TP (S1) 5 Tris

Objectifs

- Connaître les algorithmes de tris classiques.
- Déterminer les caractéristiques d'un tri donné.

1. INTRODUCTION

Important : commencez par récupérer et compiler le fichier `TPTris` dans le répertoire partagé de la classe : il contient différents outils qui vous permettront de tester, d'illustrer et de comparer vos différentes fonctions.

Un algorithme de tri est un algorithme permettant de trier par ordre croissant ou décroissant une collection d'objets. Nous avons déjà rencontré deux algorithmes de tris lors des TP précédents : le tri à bulles (TP2) et le tri par insertion (TP4). Nous allons allons étudier dans ce TP quatre nouveaux algorithmes de tri - le tri par sélection, le tri fusion, le tri rapide et le tri par comptage - et coder des fonctions permettant d'appliquer ces algorithmes afin de trier par ordre croissant des listes d'entiers et/ou de flottants voire même de chaînes de caractères, ces dernières étant nativement ordonnée en Python par ordre alphabétique. Il est bien entendu possible d'adapter nos fonctions afin de trier des listes par ordre décroissant, de trier des listes d'autres types d'éléments pour d'autres relations d'ordre et même de trier d'autres structures de données (tableaux, dictionnaires,...).

Trier des données est un aspect fondamental de l'informatique, on a vu par exemple dans le TP4 que la recherche d'un élément dans une liste pouvait se faire de façon bien plus efficace lorsque la liste était préalablement triée. Trier a bien entendu un coût mais qui est vite compensé lorsque le nombre de recherches devient conséquent. Pour vous en convaincre, saisissez dans la console la commande `compare_recherche(n)`, avec n entre 50 et 2000, pour obtenir le graphique des temps de recherche dans une liste d'entiers de taille n. Il est donc nécessaire de pouvoir trier efficacement.

Les différents tris se différencient selon les critères suivants :

- **[Comparatif ou non]** Un tri est dit *comparatif* s'il effectue des comparaisons entre les éléments de la liste à trier ; c'est le cas de l'immense majorité des tris. Nous donnerons un unique exemple de tri non-comparatif, le tri par comptage, qui n'a d'intérêt que dans une situation très particulière.
- **[Récursivité]** Un tri est dit *récursif* s'il est codé récursivement, non-récursif sinon. Nous verrons que le codage le plus naturel des tris les plus efficaces est récursif, ce qui a des conséquences regrettables (gestion de la pile). Il est possible d'en proposer une implémentation impérative mais c'est plus difficile et ne fait pas l'objet de ce TP.
- **[Stabilité]** Un tri est dit *stable* si les éléments égaux ont dans la liste triée le même ordre que dans la liste initiale. Par exemple un tri qui, appliqué à la liste `[3,2,1,2]`, renvoie la liste `[1,2,2,3]` n'est pas stable.
- **[En place ou non]** Un tri est dit *en place* lorsqu'il modifie la liste initiale, non en place lorsqu'il ne la modifie pas[1]. La version en place d'un tri est moins coûteuse en espace mémoire mais plus difficile à coder, et la liste initiale est perdue.
- **[Complexité temporelle]** La *complexité temporelle* d'une fonction de tri est le nombre moyen c_n d'opérations que celle-ci réalise pour trier une liste de taille n ; plus ce nombre est faible, plus le tri sera rapide et moins il consommera d'énergie. Il y a deux cas de figure classiques pour les tris comparatifs :
  1. On peut majorer c_n par αn^2 où α est une constante multiplicative ; on parle alors de tri quadratique ou de tri en $O(n^2)$. C'est le cas par exemple du tri à bulles et du tri par insertion dans sa version non-dichotomique.
  2. On peut majorer c_n par $\alpha n \log_2(n)$ où α est une constante multiplicative ; on parle alors de tri en $O(n\log_2(n))$. C'est le cas par exemple du tri par insertion dans sa version dichotomique.

  Nous verrons que, lorsque les conditions sont favorables, le tri par comptage a lui une complexité linéaire (en $O(n)$).

1. Il renvoie alors une nouvelle liste contenant les mêmes éléments que la liste initiales

2. EXERCICES

Dans tous les exercices, on notera n le nombre d'éléments de la liste `L` à trier.

Exercice 1 **Le tri par sélection** [Sol 1] Le principe de ce tri est le suivant :

- Première étape : on trouve le plus grand élément de `L` et on l'échange avec le dernier élément de `L`.
- Deuxième étape : on trouve le plus grand élément parmi les $n-1$ premiers éléments de `L` et on l'échange avec l'avant-dernier élément de `L`.
- ⋮
- Dernière étape : on trouve le plus grand élément parmi les 2 premiers éléments de `L` et on l'échange avec le deuxième élément de `L`.

Il s'agit donc d'un tri comparatif non-récursif, on admet que sa complexité est en $O(n^2)$.

