Optimisation des Algorithmes

Objectif de la séance : Éviter les algorithmes naïfs ou directs, au profit d'algorithmes plus élaborés pour lesquels l'analyse de complexité fournit de meilleurs résultats.

😎	Application directe du cours
😀	Facile
😖	Moyen
💀	Difficile

1 Retour sur les tris

Exercice 1 : Tri rapide (quicksort) 😀

Dans le TD précédent, nous avons vu plusieurs algorithmes de tri sur les listes. On remarque que les tris intuitifs ont une complexité de l'ordre de $O(n^2)$, où n est la taille de la liste d'entrée. D'autre part, les tris plus élaborés à base de "diviser pour régner", comme le tri fusion, sont plus intéressants car ils ont une complexité $O(n \log n)$. Nous commençons ce TD par un autre exemple de tri s'inspirant de la méthode "diviser pour régner" : le tri rapide, ou quicksort.

Le principe du tri rapide est de diviser la liste à l'aide d'un élement pivot.

1. Partager la liste selon la valeur v d'un élément arbitraire (le pivot), en pratique le premier élément de la liste, en deux sous-listes respectivement des éléments inférieurs ou égaux à v et des éléments de valeur supérieure à v.
2. Trier séparément les deux sous-listes extraites en suivant la même méthode.
3. Réarranger les sous-listes triées avec l'élément de référence (le pivot).

Question 1. Écrire une fonction `split` qui prend en argument une liste `l` et un entier `v`, et qui renvoie un couple `l1, l2` où `l1` contient les éléments de `l` inférieurs (ou égaux) à `v` et `l2` contient les éléments de `l` supérieurs (strictement) à `v`.

Par exemple, `split([2,3,8,1,5,4],4)` devra renvoyer `[2,3,1,4], [8,5]`.

On propose une version itérative et une version récursive.

```
1  def split(l, pivot):
2      l1, l2 = [], []
3      for k in range(0, len(l)):
4          if l[k] <= pivot:
5              l1 = l1 + [l[k]]
6          else:
7              l2 = l2 + [l[k]]
8      return l1, l2
9
10 def split_rec(l, pivot):
11     if l == []:
12         return [], []
13     else:
14         l1, l2 = split_rec(l[1:], pivot)
15         if l[0] <= pivot:
16             return [l[0]] + l1, l2
```

```
        else:
            return l1, [l[0]] + l2
```

Question 2. Écrire une fonction `quicksort` qui prend en argument une liste `l` et renvoie une copie triée de `l` suivant l'algorithme du tri rapide.

```
def quicksort(l):
    if l == []:
        return []
    else:
        l1, l2 = split(l[1:],  l[0])
        return quicksort(l1) + [l[0]] + quicksort(l2)
```

Question 3. On définit une opération élémentaire comme étant l'ajout d'un élément à une liste. Combien d'opérations élémentaires faut-il à la fonction `quicksort` pour trier une liste de taille n, en supposant qu'à chaque fois que l'on la divise en deux avec `split`, les deux listes obtenues `l1, l2` ont chacune pour taille $\approx n/2$?

Indication. On pourra utiliser le résultat suivant, qui est un cas particulier d'un énoncé plus général appelé "master theorem" :

Si $T : \mathbb{N} \to \mathbb{N}$ *est une fonction telle que* $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$, *alors* $T(n) = O(n \log n)$.

Si on note $T(n)$ cette complexité, l'analyse de l'algorithme montre que $T(n) = 2T\left(\frac{n}{2}\right) + n$. En effet, `quicksort` demande d'utiliser une fois la fonction `split`, qui nécessite exactement n opérations élémentaires, puis de calculer deux fois le `quicksort` sur une liste de taille $n/2$. On en déduit d'après l'indication que $T(n) = O(n \log n)$.

Question 4. Que se passe-t-il si on utilise `quicksort` sur une liste déjà triée ?

Dans ce cas, la fonction `split` renvoie `[], l[1:]`. On a donc la relation de récurrence

$$T(n) = T(n-1) + T(0) + n$$

d'où $T(n) = nT(0) + \frac{n(n+1)}{2} = O(n^2)$. Cette fois, la complexité est quadratique. On remarque que le cas où la liste est déjà triée est le pire cas possible pour `quicksort`...

Exercice 2 : Recherche du k-ème élément

Question 1. En s'inspirant de l'algorithme du tri rapide, implémenter une fonction `kth` qui prend en argument une liste `l` et un entier `k` (supérieur ou égal à 1) et qui renvoie le k-ème plus petit élément de la liste `l`.

Par exemple, `kth([1,3,2,1,4,2],4)` devra renvoyer 2.

On pourra réutiliser la fonction `split` de l'exercice précédent, mais on s'interdira de commencer par trier la liste.

Pour gérer le cas où l'utilisateur demande de chercher le k-ème élément d'une liste qui en comporte strictement moins que k, on peut définir une erreur personnalisée, ici appelée `Tooshort`.

```
class TooShort(Exception):
    pass

def kth(l, k):
    n = len(l)
    if n < k:
        raise TooShort
    else:
        l1, l2 = split(l[1:], l[0])
        if len(l1) == k-1:
            return l[0]
        elif len(l1) >= k:
            return kth(l1, k)
        else:
            return kth(l2, k - len(l1) - 1)
```

Question 2. Analyser la complexité de la fonction `kth` dans les deux cas étudiés dans l'exercice précédent.

On note toujours $T(n)$ la complexité pour une entrée de taille n. Dans le cas où `split` divise toujours la liste en deux listes de taille égale, on a la relation

$$T(n) = T(n/2) + n$$

Donc en itérant on obtient (en supposant que n est une puissance de 2) :

$$T(n) = n + n/2 + n/4 + ... + 1 = 2n - 1$$

donc la complexité est linéaire.

Dans le cas où la liste est triée à l'avance, on sait déjà que `split` se comporte assez mal. On obtient la relation

$$T(n) = T(n-1) + n$$

ce qui donne $T(n) = O(n^2)$.

2 Plus petit et plus grand élément d'une liste

Exercice 3 : Méthode naïve 😎

Question 1. Écrire une fonction `minL` (respectivement `maxL`) qui prend en argument une liste d'entiers naturels non vide `L` et renvoie son élément minimal (respectivement maximal).

▷ `minL([0,3,1,7,0,8,12,4])` renvoie `0`.
▷ `maxL([0,3,1,7,0,8,12,4])` renvoie `12`.

On s'autorisera à utiliser les fonctions `min` et `max`, prédéfinies en Python, sur les entiers. Par exemple, `min(2,3)` renvoie 2.

Quelle est la complexité de ces fonctions en nombre de comparaisons ?

```
def minL(l):
    # Returns the minimal element of the list l
    # It is assumed that L is a non-empty list of natural numbers
    small = l[0]
    for i in range(1,len(l)):
        small = min(small,l[i])
    return small

def maxL(l):
    # Returns the maximal element of the list l
    # It is assumed that L is a non-empty list of natural numbers
    big = l[0]
    for i in range(1,len(l)):
        big = max(big,l[i])
    return big
```

Voici une seconde version où l'on n'utilise pas les fonctions `min` et `max` de Python.

```
def minL2(l):
    # Returns the minimal element of the list l
    # It is assumed that L is a non-empty list of natural numbers
    small = l[0]
    for i in range(1,len(l)):
        if small > l[i]:
            small = l[i]
    return small


def maxL2(l):
    # Returns the maximal element of the list l
    # It is assumed that L is a non-empty list of natural numbers
    big = l[0]
    for i in range(1,len(l)):
        if big < l[i]:
            big = l[i]
    return big
```

Soit n la taille de la liste `L`. Chaque algorithme effectue alors $n-1$ comparaisons. La complexité est linéaire car $n-1 = \Theta(n)$.

