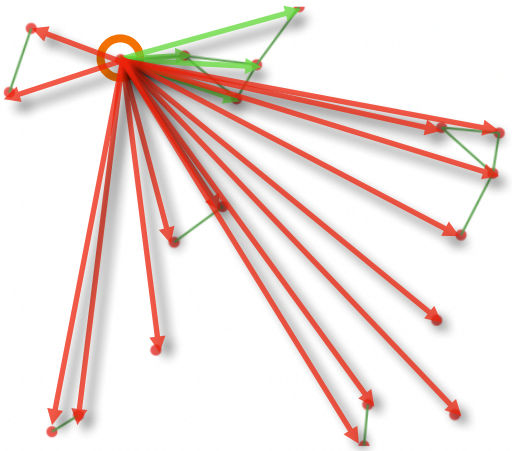
# Rapport Algo :

Azmatally Rayan
Duhamel Alexandre
Groupe 7

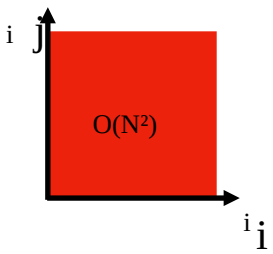
**SOMMAIRE :**



## CONCEPTS & IDÉES



x nb de points = nb opérations



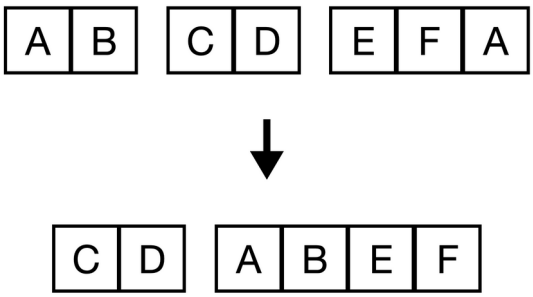

## 1ère idée : naïve

Étape 1 (**test distance)** : On sélectionne un point indice i de la liste. On parcours ensuite chaque autres points j et on compare leur distance à celle du point sélectionné. Si la distance est inférieur au seuil, on ajoute à une **tableau** indice i le points j.

```
L=[points[i] for i in range(len(points))]
for i in range(len(points)):
        for j in range(len(points)):
            if distance > points[i].distance_to(points[j]) :
                L[i].append(points[j])
```

**Problème** : Considérons A et B deux points voisins. Nous avons une liste des voisins de A et celle des voisins de B. Cependant, pour trouver les parties connexes il faut regrouper les voisins de B et ceux de A puisque si B et connecté directement à C alors A l'est également indirectement. ( Cette partie nous a posé beaucoup de problème au démarrage)

Étape 2 **(arranger la liste)** : On définit une fonction **union** (L) qui règle le problème cité au dessus. On utilise pour ce faire des **sets** python. Cela permet d'effectuer des opérations tels que l'union/l'intersections.

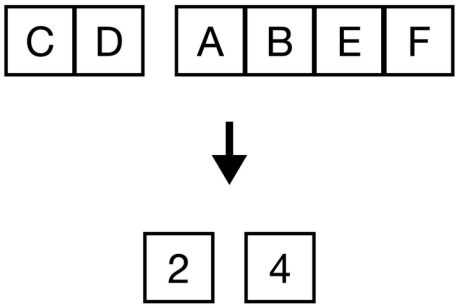



Entrée : Le tableau crée dans l'étape 1.
Sortie : Un tableau des parties connexes.

```python
def union(l) :
    out = []
    while len(l)>0:
        first, *rest = l
        first = set(first)
        lf = -1
        while len(first)>lf:
            lf = len(first)
            rest2 = []
            for r in rest:
                if len(first.intersection(set(r)))>0:
                    first |= set(r)
                else:
                    rest2.append(r)
            rest = rest2

        out.append(first)
        l = rest
    return out
```

Étape 3 : **(formater la sortie)** Prendre chaque sous liste du tableau, calculé leur taille et les rangées dans une liste. Renvoyer la liste trié par ordre décroissant.




Améliorations faites: Dans l'étape 1, la deuxième boucle peut aller de **i** à **n** ( symétrique).

