez les coordonnées

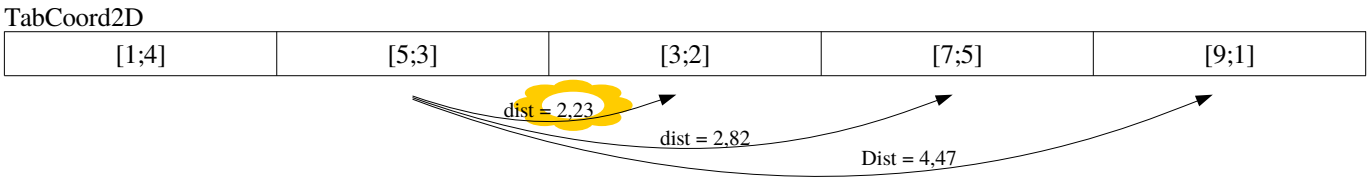
Pas besoin de réfléchir plus longtemps, il suffit de tester la distance entre tous les couples de points et puis on trouve le plus petit ! Vous pensez même à une amélioration qui serait de ne tester chaque point qu'avec les points suivants dans la liste. En effet, inutile de calculer la distance du point B au point A si on a déjà calculé la distance du point A au point B. Étant certain que cette optimisation satisfera au plus au point votre patron (peut-être même au point de décrocher une bonification sur votre fiche de paie), vous vous lancez dans la création de cette fonction, avec une méthode sans doute un peu naïve, mais qui a le mérite d'être simple.

Vous faites un petit schéma de ce qui doit se passer :

1er passage : distance de la première coordonnée à toutes les suivantes :



2nd passage : distance de la seconde coordonnée à toutes les suivantes :



À chaque nouvelle valeur minimum trouvée, on la sauvegarde, ainsi que les deux points correspondants (ou leurs emplacements dans la tableau), et on continue jusqu'à calculer la distance entre l'avant dernière coordonnée et la dernière. Vous décelez aisément la présence de deux boucles : une qui passera du premier élément à l'avant dernier et une autre de l'élément suivant de la première boucle au dernier.

Au terme de la recherche, il faudrait renvoyer un tuple qui contiendrait la distance ainsi que les deux points correspondants.

Vous débutez la création de la fonction de cette manière :

```
def pointsPlusProches(tabCoord2D):
    """
    Fonction qui renvoie la distance des deux plus proches voisins d'un ensemble d'objets
    coord2D et les deux points en question.
    """
    distmini = float ('inf')
    ...
    return distmini, coord1, coord2
```

Vous avez pensé à initialiser la valeur de distmini à l'infini (grâce à la première ligne de la fonction) et vous retournez un tuple avec cette valeur de distance minimum et les deux coordonnées correspondantes (donc un tuple avec les type `(float, coord2D, coord2D)`). Il ne reste plus qu'à faire la recherche.


Complétez la fonction pointsPlusProches.


Testez cette fonction avec le tableau de coordonnées vu plus haut dans cette page et vérifiez que la réponse est bien celle attendue (distance mini = 2,23 avec les points de coordonnées [5,3] et [3,2])


Testez aussi la fonction sur 10 valeurs aléatoires générées par genCoordsAlea et vérifiez si le résultat renvoyé a l'air de correspondre à la plus petite distance.

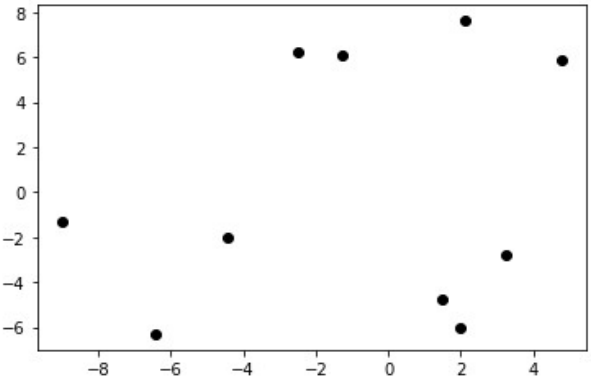
Vous aimeriez ensuite avoir un rendu visuel de votre plus petite distance calculée afin que cela soit plus simple de vérifier. Tracer un trait entre les deux points correspondants serait une bonne solution pour cela.

