Séq. 10 – Méthode « Diviser pour régner »

Objectifs

1. Comprendre le principe de la méthode « diviser pour régner »
2. Écrire un algorithme utilisant la méthode « diviser pour régner » appliqué au tri fusion
3. Aborder les notions de coût

Cette séquence s'appuie sur :

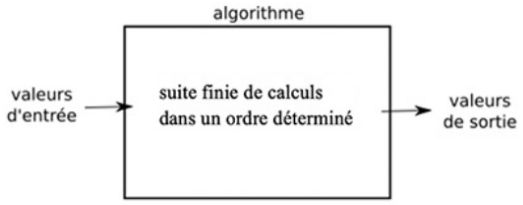
- https://pixees.fr/informatiquelycee/n_site/nsi_term_algo_diviser_pour_regner.html
- https://www.nsirenoir.fr/premiere/nsi_complexite.html

1 Avant-propos : Notion de complexité

1.1 Introduction

Qu'est-ce qu'un algorithme ? Il existe de nombreuses définitions d'un algorithme, pour ce cours, nous dirons qu'un algorithme est une suite finie de calculs (ou d'instructions élémentaires), exécutés dans un ordre déterminé, qui permettent de résoudre un problème.

L'ordre dans lequel nous allons définir ces "opérations élémentaires" va donc avoir son importance. On considérera qu'une opération élémentaire est une action simple, qui doit être facilement compréhensible pour la personne chargée d'effectuer cette action.



Un algorithme prend en entrée des données et fournit un résultat en sortie (qui sont aussi des valeurs), la "valeur de sortie" n'étant pas obligatoirement du même type que la "valeur d'entrée".

1.2 Notion de complexité d'un algorithme.

Dans cette partie, nous allons aborder la notion de complexité. C'est une notion "très mathématique" de l'informatique qui n'est pas simple à comprendre et qui utilise des outils mathématiques (on parle d'informatique théorique) d'assez bon niveau qui ne sont pas tous au programme du lycée. Ainsi ce que nous allons voir n'est qu'une approche de ces notions.

Les calculs de complexité d'un algorithme permettent d'évaluer la quantité de ressources (en temps et en mémoire) dont a besoin un algorithme pour résoudre un problème, c'est une sorte de quantification de la performance d'un algorithme.

L'objectif premier d'un calcul de complexité algorithmique est de pouvoir comparer l'efficacité d'algorithmes résolvant le même problème. Pour un même problème donné, il existe souvent plusieurs algorithmes et certains algorithmes sont plus efficaces, donc meilleurs, que d'autres.

Ce type de question est très important, car pour des données volumineuses la différence entre les durées d'exécution de deux algorithmes qui résolvent un même problème peut être de l'ordre de plusieurs jours (voire de plusieurs mois ou années voire impossible à résoudre sur des ordinateurs modernes).

Il existe 2 types de calculs de complexité : la complexité en temps et la complexité en taille (ou en mémoire). Nous nous intéresserons dans le cours uniquement à la complexité en temps.

La complexité en temps est directement liée au nombre d'opérations élémentaires qui doivent être exécutées afin de résoudre un problème donné.

Les règles que nous utilisons pour comparer et évaluer les algorithmes ne doivent pas dépendre des qualités d'une machine ou d'un choix de technologie.

On compare la "qualité" des algorithmes pour des données de grande taille ("comportement asymptotique"), ça n'a pas d' intérêt de connaître la qualité d'un algorithme pour de toutes petites quantités de données puisque dans ce cas, quelque soit l'algorithme choisi les temps d'exécutions seront rapides.

Pour calculer la complexité (en temps), on examine chaque ligne de code et on leur attribue un "coût". Le coût ainsi obtenu n'a pas d'unité, il s'agit d'un nombre d'opérations dont chacune aurait le même temps d'exécution : 1 unité.

1.2.1 Opérations élémentaires

On appelle opérations élémentaires :

- Une opération arithmétique entre deux nombres (+ - / * % //), opération "de base" sur les tableaux (listes)...
- un accès mémoire : affectation, lecture et écriture de variables,
- une comparaison de valeurs (<, >, ==, ...),
- un appel ou un retour d'une fonction.

