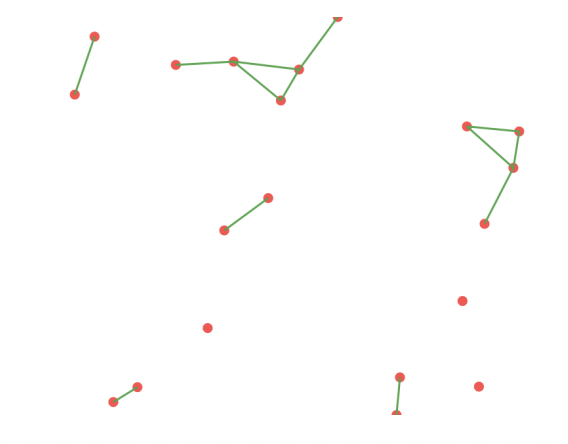
Rapport Algo :

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

1ère idée : naïve

Étape 1 : Parcourir tous les points 2 à 2 pour créer les bons segments en fonction de leur distance entre eux (2 boucles for imbriquées) :



x nb de points = nb opérations

i

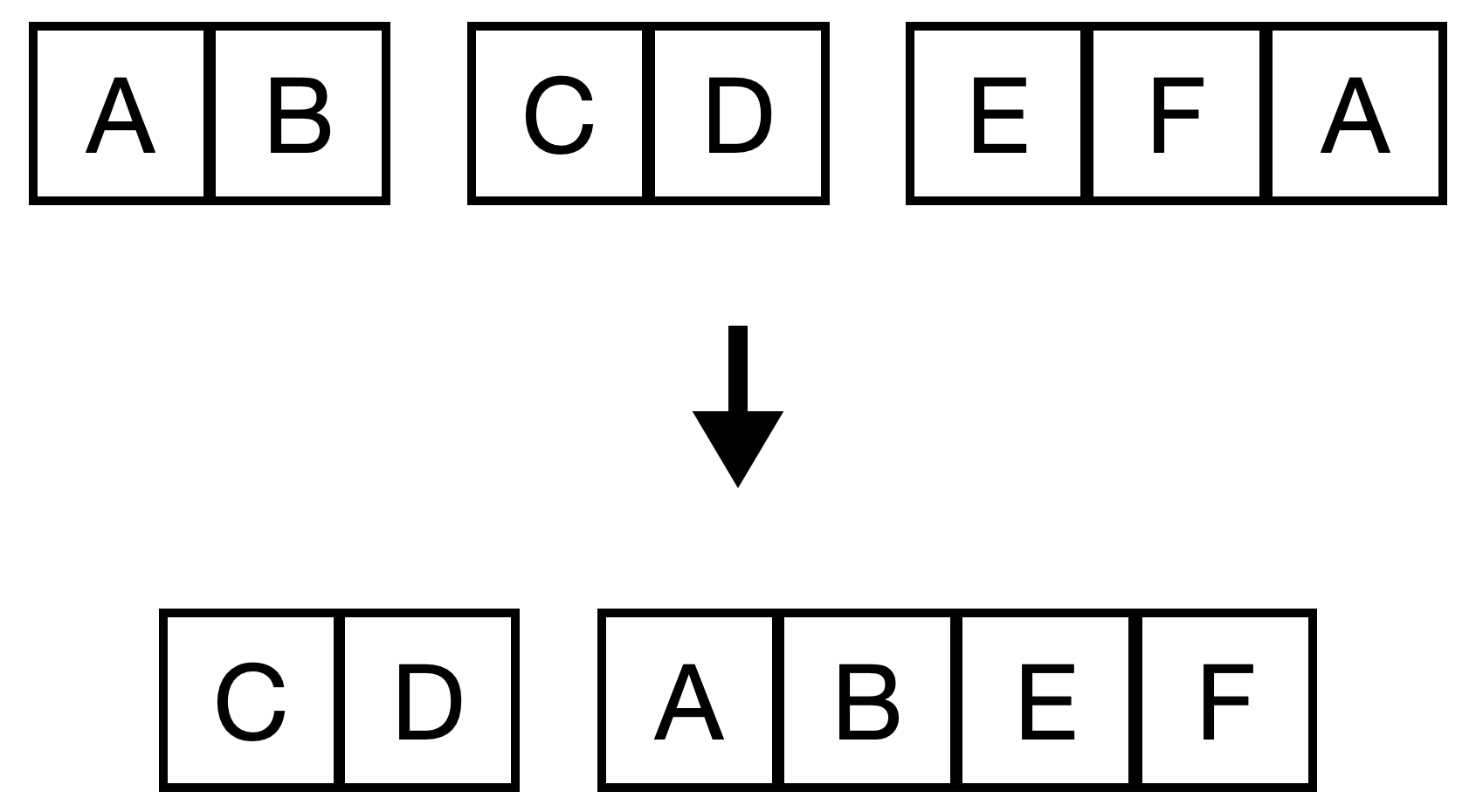
j

i

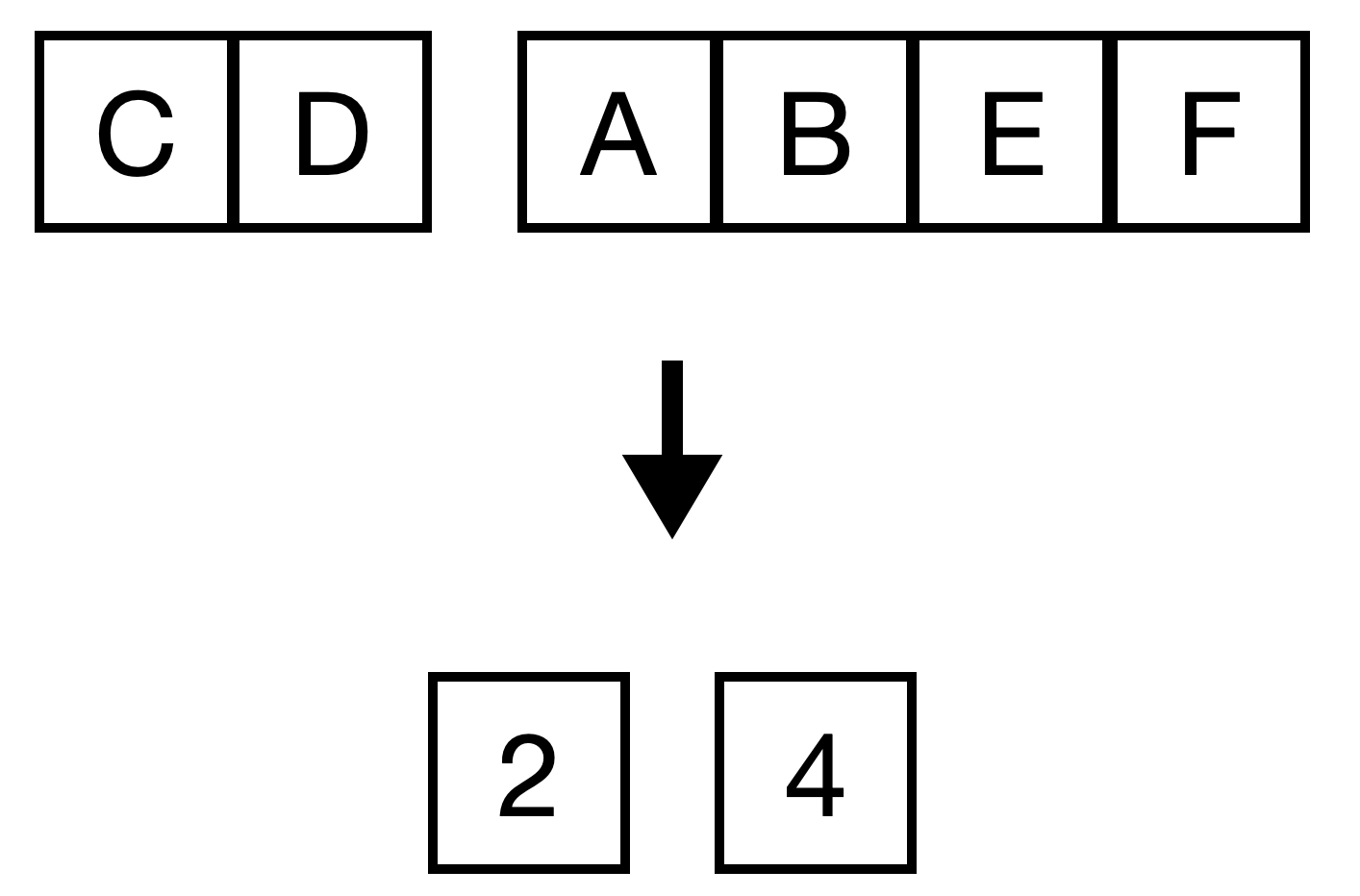
i

O(N²)