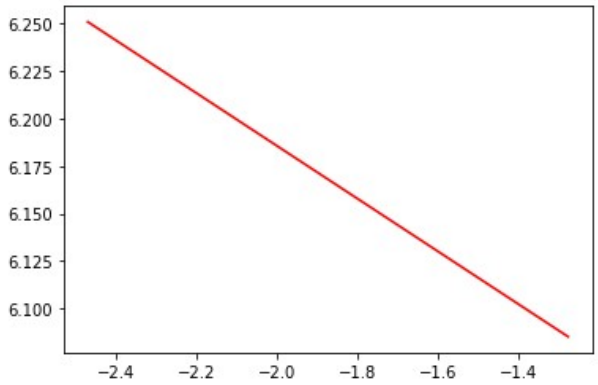
En regardant la documentation de Matplotlib, vous trouvez une méthode plot qui permet de tracer des lignes brisées (La première variable est un tableau qui contient toutes les abscisses des points et le second toutes les ordonnées.) :

```
plt.plot(tabX, tabY, color = 'red')        # plot x and y specifying line color
```

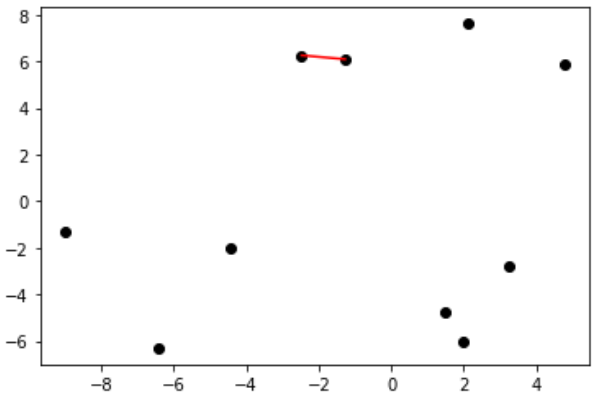
Créez un tableau ptPPX avec les deux abscisses des deux points les plus proches et ptPPY avec leur deux ordonnées. Lancez la fonction avec ce tableau et lancez de nouveau un appel à la méthode plt.show().

Les deux affichages sont alors séparés, ce qui n'est pas très intéressant :





Commentez l'appel à la méthode plt.show() dans votre fonction affichCoords pour que le trait rouge apparaisse sur le même graphique que les points et vous montre clairement les deux points les plus proches :



Content de votre travail, vous vous empressez de le copier dans le répertoire partagé du projet.

Votre petite voix intérieure vous titille quand même de vous poser la question de la complexité temporelle de votre algorithme. Sans votre amélioration, vous vous retrouveriez avec une double boucle dont chacune tourne sur le nombre de points fournis. Nombre de point, que l'on appellerait 'n' et qui serait le paramètre de la complexité. Le calcul de distance, réalisé dans la boucle, est en temps constant (puisque c'est un simple calcul).

Quelle serait la complexité alors ?

Grâce à votre amélioration, vous savez que votre double boucle ne parcourrait pas deux fois le tableau de manière imbriquée, mais lors du premier passage dans la première double boucle on effectuerait n – 1 comparaisons, lors du second passage n – 2, etc. jusqu'à réaliser uniquement une comparaison à la toute fin de l'algorithme. Soit une somme :

$$(n-1)+(n-2)+(n-3)+\ldots+1$$

Un rapide tour sur internet et vous tombez sur le site solumaths.com qui vous indique que $1+\ldots+p=p\cdot\frac{p+1}{2}$

Quelle va être la complexité au final ?

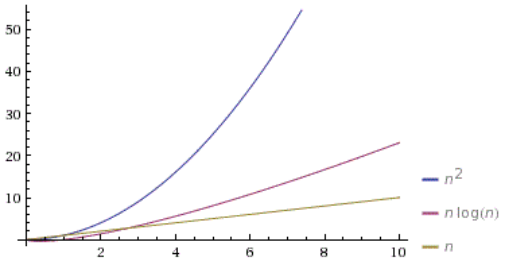
Bon, ok … ce n'est pas vraiment mieux au final … Mais vous êtes tout de même content de votre travail ! Comme vous n'avez pas mis beaucoup de temps à le coder, vous pouvez vous plonger dans l'épisode suivant de votre série préférée : Le canard de l'espace.

IV) Méthode Diviser pour régner

Le lendemain, à peine arrivé au travail, votre supérieur hiérarchique fait irruption dans votre bureau, avec sa tête des mauvais jours ...

- Monsieur Anderson ! Qu'est ce que vous nous avez fait là ? lance-t-il.
- Mais euh … c'est à dire … commencez-vous à répondre.
- L'algorithme que vous nous avez fourni pour répondre au problème qu'on vous avait confié.
- Euh, oui ?
- Je suppose que vous n'avez pas calculé sa complexité ? Parce que si oui, vous vous seriez aperçu qu'elle était en n^2 !

Vous n'osez bien entendu pas rajouter que bien sûr, vous l'aviez calculée … Une image vous revient maintenant …



Il enchaîne :

- On ne vous a jamais appris qu'une complexité en n^2, c'est bon pour des algorithmes qu'on ne lance pas souvent ou bien sur des algorithmes pour lesquels on ne va pas avoir une quantité de données importante ?
- Euh, en fait …
- Vous saviez que l'algorithme était destiné à l'analyse en temps réel d'une foule de potentiellement plusieurs milliers d'individus, et qu'en plus il serait sans doute nécessaire de le lancer à de nombreuses reprises sur la même image pour détecter les différents couples de suspects potentiels !
- Ah, c'est vrai que …
- Vous allez me reprendre votre algorithme ! Vous vous débrouillez copmme vous voulez, utilisez, je ne sais pas moi, la méthode "Diviser pour Régner", mais améliorez-moi cette complexité ! Sinon, voilà ce que je fais de votre contrat de travail !

