TD INFORMATIQUE

RECHERCHE D'ALIGNEMENTS DANS UN NUAGE DE POINTS

Le TD suivant se propose de déterminer tous les alignements de k-points ($k \geqslant 3$) présents dans un nuage de points aléatoire. Il y a quantité de méthodes pour y parvenir. Les ambitions algorithmiques de celle qui est présentée ici sont de trouver :

- tous les alignements
- pour un nombre k de points quelconque
- chaque alignement étant unique (mémorisé une seule fois)
- Et dans un minimum de temps.

Pour cela on donne le pseudo-code suivant :

Liste d'alignements vide

Faire une boucle sur tous les points :
- Trier les N-1 points restants selon l'ordre lexicographique (x, y)
- Calculer les angles[modulo pi]/horizontale de tous les points [0, pi]
- Trier les points (tri stable !) selon les angles
- **Balayer la liste des angles :**
  - identifier chaque alignement
  - si le point de référence est l'extrémité basse
    - on garde l'alignement
    - on garde en mémoire ses extrémités pour le tracé

I - ANALYSE DU PSEUDO-CODE

a - Proposer un coût algorithmique à chacune des lignes du pseudo code.
b - En déduire le coût global. Quels sont les coûts déterminants ?
Est-ce un problème de faire plusieurs tri en terme de coût, et quel en est l'intérêt ?
c - Argumenter que l'on a bien trouvé tous les alignements.
d - Justifier qu'il n'y aura pas de doublon dans les alignements trouvés, et revenir sur le coût algorithmique associé.
e - Question ouverte : can we do better ?

II - NUAGE DE POINTS ☁️

Un nuage totalement aléatoire avec des coordonnées en nombre flottants donnerait peu d'alignement parfait. Pour y remédier on considère des points (i, j) sur une grille carrée tels que $0 \leqslant i \leqslant N-1$ et $0 \leqslant j \leqslant N-1$. Chaque point est un tuple (i, j) de coordonnées entières.

On importe le module aléatoire avec **import random as rnd**.

a - Ecrire une fonction **pointAleatoire**(N, p = 0.5) qui prend en argument la taille N de la grille et p une fraction de points. [si p = 0.5 on garde en moyenne un point sur deux p = 0.01 un sur cent etc…] Pour sélectionner les points on les passe tous en revue et on garde (ou pas) chaque point aléatoirement avec une probabilité p. La fonction renvoie la liste des points aléatoires [P1, P2, etc …]. On ne connait donc sa taille qu'en moyenne : indiquer les nombres min, max et moyen de points dans le nuage.

b - Ecrire une fonction **AfficheNuage**(monNuage, option="go") qui prend en argument le nuage de points [P1, P2, P3, etc…] et l'affiche dans une fenêtre graphique.
RQ : Ne pas déclencher l'affichage avec plt.show() dans la fonction, il y aura d'autres choses à tracer.

c - Ecrire une fonction **AfficheAlignement**(mesAlignements) qui prend en argument la liste des alignements et les trace tous dans la fenêtre graphique. Chaque alignement sera la donnée d'un tuple (pStart, pStop, n) où pStart et pStop sont les points aux deux extrémités => tuples (i, j) et n le nombre de points présents dans l'alignement [information inutile pour le tracé].

III - OPÉRATIONS DE TRI

Le pseudo-code ci-dessus s'appuie sur deux tris :
- le premier tri les points dans le plan selon l'ordre lexicographique
- le second tri les N-1 points M restant autour d'un point référence O, en fonction de l'angle que fait la droite (OM) par rapport à l'horizontale [modulo π]

Pour cela nous utiliserons le tri python TimSort qui provient (a priori) de C++ : **maListe.sort( )**
C'est un **tri fusion in-place** !!! Bref le top du top … et c'est gratuit !

a - A partir du cours [tri par insertion ou tri fusion] expliquer ce qu'est un tri stable.
Quel est ici l'enjeu pour notre problème quant à la stabilité du tri ?

