TP N°12 : Les méthodes de tri

Les méthodes de tri sont évidemment fondamentales. On dispose de nombreux algorithmes qui se distinguent par leurs complexités en temps et en place mémoire. Nous allons illustrer quelques-uns de ces algorithmes avec des listes (tableaux) d'entiers (ou de flottants).

En Python, on peut trier une liste à l'aide de la méthode *sort*. Si `a` est une liste d'entiers ou de flottants, `a.sort()` modifie la liste en la liste triée. Une fois triée, elle peut servir pour d'autres traitements comme :

— L'insertion d'une nouvelle donnée à sa bonne position en prenant en considération l'ordre établi ;

— Les algorithmes dichotomiques qui utilisent les listes triées pour accélérer la recherche ...

On distingue deux grandes catégories de tri :

1. Les *Tris lents* (tri par sélection, à bulles, par insertion...)
2. Les *Tris rapides* (tri par fusion, quicksort,...)

1 Un premier exemple de tri simple : le tri à bulles

Le principe de l'algorithme de *tri à bulles* est de faire « remonter » le plus grand élément à l'extrémité droite de la liste (comme une bulle dans un verre d'eau), puis de recommencer sur le reste de la liste.
L'algorithme parcourt la liste, et compare les éléments successifs. Lorsque deux éléments successifs ne sont pas dans le bon ordre, ils sont échangés. Après chaque parcours complet de la liste, l'algorithme recommence l'opération. On donne une version en pseudo-code de cet algorithme :

Algorithme 1 : tri_bulle(liste)

```
n ← taille(liste) ;
pour i=0 à n-2 faire
    i = n-i-1                          ▷ ainsi, i va varier de n − 1 à 1 ;
    pour j=0 à i-1 faire
        si liste[j] > liste[j+1] alors
            Échanger liste[ j ] et liste[ j+1 ] ;
        sinon
    fin
fin
```

Exercice 1

1. Implémenter le pseudo-code précédent en python et vérifier son fonctionnement.
2. Mesurer le temps d'exécution pour des listes d'éléments tirés de tailles comprises entre 100 et 5000, et variant par pas de 100. Récupérer les tailles et les temps d'exécution dans deux listes et afficher le graphe du temps d'exécution en fonction de la taille de la liste à trier.

2 Tri par sélection

Le *tri par sélection* est un algorithme de tri par comparaison. Le principe de cet algorithme est le suivant :

— Commencer à chercher l'indice du minimum de la liste `L` ;

— Permuter le 1er élément de la liste avec l'élément minimum trouvé ;

— Chercher l'indice du minimum des éléments de la sous liste `L[1 :]` et le permuter avec le 2ème élément ;

— Continuer ce principe jusqu'à ce que la liste devienne triée.

D'une manière générale, on échange l'élément à la position `i` avec le minimum de la sous liste `L[i+1 :]`.
Exemple : L=[4,5,1,-6,2]

- Etape N°1 : Le minimum de `L` est -6, on le permute avec 4 : L=[-6,5,1,4,2] ;
- Etape N°2 : Le minimum de [5,1,4,2] est 1, on le permute avec 5 : L=[-6,1, 5,4,2]
- Etape N°3 : Le minimum de [5,4,2] est 2, on le permute avec 5 : L=[-6,1,2,4,5]
- Etape N°4 : Le minimum de [4,5] est 4, il est à sa bonne place. L est bien triée.

Exercice 2

1. Créer une fonction `tri_selection(L)` qui trie la liste `L` selon le principe de tri par sélection.
2. Modifier la fonction `tri_selection(L)`, pour qu'elle accepte les deux sens de tri (croissant et décroissant).
3. Créer une version récursive de la fonction `tri_selection(L)`.

3 Tri par insertion

Dans le *tri par insertion*, on place chaque nouvel arrivant dans le tableau à sa position, dans une zone déjà triée selon le principe suivant :

— On considère une liste ou un tableau dont les premiers éléments d'indices $0,1,\cdots,i-1$ sont déjà triés ;

— On cherche à placer l'élément d'indice i à sa bonne place parmi les éléments déjà triés, pour cela on procède à le comparer aux précédents jusqu'à ce que la place de $T[i]$ soit trouvée.

Exercice 3

1. Écrire une fonction `position_insertion(L,i)` prenant en entrée une liste `L` et un indice `i` , tel que $i \in [1,len(L)]$ et retournant la position où on doit insérer `L[i]` dans `L[:i]` supposée déjà triée.
2. Créer une fonction `tri_insertion(L)` qui trie la liste `L` selon le principe de tri par insertion.
3. Quelle est la complexité de ce type de tri ?

4 Tri rapide (quicksort)

Le *tri rapide* est un algorithme de tri fondé sur la méthode de conception *diviser pour régner*. Pour trier une liste `L`, le principe de ce tri est le suivant :

— Si `L` est vide ou admet un seul élément, elle est triée (c'est notre critère d'arrêt) ;

— Sinon, On choisit un élément `e` $\in$ `L`, quelconque, que l'on retire de `L` ;

— On construit deux sous listes :
- `Li` formée des éléments inférieurs à `e` ;
- `Ls` formée des éléments plus grands que `e` ;

— Le résultat est la concaténation de `Li` récursivement triée, de `[e]`, et de `Ls` qui est aussi récursivement triée.

Exercice 4

1. Créer une fonction `tri_rapide(L)` qui trie la liste `L` selon le principe de tri rapide.
2. Quelle est la complexité de ce type de tri ?

Exercice 5

On veut trier une liste d'une manière décroissante en utilisant le principe de *tri par comptage*. On va supposer que la liste ne contient que des éléments distincts. L'algorithme compte pour chaque élément de la liste `L` le nombre d'éléments qui lui sont supérieurs. On utilise une deuxième liste `A` pour insérer les éléments de `L` chacun à sa bonne position comme le montre l'exemple suivant.
Soit `L` la liste à trier : `L=[4,3,6]`

— Pour 4 : il a 1 seul élément qui lui est supérieur ; il sera inséré à la position 1

— Pour 3 : il a 2 éléments qui lui sont supérieurs ; il sera inséré à la position 2

— Pour 6 : il a 0 élément qui lui est supérieur ; il sera inséré à la position 0

— La fonction retourne la valeur de la liste `A = [6,4,3]`.

1. Écrire une fonction `remplir_distinct(L,n)` qui remplit une liste avec n entiers tous distincts.
2. Écrire une fonction `compter(L,x)` qui compte le nombre d'éléments d'une liste `L` qui sont supérieurs strictement à un entier `x` passé en paramètre.
3. Écrire une fonction `tri_comptage(L)` qui trie une liste `L` en se basant sur le principe décrit en haut.