1.2.2 Complexité en temps

Pour déterminer la complexité en temps (on dit aussi complexité temporelle) d'un algorithme, on doit compter le nombre d'opérations élémentaires, et exprimer ces valeurs en fonction des paramètres d'entrée de l'algorithme, souvent un entier n. La complexité en temps d'un algorithme sera donc une fonction qu'on note généralement T (pour time) dépendant de la taille n des données. Elle est obtenue en sommant tous les temps d' exécution des différentes opérations effectuées lors de l'exécution d'un algorithme.

En fait, on pourrait distinguer trois formes de complexité en temps :

1. la complexité dans le meilleur des cas : c'est la situation la plus favorable,
2. la complexité dans le pire des cas : c'est la situation la plus défavorable,
3. et la complexité en moyenne, mais la difficulté pour ce cas est de définir une entrée "moyenne" pour un problème particulier.

Dans la pratique on calculera le plus souvent la complexité dans le pire des cas, car elle est la plus pertinente. Il vaut mieux en effet toujours envisager le pire.

1.2.3 Exemple 1 :

L'instruction `x = c * 3` nécessite 3 opérations élémentaires : lecture de c, calcul du produit et affectation du résultat à x. On peut noter T(n)=3

1.2.4 Exemple 2 : Recherche du maximum dans une liste de n nombres.

```
max = tab[0]  # tab = liste de n nombres
for i in range(1, n):
    if tab[i] > max:
        max = tab[i]
```

On peut commencer par lister les différentes opérations élémentaires utilisées dans ce programme :

- accès à un élément d'un tableau
- lecture de la valeur d'une variable
- affectation de flottants
- affectation d'entiers
- comparaison de 2 entiers
- comparaison de 2 flottants

On peut remarquer que pour n test de la boucle tant que effectués, on exécute $n-1$ fois l'expression à l'intérieur de cette boucle.

On va maintenant compter le nombre de fois où chaque opération élémentaire est effectuée pour chaque ligne, dans le pire des cas (tableau trié par ordre croissant). On place ces calculs dans le tableau suivant :

Ligne	Accès tableau	Accès variable	Affect. entier	Affect. flottant	Comp. flottants
1	1			1	
2	n - 1		n - 1		
3	n - 1	n - 1			n - 1
4	n - 1		n - 1		

Au total on compte $7n-5$ opérations élémentaires dans le pire des cas, $T(n)=7n-5$

1.3 Notation de Landau

Pour effectuer des comparaisons entre des algorithmes, on raisonne sur de très grands nombres de données car plus il y a de données et plus les différences entre les algorithmes sont flagrantes.

Pour comparer des algorithmes, il n'est pas nécessaire de connaître précisément la fonction T. Nous allons nous intéresser uniquement à ce que l'on appelle "l'ordre de grandeur asymptotique" à l'aide de la "notation de Landau" O ("grand O") dans le pire des cas.

Remarque : La définition précise de "l'ordre de grandeur asymptotique" n'est pas au programme, il faut juste savoir que cet "ordre de grandeur asymptotique" concerne les cas où l'on prend n très très grand.

La définition mathématique de cette notation est :

On dit que $f(n)$ est $O(g(n))$ ou que $f(n) \in O(g(n))$ si il existe un entier $n_0 > 1$ et une constante positive c tels que quel que soit $n \geq n_0$, $f(n) \leq c \times g(n)$.

On peut aussi dire que " $f(n)$ est $O(g(n))$ " signifie qu'il existe un seuil n_0 à partir duquel la fonction f est toujours inférieure à la fonction g , à une constante multiplicative fixée c près. Cette notation indique que dans le pire des cas, à partir d'un certain seuil, la croissance de f ne dépassera pas celle de g .

1.3.1 Exemple :

$T(n)=2n+10$ est $O(n)$

En résolvant une inéquation très simple, on peut facilement prouver que pour tout $n \geq 10$, $2n+10 \leq 3n$ donc en choisissant $n_0=10$ et $c=3$, on a bien : pour tout entier $n \geq n_0$, $f(n) \leq c \times n$.

