Chapitre 6
Tris et recherche dichotomique

1 Recherche d'un élément dans une liste

Nous allons présenter deux algorithmes de recherche d'un élément dans une liste et comparer leur efficacité.

Les fonctions de recherche prendront en arguments une liste L et un objet x, et renverront :

- un indice k tel que $L[k]$ vaille x, si l'objet x est un élément de la liste L ;
- -1 sinon.

1.1 Recherche séquentielle

1.1.1 Principe

La méthode la plus simple, que l'on a déjà vue en cours et en TP, consiste à balayer un à un tous les éléments de la liste pour les comparer à l'élément recherché.

Plus précisément, on parcourt les indices des éléments de la liste.

⋆ Si, à l'indice considéré, l'élément de la liste est l'objet x recherché, on renvoie l'indice, ce qui arrête immédiatement la fonction.

⋆ Sinon, on continue le parcours.

À la fin du parcours, si la fonction ne s'est pas encore arrêtée, c'est que l'objet x n'apparait pas dans la liste ; on renvoie alors -1.

```
def recherche_sequentielle(L, x):
    """
    Entrées :
        - une liste L ;
        - un objet x.
    Sortie : l'indice de la première occurrence de x dans L
             si x appartient à L,
             -1 sinon.
    """
    for k in range(len(L)):
    # Pour k allant de 0 à len(L) - 1,
    # càd pour k parcourant les indices des éléments de L.
        if L[k] == x:
            return k
            # Si l'élément d'indice k dans la liste L vaut x,
            # alors on renvoie l'indice k auquel on a trouvé x,
            # et la fonction s'arrête immédiatement.
    return -1
    # Si la fonction ne s'est pas arrêtée plus tôt,
    # l'élément x n'est pas dans la liste.
```

```
recherche_sequentielle([9, -7, 3, 2, 0, -4, 2, 20 ,16], -4)

recherche_sequentielle([9, -7, 3, 2, 0, -4, 2, 20, 16], 2)

recherche_sequentielle([9, -7, 3, 2, 0, -4, 2, 20, 16], 14)
```

1.1.2 Complexité

Étudions la **complexité** de cet algorithme, c'est-à-dire le nombre d'opérations (ici, de comparaison de deux éléments) nécessaires à sa réalisation.

Dans le pire des cas, c'est-à-dire dans le cas où l'élément recherché est en dernière position ou n'apparait pas dans la liste, tous les indices de la liste auront été parcourus. Le nombre d'étapes nécessaires est alors la longueur de la liste.

1.2 Recherche dichotomique dans une liste triée

Lorsque l'on n'a aucune information sur l'agencement des éléments de la liste, on n'a pas d'autre choix que d'utiliser la recherche séquentielle pour rechercher un élément dans une liste, c'est-à-dire que de passer en revue tous les éléments de la liste.

Mais dans le cas où la liste est une liste de nombres *triée* dans l'ordre croissant, on peut faire beaucoup mieux avec une **recherche dichotomique**.

1.2.1 Principe

Le principe de la recherche dichotomique est le suivant : on coupe la liste en deux et on regarde l'élément situé au milieu de la liste.

- Si c'est le nombre x recherché, alors on renvoie son indice, ce qui arrête la fonction.
- Si l'élément est strictement inférieur au nombre x recherché, alors tous les éléments précédents de la liste le sont aussi, puisque la liste est triée dans l'ordre croissant. On cherche alors x parmi les éléments situés à droite de celui du milieu.
- Si l'élément est strictement supérieur au nombre x recherché, alors tous les éléments suivants de la liste le sont aussi, puisque la liste est triée dans l'ordre croissant. On cherche alors x parmi les éléments situés à gauche de celui du milieu.

On recommence alors la recherche, sur le même principe, dans la moitié droite ou la moitié gauche de la liste.

```
def recherche_dichotomique(L, x):
    """
    Entrées :
        - une liste L de nombres, triée dans l'ordre croissant ;
        - un nombre x.
    Sortie : un indice k tel que L[k] vaut x si x appartient à L,
             -1 sinon.
    """
    debut = 0
    fin = len(L) - 1
    # On recherchera l'élément x entre les indices debut et fin.
    while debut <= fin:
    # Tant qu'il reste des éléments à examiner :
        milieu = (fin + debut) // 2
        # Indice du milieu entre les indices debut et fin.
        if L[milieu] == x:
            return milieu
            # Si on trouve l'élément x à l'indice milieu,
            # on renvoie l'indice milieu et la fonction s'arrête.
        elif L[milieu] < x:
            debut = milieu + 1
            # Dans ce cas, comme la liste L est triée,
            # l'élément x ne peut se trouver
            # que strictement après l'indice milieu.
        else:
            fin = milieu - 1
            # Dans ce cas, l'élément x ne peut se trouver
            # que strictement avant l'indice milieu.
    return -1
    # Si la fonction ne s'est pas arrêtée plus tôt,
    # l'élément x n'est pas dans la liste.
```

Plus précisément, on commence par créer deux variables `debut` et `fin`, initialisées respectivement au premier et au dernier indices de la liste, qui seront les indices entre lesquels on cherchera le nombre x. Au début, on cherche donc le nombre x parmi tous les éléments de la liste.