Question 2. Écrire une fonction `min_maxL` qui prend en argument une liste d'entiers naturels non vide `L` et renvoie un couple constitué de l'élément minimal et l'élément maximal de `L`.

▷ `minL([0,3,1,7,0,8,12,4])` renvoie `(0,12)`.

Quelle est la complexité de cette fonction, en nombre de comparaisons ?

Première possibilité : réutiliser les fonctions `minL` et `maxL`.

```
def min_maxL(L):
```

```
    # Returns a pair (min,max) where
    # min is the minimal element of L, and
    # max is the maximal element of L.
    # It is assumed that L is a non-empty list of natural numbers
    return (minL(L),maxL(L))
```

Deuxième possibilité : simuler `minL` et `maxL` en les réécrivant directement dans le nouveau code (on appelle cela l'*inlining*).

```
def min_maxL2(L):
    # Returns a pair (min,max) where
    # min is the minimal element of L, and
    # max is the maximal element of L.
    # It is assumed that L is a non-empty list of natural numbers
    min = L[0]
    max = L[0]
    for i in range(1,len(L)):
        if min > L[i]:
            min = L[i]
        elif max < L[i]:
            max = L[i]
    return (min,max)
```

Dans la première version, quelle que soit la liste, il y a toujours $2(n-1)$ comparaisons. Dans la deuxième version, il peut y avoir jusqu'à $2(n-1)$ comparaisons dans le pire cas (élément minimal en première position), mais cela peut descendre jusqu'à $(n-1)$ comparaisons dans le meilleur cas (liste décroissante). Dans les deux cas, la complexité est donc linéaire puisque $2(n-1) = \Theta(n)$. Cependant, la deuxième version est légèrement plus optimisée.

Exercice 4 : Diviser pour régner

On rappelle que l'approche "diviser pour régner" consiste à décomposer un problème en plusieurs sous-problèmes plus faciles, à résoudre récursivement ces sous-problèmes, puis à utiliser leurs solutions pour répondre au problème initial.

En suivant une approche "diviser pour régner" proposer une fonction `min_max_DR` qui prend en argument une liste non vide d'entiers naturels `L` et calculant le couple constitué de l'élément minimal et l'élément maximal.

Indication : on pourra utiliser les fonctions Python prédéfinies `min` et `max` qui prennent en arguments deux nombres et renvoient respectivement leur minimum et leur maximum. Pour les calculs de complexité, on considérera que ces fonctions ont un coût de 1 comparaison.

Quelle est la complexité (en nombre de comparaisons d'éléments) ?

On peut penser à deux solutions différentes. La première consiste à diviser arbitrairement la liste en deux sous-listes (Diviser), puis à résoudre le problème sur ces deux sous-listes (Régner) et enfin à comparer les éléments minimaux et maximaux de ces sous-listes (Combiner).

```
def DR_aux(L,i,j):
    # Returns a pair (min,max), where
    # min is the minimal element of L[i..j], and
    # max is the maximal element of L[i..j].
    # It is assumed that L is a non-empty list of natural numbers
    if j-i <= 1: # This means the list has only one or two elements
        if L[i] <= L[j]:
```

```
            return (L[i],L[j])
        else:
            return (L[j],L[i])
    milieu = (i+j)//2
    (min1,max1) = DR_aux(L,i,milieu)
    (min2,max2) = DR_aux(L,milieu+1,j)
    return (min(min1,min2),max(max1,max2))

def min_max_DR(L):
    # Returns a pair (min,max), where
    # min is the minimal element of L, and
    # max is the maximal element of L.
    # It is assumed that L is a non-empty list of natural numbers
    return DR_aux(L,0,len(L)-1)
```

Calculons la complexité de l'algorithme ci-dessus. Par hypothèse, les fonctions `min` et `max` avec 2 arguments ont un coût de 1 comparaison. Supposons que la taille n de la liste est une puissance de 2 ($n = 2^p$ avec $p = log_2(n)$), le nombre de comparaisons $T(n)$ est alors :

$$T(n) = \begin{cases} 1 & \text{si } n = 2 \\ 2 \times T(\frac{n}{2}) + 2 & \text{sinon} \end{cases}$$

En effet, à chaque appel, il est effectué 2 comparaisons (1 appel de `min` et 1 appel de `max`) et 2 appels récursifs sur des listes de taille moitié ($n/2$).

$$\begin{aligned} T(n) &= 2 \times T(\tfrac{n}{2}) + 2 \\ &= 2 \times (2 \times T(\tfrac{\frac{n}{2}}{2}) + 2) + 2 \\ &= 4 \times T(\tfrac{n}{4}) + 4 + 2 \\ &= 2^i \times T(\tfrac{n}{2^i}) + \textstyle\sum_{j=1}^{i} 2^j \text{ pour tout } i < p \\ &= 2^{p-1} T(2) + \textstyle\sum_{j=1}^{p-1} 2^j \text{ pour } i = p - 1 \\ &= 2^{p-1} + (\textstyle\sum_{j=0}^{p-1} 2^j) - 1 \text{ comme } T(2) = 1 \\ &= \tfrac{n}{2} + (2^p - 1) - 1 \\ &= \tfrac{n}{2} + n - 2 \\ &= \tfrac{3}{2} \times n - 2 \end{aligned}$$

Ainsi l'approche *Diviser pour régner* apporte un gain dans le pire des cas de l'ordre de 25 % par rapport aux approches par simple parcours de la liste (cf. exercice précédent).

La deuxième solution consiste à couper la liste en deux sous-listes de façon intelligente (Diviser) : on s'assure que l'élément minimal se situe dans la première sous-liste et que l'élément maximal se situe dans la deuxième. Ensuite, on recherche l'élément minimal dans la première sous-liste et l'élément maximal dans la deuxième sous-liste (Régner). Enfin, on regroupe les deux éléments obtenus dans un couple et on renvoie le résultat (Combiner).

Pour couper intelligemment la liste, l'idée est de traiter les éléments par paires : on regarde les éléments placés aux indices $2i$ et $2i + 1$, puis on échange si besoin leur place pour que le plus petit soit à l'indice $2i$ et le plus grand à l'indice $2i+1$. La sous-liste comportant uniquement les indices pairs contiendra alors le plus petit élément, et la sous-liste comportant uniquement les indices impairs contiendra le plus grand élément.

```
from math import *

def swap(L,i,j):
    # Exchanges the elements of list L at indexes i and j
    # It is assumed that i and j are indexes of list L
    temp = L[i]
    L[i] = L[j]
    L[j] = temp
```

```
    return L

def min_max_DR2(L):
    # Returns a pair (min,max), where
    # min is the minimal element of L, and
    # max is the maximal element of L.
    # It is assumed that L is a non-empty list of natural numbers
    if len(L) == 1:
        return (L[0],L[0])
    for i in range(0,len(L)//2):
        if L[2*i] > L[2*i+1]:
            L = swap(L,2*i,2*i+1)
    mini = L[0]
    maxi = L[1]
    for i in range(1,len(L)//2):
        if L[2*i] < mini:
            mini = L[2*i]
        if L[2*i+1] > maxi:
            maxi = L[2*i+1]
    if len(L)%2 != 0: # If the length is an odd number
        if L[len(L)-1] > maxi:
            maxi = L[len(L)-1]
        elif L[len(L)-1] < mini:
            mini = L[len(L)-1]
    return (mini,maxi)
```

Analysons la complexité de cet algorithme.
Si n est pair ($n = 2p$), il y a

- ▷ p comparaisons pour constituer les deux sous-listes
- ▷ $p-1$ comparaisons pour calculer l'élément minimal
- ▷ $p-1$ comparaisons pour calculer l'élément maximal

Si n est impair ($n = 2p+1$), il y a

- ▷ p comparaisons pour constituer les deux sous-listes
- ▷ $p-1$ comparaisons pour calculer l'élément minimal
- ▷ $p-1$ comparaisons pour calculer l'élément maximal
- ▷ 1 ou 2 comparaisons supplémentaires pour gérer le dernier élément

La complexité totale est donc, dans les deux cas, en $\Theta(3n/2)$.