Par contre $k(n)=n^2$ n'est pas $O(n)$ car $n^2 \geq cn \Leftrightarrow n \geq c$ qui ne peut être satisfaite car c est constante (et l'inégalité doit être vraie quelque soit n).

1.3.2 Règles de la notation O :

- Si $T(n)$ est un polynôme de degré d , alors $T(n)$ est $O(n^d)$
  - On peut donc supprimer les termes de degré plus bas que d
  - On peut donc supprimer les facteurs constants.
- En pratique, on utilise la classe la plus basse possible : On dit que " $2n$ est $O(n)$ " même si on a aussi " $2n$ est $O(n^2)$ "
- et on utilise l'expression la plus simple : On dit que " $3n+5$ est $O(n)$ " même si on a aussi " $3n+5$ est $O(3n)$ ".

L'analyse asymptotique d'un algorithme détermine le temps d'exécution en notation Landau : on trouve le cas produisant le plus d'opérations élémentaires exprimé en fonction de la taille des données (en général n) et on l'exprime avec la notation de Landau.

1.3.3 Exemple :

On a vu que le programme de recherche de maximum exécute au plus $7n-3$ opérations élémentaires. On peut alors dire que l'algorithme a une complexité en $O(n)$ (on parle de complexité linéaire).

Remarque : Comme les facteurs constants et les termes de degré inférieur sont supprimés, on peut les "oublier" lors du décompte des opérations élémentaires.

1.3.4 fonctions de référence

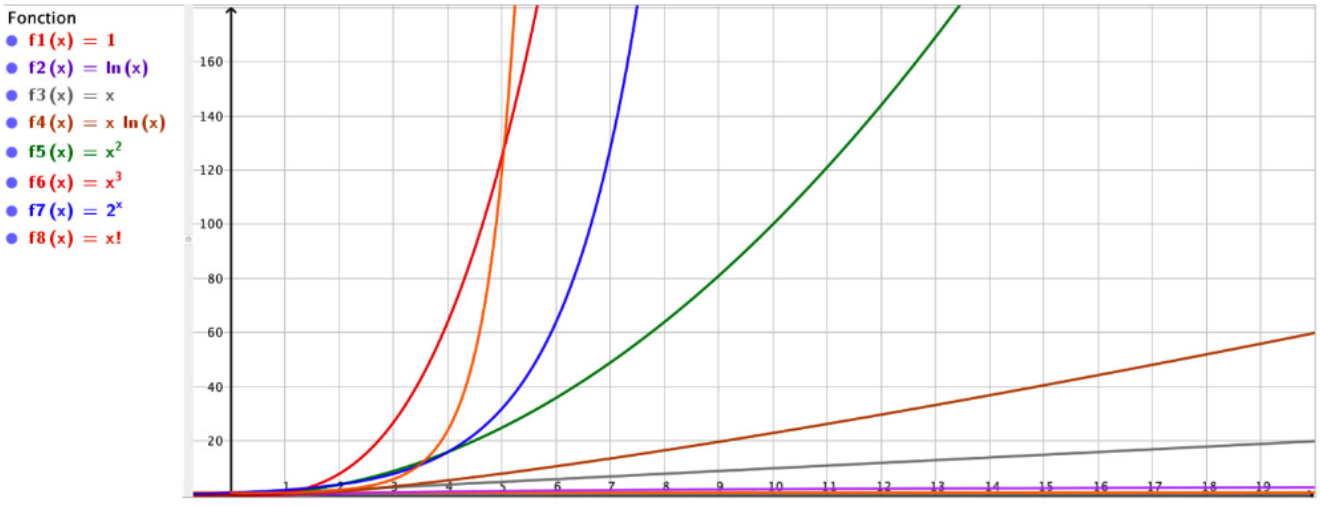
Les complexités algorithmiques que nous allons calculer vont dorénavant être exprimées comme des grand O de fonctions mathématiques considérées comme des "fonctions de référence". Cela va nous permettre de les classer en comparant la croissance de ces fonctions. Les études et les comparaisons de ces fonctions sont faites en cours de mathématiques, nous allons réduire cette partie à son minimum.

