Outre l'intérêt premier du tri de valeurs (faire un classement, afficher par date croissante, trier les résultats par pertinence lors d'une recherche sur un moteur de recherche, etc...), le fait de trier des données permet d'accélérer grandement la recherche d'élément dans une structure de données, comme on a pu le voir avec la recherche dichotomique.

Un tri peut se faire par ordre croissant, ordre alphabétique ou d'une manière générale sur toute relation d'ordre totale. Les données triées contiennent en général bien plus que juste la « clé » qui sert à effectuer le tri. Par exemple, on peut imaginer un ensemble dont les éléments sont des triplets (nom, date de naissance, numéro de sécurité sociale). Un tri peut s'effectuer sur chacune de ses trois informations, mais l'algorithme utilisé pourra très bien être le même (à la relation d'ordre près). Par conséquent, nous simplifierons le modèle à l'étude de tri sur des valeurs numériques uniquement.

1 Préliminaires

1. En utilisant la fonction `randint` du module `random`, écrire une fonction `liste_alea(n)` qui crée une liste aléatoire de taille `n` dont les éléments sont choisis uniformément entre `0` et `1000`.
2. Écrire une fonction `swap(L, i, j)` qui inverse de positions dans la liste `L` les éléments `L[i]` et `L[j]`, sans rien renvoyer.

On rappelle également la fonction de chronomètre du temps d'exécution :

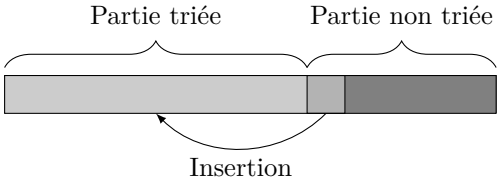
```
from time import perf_counter
def chronometre(f, x):
    debut = perf_counter()
    f(x)
    fin = perf_counter()
    return fin - debut
```

2 Tris quadratiques

2.1 Tri par insertion

Pour imaginer le fonctionnement du tri par insertion (*insertion sort*), imaginons un joueur de jeu de cartes, à qui les cartes sont données petit à petit. Pour gagner du temps, le joueur trie ses 3 premières cartes, puis rajoute les suivantes dans la partie de son jeu qui est déjà triée. Il lui suffit alors de chercher l'emplacement pour rajouter une nouvelle carte, et de l'y insérer.

Schématiquement, chaque étape du tri ressemble à :



Exercice 1

On cherche dans un premier temps à écrire une fonction qui construit une nouvelle copie triée d'une liste donnée en entrée. On rappelle les opérations suivantes sur les listes :

- si `L` est une liste et `i` et `j` sont des indices compris entre 0 et `len(L)` inclus, alors `L[i:j]` renvoie une **copie** contenant les éléments `[L[i], L[i+1], ..., L[j-1]]` ;
- si `L1` et `L2` sont deux listes, alors `L1 + L2` renvoie une **copie** qui est la concaténation des listes `L1` et `L2`.

1. Écrire une fonction `inserer(L, x)` qui prend en argument une liste `L` **supposée triée** et renvoie

une copie triée contenant les éléments de `L` et `x`, c'est-à-dire où `x` a été inséré au bon endroit dans `L`.

2. En déduire une fonction `tri_insertion(L)` qui prend en argument une liste `L` quelconque et renvoie une copie triée de `L`.

Il est possible d'implémenter le tri par insertion sans jamais créer de copies de sous-listes. C'est ce qu'on appelle un algorithme **en place**. Pour ce faire, le principe est le suivant. Si `L` est une liste dont les éléments entre les indices 0 et $i-1$ inclus sont supposés triés, alors on peut « insérer » l'élément `L[i + 1]` en le comparant avec `L[i]` :

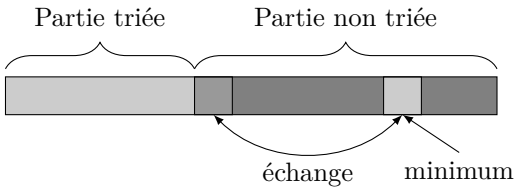
- si `L[i] > L[i - 1]`, alors la liste `L` est en fait triée jusqu'à l'indice i et il n'y a rien à faire (le calcul s'arrête) ;
- sinon, on inverse de positions `L[i]` et `L[i - 1]` et on recommence en comparant la nouvelle valeur `L[i - 1]` et `L[i - 2]` ;
- on s'arrête si on atteint le début de la liste.