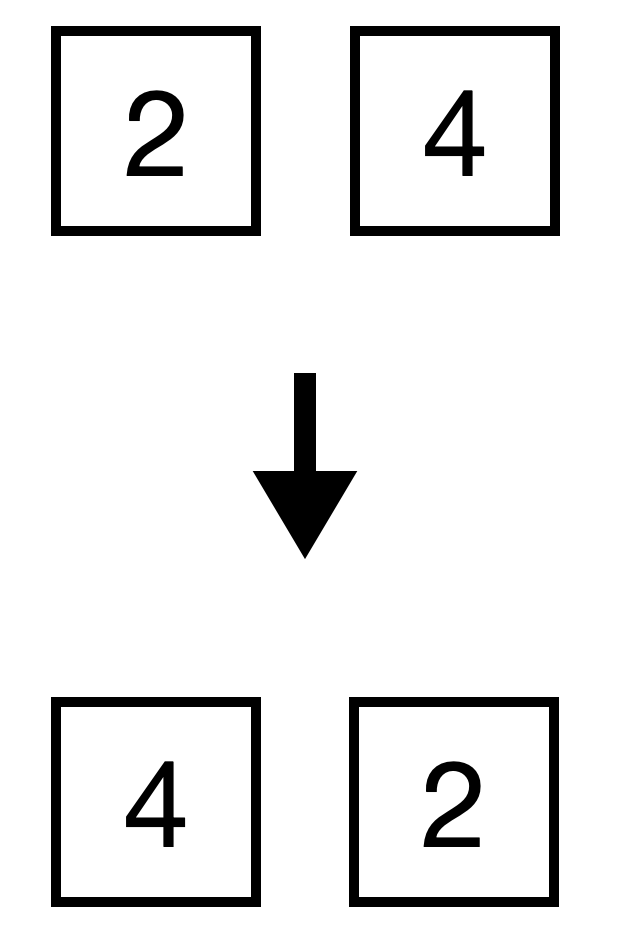
Étape 2 : Créer une fonction permettant de regrouper par paquets les doublons contenus dans plusieurs liste et de n’en garder qu’un :



Étape 3 : Prendre la taille de chacune des listes et en créer une nouvelle avec :



Étape 4 : Trier la liste dans l’ordre décroissant :



—> Cette méthode est facile à mettre en place et requiert peu de planification pour qu’elle marche. En revanche, elle est très peu performante.

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

2nd idée : naïve - opérations répétées

Étape 1 : Parcourir tous les points 2 à 2 pour créer les bons segments en fonction de leur distance entre eux (2 boucles for imbriquées). La première boucle parcours tous les points (i allant de 1 à n) et la seconde le reste des points (j allant de i à n) :

i

j

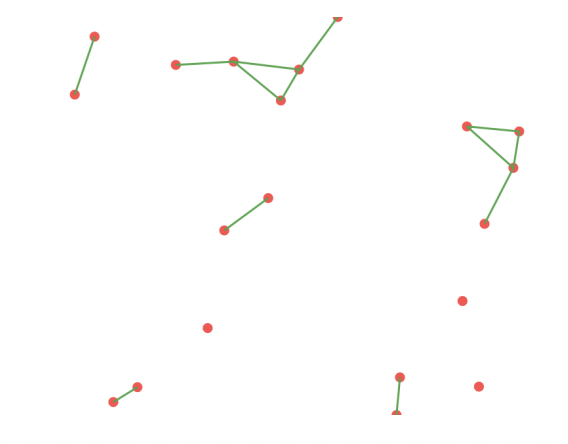
Étape 2-4 : inchangées

—> Cette méthode utilise le fait qu’un segment relie deux points entre eux, il n’est donc pas nécessaire de regarder tous les couples de points. Cela permet d’avoir un code plus rapide.

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

3ième idée : quadrants

Concept : Créer des quadrants de taille du seuil de manière à n’avoir qu’à comparer avec les points dans le quadrant voisin.

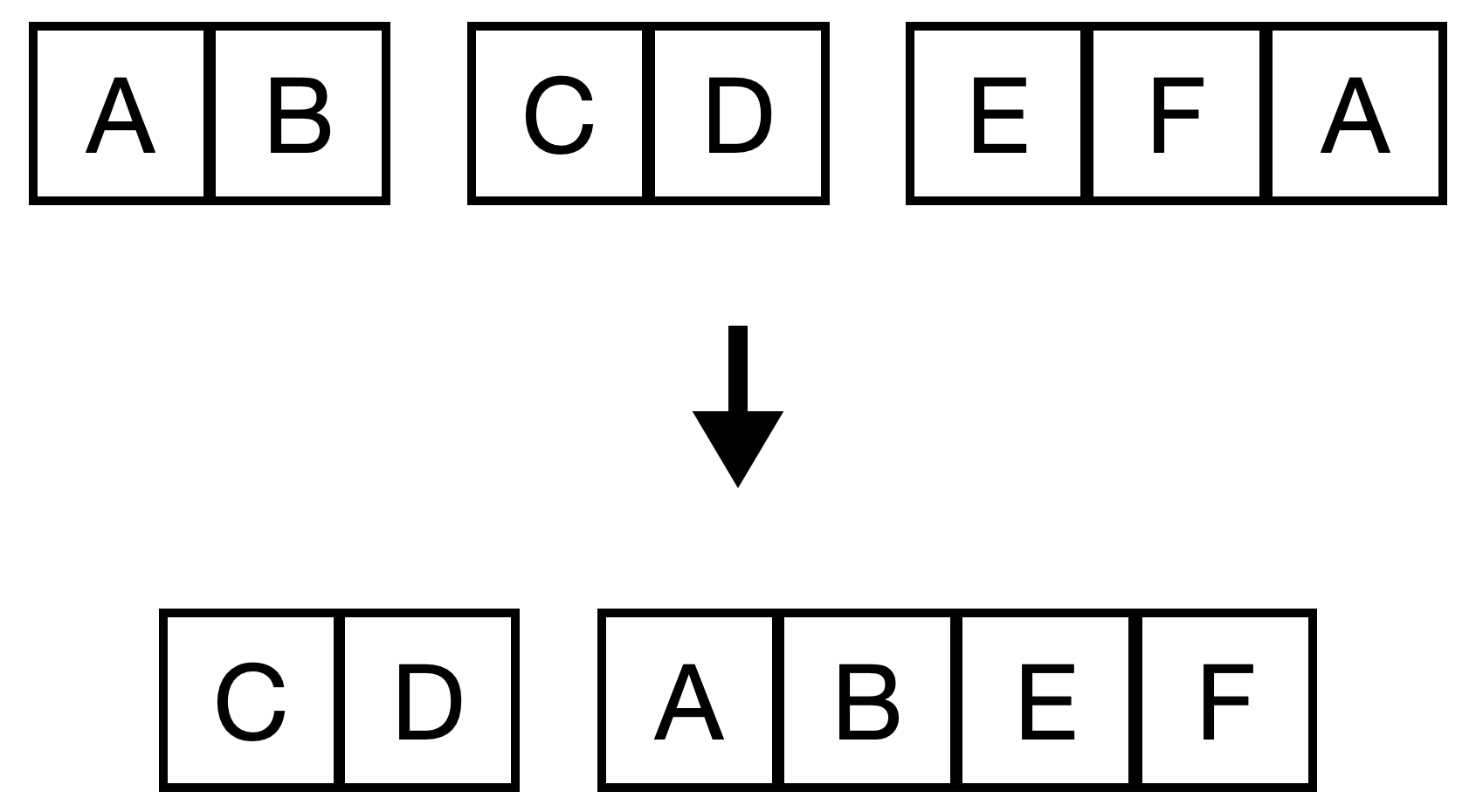


Étape 1 : Création des quadrants avec des dimensions seuil x seuil.