Des algorithmes appartenant à une même classe seront alors considérés comme de complexité équivalente et on considérera alors qu'ils ont la même efficacité.

Les fonctions de référence qu'on utilise le plus souvent sont les fonctions définies par :

- $f_1(n)=1$,
- $f_2(n)=\log(n)$ (fonction logarithme),
- $f_3(n)=n$,
- $f_4(n)=n \times \log(n)$,
- $f_5(n)=n^2$,
- $f_6(n)=n^3$,
- $f_7(n)=2^n$
- et $f_8(n)=n!$ (fonction factorielle).

Pour "visualiser" le comportement de ces fonctions, elles sont tracées ci-dessous :



Le tableau suivant récapitule les complexités de "référence" :

O	Type de complexité
$O(1)$	constant
$O(\log(n))$	logarithmique
$O(n)$	linéaire
$O(n \times \log(n))$	quasi-linéaire
$O(n^2)$	quadratique
$O(n^3)$	cubique
$O(2^n)$	exponentiel
$O(n!)$	factoriel

Pour donner un ordre d'idée sur les différentes complexités, le tableau ci-dessous présente les différentes classes de complexité, leur nom, des temps d'exécution de référence et un problème de la-dite complexité.

Temps	Type de complexité	Temps pour n = 5	Temps pour n = 10	Temps pour n = 20	Temps pour n = 50	Temps pour n = 250	Temps pour n = 1 000	Temps pour n = 10 000	Temps pour n = 1 000 000	Problème exemple
$O(1)$	complexité constante	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	accès à une cellule de tableau
$O(\log(n))$	complexité logarithmique	10 ns	10 ns	10 ns	20 ns	30 ns	30 ns	40 ns	60 ns	recherche dichotomique
$O(n)$	complexité linéaire	50 ns	100 ns	200 ns	500 ns	2.5 µs	10 µs	100 µs	10 ms	parcours de liste
$O(n \log(n))$	complexité linéarithmique	40 ns	100 ns	260 ns	850 ns	6 µs	30 µs	400 µs	60 ms	tris par comparaisons optimaux (comme le tri fusion ou le tri par tas)
$O(n^2)$	complexité quadratique (polynomiale)	250 ns	1 µs	4 µs	25 µs	625 µs	10 ms	1 s	2.8 heures	parcours de tableaux 2D
$O(n^3)$	complexité cubique (polynomiale)	1.25 µs	10 µs	80 µs	1.25 ms	156 ms	10 s	2.7 heures	316 ans	multiplication matricielle naïve
$2^{\text{poly}(n)}$	complexité exponentielle	320 ns	10 µs	10 ms	130 jours	10^{59} ans	...	...	...	problème du sac à dos par force brute
$O(n!)$	complexité factorielle	1.2 µs	36 ms	770 ans	10^{48} ans	...	...	...	...	problème du voyageur de commerce avec une approche naïve

2 Un programme est une donnée comme une autre et notion de décorateur python

2.1 Principe

Nous l' avons déjà vu, une fonction peut prendre en paramètre des types simples (entiers, flottants, chaînes de caractères) et des types plus complexes (listes, tuples, dictionnaires).

Un programme (ou une fonction) est une donnée comme une autre. Il est donc possible qu' une fonction prenne en paramètre une autre fonction.

On peut ainsi transformer n' importe qu' elle fonction en pré-traitant des données en amont et en post-traitant les données retournées en aval. C' est le principe de décorateurs python.

P. 305 ex 8

3 Principe de la méthode « diviser pour régner »

La méthode « diviser pour régner » est une méthode algorithmique basée sur le principe suivant : On prend un problème (généralement complexe à résoudre), on divise ce problème en une multitude de petits problèmes plus simples à résoudre. Une fois les petits problèmes résolus, on recombine les "petits problèmes résolus" afin d'obtenir la solution du problème de départ.

Un problème de taille n passe par la résolution de problème de taille $\frac{n}{2}$ ou une fraction de n .