**BILAN** : Complexité : **O(n²)** (deux boucles imbriquées étape1)
Méthodes fonctionnelle mais complexité trop élevée

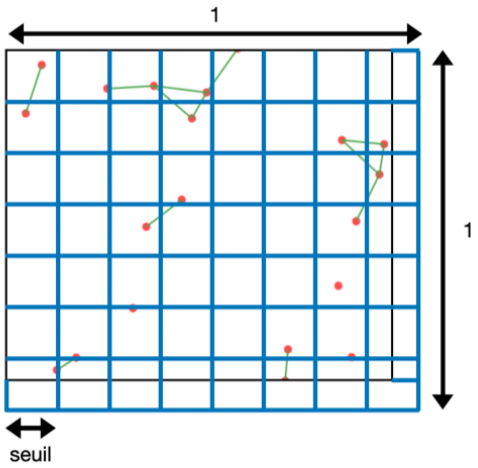
**il faut réduire la zone de recherche des voisins de chaque points**

CONCEPTS & IDÉES

## 2ième idée : quadrants **(idée retenue)**

**Concept** : Créer des quadrants de taille du seuil de manière à n'avoir qu'à comparer avec les points se situant dans des quadrants voisins.

(NB : Idée peu détaillée car c'est la version finale donnée dans le code.)

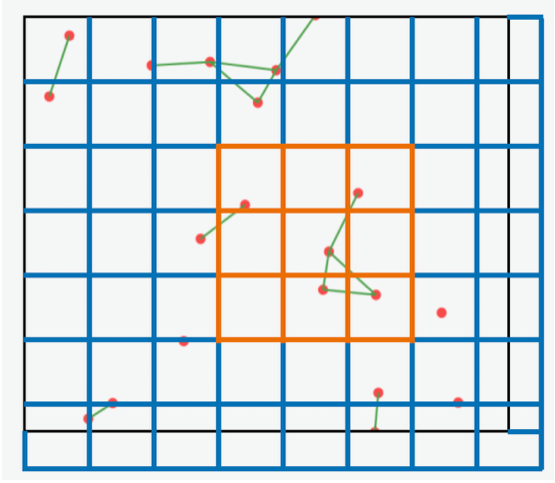



Étape 1 : **(Initialisation)** Création des quadrants rangés dans un **tableau** quad**.**
Chaque quadrant est un carré de taille **seuil**.

quad[numéro quadrant]=[(x_min,x_max),(y_min,y_max)]

Étape 2 : **(Remplissage)** On parcours chaque points du graphe (O(n)) et on lui associe un quadrant à l'aide d'un **dictionnaire** quad_point en fonction de ses coordonnées. On crée également un dictionnaire point_quad qui associe à chaque point le quadrant auquel il appartient ( évite la recherche ).

Étape 3 : **(Regroupement)** On parcours chaque points, on récupère le numéro de son quadrant = point_quad[i], on crée une liste ens_quad_voisin qui contient l'ensemble des points du quadrant concerné + ceux des quadrants voisins (ceux adjacents).



Étape 4 : **(Comparaisons)** Dans la même boucle que celle de l'étape3, on compare la distance entre le point i à tous ceux de la liste ens_quad_voisin et on les ajoute à L[i] si celle-ci est inférieur au seuil.

Étape 5 : **(formater la sortie)** Même étape que celle de l'idée 1.

**BILAN :**

**-- On a bien réduit la zone de recherche des points.**

**-- Utilisation de dictionnaires permettant d'avoir un code plus explicite ( point_quad[numéro_quad] ).**

**-- Gains de temps considérable au chronomètre** (cf perf).

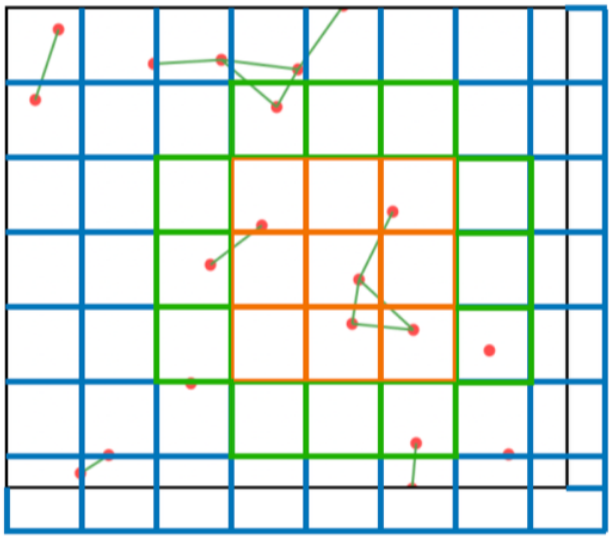
———————————CONCEPTS & IDÉES———————————

# 3ième idée : quadrants « diagonaux »

**Concept** : C'est la même idée que l'idée 2 sauf que la taille du quadrant est de **(seuil/sqrt(2))**

**Utilité :** Avec cette taille de quadrant, la taille de la diagonale est de **seuil**. Ainsi, nous pouvons directement regrouper ensemble les points d'un même quadrant.

