Recherche dichotomique et algorithmes de tris

I. Recherche dichotomique

Dans cette partie, nous allons étudier plusieurs problèmes que l'on peut résoudre par l'algorithme de dichotomie.

1) Recherche d'une solution approchée à une équation

Soit $f : [a, b] \to \mathbb{R}$ une fonction continue. On souhaite trouver une valeur approchée d'une solution de l'équation $f(x) = 0$ sur l'intervalle $[a, b]$.

- ⋆ Si $f(a) \times f(b) < 0$, alors, d'après le théorème des valeurs intermédiaires, l'équation $f(x) = 0$ admet au moins une solution dans l'intervalle $[a, b]$.
- ⋆ Si de plus f est strictement monotone, alors l'équation $f(x) = 0$ admet une unique solution dans l'intervalle $[a, b]$.

L'algorithme de dichotomie permet de trouver une solution approchée à cette équation. Le principe est le suivant :

- ⋆ On initialise deux bornes `debut = a` et `fin = b`.
- ⋆ Tant que `fin - debut > epsilon` :
  - On calcule l'indice médian `milieu = (debut + fin) / 2`.
  - Si `f(milieu) == 0` alors on a trouvé une solution exacte.
  - Si `f(milieu) * f(debut) < 0` alors on est dans le cas où la solution est dans l'intervalle `[debut, milieu]`. On pose donc `fin = milieu`.
  - Si `f(milieu) * f(fin) < 0` alors on est dans le cas où la solution est dans l'intervalle `[milieu, fin]`. On pose donc `debut = milieu`.
- ⋆ À la fin de la boucle, on a trouvé une solution approchée à l'équation $f(x) = 0$ sur l'intervalle $[a, b]$.

Remarque : L'algorithme de dichotomie est très efficace et très rapide pour trouver une solution approchée d'une équation. En effet, à chaque étape, on divise l'intervalle de recherche par 2. Ainsi, le nombre d'étapes nécessaires pour trouver une solution approchée à ε près est le plus petit entier naturel N tel que

$$\frac{b-a}{2^N} < \varepsilon \qquad \Leftrightarrow \qquad N > \log_2\left(\frac{b-a}{\varepsilon}\right)$$

donc $N = \left\lfloor \log_2\left(\frac{b-a}{\varepsilon}\right) \right\rfloor + 1$. Par exemple, si $b - a = 1$ et si $\varepsilon = 10^{-6}$, alors $N = 20$ étapes.

```python
def dichotomie(f, a, b, epsilon):
    """
    Entrée : une fonction f, deux nombres a et b tels que f(a) * f(b) < 0
             et un nombre epsilon > 0
    Sortie : une solution approchée de l'équation f(x) = 0 sur l'intervalle [a, b]
    """
    debut = a
    fin = b
    while fin - debut > epsilon:
        milieu = (debut + fin) / 2
        if f(milieu) == 0:
            return milieu
        elif f(milieu) * f(debut) < 0:
            fin = milieu
        else:
            debut = milieu
    return (debut + fin) / 2
```

2) Recherche d'un élément dans une liste triée

Considérons une liste de nombres `L` triée dans l'ordre croissant et un nombre `x`. On veut déterminer si `x` est présent dans la liste `L`.

Une première solution consiste à parcourir la liste jusqu'à trouver l'élément `x`. Si, à la fin du parcours, on n'a toujours pas trouvé l'élément `x`, alors il est absent.

```python
def recherche(L, x):
    for e in L:
        if e == x:
            return True
    return False
```

Si `len(L) = n`, alors cette fonction réalise (dans le pire des cas) $O(n)$ opérations élémentaires lors de son exécution.

Dans la fonction précédente, nous n'avons pas utilisé le fait que la liste `L` était triée. Utilisons maintenant l'algorithme de dichotomie :

- ⋆ On initialise deux bornes `debut = 0` et `fin = len(L) - 1`.
- ⋆ Tant que `debut < fin` :
  - On calcule l'indice médian `milieu = (debut + fin) // 2` (**il faut que `milieu` soit un entier**).
  - Si `x < L[milieu]` alors il faut chercher `x` dans le début de la liste (puisqu'elle est triée). On pose donc `fin = milieu - 1`.
  - Si `x > L[milieu]` alors il faut chercher `x` dans la fin de la liste (puisqu'elle est triée). On pose donc `debut = milieu + 1`.
  - Si `x == L[milieu]` alors on a trouvé `x`.
- ⋆ À la fin de la boucle, il suffit de vérifier si `x == L[debut]`.

```python
def recherche_dicho(L, x):
    """
    Entrée : une liste L triée dans l'ordre croissant et un nombre x
    Sortie : True si x est présent dans L, False sinon
    """
    debut = 0
    fin = len(L) - 1

    while debut < fin:
        milieu = (debut + fin) // 2
        if x < L[milieu]:
            fin = milieu - 1
        elif x > L[milieu]:
            debut = milieu + 1
        else:
            return True

    return x == L[debut]
```