Enfin pour pouvoir réaliser les opérations de tris à la fois sur les coordonnées et sur l'angle, nous allons devoir recréer une liste de points P avec les deux informations : P —> (angle, (i, j))
Il s'agira alors de trier la liste de tuple selon différents indices de ce tuple 0 pour l'angle et 1 pour les coordonnées. C'est justement ce que permet la commande suivante :

```
#tri la liste de tuple selon l'indice donné du tuple.
def triTuple(listeTuple, indice):
    import operator
    listeTuple.sort(key = operator.itemgetter(indice))     #LA formule magique ….
```

b - Copier la fonction **triTuple** dans votre programme.

c - Ecrire une fonction **angle**(Pref, P), qui prend en argument le point de référence Pref et un autre point P, et qui renvoie l'angle [modulo π] de la droite reliant ces deux points par rapport à l'horizontale. Rq : Il y a plusieurs façon de procéder j'ai choisi l'intervalle]-π/2, +π/2] par commodité.
On utilisera pour cela la fonction arctan et on traitera le cas particulier associé à une droite verticale. On arrondira à 10 chiffres **round**(alpha, 10) pour éviter une précision excessive.

IV - RECHERCHE DES ALIGNEMENTS

Ecrire une fonction **rechercheAlignements**(nuagePoint) qui prend en argument le nuage de points et qui renvoie deux listes : align, fullAlign

- **La première pour tracer les alignements** (cf II-c) :
align est une liste de tuple (pStart, pStop, n) où pStart et pStop sont les points aux deux extrémités => tuples (i, j) et n le nombre de points présents dans l'alignement.

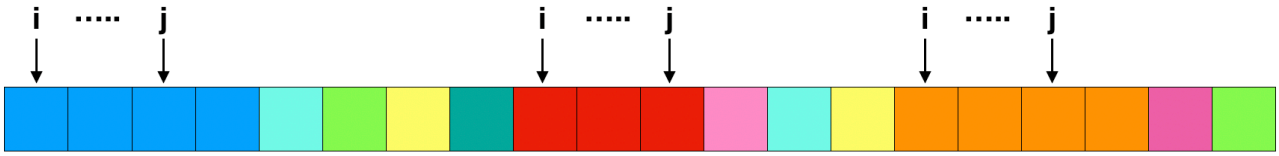
- **La seconde pour garder en mémoire tous les points alignés** :
fullAlign est la liste des points de l'alignement [pRef, P1, P2, etc…]. Noter bien que celle-ci part nécessairement de **pRef** ! Pourquoi ?

a - Initialiser les listes align, fullAlign et lancer la boucle sur les points du nuage.
b - Soit pRef : construire la liste **anglePoint** des N-1 points restant sous la forme de tuples (angle, (i, j)). On calculera donc l'angle.
c - Trier la liste obtenue avec **triTuple**, d'abord selon les positions puis selon les angles.

d - **Recherche d'alignements par balayage :** ça se complique !

Soit un alignement représenté par la liste suivante des angles obtenus autour du pRef :
On peut y voir trois zones de couleur homogène : bleue, rouge et orange, qui correspondent à des angles constants par rapport à l'horizontale, autrement dit des alignements ! Il y a donc trois alignements impliquant le point pRef. Notez bien que pRef ne peut pas être dans la liste même.

Toutefois pour ne pas compter les alignements en double, on ne retiendra l'alignement que si pRef est le plus petit des points alignés dans l'ordre lexicographique.



L'identification des alignements procède par un balayage à deux curseurs :
- Soient i et j initialement identiques à l'extrémité gauche de l'alignement (hors pRef).
- On essaye de décaler j vers la droite jusqu'à extrémité droite (sans sortir de la liste).
- Si l'angle change : on est arrivé à la fin d'un alignement !
    - Contrôler la taille de l'alignement obtenu (au moins trois points avec pRef)
    - Vérifier si pRef est bien un départ d'alignement
    - Dans ces conditions, on remplit **align** et **fullAlign** —> boucle for.

