TP Python – Structures de données et algorithmique 1

Objectifs :
- Implémentation d'un **algorithme récursif** de type « **diviser pour régner** ».
- Calculs de **complexité temporelle**.
- Comparaison de temps de calcul avec la version **itérative**.

Sous-séquence de somme maximale

Le but de ce TP est de déterminer la sous-séquence $\langle l_i, l_{i+1}, \dots, l_j \rangle$ dont la somme est maximale dans une séquence l d'entiers relatifs qui contient au moins un nombre strictement positif. Il s'agit donc de trouver les indices i et j qui permettent de calculer :

$$\max_{i \leq j} \sum_{k=i}^{j} l_k$$

1. On va d'abord écrire un algorithme **itératif** pour réaliser ce calcul :
   - on initialise la meilleure somme *best* à 0 et les indices correspondants à `None` ;
   - pour tout indice i de l :
     - on initialise la somme s_{ij} à 0 ;
     - pour tout indice $j \geq i$ de l :
       - on ajoute l_j à s_{ij} ;
       - si s_{ij} est strictement supérieure à *best*, on met à jour *best* et les indices correspondants ;
   - on renvoie *best* et les indices correspondants.

   Écrire une fonction `maxsubsum1` **non récursive** qui prend la liste en paramètre et renvoie le triplet `(best, i, j)`. On testera le code sur la liste `l = [1,-2,1,2,-1,3,-2,1]` de meilleure somme 5 correspondant aux indices 2 à 5 (inclus).
2. Quelle est la **complexité temporelle** de cet algorithme en fonction de n ?
3. On souhaite améliorer la complexité de cette recherche en utilisant un algorithme **récursif** de type « **diviser pour régner** » selon le principe suivant :
   - à chaque étape, on découpe la liste en deux parties (presque) égales :
     - soit la meilleure somme se trouve dans la sous-liste de gauche ;
     - soit elle est dans celle de droite ;
     - soit elle est à cheval entre les deux ; dans ce cas, on peut simplement la déterminer de la manière suivante :
       - on calcule la meilleure somme à gauche en partant du milieu et en ajoutant des éléments d'indices décroissant ;
       - on calcule la meilleure somme à droite en partant du milieu et en ajoutant des éléments d'indices croissant ;
       - la meilleure somme passant par le milieu sera donc la somme des deux ;

— on arrête la récursion quand la liste n'a plus qu'un seul élément (cas trivial).

Pour éviter de consommer trop de mémoire et de temps à fabriquer des nouvelles listes à chaque étape, on ne va pas réellement « découper » la liste mais passer en paramètre de la fonction récursive les indices i et j de la tranche de liste à considérer : ainsi, on effectuera les appels récursifs avec i et k pour la tranche de gauche et $k+1$ et j pour celle de droite, avec $k = \frac{i+j}{2}$. On pourra ainsi arrêter la récursion lorsque $i = j$.

Pour écrire cette algorithme, on définira une **fonction locale** (i.e. dans le corps de la fonction principale) qui prendra en paramètre les indices i et j de la tranche à considérer et renverra le triplet correspondant à la meilleure séquence entre i et j. L'appel initial sera donc réalisé sur la tranche correspondant à l'intégralité de la liste. Votre code aura donc la structure suivante :

```
def maxsubsum2(l):

    def maxsubsum2_rec(i, j):
        [...]
        return ...

    return maxsubsum2_rec(0, len(l)-1)
```

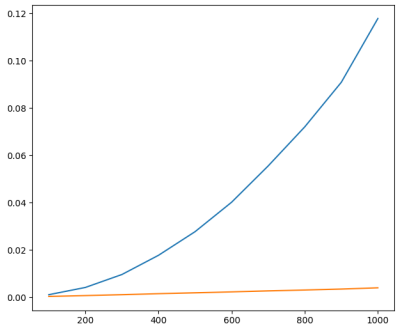
4. Écrivez en commentaires l'équation de récurrence de la complexité temporelle et la résoudre.
5. Pour vérifier les calculs de complexité, on va mesurer le temps de calcul des deux algorithmes pour des tailles de liste croissantes. Le module `time` permet d'utiliser la fonction `process_time` qui ne prend pas d'argument et renvoie le temps de processeur utilisé par le programme en secondes ; il suffit alors d'encadrer l'appel à l'algorithme par deux appels à `process_time`, puis d'en calculer la différence pour connaître son temps d'exécution.

   Pour générer des listes d'entiers aléatoires d'une taille donnée, on utilisera le module `random` qui fournit la fonction `randint(lb, ub)` renvoyant un entier compris entre `lb` et `ub` **inclus** :

   ```
   def genlist(n):
       return [random.randint(-n, n) for _ in range(n)]
   ```

   On peut ensuite tracer les courbes correspondantes avec le module `matplotlib.pyplot` :

   ```
   import matplotlib.pyplot as plt
   plt.plot(xs, ys1, xs, ys2)
   plt.show()
   ```

   avec `xs` la liste des abscisses et `ys1`, `ys2` les temps de calcul correspondants pour les deux algorithmes.
6. On peut remarquer que les deux algorithmes ne produisent pas en général la même sous-séquence maximale quand il y en a plusieurs de même somme. On souhaite à présent pouvoir déterminer la plus courte. Modifier le critère de comparaison optionnel `key` de la fonction `max` avec une **fonction anonyme** pour ajouter la longueur comme critère secondaire (à minimiser).