3 Un grand classique

Exercice 5 : La fonction $(x, n) \mapsto x^n$ 😖

Question 1. En utilisant l'approche la plus simple possible, implémenter la fonction `power` : $(x, n) \mapsto x^n$, où `x` est un flottant et `n` un entier positif. On s'interdira ici d'utiliser des fonctions prédéfinies comme `**`, `math.pow` ou `numpy.power`. Analyser la complexité en n de votre algorithme. Il s'agit du nombre $T(n)$ de calculs nécessaires pour retourner `power(x,n)`, à `x` fixé.

On propose ici une version récursive.

```
def power(x, n):
```

```
    if n == 0:
        return 1
    else:
        return x*power(x, n-1)
```

Si on note $T(n)$ le nombre de multiplications nécessaires, on a d'après le bloc `if` que $T(0) = 0$ et d'après le bloc `else` que $T(n+1) = 1+T(n)$. On en déduit que pour tout $n \in \mathbb{N}$, $T(n) = n$. La complexité est linéaire car $T(n) = O(n)$.

Question 2. En utilisant une approche diviser pour régner, implémenter une autre fonction `power2` et analyser sa complexité, qui devra être logarithmique. Pourquoi est-ce que cette solution est bien meilleure que la première ?

Indications. Si on connaît x^m, comment calculer efficacement x^{2m} ? Pour le calcul de complexité, on pourra commencer par montrer par récurrence sur k que si $2^k \leq n < 2^{k+1}$, alors $T(n) \leq 2k+2$.

L'idée est d'utiliser que $x^{2m} = x^m \times x^m$ lorsque $n = 2m$ est pair, et $x^{2m+1} = x \times x^m \times x^m$ lorsque $n = 2m+1$ est impair. On ne calcule alors qu'une seule fois le facteur x^m, ce qui permet de réduire drastiquement la complexité.

```
def power2(x, n):
    if n == 0:
        return 1
    else:
        m = n//2
        temp = power2(x,m)
        if n%2 == 0:
            return temp*temp
        else:
            return x*temp*temp
```

En notant $T'(n)$ le nombre de multiplications nécessaires pour calculer `power2(x,n)`, on a d'après le premier bloc `if` que $T'(0) = 1$, et d'après le bloc `else` qui suit, que $T'(2m) = 1+T'(m)$ et $T'(2m+1) = 2+T'(m)$. Montrons que le résultat de l'indication est vrai.

Initialisation ($k = 0$) : soit n tel que $2^0 \leq n \leq 2^1$, c'est-à-dire $n = 1$. Alors $T'(1) = 2$ (puisqu'on se retrouve dans la dernière ligne de la fonction, où il y a deux multiplications) et on a bien $T'(1) \leq 2\times 0+2$.

Hérédité : Supposons que pour tout m tel que $2^k \leq m < 2^{k+1}$, $T'(m) \leq 2k+2$. Soit $n \in \mathbb{N}$ tel que $2^{k+1} \leq n < 2^{k+2}$. On doit montrer que $T'(n) \leq 2(k+1)+2$. Pour cela, on remarque que :

- Si n est pair, $n = 2m$ avec $2^k \leq m < 2^{k+1}$ et on a $T'(n) = 1+T'(m) \leq 1+(2k+2)$ par hypothèse de récurrence. Comme $1+(2k+2) \leq 2(k+1)+2$, on en déduit que $T'(n) \leq 2(k+1)+2$.
- Si n est impair, $n = 2m+1$ avec $2^k \leq m < 2^{k+1}$ et on a $T'(n) = 2+T'(m) \leq 2+(2k+2)$ par hypothèse de récurrence. D'où $T'(n) \leq 2(k+1)+2$.

De plus, si $2^k \leq n < 2^{k+1}$, alors en passant au logarithme $k \leq \log_2(n) < k+1$ d'où $k = \lfloor \log_2(n) \rfloor$ (on utilise la notation $\lfloor y \rfloor$ pour la partie entière de y). En conclusion, on a pour tout $n \in \mathbb{N}$ que $T'(n) \leq 2\lfloor \log_2(n) \rfloor + 2$. En particulier, on peut écrire $T'(n) = O(\log_2(n))$, ce qui est une complexité logarithmique.

Répondons maintenant à la dernière partie de la question. La version diviser pour régner est bien meilleure que la version naïve car quand $n \to +\infty$, $\frac{T(n)}{T'(n)} = \frac{n}{2\lfloor \log_2(n) \rfloor + 2} \to +\infty$. En d'autres termes, la version diviser pour régner est asymptotiquement infiniment meilleure que la version naïve... En guise

de comparaison, dans l'exercice sur le min et max d'une liste, le quotient complexité naïve / complexité optimisée est seulement de 3/2.

4 Somme d'une sous-matrice

Exercice 6 : Méthode directe 😀

L'objectif des prochains exercices est de réaliser un algorithme qui calcule la somme des éléments d'une sous-matrice d'une matrice de nombres.

Pour ces exercices, nous n'allons pas utiliser des libraires adaptées au calcul de matrices telles que `numpy`. Une matrice M de l lignes et de c colonnes sera représentée en Python comme une liste de l listes de taille c. Une sous-matrice de M est définie par un intervalle de lignes de l_1 à l_2 et un intervalle de colonnes de c_1 à c_2, où $0 \leq l_1 \leq l_2 < l$ et $0 \leq c_1 \leq c_2 < c$. Notez qu'en numérotant les lignes de 0 à $l-1$ et les colonnes de 0 à $c-1$, nous avons adopté la notation Python. Par exemple, la matrice

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

sera représentée par la liste `[[1,2,3],[4,5,6]]`, ou plus visuellement :

```
[[1,  2,  3],
 [4,  5,  6]]
```

Question 1. Implémenter une fonction `ismatrix` censée prendre en argument une liste de listes d'entiers `m` et renvoyer un triplet `(b,l,c)` composé

- ▷ D'un booléen `b` qui vaut `True` uniquement si `m` représente une matrice.
- ▷ D'un entier `l` qui donne le nombre de lignes de `m` (dans le cas où `m` est bien une matrice).
- ▷ D'un entier `c` qui donne le nombre de colonnes de `m` (dans le cas où `m` est bien une matrice).

Par exemple :

- ▷ `ismatrix([[1,2,3],[4,5,6]])` renverra le triplet `(True, 2, 3)`.
- ▷ `ismatrix([[1,2,3],[4,5])` renverra le triplet `(False, None, None)` où `n` et `m` sont des nombres arbitraires de votre choix.

On commencera par donner l'algorithme en pseudo-code, puis on programmera la fonction `ismatrix` en Python en faisant les vérifications nécessaires quant au typage des arguments (revoir TD1 si besoin).

Le principe de l'algorithme est de vérifier que toutes les lignes ont la même dimension.

```
1:  procedure ISMATRIX(m)
2:      if m = 0 then
3:          return (True, None, None)
4:      else
5:          l = longueur (m)
6:          c = longueur (m(0))
7:      for i ∈ {2...l} do
8:          if c ≠ longueur (m(i)) then
9:              return (False, None, None)
10:     return (True, l, c)
```

Pour l'implémentation en Python, il faut maintenant vérifier que les types de objets sont les bons. On vérifie d'abord que `m` est une liste, puis que ses éléments sont des listes, puis que leurs éléments sont des entiers,

puis que les dimensions sont compatibles avec la structure de matrice (cette dernière étape a déjà été faite dans la question précédente).

```
def ismatrix(m):
    # Returns a triple (b,l,c)
    # b is a boolean that is true iff m is a matrix,
    # l is the number of lines of this matrix,
    # c is the number of columns of this matrix.
    # If b is false, the value of l and c do not matter.
    if type(m) == list:
        if m == []:
            return (True, 0, 0)
        l = len(m)
        if type(m[0]) == list:
            c = len(m[0])
            for e in m:
                if not (type(e) == list) or len(e) != c:
                    return (False, None, None)
                for el in e:
                    if not (type(l) == int):
                        return (False, None, None)
            return (True, l, c)
        else:
            return (False, None, None)
    else:
        return (False, None, None)
```

Question 2. Implémenter une function `sommation_dir` qui prend en argument une liste de liste d'entiers `m`, des entiers `l1`, `l2`, `c1`, `c2` et qui évalue la somme des éléments de la sous-matrice de `m` constituée des lignes `l1` à `l2` et des colonnes `c1` à `c2`.