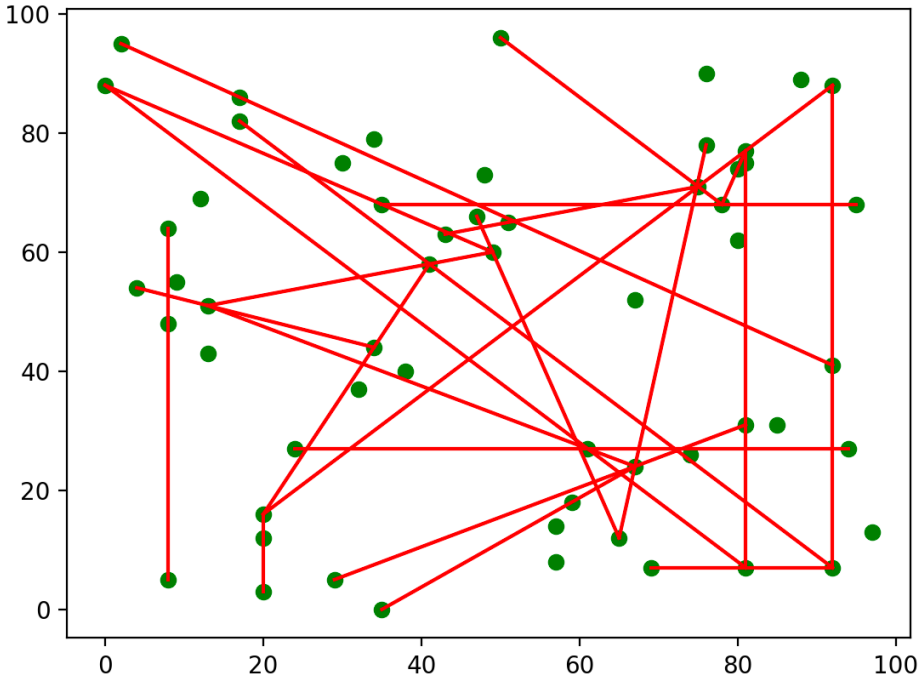
On remet i et j identiques mais sur l'indice suivant.
Etc.. jusqu'à la fin du tableau

En pratique cela doit se traduire par une **double boucle while**, a vous de montrer que le coût sera pourtant bien linéaire.

e - Tester votre code, affichages à l'appui pour N = 5 ou N = 10.
Le graphique étant complet on déclenche alors l'affichage avec **plt.show( )**

Pour des tests plus précis vous pouvez vous appuyer sur des nuages fait à la main :
nuage = [(0,0), (1,0), (2,0), (1,1), (3,1), (1,2), (0,3), (2,3), (3,3)]

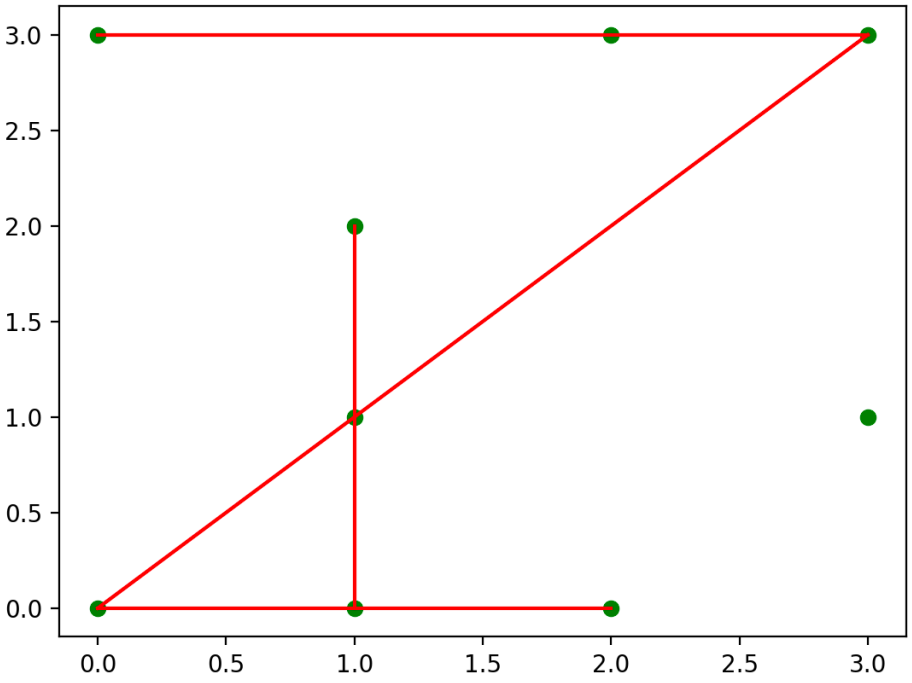
N = 100 et P = 0.006



nuage = [(0,0), (1,0), (2,0), (1,1), (3,1), (1,2), (0,3), (2,3), (3,3)]

Alignement —> [((0, 0), (2, 0), 3), ((0, 0), (3, 3), 3), ((1, 0), (1, 2), 3), ((0, 3), (3, 3), 3)]

AlignementComplet
[[(0, 0), (1, 0), (2, 0)],
[(0, 0), (1, 1), (3, 3)],
[(1, 0), (1, 1), (1, 2)],
[(0, 3), (2, 3), (3, 3)].]



FIN