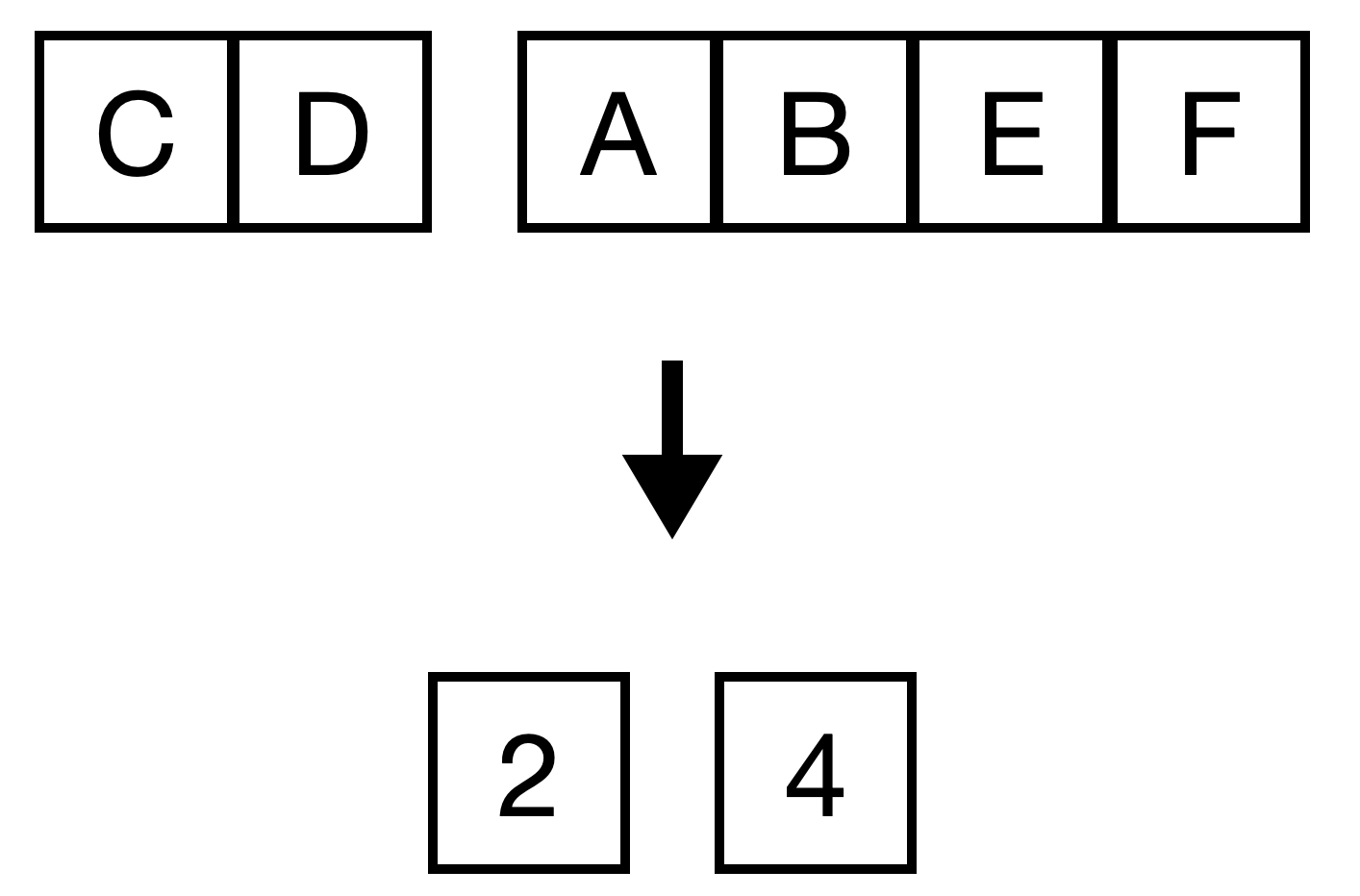
Étape 2 : Remplir les différents quadrants avec les bons points.

Étape 3 : Comparer la distance entre chaque points des quadrants voisins —> nettement moins de comparaisons qu’avec les idées précédentes.

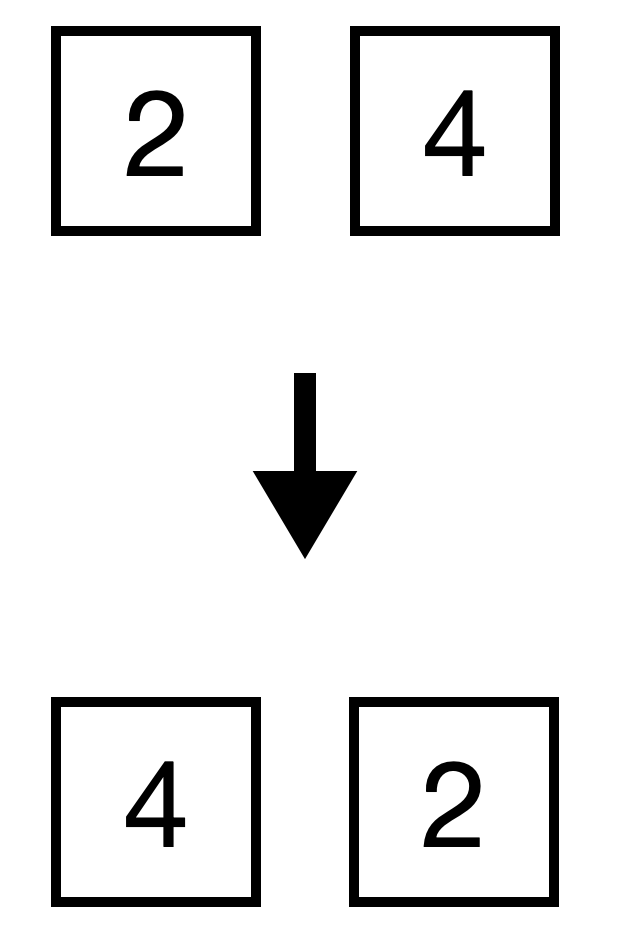
Étape 4 : Créer une fonction permettant de regrouper par paquets les doublons contenus dans plusieurs liste et de n’en garder qu’un :



Étape 5 : Prendre la taille de chacune des listes et en créer une nouvelle avec :



Étape 6 : Trier la liste dans l’ordre décroissant :

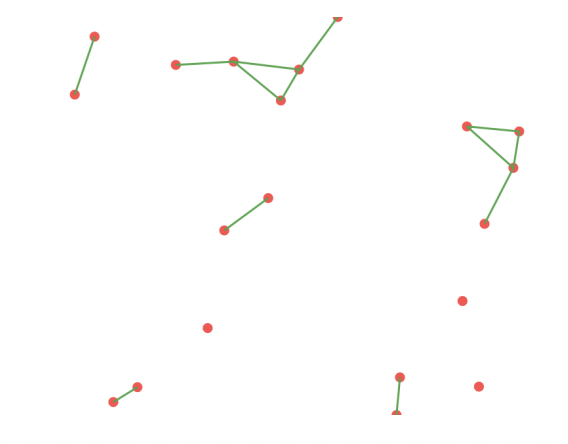


—> Cette méthode divise le plan en quadrants, pour travailler par zones et n’avoir à comparer que les points voisins. Elle utilise également comme SDD des dictionnaires, agissant comme des tables de hachage, permettant une recherche rapide des différents points. La méthode des quadrants permet de rendre l’algorithme nettement plus rapide.

