TP n°8 : Quelques algorithmes de tri

Introduction

Trier des données est une problématique très importante en informatique. Il existe de nombreux algorithmes de tri, certains livres y sont même entièrement consacrés. Il existe dèjà en Python, deux instructions prêtes à l'emploi qui permettent de trier des tableaux :

- la fonction `sorted` qui renvoie un nouveau tableau trié (le tableau pris en argument n'est pas modifié).

```
>>> t = [5,9,8,12,3]
>>> sorted(t)
[3, 5, 8, 9, 12]
>>> t # le tableau de départ n'est pas modifié
[5, 9, 8, 12, 3]
```

- la méthode `sort` qui modifie le tableau et le trie.

```
 >>> t.sort()
>>> t
[3, 5, 8, 9, 12] # le tableau de départ a été modifié
```

  Cette méthode présente l'avantage d'être économique en mémoire car on n'a pas créé de tableau supplémentaire, mais il faut avoir conscience que l'on a perdu définitivement l'ordre initial des éléments du tableau.

Dans ce TP, on se propose de découvrir trois algorithmes de tri et de déterminer leur complexité.

1 Tri par sélection du minimum

Notation : si t est un tableau, et $i \leqslant j$ des entiers naturels, on note $t[i : j]$ le sous-tableau constitué des éléments $t[i], t[i+1], \ldots, t[j-1]$ (la borne de droite n'est pas atteinte).

On désire maintenant écrire un algorithme de tri appelé **tri par sélection du minimum**.

Voici le principe (on peut si besoin consulter cette vidéo `https://www.youtube.com/watch?v=8u3Yq-5DTN8`) :

- étape 0 : on «sélectionne» le plus petit élément du tableau et on le permute avec $t[0]$ le premier élément du tableau. Le tableau $t[0 : 1]$ est ainsi trié et les éléments de $t[1 : n]$ sont supérieurs ou égaux à $t[0]$.
- étape 1 : on «sélectionne» le plus petit élément du sous-tableau $t[1 : n]$ et on le permute avec $t[1]$. Le tableau $t[0 : 2]$ est ainsi trié et les éléments de $t[2 : n]$ sont supérieurs ou égaux à $t[1]$. .

- ⋮
- étape k : on «sélectionne» le plus petit élément du sous-tableau $t[k:n]$ et on le permute avec $t[k]$. Le tableau $t[0:k+1]$ est ainsi trié et les éléments de $t[k+1:n]$ sont supérieurs ou égaux à $t[k]$.

1. Écrire une fonction `indice_min(t: [float], k: int) -> int` qui renvoie l'indice du mimimum de la partie du tableau $t[k:n]$ constitué des éléments de t d'indice $i \geqslant k$.

   Par exemple, si $t = [8, 5, 1, 12, 10]$, `indice_min(t, 0)` donne l'indice du minimum de t, donc 2 car le minimum est 1, élément d'indice 2. De même `indice_min(t, 3)` vaut 4, car 10 est le minimum parmi $t[3], t[4]$.

2. En déduire un fonction `tri_selection(t: [float]) -> None` qui trie en place la liste t (cette fonction ne renvoie rien, elle modifie la liste passée en argument). Proposer un invariant de boucle qui permette de justifier la validité de l'algorithme.

3. Déterminer le nombre de comparaisons et/ou le nombre d'échanges effectués lorsque la liste t est de longueur n. Indiquer le type de complexité.

2 Tri par insertion

Voici un deuxième algorithme de tri appelé «tri par insertion», utilisé de manière assez naturelle lorsque l'on trie des cartes à jouer.

4. Regarder la vidéo suivante qui explique le principe `https://www.youtube.com/watch?v=bRPHvWgc6YM`.

5. On prend dans cette question $t = [8, 5, 1, 12, 10]$ et donc $n = 5$. Tracer l'algorithme de tri par insertion lorsqu'on l'applique à t. On donnera quatre valeurs de t.

6. Compléter le code suivant qui implémente l'algorithme de tri par insertion :

```
def tri_insertion(t : [float])-> None:
    '''Données: t une liste de nombres.
     Cette fonction trie en place la liste t passée en argument'''
    n = len(t)
    for i in range(..., ...):
        # on va maintenir invariant que le tableau t[0:i] est triée
        # pour cela, on va insérer l'élément t[i] au bon endroit dans le tableau tr
        pivot = t[i]
        j = ....
        while ....  > pivot and j .....
            .......... # on décale la valeur vers la droite
            j =........
        # arrivé ici on place le pivot au bon endroit
        .... =  ....
```

7. Soit t une liste de longueur n. On exécute `tri_insertion(t)`. Donner en fonction de n, le nombre de comparaisons que l'on effectue dans le pire des cas. Et dans le meilleur des cas ?

3 Tri fusion

C'est un algorithme de tri efficace qui repose sur un paradigme de programmation très important qui s'appelle «diviser pour régner» («Divide and Conquer»). Il y a trois étapes :

- diviser (divide) : découper un problème initial en sous-problèmes
- conquérir (conquer) : résoudre les sous-problèmes (récursivement ou directement s'ils sont assez petits)
- combiner (combine) : calculer une solution au problème initial à partir des solutions des sous-problèmes.

8. Écrire une fonction `fusion(t1 : [float], t2: [float]) -> [float]` qui prend en argument deux listes triées par ordre croissant de longueurs n_1 et n_2 et renvoie une liste triée de longueur $n_1 + n_2$ contenant toutes les valeurs de `t1` et `t2`. On souhaite une complexité en $O(n_1 + n_2)$. Par exemple si `t1 = [3,27,38,43]` et `t2 = [9,10,82]`, `fusion(t1, t2)` renverra la liste `[3, 9, 10, 27, 38, 43, 82]`.

Voici le principe du tri fusion : on découpe la liste à trier en deux, on trie les moitiés gauche et droite séparément et récursivement, puis on fusionne ces deux moitiés.

9. Compléter le code suivant de la fonction récursive `tri_fusion(t)` qui trie la liste `t` passée en argument. Cette fonction utilisera la fonction auxiliaire `fusion(t1, t2)`.

```
def tri_fusion(t : [float]) -> None:
    if .... # cas de base
        return ...
    else:
        m = len(t)//2 # indice médian
        t1 = ..... #division
        t2 =  .....
        return .........
```

10. On suppose que la longueur n de la liste est une puissance de 2. Donner pour tout entier i, le nombre d'appels à la fonction fusion pour des listes de taille 2^i, puis donner la complexité totale de ces appels. En déduire que la complexité de ce tri est «quasi-linéaire».