**Problème** : Bien que l'on gagne du temps sur le quadrant central, ce choix implique d'englober des quadrants adjacents supplémentaires. En effet la taille en longueur n'est plus de seuil.



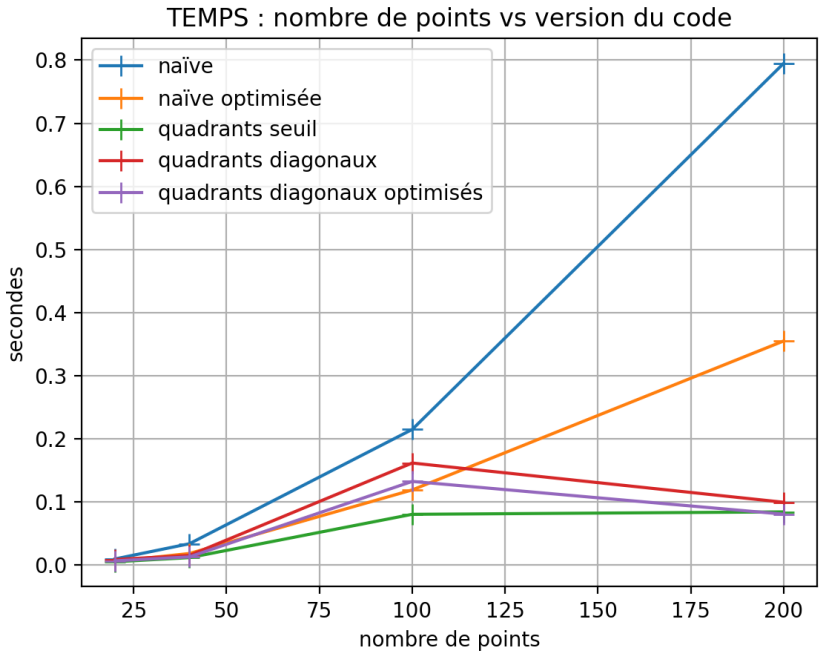
**BILAN** :

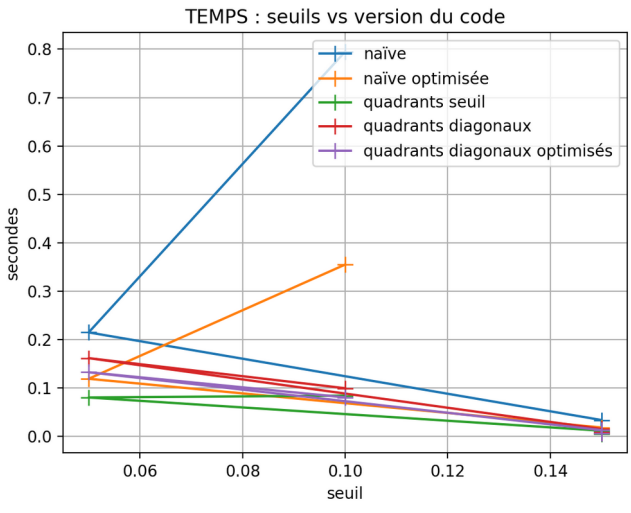
**-- Cette méthode n'est pas plus efficace, dans le cas général, en raisons du rajout de quadrants.**

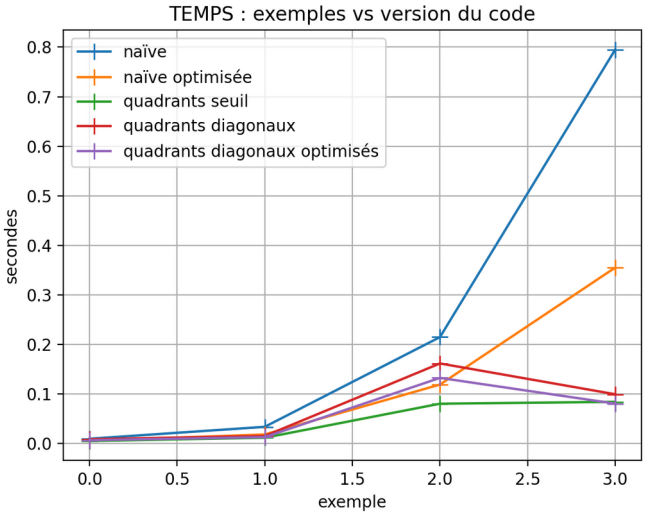
**-- Dans une situation où les points sont regroupés par paquet (ex : cluster épidémique) la méthode est très efficace.**