Le paradigme "diviser pour régner" repose donc sur 3 étapes :

1. DIVISER : le problème d'origine est divisé en un certain nombre de sous-problèmes
2. RÉGNER : on résout les sous-problèmes (les sous-problèmes sont plus faciles à résoudre que le problème d'origine)
3. COMBINER : les solutions des sous-problèmes sont combinées afin d'obtenir la solution du problème d'origine.

Les algorithmes basés sur le paradigme "diviser pour régner" sont très souvent des algorithmes récursifs.

P. 48 ex 6

P. 48 ex 7

4 Complexité typique de méthode « diviser pour régner »

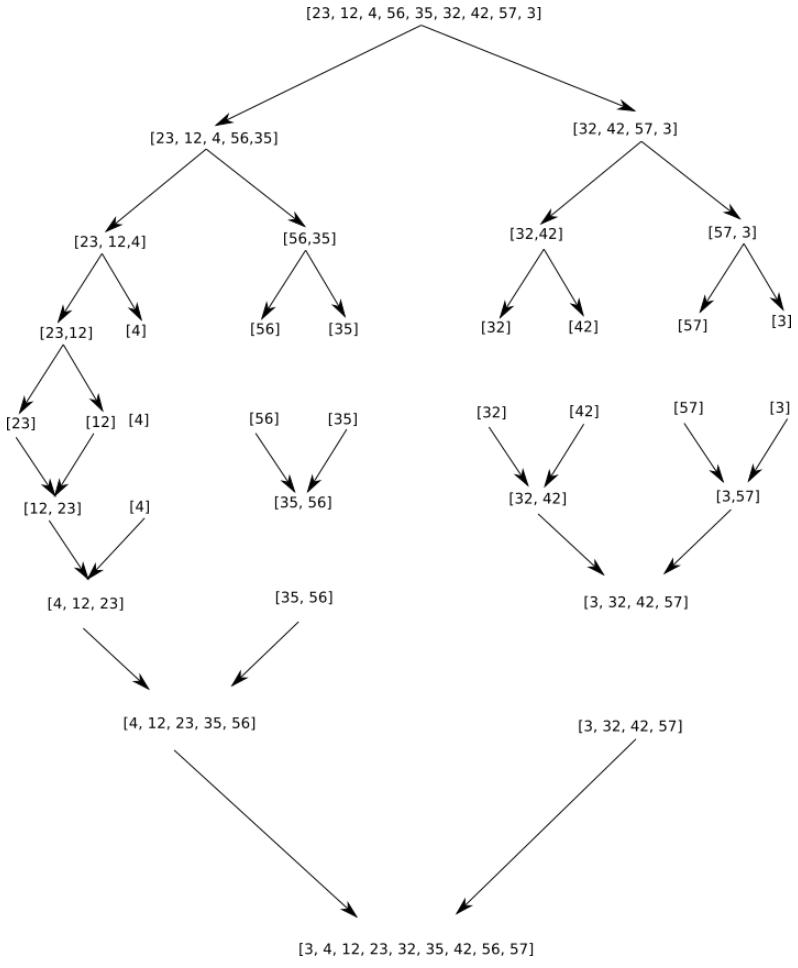
Relation de récurrence :	Complexité :
$T(n)=T(n/2)+O(1)$	$O(\log n)$
$T(n)=2T(n/2)+O(1)$	$O(n)$
$T(n)=2T(n/2)+O(n)$	$O(n\log n)$
$T(n)=aT(n/b)+O(n)(a\geq 2, b\geq 2 et a>b)$	$O(n^{\log_b(a)})$

5 L' exemple du tri-fusion

Lire cours P. 40-41

Cet algorithme est composé de deux fonctions FUSION et TRI-FUSION (fonction récursive). La fonction TRI-FUSION assure la phase "DIVISER" et la fonction FUSION assure les phases "RÉGNER" et "COMBINER".

