Les algorithmes de tris

Table des matières

1) Pourquoi trier les valeurs ?

Les tris sont une des opérations les plus souvent effectuées par les ordinateurs, que ce soit dans les applications professionnelles (gestion des bases de données, des opérations bancaires...), la compression des données ou les jeux vidéo. On estime qu'un quart des cycles d'horloge des ordinateurs sont consacrés à des opérations de tri. Le coût énergétique de ces opérations de tris représente près de 2% de la consommation totale d'électricité en Europe. Sur des effectifs de plusieurs millions de données, les temps d'exécution des différents algorithmes de tris varient entre une dizaine de secondes et une dizaine de jours. D'où l'importance d'optimiser l'efficacité de ces algorithmes.

Bien que la plupart des algorithmes de tri soient connus depuis longtemps, la recherche sur leur optimisation est encore très active. La dernière amélioration de l'algorithme de tri "Quicksort" par Bentley & Sedgewick, considéré comme indépassable au niveau des tris par comparaison, date seulement de 2002.

Les algorithmes de tris se divisent en deux grandes catégories de complexité :

- les tris lents parmis lesquels on trouve : tri par insertion, le tri par sélection, le tri par bulles..

- les tris rapides parmis lesquels on dénombre : le tri fusion, le tri rapide (Quick Sort)...

2) Travail sur les listes

2.1) Génération d'une liste aléatoire

Écrire une fonction `getRandomList` qui prend en argument le nombre d'élément `nombreElement`,ainsi que la valeur minimale et maximale prise par ces éléments `min` et `max`. Cette fonction retournera une liste `lst`.

2.2) Recherche du plus petit élément dans une liste

Écrire une fonction `getMin` qui retourne le plus petit élément d'une liste `lst` passée en paramètre. La fonction retourne ce plus petit élément. On supposera que la liste ne contient que des entiers et qu'elle n'est pas vide.

Bien-sûr, on n'utilisera pas ici la méthode `min` de Python.

2.3) Recherche de la position de la plus petite valeur

Écrire une fonction `getIndexMin` qui cette fois-ci retourne la position de la plus petite valeur dans une liste passée en paramètre.

2.4) Recherche de la position de la plus petite valeur à partir d'une position donnée

Écrire une fonction `getIndexMinDepuis` qui recherche la position du plus petit élément d'une liste `lst` donnée en paramètre, à partir d'une position de depart donnée en paramètre. La fonction retourne la position.

On supposera que les paramètres sont «cohérents», à savoir que la liste n'est pas vide et que la position de début de recherche existe bien dans la liste.

2.5) Échange de deux cases dans une liste

Écrire une fonction `echanger` qui reçoit en paramètre une liste `lst` et deux entiers `i1` et `i2` représentant deux indices de cases dans cette liste. Cette fonction ne retourne rien et échange le contenu des cases correspondant aux indices `i1` et `i2`.

On supposera ici que les paramètres sont corrects (que les deux indices `i1` et `i2` ne sont pas en dehors de la liste `lst`).

3) Algorithmes de tri

3.1) Tri par sélection

a) Principe

L'idée est simple : rechercher le plus grand élément (ou le plus petit), le placer en fin de tableau (ou en début), recommencer avec le second plus grand (ou le second plus petit), le placer en avant-dernière position (ou en seconde position) et ainsi de suite jusqu'à avoir parcouru la totalité du tableau.

Cette décision est importante car à chaque fois qu'on déplace un élément en fin de tableau, on est certain qu'il n'aura plus à être déplacé jusqu'à la fin du tri.

Cf vidéo : https://youtu.be/Ns4TPTC8whw

b) Algorithme du tri par sélection

A l'aide des fonctions mises en place lors du chapitre 2, écrire la fonction `triSelection`.

3.2) Tri par insertion

a) Principe

Le tri par insertion est un algorithme de tri classique dont le principe est très simple. C'est le tri que la plupart des personnes utilisent naturellement pour trier des cartes : prendre les cartes mélangées une à une sur la table, et former une main en insérant chaque carte à sa place.