---

## PERFORMANCES EN TEMPS



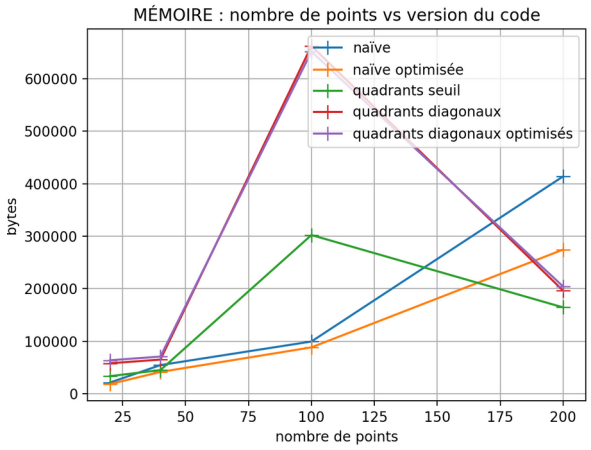
Nous avons tracé des courbes pour voir la performance en temps de nos algorithmes (méthode naïve, naïve optimisée, quadrants seuil, quadrants diagonaux, quadrants diagonaux optimisés) en fonction : du nombre de points en entrée et de la taille du seuil.

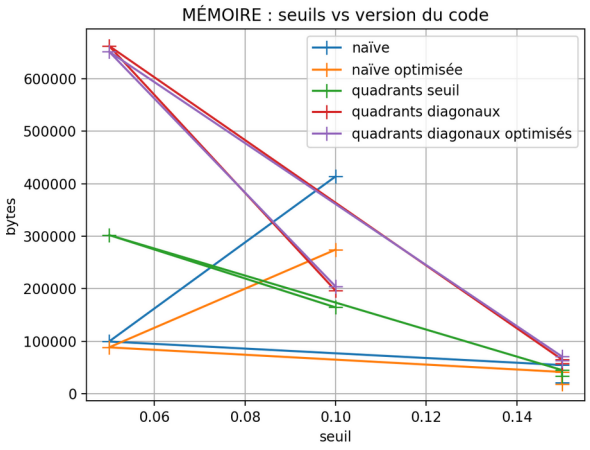
Nous pouvons voir que,en toute logique, le temps est proportionnel au nombre de points pour chacune des méthodes. En revanche l'algorithme **le plus rapide** pour n'importe quelle quantité de points en entrée est ***quadrants seuil*** (idée 2). Cela s'explique par le fait que c'est la méthode qui à besoin de faire le moins de comparaisons points à points car elle compare uniquement ses points à ceux de ses voisins, elle est donc moins affectée par le nombre de points en entrée que les autres méthodes.

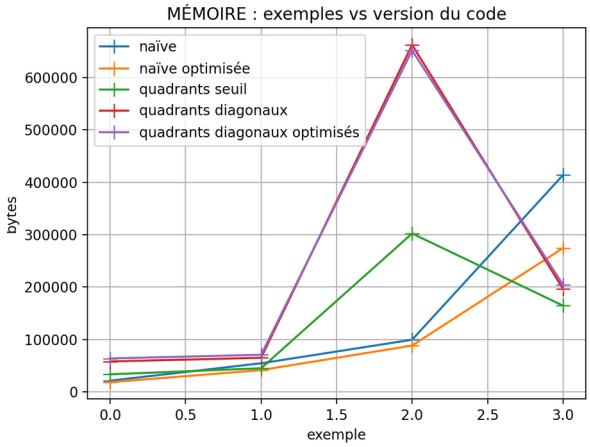
Nous avons également remarqué que plus le seuil est petit, plus nos algorithmes avec quadrants sont lents. En effet, cela implique qu'il y a plus de quadrants a former ( en revanche cela n'affecte pas la boucle principale puisque nous parcourons toujours les points et non les quadrants). Ainsi, l'algorithme le plus rapide pour n'importe quel seuil entrée est ***quadrants seuil*** (idée 2).

