TD : Les tris

Les algorithme de tri tiennent une place prépondérante en Informatique. Ils permettent d'organiser une collection d'objets selon un ordre déterminé tel que l'ordre numérique (pour les entiers et les flottants) ou l'ordre lexicographique (l'ordre du dictionnaire, pour les chaînes de caractères). Ici on cherche à trier dans l'ordre croissant des listes de flottants ou de chaînes de caractères.

1 — Préliminaire

Ex. 1 — Intérêt du tri Soit `L` une liste triée dans l'ordre croissant.

1. Proposer un algorithme pour donner le plus grand élément de la liste, puis le plus petit élément de la liste.
2. Proposer un algorithme pour tester si une valeur `x` est comprise dans la liste. Cet algorithme devra avoir une complexité nettement inférieure à $O(n)$.

Ex. 2 — Deux fonctions utiles

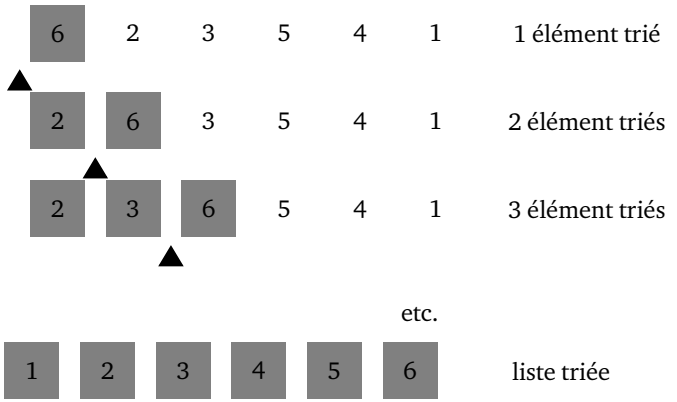
1. Écrire une fonction `random_list(n)` qui renvoie une liste de n entiers tirés au hasard entre 0 et $10 \times n$.
   On utilisera cette fonction dans toute la suite du TD pour générer des listes à trier.
2. Écrire une fonction d'en-tête `test_tri(L)` qui prend en paramètre une liste `L` et qui renvoie `True` si la liste est triée dans l'ordre croissant, `False` sinon.
   On utilisera cette fonction dans toute la suite du TD pour vérifier les résultats.

2 — Tri par insertion

C'est la méthode spontanément utilisée lors d'un tri manuel.

On considère d'abord la liste constituée par le premier élément : elle est triée. Puis on prend le second élément que l'on place par rapport au premier. On obtient ainsi une liste triée de deux éléments. De manière générale, lorsqu'on considère le k–ième élément, les précédents ayant déjà été triés on compare cet élément aux éléments de la liste d'indice compris entre 0 et $k-1$, dans l'ordre croissant, jusqu'à trouver sa place et on l'insère l'élément à cette place. On passe ensuite à l'élément suivant et on s'arrête après avoir placé le dernier élément de la liste. La liste obtenue est bien triée dans l'ordre croissant.

On va s'imposer une contrainte supplémentaire : trier sans utiliser de liste annexe (tir « en place »), afin de limiter l'empreinte mémoire du programme.



Ex. 3 — Tri par insertion

1. Écrire une fonction `insere(L,i,j)` qui déplace et insère l'élément d'indice `i` de la liste `L` à la position `j`.
2. Écrire une fonction `cherche_place(L,k)` qui renvoie la position à laquelle on va insérer l'élément `L[k]` dans la liste `L` entre les positions `0` et `k-1`.
3. Utiliser ces deux fonctions dans une fonction `tri_insertion(L)` qui trie la liste `L` selon la méthode du tri par insertion et renvoie la liste `L` ainsi triée.
4. Proposer des optimisations de ce programme.

★ **Ex. 4** — Complexité du tri par insertion On considère une liste de longueur n à trier. Les opérations considérées ici sont les comparaisons entre deux éléments.

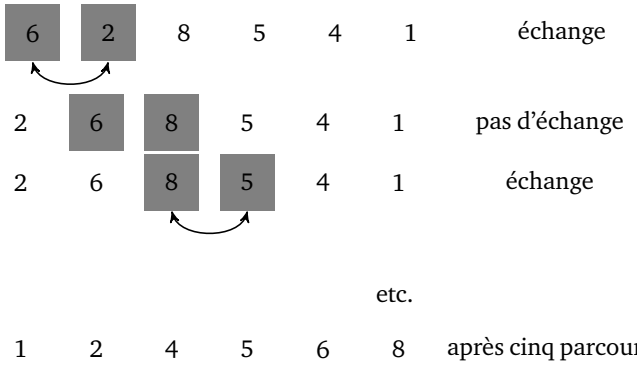
1. Dans quel cas va-t-on faire le moins d'opérations ? le plus d'opérations ?
2. a) Lorsqu'on place l'élément d'indice k, quel est, dans le pire des cas le nombre de comparaisons effectuées ?
   b) En déduire, dans le pire des cas, le nombre total de comparaisons effectuées.