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

4ième idée : quadrants diagonaux

Concept : Créer des quadrants de taille du de manière à n’avoir qu’à comparer avec les points dans le quadrant voisin. De plus les points dans un même quadrant froment une structure connexe immédiatement.

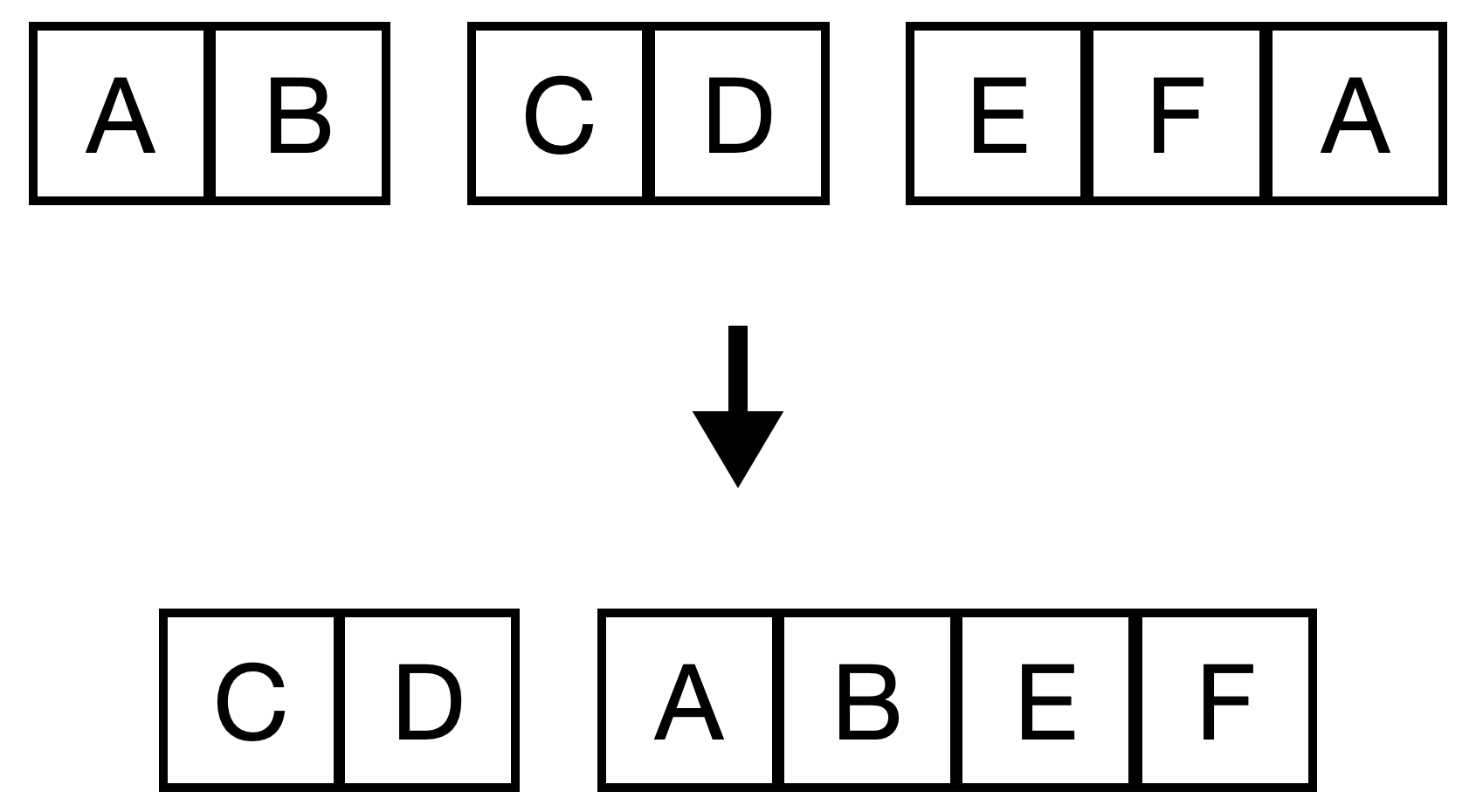


Étape 1 : Création des quadrants avec des dimensions x , tous les points dans un quadrant forment une structure connexe, cela règle les problèmes de diagonale.

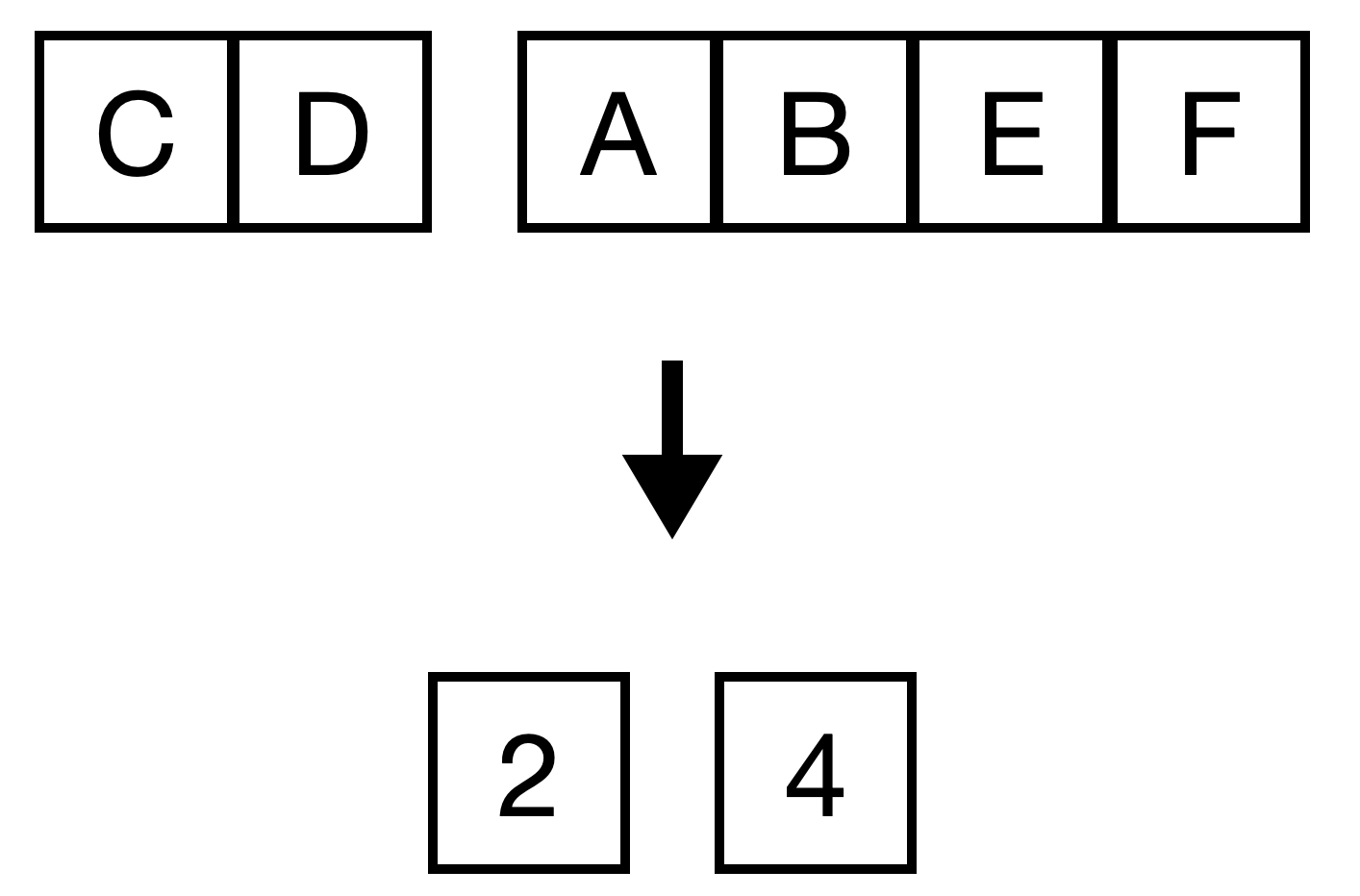
Étape 2 : Remplir les différents quadrants avec les bons points.