**L'algorithme le plus performant en temps, que nous allons alors retenir,est donc l'algorithme de *quadrants seuil.***

## PERFORMANCES EN MÉMOIRE






Nous avons également tracé des courbes pour voir la performance en mémoire de nos programmes (méthode naïve, naïve optimisée, quadrants seuil, quadrants diagonaux, quadrants diagonaux optimisés), toujours en fonction : du nombre de points en entrée et de la taille du seuil.

Nous avons remarqué que plus il y a de points, plus nos algorithmes sont coûteux en mémoire, ce qui est cohérent car nous stockons les points pour après retourner le nombre de points dans chaque structures connexes. On remarque que les algorithmes les plus coûteux sont ceux qui utilisent la méthode des quadrants. Cela s'explique par le fait qu'il faille mémoriser pour chaque points à quel quadrant il appartient mais également pour chaque quadrant, quels points il contient. En revanche, les algorithmes naïfs n'ont pas besoins de tels informations, une liste de liste fait l'affaire, l'espace en mémoire et donc directement corrélé aux nombres de points en entrée.

Nous avons également remarqué que plus le seuil est petit, plus nos algorithmes sont coûteux en mémoire. L'espace en mémoire des méthodes par quadrants est directement corrélé aux nombres de points en entrée mais également à la taille du seuil. En effet, plus le seul est petit, plus il y a de quadrant à créer, et plus de stockage sera nécessaire pour encoder notre problème.

Les algorithmes les plus **performants en mémoire** sont donc **les algorithmes naïfs.**

## ——————————————COMPLEXITÉS——————————————-

Il est assez étonnant, au vu des différences de performances, de remarquer que tous nos algorithmes sont en **O(n²).** En effet, la fonction union (cf. code) possède une complexité quadratique dans le pire des cas. Ce cas correspond à celui où tous les points seraient regroupés dans un cercle de rayon seuil. En effet, le tableau **L** (défini dans l'idée 1) serait de dimensions n x n puisque chaque point aurait pour voisin l'ensemble des points. Le parcours de ce tableau serait donc en **O(n²)**. Cependant, il est important de nuancer ce fait. Dans la recherche de graphes connexes, cette situation est rare ou du moins elle ne nécessite généralement pas d'algorithme puisqu'elle est visible à « l'œil ».

Ainsi, il existe des situations extrêmes où nos trois algorithmes ont la même complexité temporelle. Cependant, c'est bien la méthode quadrant numéro 3 qui disposerait, en **moyenne**, d'une complexité plus faible.

NB : ce cas extrême cause la même chose sur le parcours des points à l'aide des quadrants.

## ——————————————**CONCLUSION**——————————————-

En conclusion, afin de pouvoir déterminer quel algorithme choisir, il faut établir un **cahier des charges** afin de connaître la performance à optimiser. Ici, nous faisons le choix de la méthode **"quadrant seuil"** puisque l'on cherche à optimiser le temps d'exécution.

Conclusion personnel sur le projet :

Ce projet nous a appris de nombreuses compétences précieuses pour nos futurs métiers. Nous avons appris à travailler en binôme, à communiquer pour transmettre nos découvertes et nos idées, afin de capitaliser sur nos expériences respectives. Nous avons également organisé des réunions régulières pour faire le point sur l'avancement du projet et les tâches restantes.