Le script suivant permet de calculer et de comparer les temps d'exécution de chacune de ces fonctions sur 10 exemples de données aléatoires :

```python
from time import time
from random import randint

T_recherche = T_dicho = 0
for _ in range(10):
    L = [randint(0, 10**7) for _ in range(10**6)]
    L.sort()
    x = randint(0, 10**7)

    t1 = time()
    recherche(L, x)
    t2 = time()
    T_recherche += t2 - t1

    t1 = time()
    recherche_dicho(L, x)
    t2 = time()
    T_dicho += t2 - t1

print(f"Temps total pour une recherche classique : {T_recherche}")
print(f"Temps total pour une recherche dichotomique : {T_dicho}")
```

Le résultat de ce script est le suivant :

```
Temps total pour une recherche classique : 0.9277691841125488
Temps total pour une recherche dichotomique : 0.0
```

On constate donc que l'algorithme de recherche dichotomique est beaucoup plus rapide que l'algorithme de recherche basique. Cependant, si on doit trier la liste avant de rechercher l'élément, alors on perd l'avantage de cet algorithme.

II. Algorithmes de tris

Dans cette partie, nous allons étudier plusieurs algorithmes permettant de trier une liste de nombres. Nous avons vu dans la partie précédente que trier une liste permet d'avoir un algorithme de recherche efficace. Cela permet également de calculer la médiane, les quartiles et les déciles d'une série statistique.

Un enjeu important concernant les algorithmes de tris est de trouver un algorithme qui soit le plus rapide possible. On va donc étudier la complexité de ces algorithmes.

Dans toute la suite, `L` désigne une liste de nombres que l'on souhaite trier dans l'ordre croissant et on notera `n` la longueur de `L`.

1) Tri par sélection

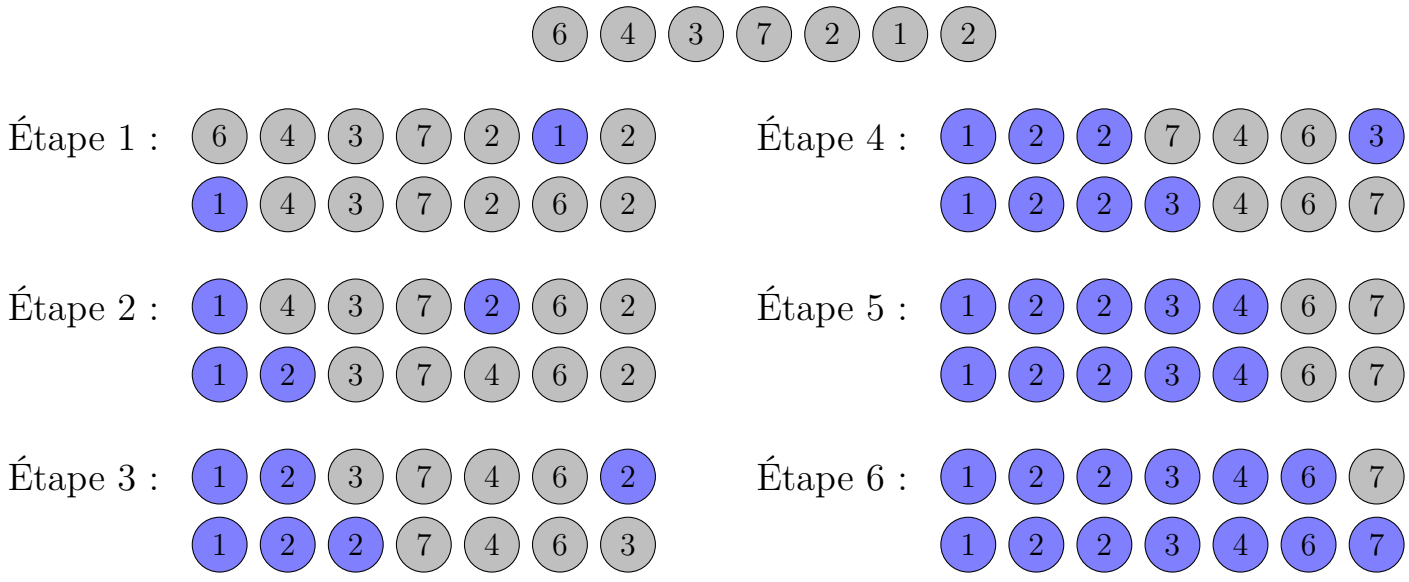
Le principe du tri par sélection est le suivant :