1. Écrire la fonction `pos_max(L:list)->int` qui renvoie l'indice de la première occurrence du maximum de la liste `L` supposée non-vide.
   Testez votre fonction en saisissant dans la console la commande `test_pos_max()`.

2. Écrire la fonction `tri_select(L:list)->none` qui réalise à l'aide de la fonction précédente le tri par sélection en place de la liste `L`.
   Testez votre fonction en saisissant dans la console la commande `test_tri_select()`.

3. Ce tri est-il stable ? Si ce n'est pas le cas, comment modifier notre code pour qu'il le soit ?

Exercice 2 **Le tri fusion** [Sol 2] Le principe de ce tri est le suivant :

- Si la liste a un élément ou moins, il n'y a rien à faire.
- Si la liste a deux éléments ou plus, on procède en deux temps :
  1. On coupe la liste en deux et on trie indépendamment les deux moitiés.
  2. On fusionne les moitiés à présent triées

Il s'agit donc d'un tri comparatif récursif, on admet que sa complexité en en $O(n\log_2(n))$.

1. Écrire la fonction `merge(L1:list,L2:list)->list` qui renvoie la liste correspondant à la fusion des deux listes triées L_1 et L_2. Par exemple `merge([0,2,2,4],[1,3,5])` doit renvoyer la liste `[0,1,2,2,3,4,5]`.
   Testez votre fonction en saisissant dans la console la commande `test_merge()`.

2. Écrire la fonction `tri_fusion(L:list)->list` qui réalise à l'aide de la fonction précédente le tri fusion non en place de la liste `L`.
   Testez votre fonction en saisissant dans la console la commande `test_tri_fusion()`.

3. Expliquez à quelle condition ce tri est stable.

Vous pouvez comparer l'efficacité de vos différents tris en saisissant[2] dans la console la commande `compare_tris(1000)`, vous obtiendrez alors le graphique des temps d'exécution des tris de listes de taille 0 à 1000 (voir aussi dernière page).

Vous pouvez constater, si votre code est correct, que les deux tris quadratiques sont bien moins efficaces que les tris en $O(n\log_2(n))$ et que, parmi ces trois derniers, le tri rapide est le plus efficace. Il est possible d'améliorer encore l'efficacité du tri rapide en choisissant de façon plus habile le pivot, mais cela se fait au prix de la stabilité.

Remarquons enfin que les tris en $O(n\log_2(n))$ utilisent tous des méthodes dichotomiques, c'est ce qui permet de "transformer" un facteur n en $\log_2(n)$ (cf TP3).

Exercice 3 **Le tri rapide** [Sol 3]

Le principe de ce tri, proche du tri fusion, est le suivant :

- Si la liste a un élément ou moins, il n'y a rien à faire.
- Si la liste a deux éléments ou plus, on procède en deux temps :
  1. On isole le premier élément de `L`, noté p pour "pivot", et on génère deux nouvelles listes : la liste Li constituée des éléments de `L[1:]` strictement inférieurs à p, et la liste Ls constituée des éléments de `L[1:]` supérieurs ou égaux à p.
  2. On trie les listes Li et Ls, notons alors Lit et Lst les listes ainsi triées, et on renvoie la concaténation de Lit, de $[q]$ et de Lst.

Il s'agit donc d'un tri comparatif récursif, on admet que sa complexité en en $O(n\log_2(n))$.

1. Écrire la fonction `split_list(L:list)->(list, list, list)` qui renvoie le triplet $(Li,[p],Ls)$; par exemple `split_list([3, 0, 3, 3, 3, 4, 5, 5, 2, 4])` doit renvoyer `([0, 2], [3], [3, 3, 3, 4, 5, 5, 4])`. Testez votre fonction en saisissant dans la console la commande `test_split_list()`.

2. Si vous n'avez pas codé tous les tris précédents la commande ne s'exécutera pas. Pour remédier à cela, vous pouvez commenter dans le code de la fonction `compare_tris(n)` les lignes qui correspondent aux tris non codés.

2. Écrire la fonction `tri_rapide(L:list)->list` qui réalise à l'aide de la fonction précédente le tri rapide non en place de la liste `L`. Testez votre fonction en saisissant dans la console la commande `test_tri_rapide()`.

3. Ce tri est-il stable? en place?

4. On souhaite écrire une fonction qui réalise le tri rapide "en place"

   4.1) Écrire une fonction `Ranger(L:list,d:int,f:int)->int` qui réordonne les éléments de la liste `L[d:f+1]` de sorte que si `p=L[d]` est le premier élément de la liste initiale, les éléments situés avant l'élément `p` dans la liste finale soient inférieurs ou égaux à `p` et qui retourne `m` la position du dernier élément inférieur ou égal à `p`. Par exemple si `L=[5,4,7,2,6,1,2]` l'exécution de `Ranger(L,1,5)` donne `L=[5,1,2,4,6,7,2]` et renvoie `3`

   4.2) Écrire une fonction récursive `TriRapideAnnexe(L:list,d:int,f:int)->None` qui tri récursivement la sous liste `L[d:f+1]`