3. On suppose que les éléments sont distincts et que toutes les permutations sont équiprobables.
   a) On place l'élément d'indice k. En utilisant un argument de symétrie, donner la loi de la variable aléatoire C_k « nombre de comparaisons effectuées ».
   b) En déduire le nombre moyen de comparaisons lorsqu'on place l'élément d'indice k, puis au total.

3 — Tri à bulles

Avec la méthode du tri à bulles, on parcourt la liste `L` de gauche à droite. Dès qu'on repère deux éléments consécutifs `L[i]` et `L[i+1]` dans le « mauvais » ordre (c'est-à-dire avec `L[i] > L[i+1]`) on échange leurs valeurs. Puis on continue le parcours de la liste.

Si on a parcouru la liste sans faire d'échanges, c'est qu'elle est triée : on arrête alors le tri.

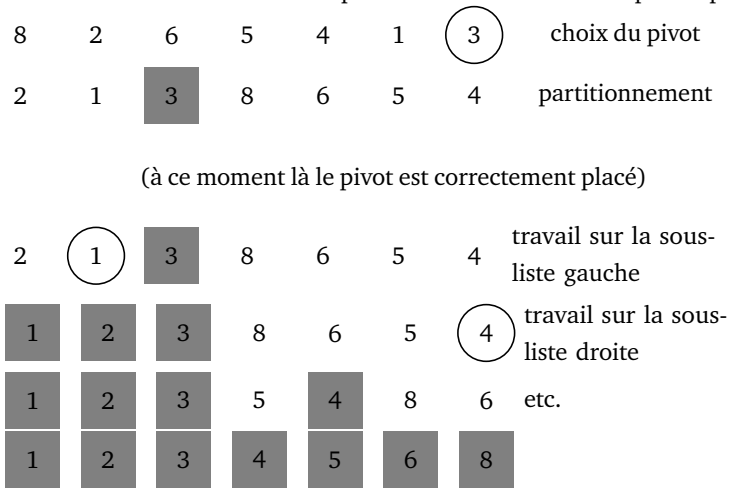


Ex. 5 — Tri à bulles Programmer une fonction `tri_bulles(L)` qui trie la liste `L` selon la méthode du tri à bulles et renvoie la liste `L` ainsi triée.

4 — Tri rapide

L'algorithme « QuickSort » (« tri rapide ») est un algorithme très efficace du type « diviser pour régner ». On choisit d'abord un élément pivot, par exemple le dernier élément de la liste [1]. Puis on *partitionne* la liste : cela consiste à fabriquer une liste comprenant tous les éléments plus petits que le pivot, puis le pivot et enfin tous les éléments plus grand que le pivot.

Le pivot est alors à sa place finale : on trouve avant lui tous les éléments plus petits, et après lui tous les éléments plus grands. On recommence ensuite ce travail sur deux sous-listes : celle des éléments avant le pivot et celle des éléments après le pivot.



Ex. 6 — Tri rapide

1. Écrire une fonction `partitionne(L, a, b)` qui partitionne une liste `L` entre les indices a et $b-1$ (le pivot est donc `L[b-1]`.
   La fonction `partitionne` renvoie la liste modifiée ainsi que l'indice finale du pivot.
2. En utilisant la fonction précédente, écrire une fonction `tri_rapide(L)` qui trie la liste `L` selon la méthode du tri rapide et renvoie la liste `L` ainsi triée.
   Pour cela, on fera une liste des indices de départ et d'arrivée des sous-listes à trier. Cette liste est donc initialisée à `[0, len(L)]`. On l'utilise ainsi : on enlève les deux derniers éléments de la liste, qui sont donc deux bornes `a` et `b`. Si $a-b<1$, on ne fait rien. Sinon on partitionne `L` entre `a` et `b` et on récupère la place du pivot. On ajoute ensuite `a, place_pivot` et `place_pivot+1, b` à la liste des bornes.
   Et on recommence... jusqu'à quand ?

1. il existe des méthodes pour optimiser le choix du pivot