3. Écrire une fonction `inserer_bis(L, i)` qui prend en argument une liste `L` et un indice `i` tels que les éléments de `L` entre les indices 0 et $i-1$ inclus sont supposés triés, et insère l'élément `L[i]` à la bonne position en procédant par permutations successives. La fonction `inserer_bis` utilisera la fonction `swap` et ne renverra rien.
4. En déduire une fonction `tri_insertion_bis(L)` qui trie une liste sans jamais créer de copie.
5. Comparer les temps d'exécution des deux versions de l'algorithme pour des listes de tailles 1000 et 10000 :
   - quelconques ;
   - triées ;
   - triées par ordre décroissant. (on pourra utiliser la commande `L.sort(reverse = True)` pour trier une liste par ordre décroissant).

2.2 Tri par sélection

Pour comprendre le fonctionnement de l'algorithme de tri par sélection (*selection sort*), imaginons une photo de classe. Le photographe veut que tout le monde soit bien visible, et place chaque personne une par une. Naturellement, il commence par les plus petits, qu'il place devant, et à chaque étape, il choisit le prochain élève le plus petit pour le placer.

Schématiquement, chaque étape du tri ressemble à :



Exercice 2

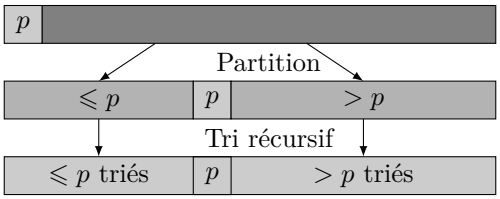
1. Écrire une fonction `minimum(L)` qui détermine l'indice de l'élément minimal dans la liste `L`.
2. En déduire une fonction `tri_selection(L)`. (On autorisera les tranches de listes et la concaténation).
3. Écrire une version **en place** du tri par sélection. On pourra pour cela écrire une fonction qui renvoie l'élément minimal dans une liste à partir d'un certain indice.

3 Tris améliorés

3.1 Tri rapide

Le tri rapide (*quick sort*) utilise le paradigme diviser pour régner. Le principe est le suivant : on désigne un élément de la liste comme étant un *pivot*. On partitionne la liste en trois parties : les éléments inférieurs au pivot, le pivot lui-même, et les éléments supérieurs. On trie ensuite les deux parties contenant les éléments non-pivot récursivement, qu'on concatène pour obtenir une liste triée. Par défaut, le pivot sera le premier élément de la liste.

Schématiquement, on a :



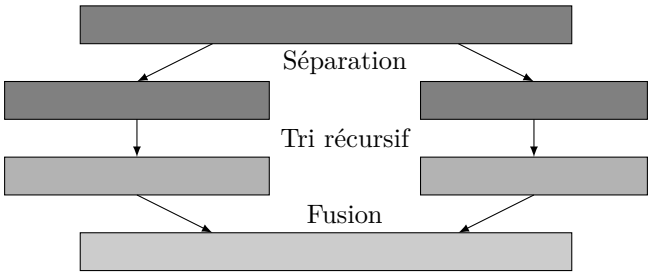
Exercice 3

1. Écrire une fonction `partition(L)` qui prend en argument une liste non vide `L` et renvoie un triplet `L1, p, L2` telle que `p = L[0]`, `L1` est la liste des éléments de `L` inférieurs ou égaux à `p` (sauf `L[0]`) et `L2` est la liste des éléments de `L` strictement supérieurs à `p`.
2. En déduire une fonction récursive `tri_rapide(L)` qui trie une liste selon le principe du tri rapide (on partitionne et on trie chaque moitié récursivement avant de recoller les morceaux).
3. Déterminer les temps d'exécution du tri rapide sur des listes de taille 500 :
   - quelconques ;
   - triées ;
   - triées par ordre décroissant.
4. Quel problème rencontre-t-on si on effectue ces tests pour le tri rapide pour des listes de taille 5000 ?

3.2 Tri fusion

Pour illustrer le principe du tri fusion (*merge sort*), imaginons qu'un enseignant doive trier un tas de copies par ordre alphabétique. Chance pour lui, deux élèves sont à la traîne pour quitter la salle de classe, et il les sollicite pour ce travail. Il donne une moitié du tas à l'un, l'autre moitié à l'autre, et leur demande de trier chacun sa partie. Une fois les deux parties triées, il est « facile » de recomposer le tas final en comparant à chaque fois le plus petit des deux. On peut imaginer un procédé similaire avec quatre élèves disponibles, et ainsi de suite...