Cf vidéo : https://youtu.be/ROalU379l3U

b) Algorithme du tri par insertion

```
def triInsertion(lst: list) -> None:
    for i in range(1, len(lst)):
        j = i
        elmt = lst[i]
        while j > 0 and lst[j-1]>elmt:
            lst[j] = lst[j-1]
            j -= 1
        lst[j] = elmt
    return None
```

c) Test de la fonction `triInsertion`

Implémenter cette fonction et la tester.

3.3) Tri par fusion

a) Principe

Le tri fusion est construit suivant la stratégie "diviser pour régner", en anglais "divide and conquer". Le principe de base de la stratégie "diviser pour régner" est que pour résoudre un gros problème, il est souvent plus facile de le diviser en petits problèmes élémentaires. Une fois chaque petit problème résolu, il n'y a plus qu'à combiner les différentes solutions pour résoudre le problème global. La méthode "diviser pour régner" est tout à fait applicable au problème de tri : plutôt que de trier le tableau complet, il est préférable de trier deux sous tableaux de taille égale, puis de fusionner les résultats.

Cf vidéo : https://youtu.be/XaqR3G_NVoo

b) Algorithme du tri par fusion

```
 # fusion de deux tableaux lst1 et lst2
def fusion(lst1:list,lst2:list)-> list:
    i1,i2,n1,n2=0,0,len(lst1),len(lst2)
    t=[]
    while i1<n1 and i2<n2:
```

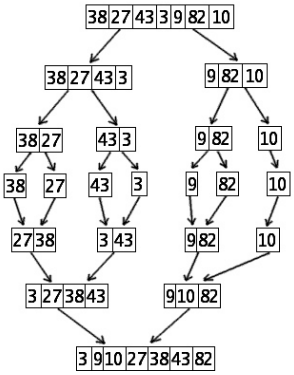

FIGURE 1 – tri fusion appliqué à un tableau de 7 éléments

```
        if lst1[i1]<lst2[i2]:
            t.append(lst1[i1])
            i1+=1
        else:
            t.append(lst2[i2])
            i2+=1
    if i1==n1:
        t.extend(lst2[i2:])
    else:
        t.extend(lst1[i1:])
    return t


#fonction du tri par fusion
def triFusion(lst:list)-> list:
    n=len(lst)
    if n<2:
        return lst
    else:
        m=n//2

        return fusion(triFusion(lst[:m]),triFusion(lst[m:]))
```

c) Test de la fonction `triFusion`

Implémenter cette fonction et la tester.

4) Comparaison entre les différents algorithmes de tri

Nous allons chronométrer les fonctions de tri une à une sur un même tableau. Pour réaliser le chronométrage, nous allons utiliser la fonction `timeit`. En effet, en général le processeur n'exécute pas qu'une seule tâche , ce qui rend la programmation de cette fonction très complexe. Cette instruction fonctionne ainsi :

```
import timeit
timeit.timeit(stmt="1",setup="from __main__ import (2)",number=3)
```

Avec :

- 1 : instructions à chronométrer avec des arguments
- 2 : fonctions ou arguments à prendre en compte au moment de l'exécution (séparés par une virgule)
- 3 : nombre de fois où l'intruction est à repéter

Exemple :pour calculer le temps mis par le tri par selection pour ordonner le tableau lst

```
temps=timeit.timeit(stmt="triSelection(lst.copy())",
                    setup="from __main__ import (triSelection,lst)",
                    number=1)
```

Pour réaliser cette comparaison :

- Créer une liste à l'aide de la fonction getRandomList(...). Les valeurs seront prises dans l'intervalle [-1000, 1000].
- Pensez à faire une copie de la liste à chaque fois si vous enchaînez les algorithmes (pour ne pas trier une liste qui serait déjà triée).
- Compléter le tableau ci-dessous :

Algorithme de tris	10 éléments temps (μs)	50 éléments temps (μs)	100 éléments temps (μs)	1000 élément temps (μs)
tri par sélection				
tri par insertion				
tri par fusion				

Exercice 1 :

réaliser grâce à `matplotlib` un graphique ayant comme ordonnée le nombre d'élément dans le tableau et comme abcisse le temps mis par les différents algorithmes.

Exercice 2 :

Évaluer la complexité de ces trois algorithmes