Votre patron, M. Smith, sort de votre bureau en claquant la porte. Vous poussez un soupir. Vous pensiez avoir une augmentation et vous la couler douce quelques jours à mater le Canard de l'espace en douce et finalement vous êtes à deux doigt d'être mis à la porte !

Après un moment d'abattement, vous décidez de vous retrousser les manches et de suivre une des indications fournies par votre chef de service : il avait parlé de la méthode "Diviser pour Régner".

Quelques recherches sur internet vous précisent l'esprit de cette méthode :

- au lieu de travailler sur un grand ensemble de données, on le divise en deux successivement ;
- une fois que la taille est suffisamment petite, on résout le problème ;
- une fois les deux parties d'un problème divisé en deux résolues, on en combine les résultats.

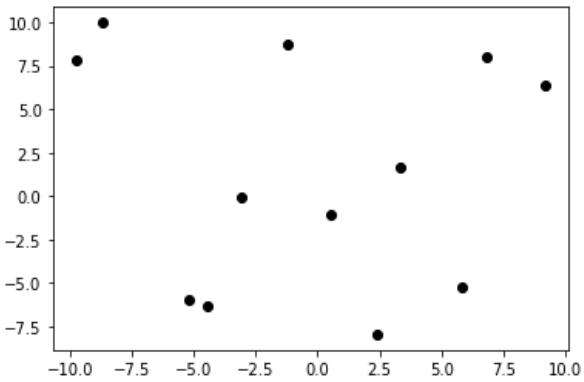
Cela vous rappelle l'algorithme de recherche dichotomique et vous voyez bien la récursivité possible dans cette méthode.

Une journée de travail intensif plus tard, vous voyez la méthode à appliquer pour votre problème précis :

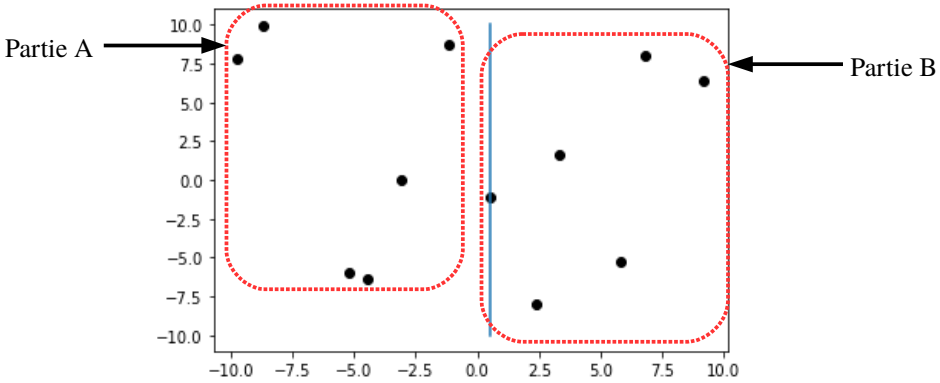
- Étape 1 : comme pour la recherche dichotomique, il faut comme étape préliminaire que les données d'entrées soient triées. Vous pensez donc les trier selon les abscisses, mais aussi selon les ordonnées. Vous pensez utiliser la fonction `sorted` de python, mais quelle est sa complexité ?

- Étape 2 : la complexité de la fonction sorted n'est donc pas un problème, étant meilleure que celle qui faisait tiquer votre boss, vous enchaînez sur l'étape "Diviser" : si le nombre de données est trop grand, on découpe en deux parties égales (à un élément près) et on lance la résolution du problème sur les deux parties (à priori, il faudrait découper tant que l'on est au dessus de 3 valeurs). Si la quantité de données est inférieure ou égale à 3 éléments, on utilisera l'algorithme naïf (finalement, il servira quand même !! Ah ah !)
- Étape 3 : l'étape "Régner" : des deux résultats renvoyés par les deux parties, on trouve le meilleur des deux (celui dont la distance est la plus petite).
- Étape 4 : l'étape "Combiner" : maintenant que l'on a la plus petite distance de chaque partie, il faut vérifier s'il n'existe pas une plus petite distance entre deux points se trouvant respectivement sur la première et la seconde partie.
- Il ne reste plus à la fin qu'à renvoyer le meilleur résultat trouvé : la distance et les deux points permettant de l'obtenir.