Tant qu'il y a des éléments entre l'indice `debut` et l'indice `fin`, c'est-à-dire tant que `debut` est inférieur à `fin`, on procède de la manière suivante.
On considère l'indice `milieu` qui est la partie entière de la moyenne entre `debut` et `fin`.

- Si l'élément d'indice `milieu` est le nombre x recherché, alors on renvoie l'indice `milieu`, ce qui arrête la fonction.
- Si l'élément d'indice `milieu` est strictement inférieur au nombre x recherché, on recherche x parmi les éléments dont l'indice est strictement supérieur à l'indice `milieu`. L'indice `debut` prend donc la valeur `milieu` $+ 1$, et l'indice `fin` reste inchangé.
- Si l'élément d'indice `milieu` est strictement supérieur au nombre x recherché, alors on recherche x parmi les éléments dont l'indice est strictement inférieur à l'indice `milieu`. L'indice `debut` reste inchangé et l'indice `fin` prend la valeur `milieu` $- 1$.

Si on arrive à la fin de la boucle sans que la fonction se soit arrêtée, alors le nombre x n'apparait pas dans la liste, donc on renvoie -1.

```
recherche_dichotomique([-7, -4, 0, 2, 2, 3, 9, 16, 20], -4)

recherche_dichotomique([-7, -4, 0, 2, 2, 3, 9, 16, 20], 2)

recherche_dichotomique([-7, -4, 0, 2, 2, 3, 9, 16], 2)

recherche_dichotomique([-7, -4, 0, 2, 2, 3, 9, 16, 20], 14)
```

1.2.2 Complexité

Étudions la complexité dans le pire des cas de la recherche dichotomique.

Si la liste est de longueur 2^n, alors après un tour de boucle, on se ramène à une recherche dans l'une des moitiés de la liste, c'est-à-dire dans une liste de longueur 2^{n-1}. Après un deuxième tour de boucle, on se ramène à une recherche dans une liste de longueur 2^{n-2}. Au bout de n étapes, la longueur de la sous-liste considérée est réduite à 1 ; la boucle s'arrêtera donc au prochain tour.

Ainsi, sur une liste de longueur 2^n, la recherche dichotomique nécessite n tours de boucle dans le pire des cas (chaque tour de boucle demandant trois comparaisons, deux affectations et un calcul de moyenne, qui sont des opérations peu couteuses).

1.2.3 Comparaison avec la recherche séquentielle

Sur une liste de longueur 2^n, la recherche séquentielle nécessitait 2^n tours de boucle, contre seulement n pour la recherche dichotomique.

La recherche dichotomique est donc nettement plus efficace que la recherche séquentielle !

On peut l'observer expérimentalement en utilisant le module `time`, qui permet de mesurer le temps d'exécution d'un programme.
Après avoir importé la bibliothèque `time` :

```
import time
```