▷ `sommation_dir([[1,2,3],[4,5,6]],1,1,0,1)` renvoie 9.
▷ `sommation_dir([[1,1,1,1],[1,1,1,1],[1,1,1,1],[1,1,1,1]],1,3,1,2)` renvoie 6.

Il est demandé de retourner un message d'erreur en cas de mal-formation des arguments. Quelle est la complexité de l'algorithme ?

```
def sommation_dir(m, l1, l2, c1, c2):
    # Returns the sum of the elements of the submatrix [l1..l2] x [c1 .. c2] of m.
    # It is assumed that c is a list of lists of natural numbers
    # and that l1,l2,c1,c2 are natural numbers
    (b, l, c) = ismatrix(m)
    if b and 0 <= l1 and l1 <= l2 and l2 < l and 0 <= c1 and c1 <= c2 and c2 < c:
        sum = 0
        for c_index in range(c1, c2 + 1):
            for l_index in range(l1, l2 + 1):
                sum += m[l_index][c_index]
        return sum
    else:
        return 'sommation_dir : bad arguments'
```

Par la suite, le cas `return 'sommation_dir : bad arguments'` sera plutôt géré en utilisant les mécanismes de traitement d'exceptions. En effet, la solution ci-dessus présente l'inconvénient que la fonction `sommation_dir` retourne selon les cas un nombre ou une chaîne de caractères (ce qui peut être gênant lors

des utilisations ultérieures de la fonction.

La complexité en nombre d'additions est $(l_2 - l_1 + 1) \times (c_2 - c_1 + 1)$, soit $\Theta((l_2 - l_1) \times (c_2 - c_1))$.

Exercice 7 : Version optimisée

Question 1. Implémenter une fonction `table_sommation` qui prend en argument une matrice `m` et renvoie une matrice `ts` de mêmes dimensions vérifiant

$$\texttt{ts[i][j]} = \sum_{0 \leq \texttt{u} \leq \texttt{i}, 0 \leq \texttt{v} \leq \texttt{j}} \texttt{m[u][v]}$$

En cas de mal-formation des arguments, la fonction `table_sommation` renvoie un message d'erreur. Il est attendu que la complexité de la fonction soit en $O(n \times m)$ où n est le nombre de lignes et m le nombre de colonnes.

- ▷ `table_sommation([[1,2,3],[4,5,6]])` renvoie `[[1,3,6],[5,12,21]]`.
- ▷ `table_sommation([[1,1,1,1],[1,1,1,1],[1,1,1,1],[1,1,1,1]])` renvoie `[[1,2,3,4],[2,4,6,8],[3,6,9,12],[4,8,12,16]]`.

Si on calcule brutalement les coefficients de la nouvelle matrice un par un, la complexité sera de

$$\begin{aligned}\sum_{0 \leq i \leq n-1, 0 \leq j \leq m-1} i \times j &= \left(\sum_{i=0}^{n-1} i\right) \times \left(\sum_{j=0}^{m-1} j\right) \\ &= \frac{(n-1)(n-2)}{2} \times \frac{(m-1)(m-2)}{2} \\ &= \Theta(n^2 m^2)\end{aligned}$$

ce qui est bien trop grand par rapport à ce qui est demandé. Il va donc falloir faire preuve de ruse et utiliser les coefficients déjà calculés par l'algorithme pour calculer les suivants.

```
1  def copy_matrix(m):
2      # Returns a copy of the matrix m
3      copie = []
4      for ligne in m:
5          copie += [ligne.copy()]
6      return copie
7
8
9  def table_sommation(m):
10     # Returns a matrix ts such that ts[i][j] is the sum of m[u][v]
11     # for all (u,v) above left of (i,j)
12     (b, l, c) = ismatrix(m)
13     if b:
14         ts = copy_matrix(m)
15         for l_index in range(0, l):
16             lsum = 0  # l_sum is the partial sum of elements of line l_index
17             for c_index in range(0, c):
18                 if l_index == 0:
19                     if c_index == 0:
20                         ts[l_index][c_index] = m[0][0]
21                     else:
22                         ts[l_index][c_index] = ts[l_index][c_index - 1] \
23                                                + m[l_index][c_index]
24                 else:
```

```
                    lsum += m[l_index][c_index]
                    ts[l_index][c_index] = lsum + ts[l_index - 1][c_index]
        return ts
    else:
        return 'table_sommation : bad_arguments'
```

La raison pour laquelle on commence par créer une fonction qui crée une copie d'une matrice est que si l'on écrivait directement `ts = m`, modifier `ts` modifierait aussi `m`, ce que l'on ne souhaite pas. Dans le cas d'une liste simple, on pourrait utiliser `ts = m.copy()` pour régler ce problème, mais ici, `m` est une liste *de listes*, donc le problème se pose encore. La fonction `copy_matrix` utilise donc la fonction `copy` directement au niveau des lignes de la matrice. Ce défaut un peu désagréable de Python sera revu dans un prochain TD.

Les deux `for` imbriqués donnent bien une complexité en $\Theta(n \times m)$.

Question 2. En considérant que la table de sommation `ts` d'une matrice `m` est précalculée, implémenter une fonction `sommation_tab` qui prend en arguments `m`, `l1`, `l2`, `c1`, `c2` et qui renvoie la somme des éléments de la sous-matrice `[l1..l2] x [c1..c2]` de `m`. Cette fonction a les mêmes entrées/sorties que celle de l'exercice précédent, mais cette fois, il est attendu que la complexité soit en $O(1)$. Cette approche peut être avantageuse si l'on est amené à faire appel de nombreuses fois au calcul de la somme des éléments d'une sous-matrice. En effet, pour ce genre de problèmes, on va d'abord effectuer une étape d'initialisation où l'on calcule une bonne fois pour toutes la table de sommation, ce qui permet ensuite d'avoir des accès très rapides aux données.

L'expression générique de

$$\sum_{\texttt{l1} \leq \texttt{i} \leq \texttt{l2}, \texttt{c1} \leq \texttt{j} \leq \texttt{c2}} \texttt{m[i][j]}$$

est donnée par

$$\texttt{ts[l2][c2]} - \texttt{ts[l2-1][c2]} - \texttt{ts[l2][c2-1]} + \texttt{ts[l2-1][c2-1]}$$

en prenant soin de traiter à part les cas particuliers (`l1 == 0` ou/et `c1 == 0`).

```
def sommation_tab(m, l1, l2, c1, c2):
    (b, l, c) = ismatrix(m)
    if b and 0 <= l1 and l1 <= l2 and l2 < c and 0 <= c1 and c1 <= c2 and c2 < c:
        ts = table_sommation(m)
        if l1 == 0:
            if c1 == 0:
                return ts[l2][c2]
            else:
                return ts[l2][c2] - ts[l2][c1 - 1]
        else:
            if c1 == 0:
                return ts[l2][c2] - ts[l1 - 1][c2]
            else:
                return ts[l2][c2] - ts[l1 - 1][c2] \
                       - ts[l2][c1 - 1] + ts[l1 - 1][c1 - 1]
    else:
        return ' sommation_tab : bad arguments '
```

5 Recherche de deux éléments d'une liste à partir de leur somme

On souhaite répondre à la question suivante : étant donné une liste `l` et un entier `x`, existe-t-il deux éléments de `l` d'indices distincts dont la somme est égale à `x` ?