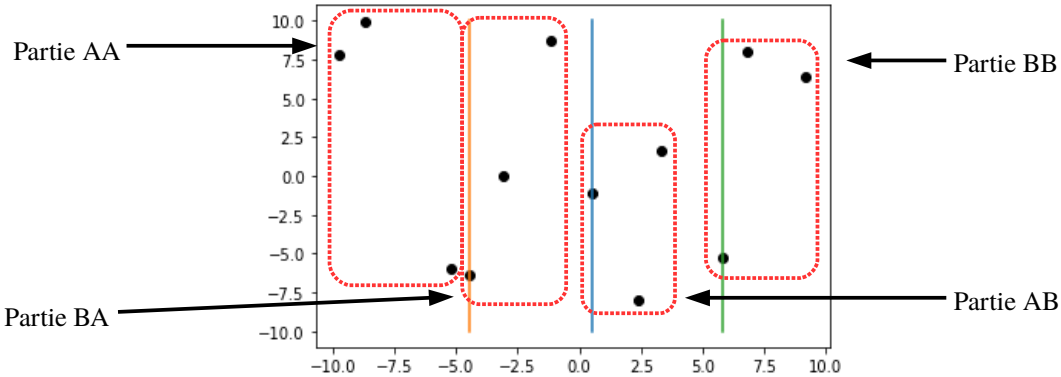
Vous arrivez à illustrer ces étapes mentalement grâce à un exemple comprenant12 points :



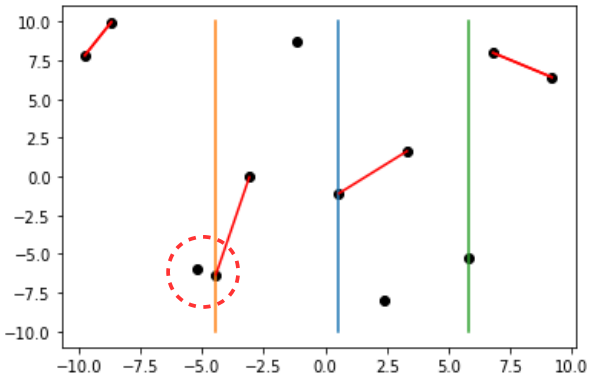
Le nombre de points est trop grand, il faut diviser le problème en deux parties (à gauche nommée A, à droite nommée B) :



Dans les deux parties découpées, le nombre de points est encore trop grand, il faut de nouveau couper en deux :

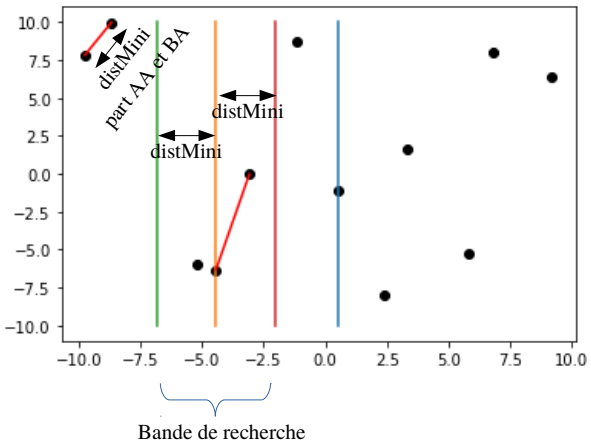


On a maintenant le nombre de points de 3 maximum par partie, on peut donc utiliser la fonction naïve. Elle permet de calculer les points les plus proches dans chacune des parties :



On pourrait se dire que notre solution se trouve parmi les réponses apportées par la méthode naïve, mais dans ce cas particulier, on a un problème au niveau de la section entourée en pointillé : le point de gauche se trouve dans la partie AA et le point de droite se trouve dans la partie BA, ce qui fait que notre recherche ne s'est pas appliquée à eux deux et que donc cette distance, pourtant plus petite, n'aie pas été détectée.

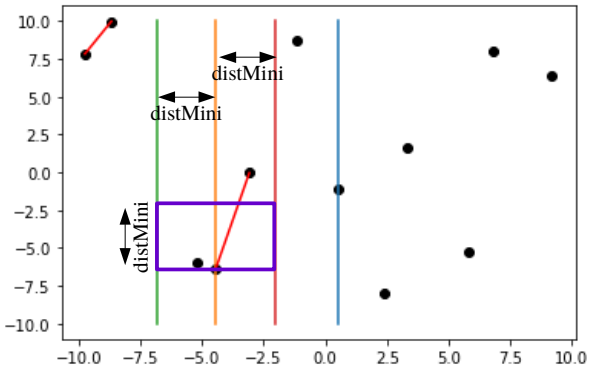
Il faut donc, lors de l'étape combiner, vérifier qu'il n'existe pas de points plus proches dans une bande située à gauche et à droite du point médian. La largeur de la bande sera fonction de la plus petite distance trouvée de chaque coté :



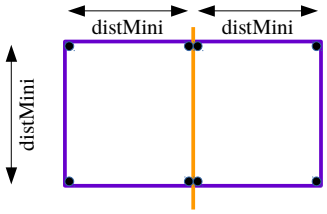
Parmi les points situés sur cette bande, il va falloir faire une nouvelle recherche d'un minimum. Problème : le nombre de points dans cette bande peut être très grand et lancer une recherche naïve sur la totalité des points peut être très peu efficace. Faut-il refaire une recherche avec de nouveau la méthode "Diviser pour Régner", mais cette fois-ci sur les données triées selon les ordonnées ? Ou bien existe-t-il un moyen plus efficace ?