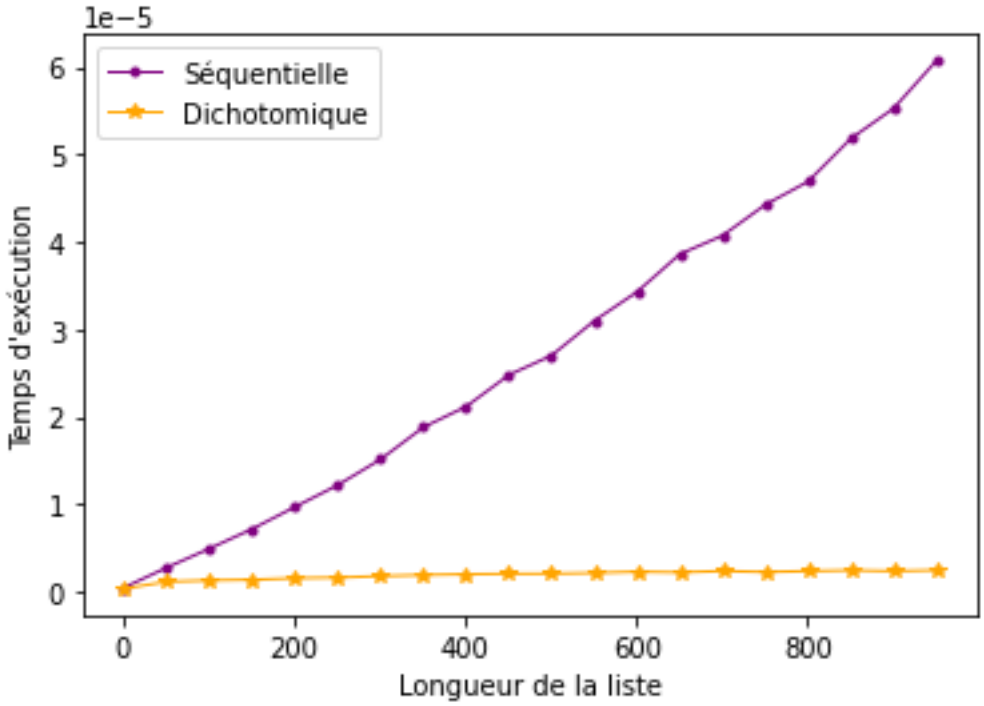
l'instruction

```
time.time()
```

renvoie le temps écoulé, en secondes, depuis la date du 1er janvier 1970 à 00h00min00s.

Pour mesurer le temps d'exécution d'un programme, on note alors le temps avant puis après exécution du programme, et on calcule la différence des deux.

Le graphe suivant représente les temps d'exécution moyens des fonctions de recherche séquentielle et dichotomique sur des listes d'entiers aléatoires en fonction de leur taille.



Si l'on souhaite faire un grand nombre de recherches dans une même liste, il sera donc très intéressant de commencer par la trier (avec l'une des méthodes de la partie suivante, par exemple), afin de pouvoir appliquer l'algorithme de recherche dichotomique, qui est plus rapide.

2 Algorithmes de tri

Dans cette partie, on présente plusieurs algorithmes permettant de trier une liste de nombres. En plus de permettre des recherches efficaces, trier une liste permettra aussi de calculer la médiane, les quartiles et les déciles d'une série statistique.

Dans la suite, L désignera la liste de nombres que l'on souhaite trier dans l'ordre croissant et n désignera la longueur de la liste L.

⚠ Les fonctions de tri par sélection et par insertion que nous présenterons modifient directement la liste donnée en argument (on dit que le tri se fait **en place**).
Pour ne pas perdre la liste de départ (et pouvoir lui appliquer d'autres fonctions, par exemple), on pourra en créer une copie avant de la trier. On pourra pour cela utiliser la technique du slicing :

```
Copie = L[ : ]
```

ou d'autres méthodes de copie de listes que nous verrons au chapitre 10.

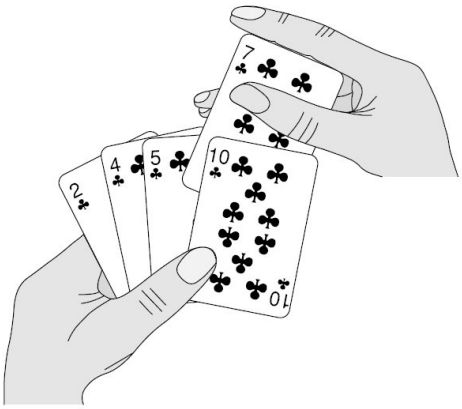
2.1 Tri par sélection

Le principe du **tri par sélection** est le suivant.

- On commence par trouver l'indice i_0 d'une occurrence du minimum de la liste.
  On échange alors l'élément à l'indice i_0 avec le premier élément de la liste : on échange les éléments $L[0]$ et $L[i_0]$.
  Après cette première étape, le plus petit élément est donc à sa place (en première position).
- Puis on trouve l'indice i_1 d'une occurrence du minimum des éléments qui restent, c'est-à-dire du minimum de la sous-liste $L[1:]$.
  On échange alors l'élément à l'indice i_1 avec le deuxième élément de la liste : on échange les éléments $L[1]$ et $L[i_1]$.
  Après cette deuxième étape, les deux plus petits éléments sont à leur place.
- Et ainsi de suite.
- Après $n-1$ étapes, les $n-1$ plus petits éléments sont à leur place, donc nécessairement, le dernier aussi !

Nous implémenterons le tri par sélection en TP.

2.2 Tri par insertion



Le principe du **tri par insertion** est de trier progressivement les premiers éléments de la liste.