- ⋆ Étape 1 : on commence par chercher le minimum de la liste `L` et on l'échange avec le premier élément de la liste.
- ⋆ Étape 2 : ensuite, on cherche le minimum de la liste `L[1:]`, c'est-à-dire de la sous-liste `L` sans son premier élément, et on l'échange avec le second élément de la liste.
- ⋆ Étape 3 : on continue en cherchant le minimum de la liste `L[2:]`, c'est-à-dire de la sous-liste `L` sans ses deux premiers éléments, et on l'échange avec le troisième élément de la liste.
- ⋆ Étape k : on cherche le minimum de la liste `L[k-1:]`, c'est-à-dire de la sous-liste `L` sans ses `k-1` premiers éléments, et on l'échange avec le `k`-ième élément de la liste.
- ⋆ Étape $n-1$: on cherche le minimum de la liste `L[n-2:]`, c'est-à-dire de la sous-liste `L` sans ses `n-2` premiers éléments, et on l'échange avec le `n-1`-ième élément de la liste.

Cet algorithme alterne donc deux opérations déjà vues dans les cours précédents :

- ⋆ recherche de l'indice du minimum d'une liste
- ⋆ échange de deux éléments d'une liste

Appliquons le tri par sélection à la liste `L = [6, 4, 3, 7, 2, 1, 2]` :



Passons maintenant à l'implémentation (code du programme) de cet algorithme en Python :

```python
def tri_par_selection(L):
    """
    Entrée : une liste numérique
    Sortie : None
    La liste est triée dans l'ordre croissant.
    """
    n = len(L)
    for k in range(n-1):
        # On détermine l'indice i d'une occurrence dans L du minimum de L[k:]
        i = k
        for j in range(k, n):
            if L[j] < L[i]:
                i = j

        # On échange les éléments L[k] et L[i]
        L[k], L[i] = L[i], L[k]
```

Pour utiliser ce tri :

```
>>> L = [6, 4, 3, 7, 2, 1, 2]
>>> tri_par_selection(L)
>>> L
[1, 2, 2, 3, 4, 6, 7]
```

Remarque :

- ⋆ Ce tri est un tri sur place c'est-à-dire que la liste passée en argument est directement modifiée. Si l'on souhaite conserver la liste dans l'ordre initial et obtenir la liste triée dans une autre liste, il faut faire une copie de la liste avant d'appliquer la fonction `tri_par_selection` : `L_copie = L[:]`.
- ⋆ Pour évaluer le temps d'exécution de la fonction `tri_par_selection` sur une liste de taille `n`, on peut compter le nombre d'opérations élémentaires nécessaires à l'exécution de la fonction :
  - `n = len(L)` : 1 opération
  - `i = k` : 1 opération mais répétée `n-1` fois
  - `L[j] < L[i]` et `i = j` : 2 opérations répétées `n-k` fois pour chaque `k` entre `0` et `n-2`
  - `L[k], L[i] = L[i], L[k]` : 1 opération répétée `n-1` fois

  Ainsi, en notant $C(n)$ le nombre total d'opérations, on a

$$\begin{aligned} C(n) &= 1 + 1 \times (n-1) + \sum_{k=0}^{n-2} 2(n-k) + 1 \times (n-1) \\ &= 1 + 2(n-1) + 2n(n-1) - 2\sum_{k=0}^{n-2} k \\ &= 1 + 2(n-1)(n+1) - 2\frac{(n-2)(n-1)}{2} \\ &= n^2 + 3n - 3 \end{aligned}$$

  Ainsi, $C(n)$ est de l'ordre de n^2 ce qui se note $C(n) = O(n^2)$. La fonction C est la complexité temporelle de l'algorithme de tri par sélection. On dit que **cet algorithme est de complexité quadratique**.