Étape 3 : Comparer la distance entre chaque points des quadrants voisins —> nettement moins de comparaisons qu’avec les idées précédentes.

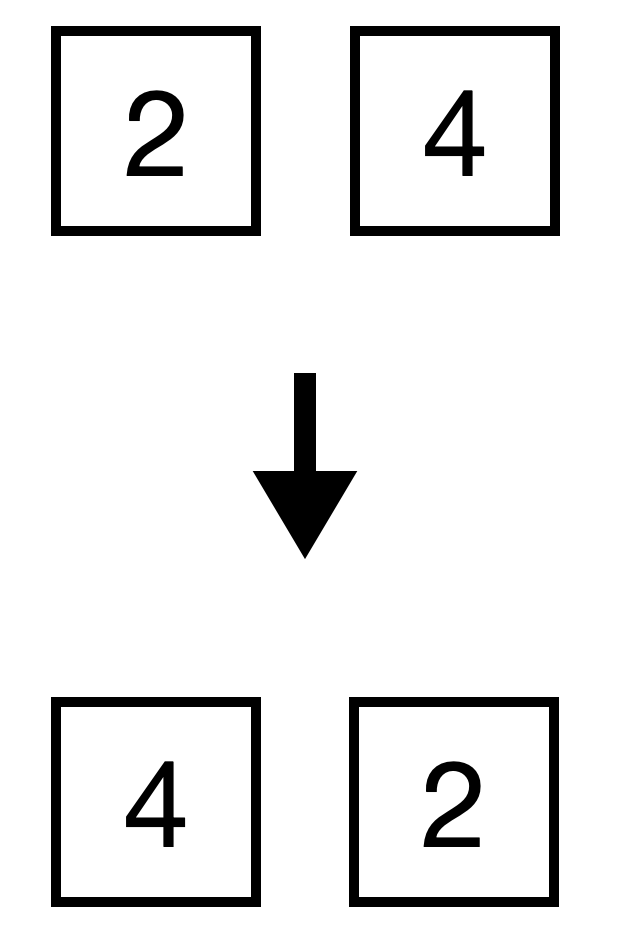
Étape 4 : Créer une fonction permettant de regrouper par paquets les doublons contenus dans plusieurs liste et de n’en garder qu’un :



Étape 5 : Prendre la taille de chacune des listes et en créer une nouvelle avec :

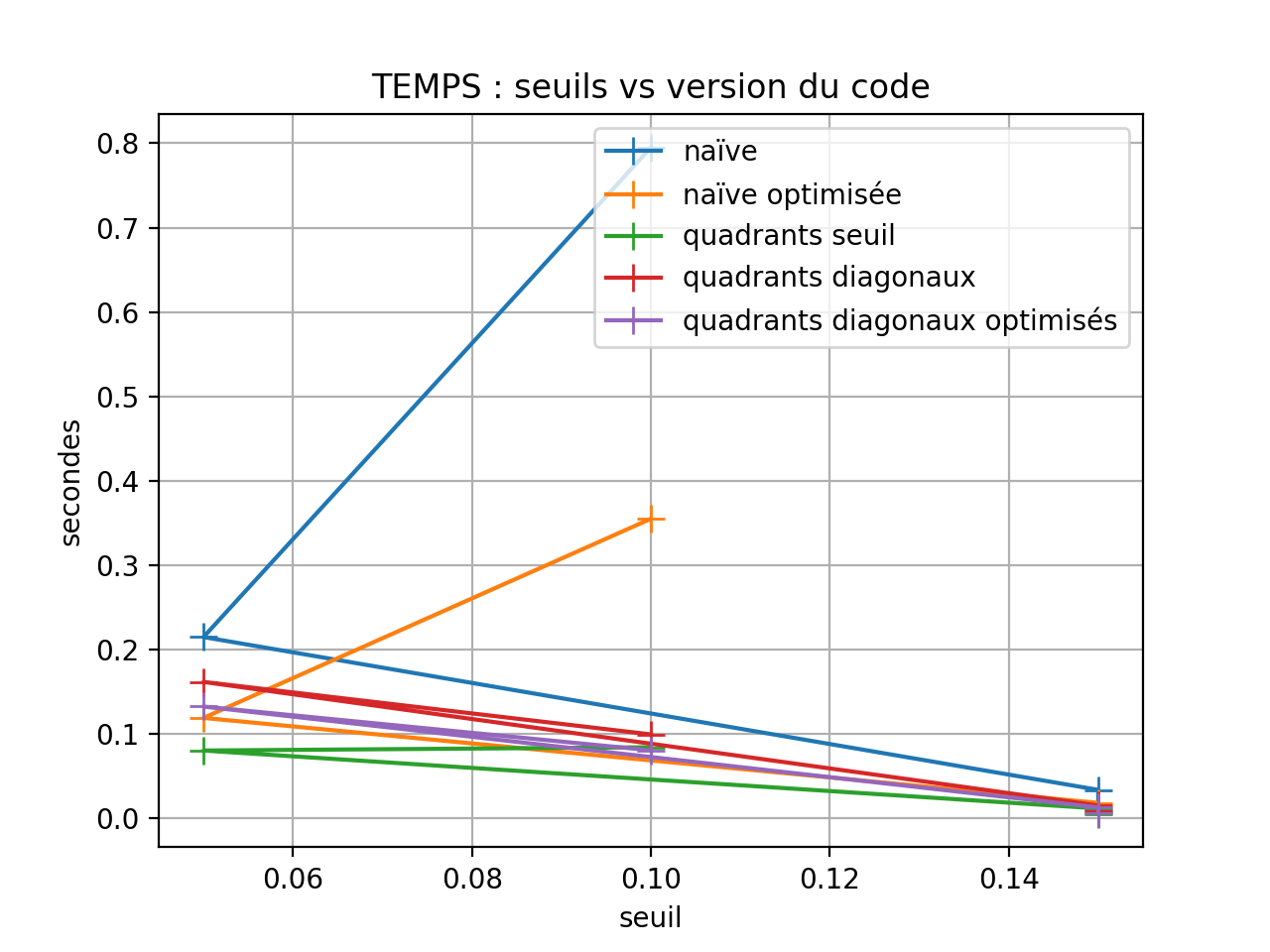
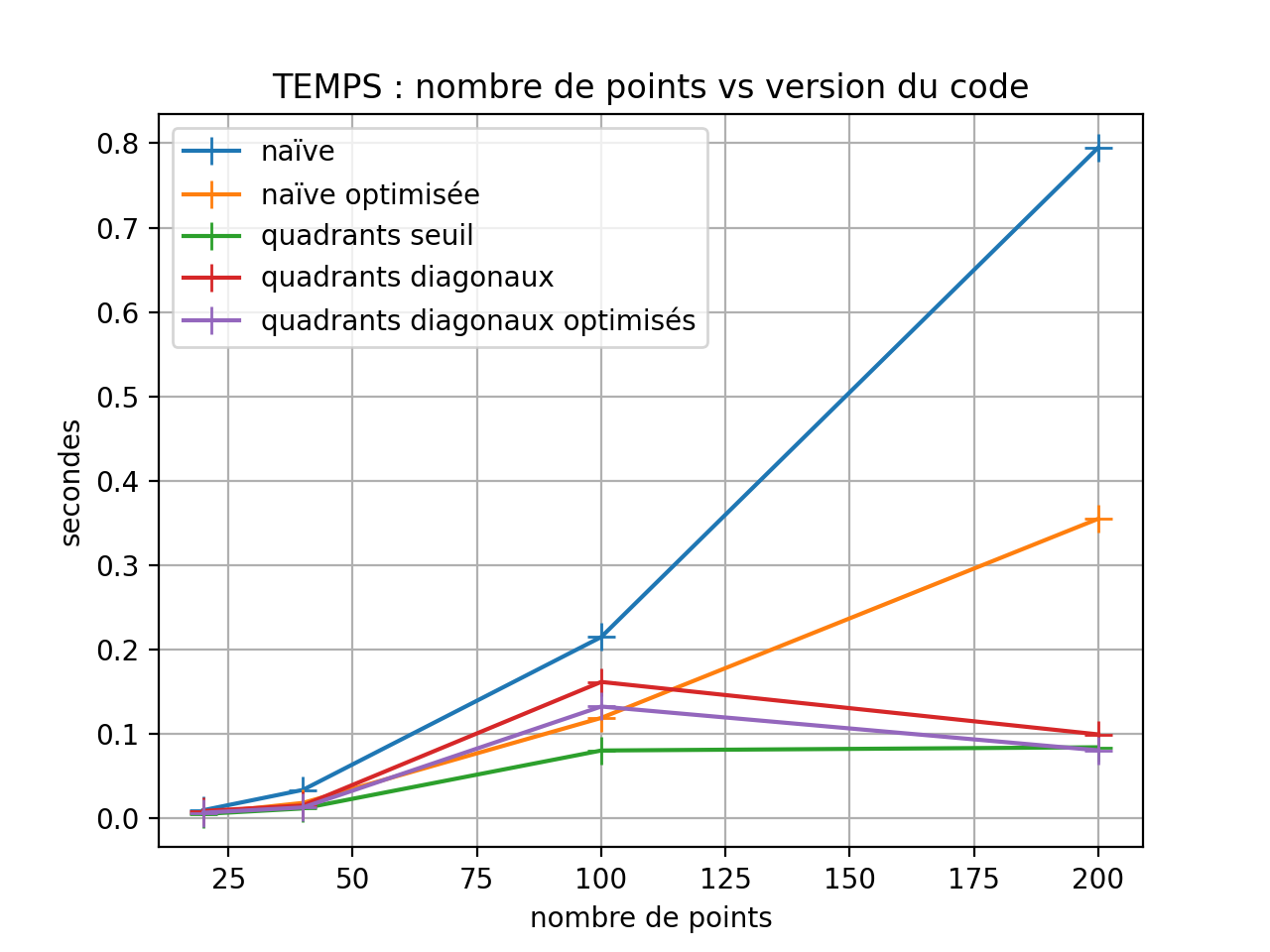