Exercice 8 : Méthode naïve

Question 1. Écrire une fonction `testSum` qui prend en argument une liste `l` et un entier `x` et renvoie un booléen qui indique si il existe deux éléments de `l` d'indices distincts dont la somme est égale à `x`. L'algorithme doit être le plus simple possible et se baser sur des comparaisons. La complexité dans le pire des cas doit être de $\Theta(n^2)$ comparaisons.

```
def testSum(l, x):
    for i in range(0,len(l)-1):
        first = l[i]
        for j in range(i+1,len(l)):
            if i != j and x == l[i] + l[j]:
                return True
    return False
```

Dans la deuxième boucle `for`, on peut commencer directement à `i+1` (et pas à `0`) puisque les couples (i, j) pour lesquels $j \leq i$ ont déjà été traités précedemment par l'algorithme. En faisant cela, on divise la complexité en moyenne par 2 (ce qui ne change pas l'ordre de grandeur $\Theta(n^2)$).

Question 2. Prouver que la complexité dans le pire des cas est en $\Theta(n^2)$.

Le pire des cas correspond à une situation où les boucles tournent jusqu'au bout, ce qui se produit quand le résultat est `False`. Dans ce cas, le nombre de comparaisons (c'est-à-dire le nombre de fois où l'on teste si `x == l[i] + l[j]`) est de

$$\sum_{0\leq i<n-1} \sum_{i+1\leq j<n} 1 = \sum_{0\leq i<n-1} (n-i-1) = (n-1)^2 - \frac{(n-1)(n-2)}{2} = \frac{n(n-1)}{2} = \Theta(n^2)$$

Exercice 9 : Solution optimisée

Il est possible de répondre à ce problème avec une complexité en $O(n \log n)$ comparaisons dans le pire des cas.

Question 1. Écrire une fonction `testSumOpti` qui renvoie le même résultat que `testSum`, mais avec une complexité en $O(n \log n)$ dans le pire des cas. On pourra utiliser le tri rapide de Python `sorted` qui prend en argument une liste et renvoie la liste triée dans l'ordre croissant en complexité $O(n \log n)$.

Il faut être rusé. L'énoncé nous indique que l'on peut utiliser un tri en $O(n \log n)$. Comme on veut une complexité globale en $O(n \log n)$, cela ne coûte donc rien que de commencer par trier la liste. Le problème revient donc à vouloir trouver deux éléments d'une liste triée dont la somme fait `x`. Essayons de faire comme l'algorithme précédent et fixons un indice `i`. Le problème revient alors à effectuer la recherche de l'élément `x - l[i]` dans la liste triée `l`, ce qui peut se faire en complexité $O(\log n)$ à l'aide d'une recherche dichotomique.

On obtient donc l'algorithme suivant.

```python
def dichotomicSearch(l, y, begin, end):
    # Finds an index of l between begin and end
    # such that l[index] = y
    # If there is no such index, returns -1
    # The list l must be sorted!
    if end < begin:
        return -1
    else:
        m = (begin + end)//2
        if l[m] == y:
            return m
        elif l[m] < y:
            return dichotomicSearch(l, y, m+1, end)
        else:
            return dichotomicSearch(l, y, begin, m-1)


def testSumOpti(l, x):
    l1 = sorted(l)
    length = len(l)
    for i in range(0,length-1):
        y = x - l1[i]
        j = dichotomicSearch(l, y, 0, length-1)
        if j >= 0 and not (j == i):
            return True
    return False
```

Question 2. Justifier proprement que la complexité dans le pire des cas est bien en $O(n \log n)$.

On sait déjà que la complexité du tri est en $O(n \log n)$. Ensuite, il y a une boucle de taille n dans laquelle on effectue à chaque itération une recherche dichotomique (complexité $O(\log n)$) et deux comparaisons (complexité $O(1)$). La complexité totale de l'algorithme est donc

$$O(n \log n) + n \times (O(\log n) + O(1)) = O(n \log n) + n \times O(\log n) = O(n \log n) + O(n \log n) = O(n \log n)$$

6 Sous-liste de somme maximale

On cherche à résoudre le problème suivant. Étant donné une liste d'entiers `l`, une sous-liste de `l` est une liste de la forme `l[i:j]` où $0 \leq$ `i` $<$ `j` $\leq$ `len(l)`. Quelle est une sous-liste de somme maximale de `l` ? Autrement dit, comment trouver deux `i` et `j` tels que $\sum_{\texttt{i} \leq k < \texttt{j}} \texttt{l}[k]$ est maximale ?

Par exemple, la liste $(1, -1, 1, 2, 3, -2)$ a pour sous-liste de somme maximale $(1, 2, 3)$ de somme 6. Ce n'est pas la seule possibilité, la sous-liste $(1, -1, 1, 2, 3)$ a aussi pour somme 6.

Ce problème peut être résolu par beaucoup d'algorithmes de complexité très variable. Nous allons en voir trois.

Exercice 10 : Méthode brutale 😀

Question 1. Décrire informellement l'algorithme de type force brute (= tester toutes les combinaisons possibles) le plus simple qui vous vient à l'esprit pour répondre à la question posée. Estimer sa complexité.

Raisonnement rapide :

- On calcule toutes les sous-listes possibles : complexité $\Theta(n^2)$ puisque cela revient à faire varier les deux indices `i` et `j`, donc à imbriquer deux boucles.
- On calcule la somme de chaque sous-liste : $\Theta(n)$ pour chaque sous-liste.
- On trouve la plus grande de ces n^2 sommes : $\Theta(n^2)$, c'est une recherche de maximum. Enfin, on renvoie les indices correspondants.

La complexité totale est donc $\Theta(n^2) \times \Theta(n) + \Theta(n^2) = \Theta(n^3)$.

Raisonnement détaillé : étant donné $k \in [1, n]$, il y a $n - k + 1$ sous-listes de taille k (car on ne considère que les sous-listes du type `l[i:j]`). Calculer la somme d'une sous-liste de taille k nécessite k additions. Le nombre total d'additions est donc de

$$\sum_{k=1}^{n}(n-k+1)k = (n+1)\sum_{k=1}^{n}k - \sum_{k=1}^{n}k^2 = (n+1)\frac{n(n+1)}{2} - \frac{n(n+1)(2n+1)}{6} = \frac{n(n+1)(n+2)}{6}$$

Il faut ajouter à cette complexité celle de la recherche de maximum, ce qui nécessite de parcourir une liste de taille le nombre de sous-listes, qui est de $\binom{n+1}{2}$ (on choisit deux indices parmi $\{0, ..., n\}$). Cette complexité se noie dans la précédente pour donner une complexité totale de

$$\frac{n(n+1)(n+2)}{6} + \frac{n(n+1)}{2} = \Theta(n^3)$$

Question 2. Écrire une fonction `maxsublist` qui implémente cet algorithme.

```python
def sum_list(l):
    if l == []:
        return 0
    else:
        return l[0] + sum_list(l[1:])


def maxsublist(l):
    n = len(l)
    max_sum = l[0]
    low = 0
    high = 1
    for i in range(0, n):
        for j in range(i+1, n+1):
            sub_sum = sum_list(l[i:j])
            if sub_sum > max_sum:
                max_sum = sub_sum
                low = i
                high = j
    return max_sum, low, high
```

Dans `maxsublist`, on initialise à `(l[0], 0, 1)`. Cela correspond d'ailleurs aux valeurs qui seront prises lors du premier passage dans la boucle.

Question 3. Si à la question précédente vous avez trouvé une complexité de $\Theta(n^3)$, il est encore possible d'améliorer facilement votre algorithme. En économisant des opérations lors du calcul des sommes des sous-listes, modifier l'algorithme pour que sa complexité soit $\Theta(n^2)$.