Il est en fait possible de limiter le nombre de calculs dans cette recherche en remarquant deux choses :

- la première est de s'apercevoir que pour chaque point présent dans cette bande, il est inutile de rechercher une distance minimum plus loin qu'à une hauteur de distMini, c'est dire dans le rectangle suivant :



- la seconde est de se dire que dans le pire cas, on peut avoir au maximum 8 points dans ce rectangle : 4 cotés gauche éloignés d'au moins distMini et 4 cotés droits éloignés eux aussi d'au moins distMini :



Cela veut donc dire que l'on peut se limiter à évaluer les distances entre chacun des points présents dans la bande et les 7 suivants dans la liste triée selon les ordonnées. Cette recherche pourra être réalisée grâce à la méthode naïve.

Une fois la recherche effectuée sur tous les points de la bande, une nouvelle distance minimale est potentiellement trouvée et donc doit remplacer le meilleur résultat trouvé des deux parties.

Le meilleur résultat des trois recherches (partie A, partie B et bande) sera renvoyé.

Maintenant que la démarche pour résoudre le problème est déterminée, il faut la coder. Chacune des étapes sera codée puis testée dans la foulée.

A) Étape préliminaire

Il nous faudra, pour résoudre le problème avec la méthode Diviser pour Régner, créer deux autres fonctions :

- une fonction qui sera récursive, réalisera les étapes "Diviser", "Régner" et "Combiner" (fonction `pointsPlusProchesRec`) ;
- une autre concernant l'étape préliminaire, c'est à dire créer, à partir du tableau de points, deux tableaux triés, l'un selon les abscisses et l'autre selon les ordonnées et ensuite lancer la fonction récursive et en renvoyer le résultat (fonction pointsPlusProchesDepart).

Vous codez tout d'abord la seconde fonction qui vous semble un meilleur point de départ et vous essayez de l'exécuter à la place de votre fonction utilisant la méthode naïve :

```
def pointsPlusProchesDepart(tabCoord):
    tabCoordTriX = sorted(tabCoord)
    tabCoordTriY = sorted(tabCoord)
```

Mais lors de l'exécution, l'interpréteur Python vous rappelle à l'ordre :

```
    tabCoordTriX = sorted(tabCoord)
TypeError: '<' not supported between instances of 'coord2D' and 'coord2D'
```

Le message d'erreur vous indique que la fonction sorted n'arrive pas à trier les objets de type coord2D, car l'opérateur '<' n'est pas supporté. En plus, il faut que vous puissiez dire que dans un cas, c'est un tri selon les abscisses qu'il faut faire et dans l'autre selon les ordonnées.

En regardant dans la documentation de cette fonction (https://python-reference.readthedocs.io/en/latest/docs/functions/sorted.html), vous remarquez dans l'exemple 4 qu'il est possible d'indiquer une clef pour le tri sous forme d'une fonction lambda :

```
>>> sorted(['A', 'b', 'C'])
['A', 'C', 'b']
>>> sorted(['A', 'b', 'C'], key=lambda x: x.lower())
['A', 'b', 'C']
```

Vous essayez donc la modification suivante pour l'utilisation de la fonction sorted :

```
tabCoordTriX = sorted(tabCoord, key = lambda c: c.getX())
```

L'argument supplémentaire à sorted `key = lambda c: c.getX()` crée une fonction temporaire (lambda) à qui l'on passe l'objet coord2D (`c`) et qui en extrait la coordonnée x (`c.getX()`). La fonction `sorted` utilise cette valeur pour réaliser le tri.

Réalisez la même modification pour le tri selon les ordonnées (en la modifiant très légèrement).