Voici un exemple d'application de cet algorithme sur le tableau `A = [23, 12, 4, 56, 35, 32, 42, 57, 3]` :



6 L' exemple de la Paire des points les plus proches

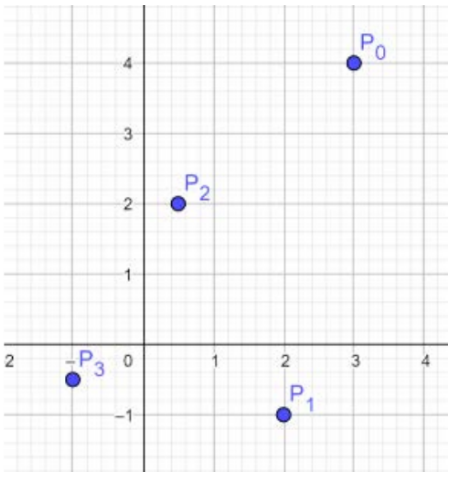
6.1 Description du problème

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 P2 et P3 sur le dessin.



6.2 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.

6.3 Algorithme « diviser pour régner »

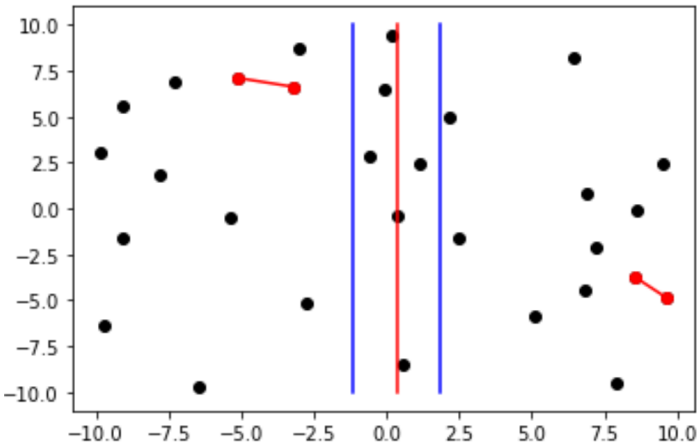
Soit P un ensemble de n points. On note d_P la distance minimale entre deux points de P.

6.3.1 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.

6.3.2 Description de l'algorithme

1ère étape (préliminaire)

On crée un tableau P_x contenant les points de P triés dans l'ordre croissant des abscisses et P_y 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 ≤ 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 A_x et B_x en partageant P_x en deux, puis A_y et B_y à partir de la liste P_y .

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 $]x-\delta, x+\delta[$ (bande délimitée par les lignes bleues).

On crée le tableau Q_y 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 Q_y, 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 Q_y, on ne considère que les points situés au-dessus afin de ne pas considérer chaque paire deux fois : (p, q) et (q, p).

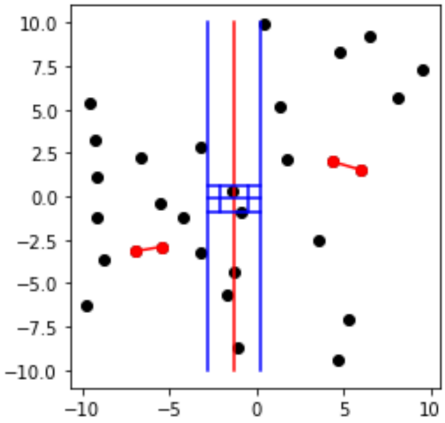
Pourquoi se contente-t-on de 7 points ?

Pour chaque point p de Q_y, 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 Q_y 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 Q_y car les points suivants sont forcément en dehors de R.



6.3.3 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.

6.3.4 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.

1. Ouvrir le script distance_points.py : http://ninoo.fr/LC/Term_NSI/seq10_methode_diviser_pour_regner/distance_points.py.
2. Écrite la fonction distance renvoyant la distance entre deux points.
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 `[(x1, y1), …, (xn, yn)]` et renvoyant la distance minimale et les coordonnées des deux points les plus proches en utilisant l'algorithme naïf.
4. Exécuter le script. Si le doctest ne renvoie pas de message d'erreur, alors vos fonctions passent les tests.
5. Compléter la fonction `plus_proches_dpr`. Montrer votre travail au professeur après chacune des trois étapes.
6. Exécuter le programme principal avec `doctest` et la fonction `graphique_ppdpr` pour tester votre code.