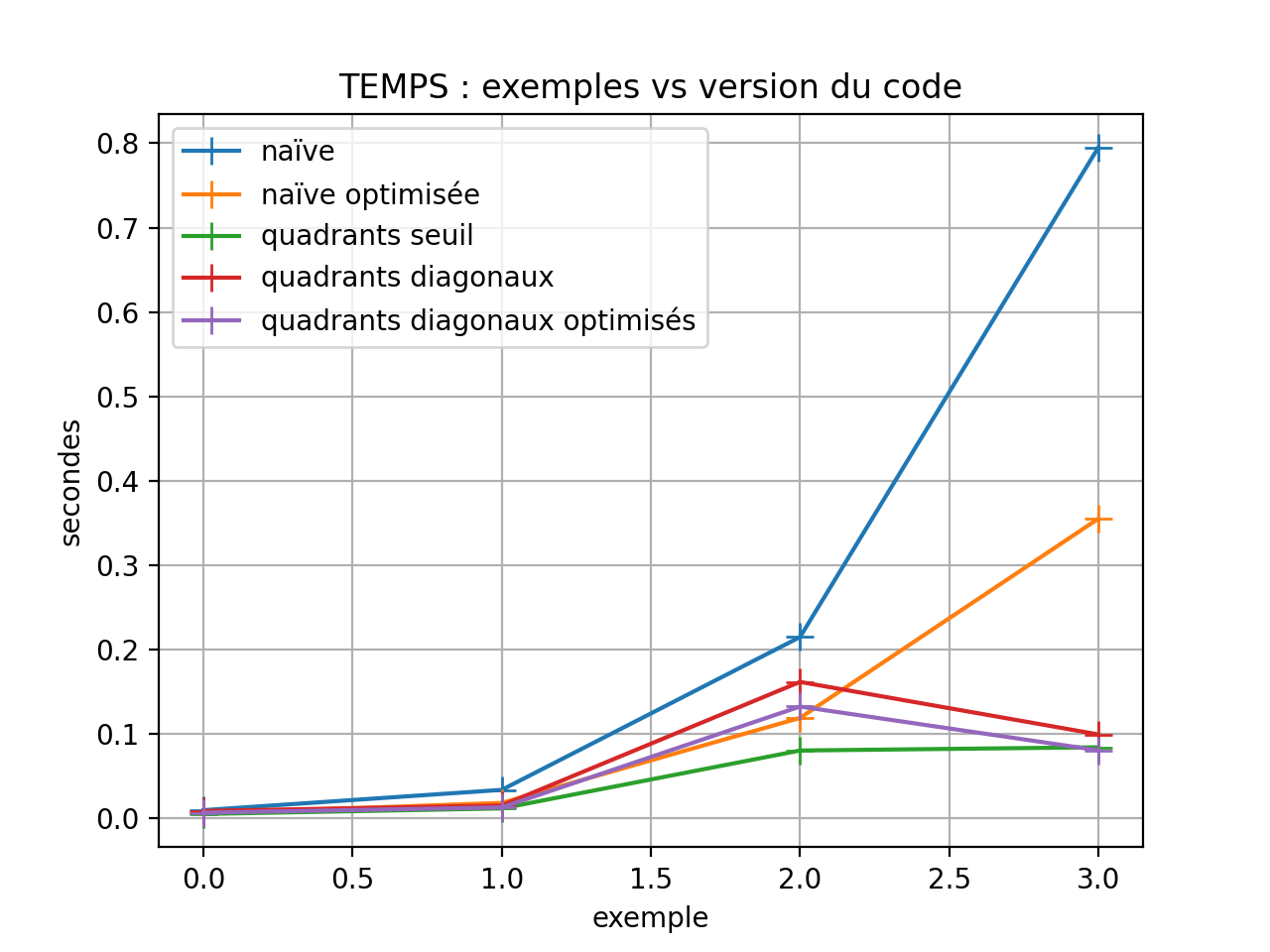
Étape 6 : Trier la liste dans l’ordre décroissant :



—> Cette méthode utilise des quadrants de diagonale de la taille du seuil, ainsi tous les points dans un même quadrant est une structure connexe, en construisant les quadrants on peut facilement reconnaitre des structures connexes et limiter les comparaisons avec d’autres points. En revanche, nous avons remarqué par notre expérience que cela rajoute des quadrants et donc plus de voisins et de comparaisons, cet algorithme est donc légèrement moins rapide que le précédent.