Ajoutez les quelques lignes nécessaires pour afficher ces données triées de cette manière (encore plus simple si vous avez une méthode __str__ ou __repr__ fonctionnelle dans la classe coord2D :

```
tri selon x:
X = -9.564892097685352 / Y = 8.847840220740093
X = -7.469160156199881 / Y = -4.321550422301382
X = -4.936917393589635 / Y = -5.267886677779947
X = -3.233608890987723 / Y = 9.538924445234969
X = 1.6744055828059086 / Y = 4.0261324435314165
X = 2.2098684053521147 / Y = -3.4239353919339237
X = 3.204771317659315 / Y = 6.063096774218629
X = 4.493724929811096 / Y = -8.613490397631839
X = 5.03490022948265 / Y = -7.193541086417996
X = 8.634400290737641 / Y = 7.127217084165412
tri selon y:
X = 4.493724929811096 / Y = -8.613490397631839
X = 5.03490022948265 / Y = -7.193541086417996
X = -4.936917393589635 / Y = -5.267886677779947
X = -7.469160156199881 / Y = -4.321550422301382
X = 2.2098684053521147 / Y = -3.4239353919339237
X = 1.6744055828059086 / Y = 4.0261324435314165
X = 3.204771317659315 / Y = 6.063096774218629
X = 8.634400290737641 / Y = 7.127217084165412
X = -9.564892097685352 / Y = 8.847840220740093
X = -3.233608890987723 / Y = 9.538924445234969
```

Si l'affichage montre bien que les tableaux tabCoordTriX et tabCoordTriY sont bien triés selon les abscisses et les ordonnées, vous pouvez mettre en commentaires les lignes permettant cet affichage.

À la fin de la fonction `pointsPlusProchesDepart`, vous pouvez maintenant ajouter le code pour lancer la fonction `pointsPlusProchesRec` avec comme paramètres les deux tableaux que vous venez de créer.

B) Étape Diviser

L'étape diviser est le début de la fonction récursive pointsPlusProchesRec. Cette fonction prend comme paramètres deux tableaux tabTriX et tabTriY et renvoie un tuple : (distance (float), point1 (coord2D), point2 (coord2D)). Son entête de définition sera donc :

```
def pointsPlusProchesRec (tabTriX, tabTriY):
    ...
```

Elle va devoir réaliser les opérations suivantes :

- si le nombre de points est inférieur ou égal à 3, lancement du calcul via la méthode naïve et renvoi du résultat ;
- sinon il faudra découper le problème en deux parties :
  - copie dans un tableau `partieATriX` de la première moitié (à une valeur près) du tableau `tabTriX` ;
  - copie dans un tableau `partieBTriX` de la seconde moitié du tableau `tabTriX` ;
  - copie dans une variable `valMoitieX` le premier élément de `partieBTriX` ;
  - copie dans un tableau `partieATriY` des éléments de `tabTriY` dont l'abscisse est inférieure strictement à `valMoitieX` ;
  - copie dans un tableau `partieBTriY` des éléments de `tabTriY` donc l'abscisse est supérieure ou égale à `valMoitieX`.

Comme `tabTriY` est trié selon les ordonnées, et si on le parcourt dans l'ordre pour remplir les deux tableaux `PartieAY` et `PartieBY`, ils seront eux aussi triés selon les ordonnées.

Codez dans la fonction `pointsPlusProchesRec` les instructions permettant de réaliser les opérations ci-dessus.

Afin de vérifier si les deux tableaux sont correctement découpés, à la suite de la création des tableaux Partie…, ajoutez les lignes de code pour réaliser un affichage de la sorte :

```
tableau Partie A trié selon X
X = -7.778893056128902 / Y = 6.6597273679794995
X = -3.395258865694162 / Y = 1.8326542026009527
X = -1.7041223852474463 / Y = 6.04140705445343
X = 1.7453738653428417 / Y = -6.846823918236431
X = 2.2307706717154794 / Y = 9.564749946158862
tableau Partie B trié selon X
X = 5.096819849976075 / Y = -2.189765343019201
X = 7.657479409120555 / Y = -6.780113596404746
X = 8.46163977967845 / Y = 4.980291202843507
X = 8.656978789158899 / Y = 3.0565490557020603
X = 9.160400281187506 / Y = 8.966837080089412
tableau Partie A trié selon Y
X = 1.7453738653428417 / Y = -6.846823918236431
X = -3.395258865694162 / Y = 1.8326542026009527
X = -1.7041223852474463 / Y = 6.04140705445343
X = -7.778893056128902 / Y = 6.6597273679794995
X = 2.2307706717154794 / Y = 9.564749946158862
tableau Partie B trié selon Y
X = 7.657479409120555 / Y = -6.780113596404746
X = 5.096819849976075 / Y = -2.189765343019201
X = 8.656978789158899 / Y = 3.0565490557020603
X = 8.46163977967845 / Y = 4.980291202843507
X = 9.160400281187506 / Y = 8.966837080089412
```

Mêmes couples de points

Mêmes couples de points

Testez votre fonction `pointsPlusProchesRec` et vérifiez que les tableaux sont bien divisés en deux et toujours triés correctement. Si l'affichage des valeurs est correct, vous pouvez commenter celui-ci.

On supposera ici que tous les points ont une abscisse différente, ce qui permettra d'éviter le cas particulier où deux points de même abscisse se retrouvent un sur `partieATriX`, l'autre sur `partieBTriX` car à ce moment-là, les deux points basculeraient sur `partieBTriY`, ce qui entraînerait une erreur. Une solution à ce problème serait d'affecter un identifiant à chaque point et de ce servir de cet identifiant pour répartir des points dans `partieATriY` et `partieBTriY`.