Quand on incrémente le deuxième indice, ce n'est pas la peine de recalculer la somme de la sous-liste depuis le début, il suffit d'ajouter `l[j]` à la somme précédente. Cela donne la fonction suivante :

```
def maxsublist2(l):
    n = len(l)
    max_sum, low, high = l[0], 0, 1
    for i in range(0, n):
        sub_sum = 0
        for j in range(i, n):
            sub_sum = sub_sum + l[j]
            if sub_sum > max_sum:
                max_sum = sub_sum
                low = i
                high = j+1
    return max_sum, low, high
```

Avec cette nouvelle solution, la complexité n'est plus que de $\Theta(n^2) \times \Theta(1) + \Theta(n) = \Theta(n^2)$.

Exercice 11 : Diviser pour régner

En fait, la complexité optimale pour résoudre ce problème est $\Theta(n)$. Ici, on va seulement montrer qu'il est possible de le résoudre en complexité $\Theta(n \log n)$ grâce à la méthode "diviser pour régner".

On cherche à écrire une fonction `optisublist` qui prend en argument une liste `l` et deux indices `low` et `high`, et qui renvoie la sous-liste de somme maximale *incluse dans* `l[low:high]`.

- On *divise* en séparant la liste `l[low:high]` en deux listes de même taille (on coupe au milieu `mid` $= \frac{\texttt{low+high-1}}{2}$).
- On *règne* en trouvant récursivement la sous-liste de somme maximale dans la première liste, `optisublist(l,low,mid+1)` puis dans la deuxième, `optisublist(l,mid+1,high)`.
- Enfin, on utilise ces deux sous-résultats pour déduire quelle est la sous-liste de somme maximale dans `l[low:high]`. C'est l'étape difficile.

On définit ensuite tout simplement la fonction `maxsublist3(l) = optisublist(l,0,len(l))`.

Question 1. Implémenter la fonction `optisublist`.

Il y a trois possibilités. Soit la sous-liste que l'on recherche est incluse dans `l[low:mid+1]` et dans ce cas on renverra `optisublist(l,low,mid+1)`. Soit la sous-liste que l'on recherche est incluse dans `l[mid+1:high]` et dans ce cas on renverra `optisublist(l,mid+1,high)`. Soit la sous-liste que l'on recherche chevauche à la fois `l[low:mid+1]` et `l[mid+1:high]`, autrement dit, elle contient à la fois les indices `mid` et `mid+1`.

Il faut donc gérer ce troisième cas à l'aide d'une fonction dédiée `crossingsublist`.

```
def crossingsublist(l,low,mid,high):
    left_sum = l[mid]
    max_left = mid
```

```
    sub_sum = 0
    for i in range(mid,low-1,-1):
        sub_sum = sub_sum + l[i]
        if sub_sum > left_sum:
            left_sum = sub_sum
            max_left = i
    right_sum = l[mid+1]
    max_right = mid+1
    sub_sum = 0
    for i in range(mid+1,high):
        sub_sum = sub_sum + l[i]
        if sub_sum > right_sum:
            right_sum = sub_sum
            max_right = i
    return (left_sum + right_sum), max_left, (max_right + 1)


def optisublist(l,low,high):
    if low == high - 1:
        return l[low], low, low
    else:
        mid = (low + high - 1)//2
        s_left, i_left, j_left = optisublist(l, low, mid+1)
        s_right, i_right, j_right = optisublist(l, mid+1, high)
        s_cross, i_cross, j_cross = crossingsublist(l, low, mid, high)
        s = max(s_left, s_right, s_cross)
        if s == s_left:
            return s_left, i_left, j_left
        elif s == s_right:
            return s_right, i_right, j_right
        else:
            return s_cross, i_cross, j_cross

def maxsublist3(l):
    return optisublist(l,0,len(l))
```

Question 2. Prouver que la complexité de `maxsublist3` est en $\Theta(n \log n)$, où n est la taille de la liste d'entrée.

La complexité de `crossingsublist` est clairement linéaire. Si on note $T(n)$ la complexité de `maxsublist3` pour une entrée de taille n, on a (lorsque n est pair, mais pour simplifier on néglige les effets de bord lorsque n est impair) :

$$T(n) = 2T(n/2) + O(n)$$

Il s'agit de la même relation que celle que l'on trouve pour le tri fusion, dont on sait que la complexité est en $\Theta(n \log n)$. Par conséquent, la complexité de `maxsublist3` est également en $\Theta(n \log n)$.

7 Diviser pour reigner (avancé)

"Diviser pour régner" est une technique algorithmique qui consiste à décomposer un problème en plusieurs sous-problèmes plus faciles (diviser), à résoudre récursivement ces sous-problèmes (réfgner), puis à utiliser leurs solutions pour répondre au problème initial (combiner). Il s'agit ici de problèmes légèrement plus difficiles.

Exercice 12 : Calcul d'une ligne de gratte-ciels (skyline) 😖

Le problème de la ligne de gratte-ciels (skyline problem) consiste à partir d'un ensemble de gratte-ciels (donnés sous la forme d'un triplet $[x, h, y]$ avec x abscisse gauche, h hauteur, et y abscisse droite) à construire la ligne de séparation entre les bâtiments et le ciel sous la forme $[x1, h1, x2, h2, ..., xi, hi, ..., xn, 0]$ représentant la ligne de séparation sous forme d'escalier.

Pour l'exemple de la figure 7, la ligne de gratte-ciels associée à l'ensemble de gratte-ciels

$$[[3, 13, 9], [1, 11, 5], [12, 7, 16], [14, 3, 25], [19, 18, 22], [2, 6, 7], [23, 13, 29], [23, 4, 28]]$$

est :

$$[1, 11, 3, 13, 9, 0, 12, 7, 16, 3, 19, 18, 22, 3, 23, 13, 29, 0]$$

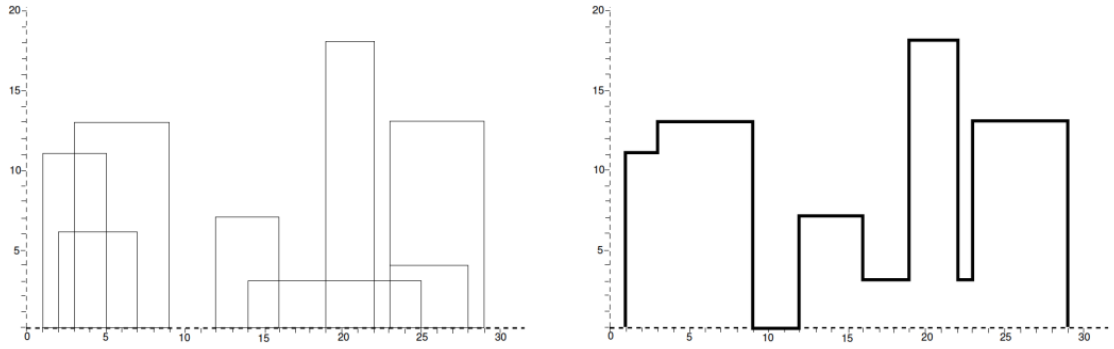


Figure 1: Ligne de gratte-ciels.

Question 1 : Ecrire l'algorithme qui à partir d'une liste de gratte-ciels, fournit la ligne de gratte ciel.