Nous nous sommes également améliorés en programmation Python, en utilisant les fonctions built-in de Python pour alléger notre code et le rendre plus clair et lisible. Nous avons également commenté nos différentes versions de code pour décomposer ce problème complexe en différents sous-problèmes plus abordables. Comme on dit, "un problème bien posé est à moitié résolu".
De plus, nous avons tous deux compris l'importance de la complexité d'un code. Un code plus lent sur un exemple simple devient vraiment lent sur un exemple beaucoup plus complexe. Nous avons donc dû changer de point de vue sur le problème, prendre du recul et changer de stratégie. C'est ainsi que nous nous sommes réellement penchés sur la complexité d'un algorithme pour chercher à l'optimiser. Cela a été un challenge intéressant, compétitif, avec une envie de se dépasser pour obtenir de meilleurs résultats .

```python
#!/usr/bin/env python3
import numpy as np
from timeit import timeit
from sys import argv
from geo.tycat import tycat
from geo.point import Point
from itertools import combinations
from geo.point import Point
from geo.segment import Segment


def load_instance(filename):
    # Entrée : le nom du fichier qu'on veut traiter en format
".pts"
    # Sortie : le seuil à partir duquel 2 points forment une
structure connexe et la liste des points

    with open(filename, "r") as instance_file:
        lines = iter(instance_file)
        distance = float(next(lines))
        points = [Point([float(f) for f in l.split(",")]) for l in
lines]

    return distance, points


def creation_quadrant(distance,points) :
    # Entrée : le seuil à partir duquel 2 points forment une
structure connexe et la liste des points
    # Sortie : quad_point et point_quad
    #
    # Cette fonction créer une liste de quadrant et renvoie deux
dictionnaires :
    #       1) quad_point qui associe à chaque quadrant la liste
de points qu'il contient
    #       2) point_quad qui associe à chaque point le quadrant
auquelle il appartient

    quad=[]
    quad_point={}
    quad_voisin={}
    point_quad={}
    numero_point=-1
    taille_quad= distance
    if int(1/taille_quad) < 1/taille_quad:
        nb_quadrant=int(1/taille_quad)+1
    else:
        nb_quadrant=int(1/taille_quad)
    for i in range(0,nb_quadrant):
        for j in range(0,nb_quadrant):
            p=j+nb_quadrant*i
```

```
            quad.append([(i*taille_quad,(i+1)*taille_quad),
(j*taille_quad,(j+1)*taille_quad)])
            quad_point[j+nb_quadrant*i]=[]
            quad_voisin[p]=[p,p-1,p+1,p-
nb_quadrant,p+nb_quadrant,p+nb_quadrant-1,p+nb_quadrant+1,p-
nb_quadrant+1,p-nb_quadrant-1]
    for point in points:
        numero_point+=1
        x= point.coordinates[0]
        y= point.coordinates[1]
        for k in range( len(quad) ):
            x_min,x_max=quad[k][0]
            y_min,y_max=quad[k][1]
            if x  < x_max and x >= x_min:
                if y  < y_max and y >= y_min:
                    quad_point[k].append(numero_point)
                    point_quad[numero_point]=k
                    break
    return quad_point,point_quad,nb_quadrant,quad_voisin



def
voisin(points,point_quad,quad_point,nb_quadrant,distance,quad_vois
in):
    # Entrée : liste des points, un dicotionnaire qui associe à
chaque quadrant la liste de points qu'il contient,
    #          un autre dictionnaire qui associe à chaque point le
quadrant auquelle il appartient, le seuil à partir duquel 2 points
    #          forment une structure connexe et la liste des
points
    # Sortie : liste des potentiels voisins
    #
    # Cette fonction parcourt chaque points et créer une liste
ens_quad_voisin qui regroupe
    # l'ensemble des potentiels voisins de ce points ( ceux qui se
trouve dans un quadrant adjacent )

    seg=[]
    L = [[] for _ in range(len(points))]
    for i in range (len(points)):
            ens_quad_voisin=[]
            k=point_quad[i]
            for h in quad_voisin[k]:
                try:
                    ens_quad_voisin+=quad_point[h]
                except Exception:
                    pass
            for elem in ens_quad_voisin:
                if distance >
points[i].distance_to(points[elem]) :
                    seg.append(Segment([points[i],points[elem]]))
```

```
                    L[i].append(points[i])
                    L[i].append(points[elem])
    return L


def union(l) :
    # Entrée : Liste des multiples structures connexes trouvées
avec des points partagés
    # Sortie : Création des connexes et suppression des doublons

        out = []
        while len(l)>0:
            first, *rest = l
            first = set(first)
            lf = -1
            while len(first)>lf:
                lf = len(first)

                rest2 = []
                for r in rest:
                    if len(first.intersection(set(r)))>0:
                        first |= set(r)
                    else:
                        rest2.append(r)
                rest = rest2

            out.append(first)
            l = rest
        return out

def print_components_sizes(distance, points):
    # Entrée : le seuil à partir duquel 2 points forment une
structure connexe et la liste des points
    # Sortie : void function
    #
    # Affichage des tailles triees de chaque composante

    pass

quad_point,point_quad,nb_quadrant,quad_voisin=creation_quadrant(di
stance,points)

L=voisin(points,point_quad,quad_point,nb_quadrant,distance,quad_vo
isin)
    LL = union(L)
    LL = [len(LL[i]) for i in range(len(LL))]
    LL.sort()
    LL.reverse()
    print(LL)
    #tycat(points,seg)
```

```
def main():
    """
    ne pas modifier: on charge une instance et on affiche les
tailles
    """
    for instance in argv[1:]:
        distance, points = load_instance(instance)
        print_components_sizes(distance, points)

main()
```