C) Étape Régner

Cette étape n'est pas très compliquée : il suffit de relancer deux fois la fonction récursive `pointsPlusProcheRec` avec comme paramètres les deux tableaux trié en X et en Y des parties respectives A et B. Le résultat sera rangé respectivement pour chaque appel dans deux tuples à trois éléments : (`distMiniPartieA`, `point1PartieA`, `point2PartieA`) et (`distMiniPartieB`, `point1PartieB`, `point2PartieB`).

Ajoutez à la suite de votre code le double appel récursif avec les bons paramètres et récupérez les résultats dans les tuples.

Comme la fonction récursive est censée renvoyer quelque chose, pour l'instant, ajoutez, à la fin de votre fonction le code pour renvoyer le tuple (`distMiniPartieA`, `point1PartieA`, `point2PartieA`) ;

Pour tester le bon fonctionnement de votre code, vous pouvez utiliser la méthode plt.plot(tabX, tabY, color) afin d'afficher les résultats renvoyés par l'étape régner.

Les échelles des graphiques par défaut dans matplotlib ne sont pas forcément normées : parfois les abscisses sont plus grandes et parfois ce sont les ordonnées. Faites-y attention lorsque vous vérifierez visuellement vos résultats.

D) Combiner

La première phase de l'étape Combiner, n'est pas très compliquée non plus. Il faut déterminer la plus petite distance parmi les deux résultats renvoyés par les deux appels à `pointPlusProcheRec`. On compare donc les deux valeurs `distMiniPartieA` et `distMiniPartieB` et en fonction de la plus petite des deux, on copie dans le tuple (`distMini`, `point1` et `point2`) le meilleur résultat.

Codez cette première phase et modifiez la valeur renvoyée pour (`distMini`, `point1` et `point2`) ;

Vient ensuite la recherche d'une éventuelle plus petite distance à cheval sur les deux partie A et B. Plusieurs sous-étapes pour réaliser cela :

- copier dans un tableau `partieBandeTriY` les points de `tabTriY` dont les abscisses sont situées sur l'intervalle $]valMoitieX - distMini\,;\, valMoitieX + distMini[$ (ici aussi, comme `tabTriY` est trié selon les ordonnées, `partieBandeTriY` le sera également);
- il faut maintenant relancer une recherche équivalente en tout point à la méthode naïve sauf qu'il faut limiter la recherche à 7 points suivants au maximum. Un algorithme permettant cela pourrait être celui-ci :

```
POUR i ALLANT DE 0 À taille(partieBandeTriY) - 1
    nbPointsATraiter = minimum (7, taille(partieBandeTriY) - i - 1)
    POUR j ALLANT DE i + 1 À i + nbPointsATraiter + 1
        distance = partieBandeTriY[i].distance(partieBandeTriY[j])
        SI distance < distMini
            distMini = distance
            point1 = partieBandeTriY[i]
            point2 = partieBandeTriY[j]
        FIN SI
    FIN POUR
FIN POUR
```

- il ne reste plus ensuite qu'à vérifiez que la dernière ligne de votre fonction est bien le renvoi du tuple (`distMini`, `point1` et `point2`).

Codez la création du tableau `partieBandeTriY` et son remplissage.

Transcodez l'algorithme de recherche.

Tester votre programme amélioré sur des valeurs aléatoires et vérifiez que vous obtenez visuellement des résultats cohérents. Vous pouvez utiliser des instructions `plt.plot(…)` pour relier les points les plus proches.

V) Test final et vérification d'efficacité comparée à la méthode naïve

Ça y est, vous avez sué sang et eau pour arriver à ce nouveau programme. Vous vous êtes investi corps et âme dedans. Vous en avez même été jusqu'à oublier qui est le Canard de l'espace ! Il est temps maintenant de vérifier que tous ces sacrifices ont bien servi à quelque chose !

La première chose est de vérifier que votre méthode diviser pour régner donne bien le même résultat que la méthode naïve.

Réalisez un bout de programme qui lance dix fois les deux méthodes sur des valeurs aléatoires et vérifie que le résultat est le même dans les deux cas.