————————————————————————————————————————

————————————PERFORMANCES EN TEMPS————————————

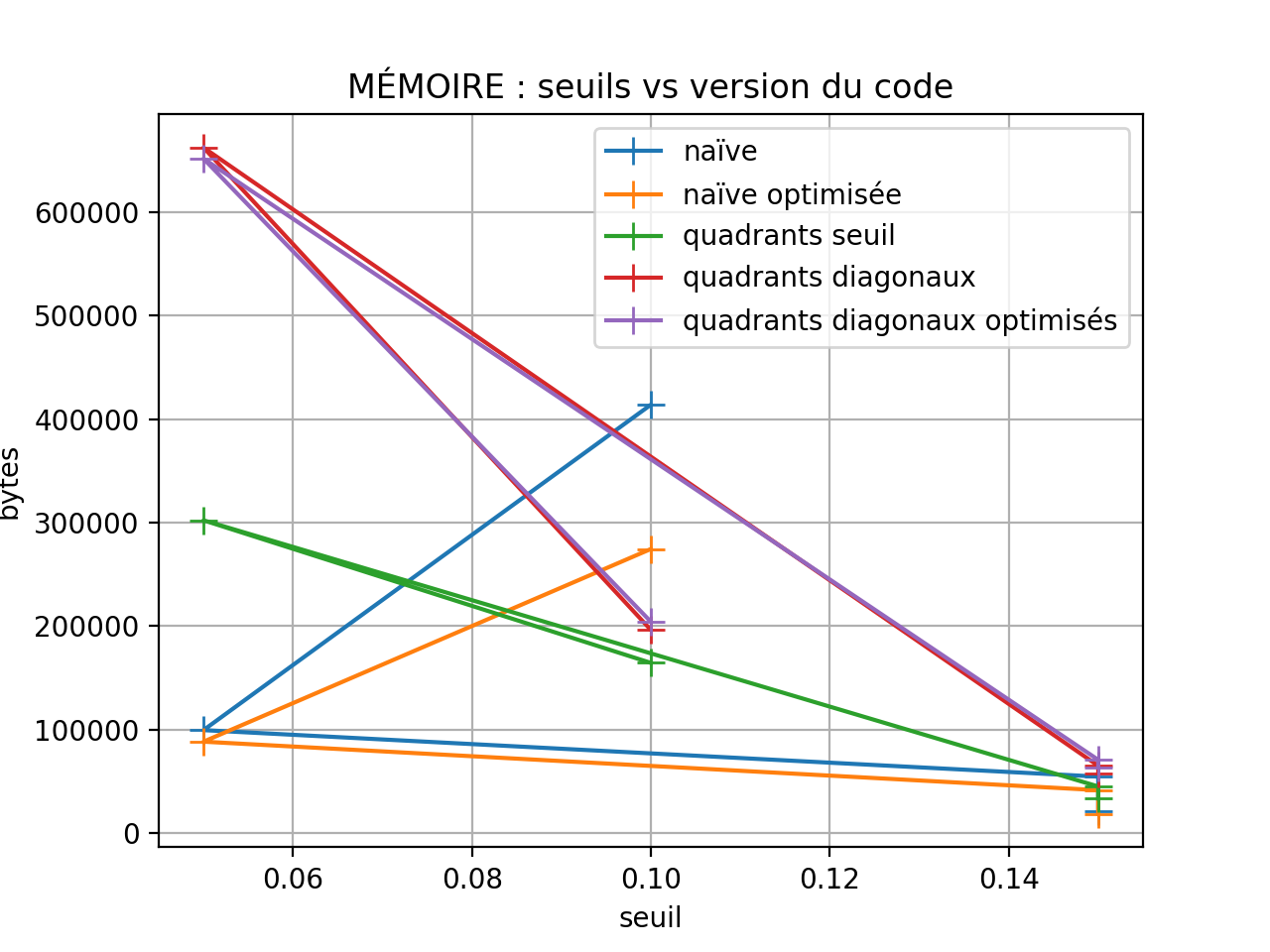
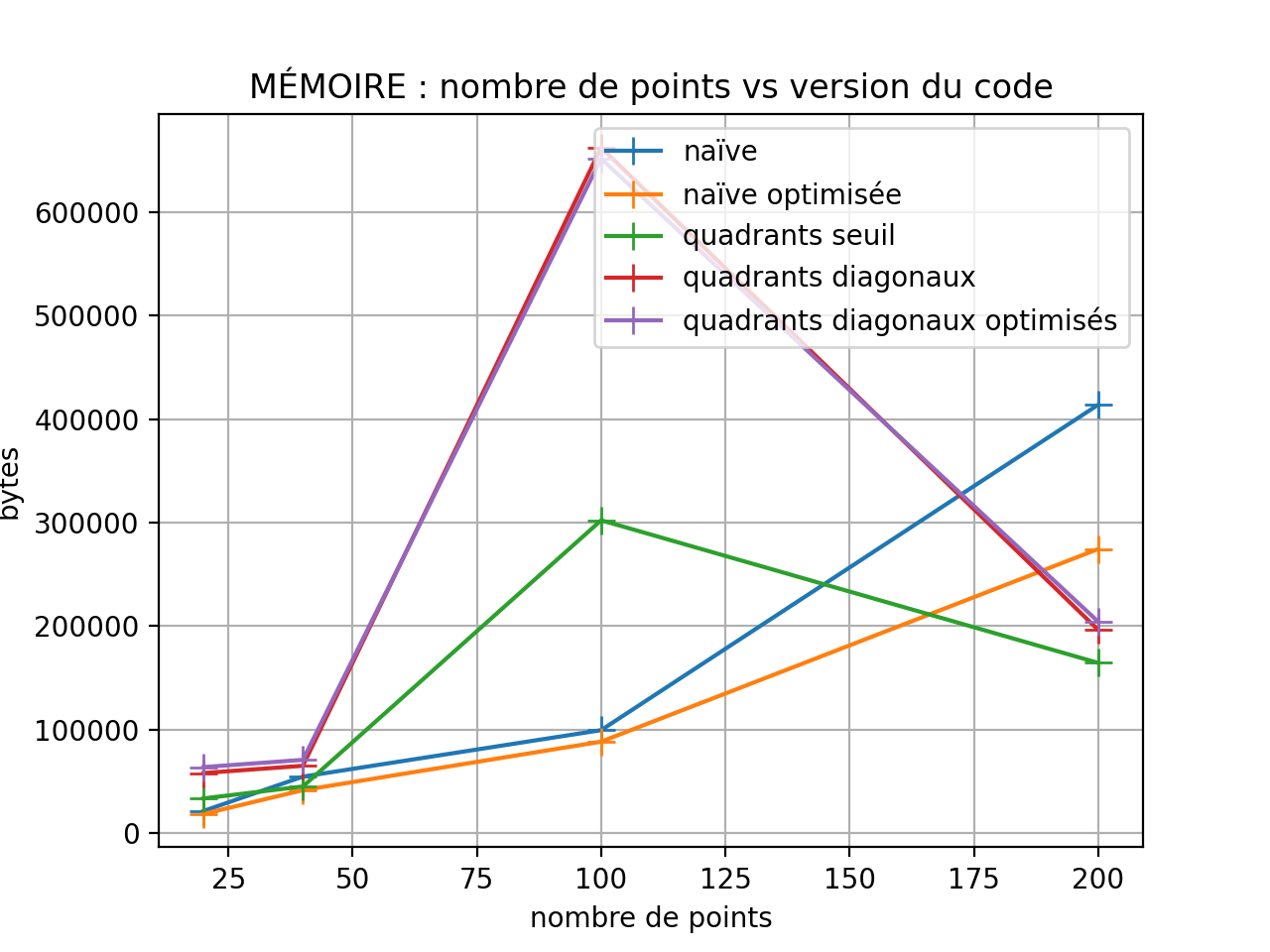


