TP Informatique n° 4
Algorithmes de tri

I. Tri fusion et tri rapide

I.1. Diviser pour régner

On emploie le terme **diviser pour régner** pour désigner l'approche algorithmique consistant à :

1) **Diviser** : découper un problème initial en sous-problème.

2) **Régner** : résoudre les sous-problèmes.
(de manière récursive -*cf* TP précédent- ou directement s'ils sont assez petits)

3) **Combiner** : la solution du problème initiale est obtenue en combinant les solutions des sous-problèmes.

I.2. Tri fusion

Le tri fusion est un algorithme récursif basé sur le principe « diviser pour régner ».

Étant donnée une liste L, on procède comme suit.

1) **Diviser** : la liste L est séparée en 2 (au milieu), à chaque appel.

2) **Régner** : chaque sous-liste est alors triée récursivement en utilisant le principe de découpage précédent. Cas initial : une liste qui ne contient qu'un élément est triée.

3) **Combiner** : les deux sous-listes triées sont alors combinées de sorte à ce que la liste résultat soit triée. Cette étape est appelée **fusion**.

▶ Implémenter la fonction `fusion` qui prend en paramètre deux listes `L1` et `L2` triées et renvoie la liste `L` qui réalise la fusion des deux listes précédentes.

- Implémenter la fonction `tri_fusion` qui prend en paramètre une liste `L`, trie la liste selon le principe du tri fusion et renvoie la liste obtenue.

- Nous avons implémenté la fonction `fusion` de manière itérative. Écrire une fonction `fusionRec` permettant de réaliser la fusion de deux listes triées `L1` et `L2` de manière récursive.

Remarque
On peut démontrer que ce tri est optimal en terme de comparaisons. Il nécéssite de l'ordre de $n\log(n)$ comparaisons/affectations pour trier une liste de n nombres.

I.3. Tri rapide

Le tri rapide est un algorithme récursif lui aussi basé sur le principe « diviser pour régner ».

Étant donnée une liste L, on procède comme suit.

1) **Diviser** : on choisit le 1^{er} élément de L comme *pivot* permettant de partitionner L en deux sous-listes.
   - La $1^{\text{ère}}$ contient des éléments inférieurs ou égaux au pivot.
   - La $2^{\text{ème}}$ contient des éléments supérieurs (strictement) au pivot.

2) **Régner** : on lance l'algorithme du tri rapide sur chacune des deux sous-listes de sorte à obtenir récursivement deux sous-listes triées.

3) **Combiner** : le pivot est replacé au bon endroit

Remarque
L'algorithme du tri rapide peut être implémenté en place. Ceci signifie que l'algorithme permet de trier une liste en paramètre en modifiant directement cette liste et sans avoir besoin d'en créer une autre. Nous nous intéressons à cette implémentation.

- Écrire une fonction `echange(L,i,j)` qui échange les éléments d'indices `i` et `j` de la liste `L`.

- Écrire une fonction `partition(L, g, d)` qui considère la liste `L[g : d+1]` et effectue la partition sur cette sous-liste. Plus précisément, les éléments de cette sous-liste devront être modifiés de sorte à ce que :
  - × les premiers éléments soient inférieurs ou égaux à `L[g]`, pivot de cette partition.
  - × les éléments suivants soient strictement supérieurs à `L[g]`.

  Le pivot sera placé au bon endroit et on renverra la position finale de cet élément.
  (indication : on pourra utiliser un compteur `i` *parcourant la liste de gauche à droite et un compteur* `j` *parcourant la liste de droite à gauche)*

- Écrire alors une fonction récursive `tri_rapide(L, g, d)`.

Remarque
La complexité du tri rapide est en moyenne (et à une constante près) de l'ordre de $n\log(n)$ comparaisons/ affectations, et de n^2 comparaisons/affectations dans le pire cas. On notera qu'il est possible de démontrer que l'ordre de grandeur optimal d'un tri est de $n\log(n)$ comparaisons.

II. D'autres algorithmes de tris

II.1. Tri par insertion

C'est la forme de tri que l'on utilise naturellement pour trier sa main lorsque l'on joue aux cartes. L'idée est de trier sa main de gauche à droite en insérant successivement au bon endroit une carte de droite dans la partie de gauche. Plus précisément, on procède comme suit.

- Initialement, la partie gauche contient 1 élément (elle est donc triée) : la carte la plus à gauche.
- La carte suivante de la main est alors placée soit avant la carte précédente, soit après.
- On procède ainsi de suite de sorte qu'au début de la $i^{\text{ème}}$ étape, les i cartes les plus à gauche sont classées dans l'ordre croissant. On place alors la $i^{\text{ème}}$ carte au bon endroit dans cette partie de gauche classée.
- La dernière étape consiste à placer la carte la plus à droite parmi les $n-1$ cartes les plus à gauches qui sont alors classées correctement.

Ce tri peut être codé comme suit.

```
1  def tri_insertion(L):
2      n = len(L)
3      for i in range(1,n):
4          aux = L[i]
5          j = i
6          while (j>0 and L[j-1]>aux):
7              L[j] = L[j-1]
8              j = j-1
9          L[j] = aux
```

▶ Trier à la main la liste `[2, 5, 3, 7, 1]` à l'aide de cet algorithme.

▶ Quelle est la complexité dans le pire cas de cet algorithme ?

▶ Que renvoie cette fonction ?

II.2. Tri à bulles

- Cet algorithme est basé sur un mécanisme de remontée permettant, par échange de deux valeurs successives, de placer le plus grand élément en fin de liste.
  Plus précisément, la remontée s'effectue comme suit.

  On considère a le premier élément de la liste,
  - × s'il est plus grand que l'élément b suivant, on échange ces deux éléments, et on continue la remontée avec a.
  - × sinon, on n'éffectue pas d'échange et on continue la remontée avec b.
- L'algorithme consiste alors à effectuer n remontées successives.
- Ce tri tire son nom de ce mécanisme de remontée que l'on peut rapprocher du parcours d'une bulle qui remonte progressivement à la surface d'un verre contenant une boisson gazeuse.

▶ Trier à la main la liste `[2, 5, 3, 7, 1]` à l'aide de cet algorithme.

On propose alors le code suivant suivant.

```
1  def tri_bulles(L):
2      n = len(L)
3      for i in range(n-1,0,-1):
4          nb_echange = 0
5          for j in range(i):
6              if L[j] > L[j+1]:
7                  echange(L, j, j+1)
8                  nb_echange = nb_echange + 1
9          if nb_echange == 0:
10             break
```

- Combien de remontées successives doit-on faire ? On détaillera notamment la ligne 3.

- Que réalise la boucle en ligne 5 ?

- Quel est l'intérêt de l'instruction `break` dans ce code ?
  Quel est l'intérêt de cette instruction de manière générale ?

- Quel est la complexité dans le pire cas de cet algorithme ?