Cet exercice est intéressant à deux titres :

- Sans avoir l'air d'y toucher, 2 structures de données sont manipulées, à savoir une liste de gratte-ciels, ces derniers modélisés sous la forme d'une liste à trois éléments, et la ligne de séparation entre l'horizon et les bâtiments sous forme d'une liste de longueur paire (représentant une fonction en escalier qui commence et finit à 0). La liste se présente sous la forme $[abs_1, h_1, abs_2, h_2, \ldots, abs_k, h_k]$, où abs_i indique l'abscisse à laquelle l'horizon se positionnera à la hauteur h_i, avec la convention que la dernière abscisse abs_k est associée à la hauteur $h_k = 0$ indiquant la fin de la ligne d'horizon.
- rien *a priori* dans l'énoncé ne semble suggérer d'adopter la méthode "diviser pour régner" (sauf le titre de la feuille d'exercice).

```
1  def mergeskyline(s1,s2,h1,h2,prev_h):
2      """
3      input :
4          s1 premiere skyline
5          s2 seconde skyline
6          h1 hauteur de la premiere skyline a l'abscisse courante
7          h2 hauteur de la seconde skyline a l'abscisse courante
8          prev_h hauteur de la skyline fusionnee avant l'abscisse courante
9
10     output :
```

```
        fusion des deux skylines
    """
    if s1 == []:
        return s2
    elif s2 == []:
        return s1

    # la skyline s1 est modifiee avant la skyline s2
    elif s1[0] < s2[0]:
        cur_h = max(s1[1],h2)

        # on ne change pas la hauteur on peut donc continuer
        if cur_h == prev_h :
            return mergeskyline(s1[2:],s2,s1[1],h2,cur_h)

        # on change la hauteur et on continue
        else :
            return [s1[0],cur_h] + mergeskyline(s1[2:],s2,s1[1],h2,cur_h)

    # les deux skylines sont modifiees a la meme abscisse
    # la hauteur est necessairement changee
    elif s1[0] == s2[0]:
        cur_h = max(s1[1],s2[1])
        return [s1[0],cur_h] + mergeskyline(s1[2:],s2[2:],s1[1],s2[1],cur_h)

    # la skyline s2 est modifiee avant la skyline s1
    else:
        cur_h = max(s2[1],h1)
        if cur_h == prev_h:
            return mergeskyline(s1,s2[2:],h1,s2[1],cur_h)
        else:
            return [s2[0],cur_h] + mergeskyline(s1,s2[2:],h1,s2[1],cur_h)


def skyline(set_skycraper):
    """
    input :
        set_skycraper liste des grattes-ciels

    output :
        skyline
    """
    if set_skycraper == []:
        return [0,0]
    elif len(set_skycraper) == 1:
        return set_skycraper[0] + [0]
    else:

        # Divide
        half = len(set_skycraper)//2
        set_s1 = set_skycraper[:half]
        set_s2 = set_skycraper[half:]

        # Conquer
        skyline1 = skyline(set_s1)
        skyline2 = skyline(set_s2)

        # Combine
        # par convention, on commence avec la ligne 1
        line = mergeskyline(skyline1,skyline2,0,0,0)
```

```
        return line


set_skycraper = [[3,13,9],[1,11,5],[12,7,16],
                 [14,3,25],[19,18,22],[2,6,7],
                 [23,13,29],[23,4,28]]

# resultat attendu
# [1, 11, 3, 13, 9, 0, 12, 7, 16, 3, 19, 18, 22, 3, 23, 13, 29, 0]
```

L'algorithme est très similaire à celui du tri fusion :

- l'ensemble des gratte-ciel est séparé en 2, jusqu'à obtenir un ensemble réduit à 0 ou 1 seul gratte-ciel ;
- pour les cas de base (0 ou 1 gratte-ciel), la ligne d'horizon est construite directement
- l'étape standard consiste à appeler récursivement la construction de la ligne d'horizon sur les deux ensembles de taille moitié, et à combiner deux lignes d'horizon
- pour l'étape de combinaison des lignes d'horizon, il s'agit d'un parcours linéaire des deux lignes d'horizon en mettant à jour la ligne d'horizon selon les abscisses rencontrées dans les lignes d'horizon, selon l'ordre croissant. Un cas particulier notable : si la hauteur courante de la ligne d'horizon en construction coïncide avec la hauteur que l'on souhaite assigner à la ligne d'horizon, alors on passe à l'abscisse suivante.

Question 2 : Quelle en est la complexité ? Peut-on faire mieux ?

L'étape de combinaison est linéaire (en la somme des longueurs des deux lignes d'horizon à fusionner), et par conséquent, la complexité globale est comparable à celle du tri fusion, soit $O(n \log n)$.
En fait, on ne peut pas faire mieux, car sinon, cela voudrait dire que l'on saurait trier en moins que $n \log n$. Si on considère une liste (désordonnée) contenant les n gratte-ciels $[i, i, 2n]$ pour i de 1 à n, construire la ligne de gratte-ciels permet de les trier. Comme $O(n \log n)$ est une complexité optimale pour le tri, il n'est donc pas possible de construire la ligne de gratte-ciels en moins de $O(n \log n)$.

Exercice 13 : Points 2D les plus proches

Considérons un ensemble E de n points p donnés par leurs coordonnées planaires (p_x, p_y) vérifiant (pour des soucis de simplicité) que deux points distincts p^1 et p^2 de E ont des abscisses et des ordonnées différentes ($p^1_x \neq p^2_x$ et $p^1_y \neq p^2_y$).

Question 1 : Ecrire une fonction naïve **dmin(L):list** $\rightarrow$ **float** qui calcule la plus petite distance possible entre deux points distincts de E. L'entrée est une liste de liste de la forme $[[x1, y1], [x2, y2], \ldots, [xi, yi], \ldots, [xn, yn]]$.

On donne le code de calcul de la distance entre deux points :

```
import math

def distance(p1,p2):
    return math.sqrt(abs(p1[0] - p2[0])**2 + abs(p1[1] - p2[1])**2)
```

On propose aussi un code pour générer une liste de points pseudo-aléatoires:

```
import random
random.seed(0)

def build_point(min_x, max_x, min_y, max_y):
    """
```

```
    input :
        min_x abscisse minimale
        max_y abscisse maximale
        min_y ordonnee minimale
        max_y ordonnee maximale
    output :
        un point pseudo-aleatoire
        d'abscisse comprise entre min_x et max_y
        et d'ordonnee comprise entre min_y et max_y
    """
    x = min_x + random.random() * (max_x - min_x)
    y = min_y + random.random() * (max_y - min_y)
    return [x,y]

def build_points(min_x, max_x, min_y, max_y, nb_points):
    """
    input :
        min_x abscisse minimale
        max_y abscisse maximale
        min_y ordonnee minimale
        max_y ordonnee maximale
        nb_points nombre de points souhaites
    output :
        une liste de points pseudo-aleatoires
        d'abscisse comprise entre min_x et max_y
        et d'ordonnee comprise entre min_y et max_y
    """
    return [build_point(min_x, max_x, min_y, max_y) for _ in range(nb_points)]

points = build_points(-50,50,-50,50,100)
```

```
# fonction naive
def dmin(points):
    """
    input :
        liste de points
    output :
        distance minimale entre deux points
    """
    n = len(points)

    if n < 1:
        print('Il faut au moins deux points')
        # renvoyer une distance infinie
        return float("inf")

    # initialiser la distance minimale
    min_d = distance(points[0], points[1])

    # double parcours des points
    for i in range(n):
        for j in range(i):

            # mettre a jour la distance
            cur_d = distance(points[i], points[j])
            if cur_d < min_d:
                min_d = cur_d
    return min_d
```

```
28
29 # resultat attendu avec une seed initialisee a 0
30 # 1.172871557467097
```

Question 2 : Quelle est la complexité en temps de votre algorithme en supposant que le calcul de distance entre deux points se fait en temps constant ?

On effectue $\sum_{i=0}^{n-1}\sum_{j=0}^{i-1} 1 = \sum_{i=0}^{n-1} i = \frac{(n-1)n}{2}$ calculs de distance. La solution donnée est donc en $\Theta(n^2)$.