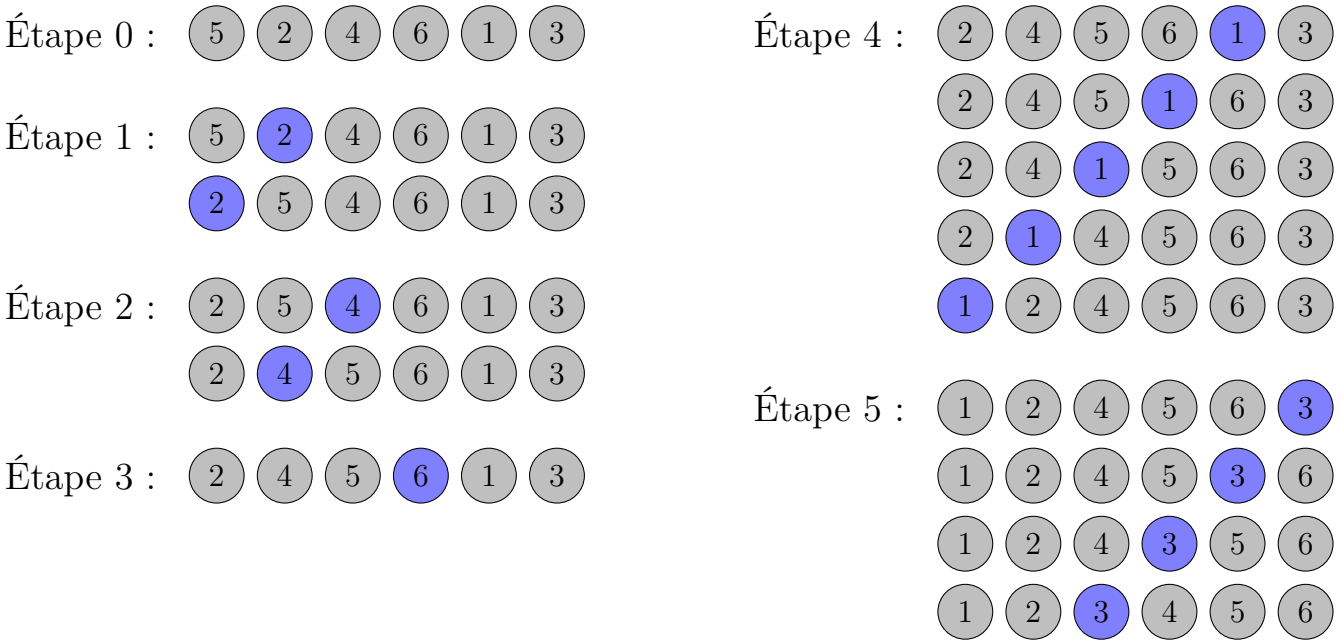
2) Tri par insertion

Le principe du tri par insertion est le suivant :

- ⋆ On considère un élément de la liste que l'on mémorise dans une variable `pivot`.
- ⋆ On parcourt la liste de droite à gauche à partir de la position du pivot en décalant vers la droite tous les éléments plus grands que `pivot`.
- ⋆ On insère alors `pivot` à la position du dernier élément décalé.

En considérant le pivot comme étant successivement les éléments `L[1]`, puis `L[2]`, puis `L[3]` etc, la liste `L` sera alors triée au bout de `n-1` étapes.

Appliquons le tri par insertion à la liste `L = [5, 2, 4, 6, 1, 3]` :



Passons maintenant à l'implémentation de cet algorithme en Python :

```
def tri_par_insertion(L):
    """
    Entrée : une liste numérique
    Sortie : None
    La liste est triée dans l'ordre croissant.
    """
    n = len(L)
    for k in range(1, n):
        pivot = L[k]
        i = k
        while i > 0 and L[i-1] > pivot:
            L[i] = L[i-1]
            i -= 1
        L[i] = pivot
```

Pour utiliser ce tri :

```
>>> L = [5, 2, 4, 6, 1, 3]
>>> tri_par_insertion(L)
>>> L
[1, 2, 3, 4, 5, 6]
```

Remarque :

- ⋆ Comme pour le tri par sélection, ce tri est aussi un tri sur place.
- ⋆ Pour évaluer le temps d'exécution de la fonction `tri_par_insertion` sur une liste de taille `n`, on peut compter le nombre d'opérations élémentaires nécessaires à l'exécution de la fonction dans le pire des cas (la boucle `while` peut être interrompue plus ou moins rapidement) :
  - Il y a 1 opération élémentaire en dehors de la boucle.
  - La boucle `for` est répétée `n-1` et dans cette boucle :
    - ▶ Il y a 3 opérations élémentaires.
    - ▶ Une boucle `while` qui est répétée au plus `i+1` fois dans laquelle il y a 4 opérations élémentaires.