A) Étude théorique de la complexité

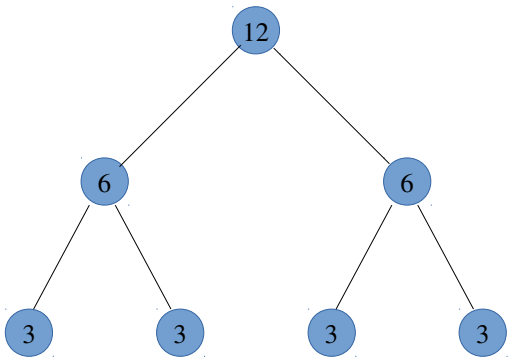
Le but de tout ce travail était d'améliorer la complexité du programme d'origine. Une première étude théorique de la complexité **en pire cas** donnerait :

1) Étape préliminaire

Au début de la méthode, nous trions les tableaux dans l'étape préliminaire. Ce double tri est lancé une seule fois et sa complexité est :

2) Récursivité

Au cours de l'exécution de la méthode, nous divisons le tableau en deux jusqu'à ne plus avoir que des tableaux de 2 ou 3 cases. On peut modéliser ces divisions par un arbre. Par exemple pour un nombre de points de 12 :



Il faudra réaliser récursivement deux fois la division, et nous nous retrouvons avec un arbre d'une hauteur de 2, donc on aura a réaliser les opérations récursives sur 3 niveaux (niveau 0, la racine, et niveau 1 et 2).

Pour un nombre de points de 1024, quel sera le nombre de niveaux ?

Pour un nombre de points n, quel sera le nombre de niveaux ?

Nous avons donc ici la complexité de la récursivité, trouvons maintenant la complexité des opérations la constituant.

3) Chaque niveau (sauf dernier)

Nous allons essayer de calculer la complexité **à chaque niveau** de l'arbre de récursivité (sauf le dernier). À chaque niveau, nous allons devoir réaliser les opérations suivantes :

a) Phase Diviser :

- couper en deux parties le tableau trié en X, quelle est la complexité ?

- trouver l'abscisse du point milieu (`valMoitieX`), quelle est la complexité ?

- répartir en deux parties le tableau trié en Y, quelle est la complexité ?

b) Phase Régner

On supposera que le lancement des deux fonctions récursives sont de complexité constante O(1) (sans s'occuper de ce qui est réalisé dedans, on sera au niveau inférieur de l'arbre, ce sera comptabilisé par la complexité de la récursivité).

c) Phase Combiner

- copier dans le tableau `partieBandeTriY` les points faisant partie de la bande $]valMoitieX - distMini\,;\, valMoitieX + distMini[$. Quelle est la complexité ?
- rechercher un nouveau minimum dans la bande, sachant que, dans le pire cas, pour chaque point de la bande, on aura au maximum 7 calculs. Quelle est la complexité ?

4) Dernier Niveau

Pour le dernier niveau, de l'arbre, il faut utiliser la méthode naïve,

- si le dernier niveau comporte 2 points, combien de calculs seront à réaliser ?
- si le dernier niveau comporte 3 points, combien de calculs seront à réaliser ?

Donc, dans le pire cas, la complexité pour la totalité de ce niveau sera de :

5) Complexité finale

Au final, la complexité donne :

Etape préliminaire		**Récursivité (sauf dernier niveau)**		**Phase Diviser**		**Phase Régner**		**Phase Combiner**		**Dernier niveau (méthode naïve)**
	+		* (		+		+		) +	

Quelle est la complexité si on simplifie ?

B) Étude pratique de la complexité

Bon, cette étude théorique montre bien que l'on gagne sur la complexité, mais qu'en est-il des gains en pratique ? Ce genre de choses pourraient intéresser votre patron !

Testez en pratique ce que nous permet de gagner en temps la méthode Diviser pour Régner par rapport à la méthode naïve avec le code suivant :

```
import timeit
npoints = 10 # nombre de points dans le lot
nrepeat = 10 # nombre de fois où on va répéter le test
print('Temps de calcul lors de {} recherches sur des nuages de {} points'.format(nrepeat,
npoints))
t1 = timeit.timeit('pointsPlusProches(genCoordsAlea(-100, 100, -100, 100,
{}))'.format(npoints), globals = globals(), number=nrepeat)
t2 = timeit.timeit('pointsPlusProchesDepart(genCoordsAlea(-100, 100, -100, 100,
{}))'.format(npoints), globals = globals(), number=nrepeat)
print('Recherche naïve : {} - Algorithme diviser pour régner : {}'.format(t1, t2))
```

La méthode Diviser pour Régner permet-elle de gagner du temps lors d'un calcul sur 10 points ?

Testez maintenant le temps lors d'un calcul sur 100 points puis sur 1000 points, comment semble évoluer la méthode naïve ? Et la méthode Diviser pour Régner ?

L'évolution des temps d'exécution vous semble-t-elle cohérente avec les calculs théoriques sur la complexité ?

Maintenant que vous avez vérifié que la complexité est bien améliorée, vous envoyez votre travail à votre patron. 10 minutes plus tard, il vous répond que votre algorithme a été implémenté dans la chaîne de recherche du programme

complet. Le soir même vous recevez un mail vous indiquant que les espions ont été appréhendés. Vous vous replongez alors dans le Canard de l'espace avec le sentiment d'avoir accompli votre travail. Bien joué, M. Anderson !