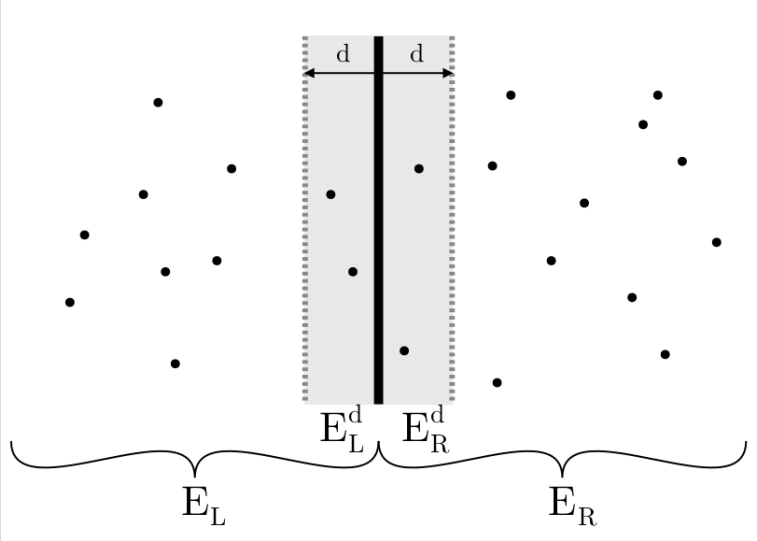
Question 3 : Proposer un algorithme de type *diviser pour régner* pour résoudre le problème de recherche des plus proches points en 2D.

Approche *Diviser pour régner :*

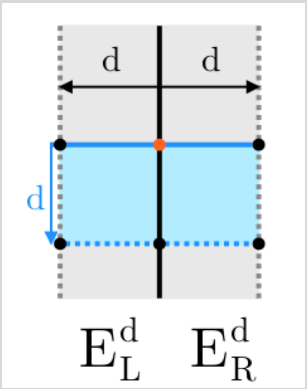
- Trier des points selon leur abscisse
- Partager l'ensemble des points en deux ensemble de taille (quasi) égale selon l'abscisse médiane (x_m), soit en E_L ensemble des points situés à gauche de l'abscisse médiane (i.e. dont l'abscisse est inférieur à l'abscisse médiane), E_R ensemble des points situés à droite de l'abscisse médiane.
- Soit d_L et d_R les plus petites distances entre deux points respectivement dans les ensembles E_L et E_R
- L'étape de combinaison requiert de regarder s'il y a un couple (p^1, p^2) de points dans $E_L \times E_R$ vérifiant $d(p^1, p^2) < d$ avec $d = min(d_L, d_R)$.

Question 4 : Montrer que la complexité de l'algorithme proposé pour la question précédente est en $O(n \log n)$. Il est attendu que l'étape de combinaison des solutions des sous-problèmes soit soigneusement décrite et justifiée.

On peut remarquer qu'un couple de points (p^1, p^2) de points dans $E_L \times E_R$ vérifiant $d(p^1, p^2) < d$ avec $d = min(d_L, d_R)$ vérifie nécessairement que $p_x^1 \geq x_m - d$ et $p_x^2 \leq x_m + d$.
En considérant E_L^d (resp. E_R^d) le sous-ensemble de E_L à une distance au plus de d de la droite $x = x_m$, on peut remarquer que pour tout point de E_L^d, il y a forcément un nombre borné (constant) de points dans E_R^d situés à une distance inférieure à d. Cette construction est illustrée sur le schéma suivant :



En effet, deux points p^1 et p^2 de E_L^d (resp. E_R^d) ne peuvent être à distance strictement inférieure à d. Par suite, pour chaque point p^L de E_L^d (resp. E_R^d), un point p^R de distance inférieure à d est nécessairement dans E_R^d (resp. E_L^d). Comme de tels points p^R sont à distance au plus d de p^L et au moins d les uns des autres, leur nombre est borné.
Pour chercher ces points, il convient de trier les points p de $E_L^d \cup E_R^d$ selon leur ordonnée p_y. On compare alors un point avec les points suivants tels que la différence d'ordonnées est au plus d. On obtient alors des carrés de côté d comme sur le schéma suivant :



Chaque carré est inclus complètement dans E_L^d ou dans E_R^d et est de côté d. Il ne peut donc pas y avoir de point à l'intérieur du carré. On peut donc placer 4 points dans chaque carré, soit 8 points au total donc 7 en excluant le point considérer.

Analyse de la complexité : La séparation est réalisée en temps $\Theta(n)$. Trouver les points dans la bande centrale peut être réalisé en temps $\Theta(n)$. Si l'on ordonne les points selon y dans un prétraitement, trouver la distance minimale dans la bande prend un temps $\Theta(n)$. On a donc une relation de récurrence de la forme $c_n = 2 * c_{n/2} + \Theta(n)$, ce qui donne une complexité en $\Theta(n \log n)$. Le prétraitement pour ordonner les points selon y est aussi réalisé en temps $\Theta(n \log n)$ donc la complexité totale est en $\Theta(n \log n)$.

Question 5: Implémenter la version "diviser pour régner" de l'algorithme en Python.

```python
def sortByX(points):
    """
    renvoie une copie de la liste des points triee par abscisse croissante
    """
    return sorted(points, key=lambda x : x[0])

def sortByY(points):
    """
    renvoie une copie de la liste des points triee par abscisse croissante
    """
    return sorted(points, key=lambda x : x[1])


def dminDivideAndConquer(points):
    """
    input :
        points liste de points
    output :
        distance minimale entre deux points
    """

    # copie triee par abscisses croissantes
    points_par_x = sortByX(points)

    # copie triee par ordonnees croissantes
    points_par_y = sortByY(points)

    return dminDivide(points_par_x, points_par_y)

def dminDivide(ptsX, ptsY):
    """
    input : liste de points
        ptsX liste des points triee selon X
        ptsY liste des points triee selon Y
    output :
        distance minimale entre deux points de la liste
    """

    # taille de la liste
    n = len(ptsX)

    # Brute Force sur les petites instances
    if n <= 3 :
        return dmin(ptsX)

    # Divide
    mid = n//2

    # abscisse du point median
    x_med = ptsX[mid][0]

    # separe selon X
    leftX   = ptsX[:mid]
    rightX  = ptsX[mid:]

    # separe selon Y
    leftY   = []
    rightY  = []
```

```python

    for point in ptsY:
        if point[0] < x_med :
            leftY.append(point)
        else :
            rightY.append(point)

    # appels recurifs
    d_1 = dminDivide(leftX,leftY)
    d_2 = dminDivide(rightX,rightY)

    min_d = min(d_1,d_2)

    # calcul de la distance minimale dans la bande du milieu.
    return dminMerge(ptsX, ptsY, x_med, min_d)

def dminMerge(ptsX, ptsY, x_med, d):
    """
    Calcul de la distance minimale dans la bande du milieu.
    Renvoie d si aucune distance plus petite n'est trouvee.
    """

    # calcul des points dans la bande autour du point median
    bande_milieu = [pt for pt in ptsY if abs(pt[0] - x_med) < d ]

    """"
    on aurait pu ecrire :
    bande_milieu = []
    for pt in ptsT:
        if  abs(pt[0] - x_med) < d :
            bande_milieu.append(pt)
    """

    min_d = d

    # calul de la distance minimale dans cette bande (au plus 7 points)
    n_bande = len(bande_milieu)
    for i in range(n_bande - 1):
        for j in range(i+1, min(n_bande, i+7)):
            cur_d = distance(bande_milieu[i], bande_milieu[j])
            if cur_d < min_d:
                min_d = cur_d

    return min_d

# resultat attendu avec une seed initialisee a 0
# 1.172871557467097
```

Pour information, la solution naïve proposée s'exécute en $\sim$ 22 secondes pour 10000 points et la solution "diviser pour régner" en $\sim$ 0.1 secondes (avec python 3.6.8 sur un processeur Intel Core i5-7200U CPU à une fréquence de 2.50GHz).