  Ainsi, en notant $C(n)$ le nombre total d'opérations, on a

$$\begin{aligned} C(n) &= 1 + 3 \times (n-1) + \sum_{i=1}^{n-1} 4(i+1) \\ &= 3n - 2 + 4\sum_{i=1}^{n-1} i + 4(n-1) \\ &= 7n - 6 + 4\frac{(n-1)n}{2} \\ &= 2n^2 + 5n - 6 \end{aligned}$$

  Ainsi, $C(n)$ est de l'ordre de n^2 ce qui se note $C(n) = O(n^2)$. La fonction C est la complexité temporelle de l'algorithme de tri par insertion. On dit que **cet algorithme est de complexité quadratique**.

3) Tri par comptage

Le principe du tri par comptage est le suivant :

- ⋆ On considère une liste `L` de taille `n` et on suppose que les éléments de `L` sont des entiers compris entre `0` et `N` où $N \in \mathbb{N}$.
- ⋆ On calcule les effectifs de chaque valeur :
  - On initialise une liste `E` de taille `N+1` dont tous les éléments sont égaux à `0`.
  - On parcourt la liste `L` et pour chaque élément `e`, on incrémente la case `E[e]` de `1`.
- ⋆ On reconstruit la liste `L` en ajoutant chaque indice `i` de `E` autant de fois que le nombre d'occurrences `E[i]` de `i` dans `L`.

Appliquons le tri par comptage à la liste `L = [3, 0, 0, 5, 1, 0, 1, 5, 5, 1]` en prenant `N = 5` :

- ⋆ On initialise la liste `E = [0, 0, 0, 0, 0, 0]`.
- ⋆ On parcourt la liste `L` et on incrémente les cases de `E` :
  - `E[3]` est incrémenté de `1` donc `E = [0, 0, 0, 1, 0, 0]`.
  - `E[0]` est incrémenté de `1` donc `E = [1, 0, 0, 1, 0, 0]`.
  - `E[0]` est incrémenté de `1` donc `E = [2, 0, 0, 1, 0, 0]`.
  - `E[5]` est incrémenté de `1` donc `E = [2, 0, 0, 1, 0, 1]`.
  - ⋮
  - `E[5]` est incrémenté de `1` donc `E = [3, 3, 0, 1, 0, 3]`.
  - `E[1]` est incrémenté de `1` donc `E = [3, 4, 0, 1, 0, 3]`.

- ⋆ On reconstruit la liste `L` en ajoutant chaque élément `e` de `E` autant de fois que le nombre d'occurrences de `e` dans `L` :

```
L = 3 * [0] + 4 * [1] + 1 * [3] + 3 * [5] = [0, 0, 0, 1, 1, 1, 3, 5, 5, 5]
```

Passons maintenant à l'implémentation de cet algorithme en Python :

```python
def effectifs(L, N):
    """
    Entrée : une liste numérique et un entier N
    Sortie : la liste des effectifs de chaque valeur
    """
    E = (N + 1) * [0]

    for e in L:
        E[e] += 1

    return E

def tri_comptage(L, N):
    """
    Entrée : une liste numérique et un entier N
    Sortie : une copie de la liste triée dans l'ordre croissant
    """
    E = effectifs(L, N)

    L_triee = []
    for i in range(len(E)):
        # On ajoute à L_triee E[i] fois la valeur i
        L_triee = L_triee + E[i] * [i]

    return L_triee
```

Remarque :

- ⋆ Ce tri n'est pas un tri sur place : la liste passée en argument n'est pas modifiée mais une nouvelle liste est créée.
- ⋆ Il est nécessaire de connaître la valeur maximale des éléments de la liste pour pouvoir utiliser ce tri.
- ⋆ La complexité temporelle de ce tri est $C(n, N) = O(n + N)$. En effet, on parcourt la liste `L` une fois pour calculer les effectifs et on parcourt la liste `E` une fois pour reconstituer la liste triée.

  Cette complexité est donc bien meilleure que la complexité quadratique des tris précédents mais le cadre d'application de ce tri est assez restreint.

En général, ce tri est utilisé dans le cadre de l'analyse statistique de données où la liste des effectifs permet de calculer la médiane ou les quartiles de la liste.