Schématiquement, le principe est le suivant :



Exercice 4

1. Écrire une fonction `separation(L)` qui prend en argument une liste `L` et renvoie un couple de liste `L1, L2`, dont les tailles diffèrent d'au plus 1, correspondant respectivement à la première et la deuxième moitié de la liste `L`.
2. Écrire une fonction `fusion(L1, L2)` qui prend en arguments deux listes **supposées triées** et renvoie une copie triée contenant les éléments des deux listes. On utilisera deux indices `i1` et `i2` qui progressent dans les deux listes en repérant à chaque instant le plus petit élément qui n'a pas encore été rajouté à la nouvelle liste.
3. En déduire une fonction récursive `tri_fusion(L)` qui trie une liste selon le principe du tri fusion.
4. Déterminer les temps d'exécution du tri rapide sur des listes de taille 5000 :
   - quelconques ;
   - triées ;
   - triées par ordre décroissant.

   Que remarque-t-on ?

4 Tri de comptage

Les tris précédemment étudiés ne permettent pas d'avoir de complexité dans le pire des cas (et moyenne) inférieure à $\mathcal{O}(n \log n)$ pour une liste de taille n. On peut montrer que c'est le cas pour tout tri qui fonctionne en comparant les éléments entre eux. Cependant, avec des hypothèses supplémentaires sur les données à trier, il est possible d'effectuer le tri sans les comparer. On peut alors espérer obtenir des complexités de l'ordre de $\mathcal{O}(n)$.

Le tri de comptage est le plus intuitif des tris linéaires. On suppose pour cela que la liste à trier contient n éléments compris entre 0 et p compris ($p \in \mathbb{N}$). On se contente alors de compter pour k entre 0 et p le nombre d'éléments de la liste égaux à k, puis de reconstituer la liste.

Exercice 5

1. Écrire une fonction `tri_comptage(L)` qui prend en argument une liste et le trie avec cet algorithme. On commencera par déterminer l'intervalle des valeurs et on utilisera un tableau intermédiaire pour compter les éléments.
2. Déterminer la complexité de cet algorithme.

5 Variantes des tris quadratiques

5.1 Tri à bulles (*bubble sort*)

Le principe du tri à bulles est le suivant : on parcourt la liste un certain nombre de fois, et chaque fois qu'on trouve deux éléments **consécutifs** qui sont dans le mauvais ordre, on les permute. On arrête les parcours dès que la liste est triée. Les « bulles » (ou lièvres) sont alors les grands éléments de la liste, qui sont les plus rapide à remonter jusqu'à leur bonne position. Les « tortues » sont les petits éléments qui nécessitent plusieurs passages pour arriver à la bonne position.

Exercice 6

3. Écrire une fonction `tri_bulles(L)` qui effectue un tri à bulles.
4. Tester le temps de calcul de la fonction pour différentes tailles de liste `L`, et en déduire la complexité en fonction du nombre d'éléments de `L`.

5.2 Tri de Shell

En 1959, Shell propose d'améliorer le tri par insertion en remarquant deux choses :

- Le tri par insertion est efficace si la liste est presque triée.
- Il est inefficace en moyenne car on n'effectue de permutations que sur des éléments adjacents.

Sa version de tri d'une liste L est alors la suivante :

(a) on choisit un « gap » g ;

(b) on trie les sous-listes de L des éléments espacés du gap (c'est-à-dire on trie d'abord les éléments d'indices 0, g, $2g$, ..., puis les éléments d'indices 1, $g+1$, $2g+1$, ..., jusqu'aux éléments d'indices $g-1$, $2g-1$, ...) ;

(c) on diminue le gap et on recommence jusqu'à $g = 1$.

L'intérêt de commencer avec un gap élevé résout le problème de l'inefficacité en moyenne. L'intérêt de terminer par $g = 1$ est que la liste est presque triée.

Exercice 7

1. Écrire une fonction `tri_insertion_gap(L, debut, g)` qui trie **en place** les éléments de la liste d'indices `debut`, `debut + g`, ..., sans toucher aux autres éléments de la liste.
2. En déduire une fonction `tri_shell(L)` qui trie une liste en place avec le tri de Shell. On choisira des gaps décroissants de la forme $g_k = 2^k - 1$, et on commencera par le plus grand gap inférieur à la taille de la liste.

Remarque

Le choix de cette suite de gaps permet d'atteindre une complexité $\mathcal{O}(n^{1,5})$ dans le pire des cas. Il existe des suites permettant d'atteindre $\mathcal{O}(n(\log n)^2)$, mais l'étude de ces suites est à ce jour un problème ouvert, notamment l'existence d'une suite permettant d'obtenir $\mathcal{O}(n \log n)$.