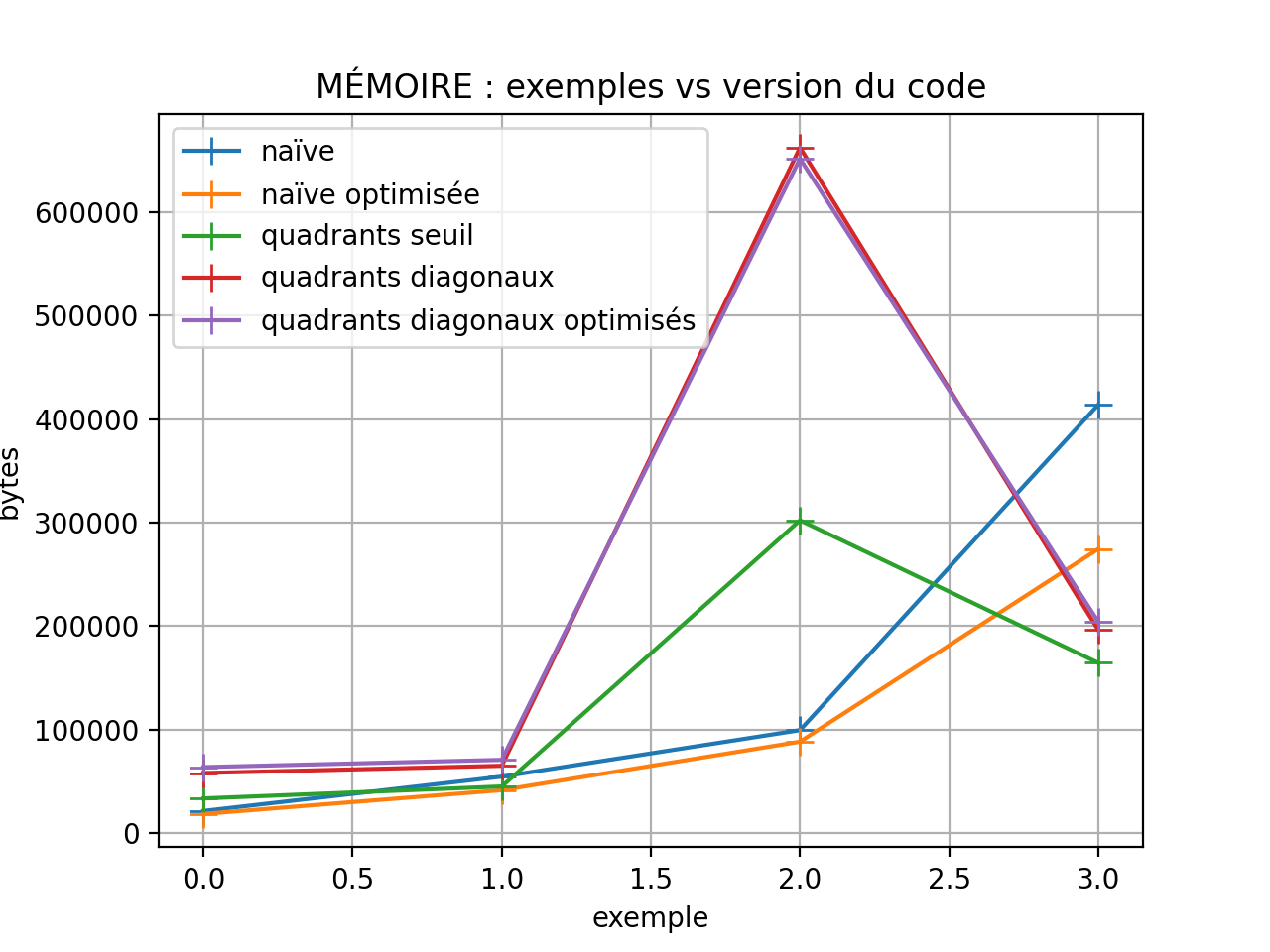
Nous avons tracé des courbes pour voir la performance en temps de nos algorithmes (méthode naïve, naïve optimisée, quadrants seuil, quadrants diagonaux, quadrants diagonaux optimisés) en fonction : du nombre de points en entrée et de la taille du seuil.

Nous avons remarqué que plus il y a de points, plus nos algorithmes sont lents, en revanche l’algorithme le plus rapide pour n’importe quelle quantité de points en entrée est *quadrants seuil* (idée 3). 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 sont lents, en effet, cela implique qu’il y a plus de quadrants a former et donc plus de voisins à comparer. Ainsi, l’algorithme le plus rapide pour n’importe quel seuil entrée est *quadrants seuil* (idée 3).

L’algorithme le plus performant en temps est donc l’algorithme de *quadrants seuil.*

————————————PERFORMANCES EN MÉMOIRE————————————



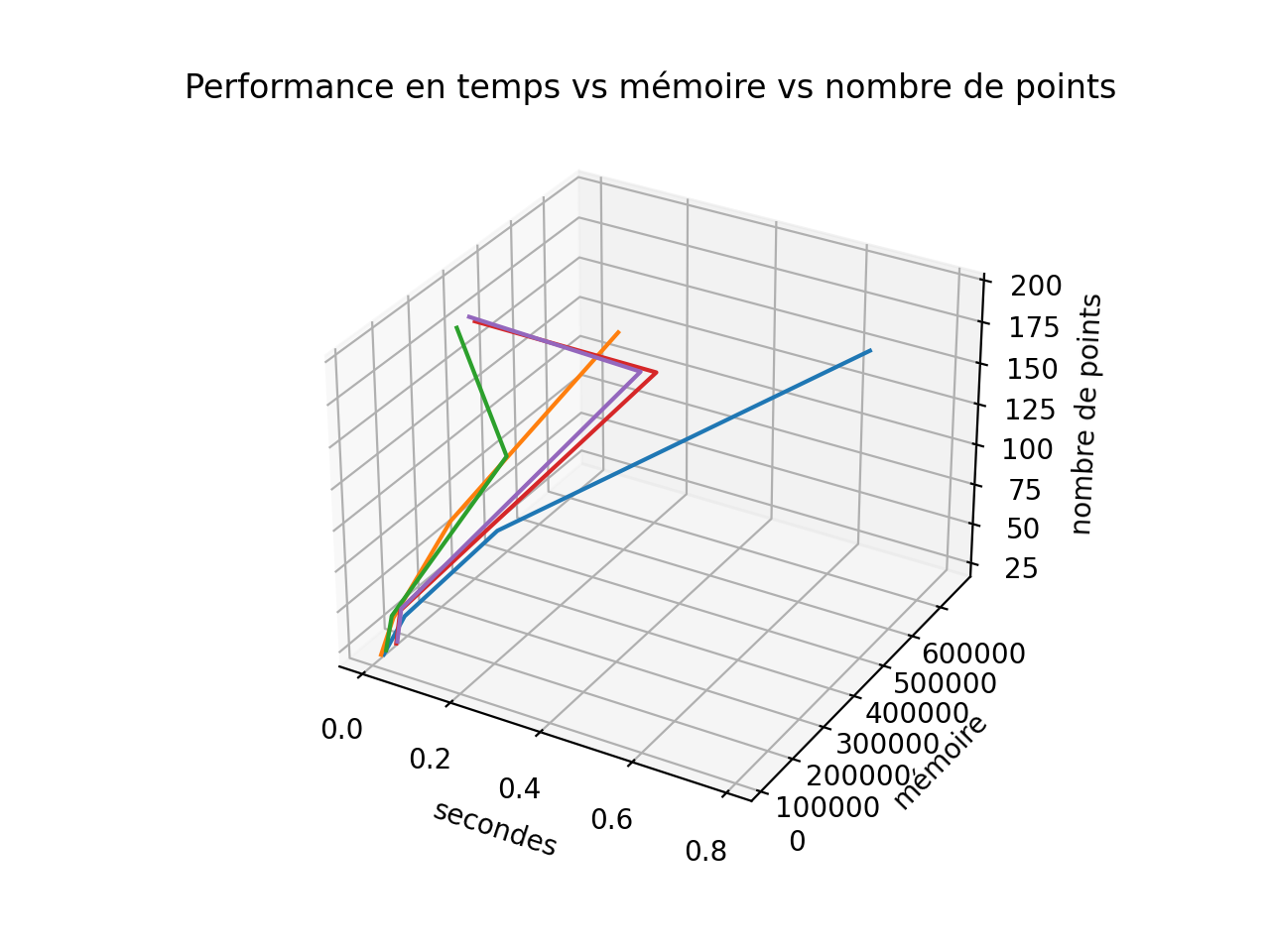
Nous avons également tracé des courbes pour voir la performance en mémoire de nos programmes (méthode naïve, naïve optimisée, quadrants seuil, quadrants diagonaux, quadrants diagonaux optimisés), toujours en fonction : du nombre de points en entrée et de la taille du seuil.

Nous avons remarqué que plus il y a de points, plus nos algorithmes sont couteux 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 couteux 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 couteux 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.

————————————————PERFORMANCES———————————————-



En conclusion, afin de pouvoir déterminer quel algorithme choisir, il faut établir un cahier des charges et savoir pourquoi on a en besoin. Si on a besoin qu’il soit rapide (ex : d’autres algorithmes dépendent du résultat du notre ou si l’algorithme doit être utiliser beaucoup de fois d’affilée), on privilégiera une méthode par *quadrants seuil* (pour ne pas accumuler les pertes en temps)*,* alors que si on a besoin qu’il soit performant en mémoire, c’est a dire qu’il ne prenne pas trop de place en mémoire (ex : si on a de nombreux programmes qui tournent en même temps et pas suffisamment d‘espace pour faire tourner une méthode par quadrant), on privilégiera un méthode naïve.