Pour tout indice $k \in [\![1; n-1]\!]$, on insère l'élément $L[k]$ parmi les k premiers éléments de la liste $L[0]$, ..., $L[k-1]$, dont on suppose qu'ils sont déjà triés dans l'ordre croissant (même si ce ne sont pas nécessairement les k plus petites valeurs de la liste).

Détaillons les différentes étapes de l'algorithme.

- La sous-liste constituée uniquement du premier élément $L[0]$ de L est déjà triée (puisqu'elle ne contient qu'un seul élément).
- **Étape** 1 **: insertion de** $L[1]$ **en début de liste**
  On souhaite insérer l'élément $L[1]$ à sa place dans la liste $[L[0]]$.
  Pour cela, on compare les éléments $L[0]$ et $L[1]$:
  - ⋆ si l'élément $L[1]$ est plus petit que $L[0]$, alors on les permute ;
  - ⋆ sinon, on les laisse à leur place.

  Au terme de cette étape, la liste $[L[0], L[1]]$ constituée des deux premiers éléments de la liste est triée dans l'ordre croissant.
- ...
- **Étape** k **: insertion de l'élément** $L[k]$ **en début de liste**
  Au début de l'étape k, la liste $[L[0], ..., L[k-1]]$ constituée des k premiers éléments de la liste est triée dans l'ordre croissant.
  Pour insérer l'élément $L[k]$ à sa place dans la liste $[L[0], ..., L[k-1]]$, on peut alors procéder de deux manières différentes.
  - ⋆ **Première méthode : par échanges successifs**
    La première méthode consiste à procéder par échanges successifs : tant que l'élément $L[k]$ est plus petit que l'élément qui le précède, on l'échange avec l'élément précédent.
  - ⋆ **Seconde méthode : par décalages successifs**
    La seconde méthode consiste à déterminer tout d'abord à quelle position i l'élément $L[k]$ devra être inséré, à l'aide d'une boucle `while`.

On stocke alors la valeur de $L[k]$ dans une variable.
On décale d'un cran vers la droite les éléments $L[k-1]$, puis $L[k-2]$, ..., puis $L[i]$.
Puis on affecte la valeur stockée à l'élément $L[i]$.

À l'issue de cette étape, la liste $[L[0], ..., L[k]]$ des $k+1$ premiers éléments de L est triée.

- ...
- On procède de même jusqu'à l'étape $n-1$.

Implémentons les deux versions du tri par insertion.

⋆ **Tri par insertion avec échanges successifs**

Commençons par définir une fonction d'échange pour alléger l'écriture du programme.

```
def echange(L, i, j):
    """
    Arguments : une liste L et deux indices i et j de la liste L.
    Échange les éléments d'indices i et j dans la liste L.
    """
    aux = L[i]
    L[i] = L[j]
    L[j] = aux
```

```
def tri_insertion(L):
    """
    Entrée : une liste L de nombres.
    Trie la liste avec l'algorithme de tri par insertion.
    Sortie : aucune, la liste L est directement modifiée.
    """
    n = len(L)
    for k in range(1, n):
    # Pour k allant de 1 à n - 1,
        # on insère l'élément L[k] dans le début de la liste
        # de la manière suivante.

        pos = k # Position à laquelle se trouve
                # l'élément que l'on cherche à insérer
                # (initialement, il est à l'indice k).

        while (pos > 0) and (L[pos] < L[pos - 1]):
            echange(L, pos, pos - 1)
            pos = pos - 1
        # Tant que l'élément n'est pas en première position
        # et tant que l'élément d'indice pos est inférieur
        # à l'élément précédent dans la liste,
            # on échange les éléments d'indices pos et pos - 1.
            # L'élément qui nous intéresse se trouve
            # désormais à l'indice pos - 1.
```

⋆ **Tri par insertion avec décalages successifs**

```
def tri_insertion_bis(L):
    """
    Entrée : une liste L de nombres.
    Trie la liste avec l'algorithme de tri par insertion.
    Sortie : aucune, la liste L est directement modifiée.
    """
    n = len(L)
    for k in range(1, n):
    # Pour k allant de 1 à n - 1 :

        # on cherche tout d'abord la position i
        # à laquelle on insèrera L[k].
        i = 0
        element = L[k] # On stocke la valeur de L[k].
        while (i < k) and (L[i] < element):
            i = i + 1

        # On va insérer l'élément L[k] à l'indice i.
        for j in range(k, i, -1):
            L[j] = L[j - 1]
        # Pour j allant de k à i + 1 en reculant,
            # l'élément d'indice j prend la valeur
            # de l'élément précédent (d'indice j - 1).

        L[i] = element
        # On place finalement l'élément (initialement L[k])
        # à l'indice i.
```