   4.3) Écrire une fonction récursive `TriRapide(L:list)->None` qui tri récursivement la liste `L` en utilisant la fonction précédente

Exercice 4 **Le tri par comptage** [Sol 4] Ce tri peut être mis en place quand les éléments de la liste appartiennent à un ensemble ordonné fini $E = \{e_1, e_2, \dots, e_m\}$ avec $e_1 < e_2 < \dots < e_m$. Le principe de ce tri est le suivant :

1. On crée une liste `C` de $m = \text{Card}(E)$ zéros qui sera amenée à contenir le nombre d'occurrences de chaque élément de E dans `L`. Ainsi, à l'issue **d'un unique parcours** de `L`, `C[0]` sera égal au nombre de fois où e_1 est présent dans `L`, `C[1]` sera égal au nombre de fois où e_2 est présent dans `L` etc...
2. On crée alors par concaténations successives une nouvelle liste `Lt` triée par ordre croissant et qui contient exactement les mêmes éléments que `L`.

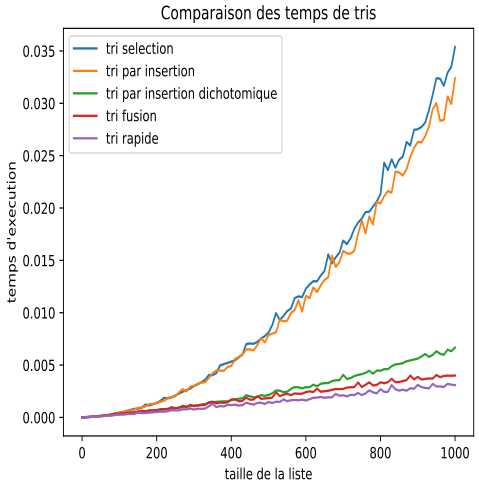
Il s'agit donc d'un tri non comparatif, non récursif et non en place ; parler de stabilité ici n'a pas de sens.

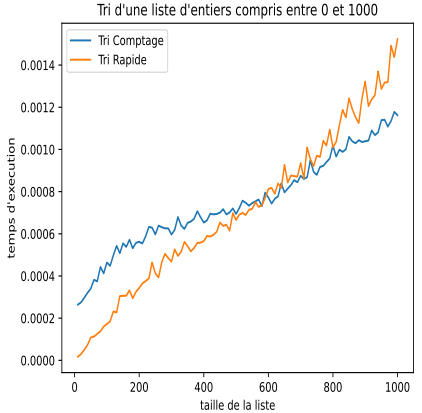
Écrire la fonction `tri_comptage(L:list,a:int,b:int)->list` qui réalise le tri par comptage de la liste `L` dont les éléments sont dans l'intervalle d'entiers $[\![a, b]\!]$. Testez votre fonction en saisissant dans la console la commande `test_tri_comptage()`.

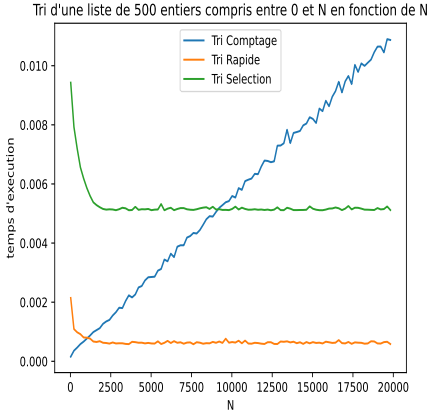
Le tri par comptage a une complexité temporelle linéaire, c'est-à-dire de forme αn où α est indépendant de n, soit mieux que le tri rapide comme vous pouvez le constater sur le graphique obtenu en saisissant dans la console la commande `compare_tris2(1000)`.

On observe toutefois dans le graphique que quand la taille de la liste est « petite » devant le cardinal de E, le tri rapide est meilleur. En effet, si la constante α est indépendante de la taille de la liste, elle dépend en revanche du cardinal de E.

Saisissez dans la console la commande `compare_tris3(500)` pour vous en convaincre (voir aussi ci-dessous) : à taille de liste constante, le temps d'exécution du tri comptage augment avec le cardinal de E tandis qu'il est constant pour les autres tris.







SOLUTIONS DES EXERCICES

Solution 1

1.
```
def pos_max(L):
    """Renvoie l'indice de la premiere occurence du max de L"""
    m, p, n = L[0], 0, len(L) #initialisation avec le premier \
    ↪ élément
    for i in range(1, n): #on parcourt les éléments suivants
        if L[i] > m:   #si on trouve un élément strict. sup.
            m, p = L[i], i  #on met à jour nos variables
    return p
```

2.
```
def tri_select(L):
    """Réalise le tri par selection en place de la liste L"""
    n=len(L)
    #la première étape se fait sur la liste en entier: L[:n]
    #la deuxième sur ses n-1 premiers éléments: L[:n-1]
    #la dernière sur ses deux premiers éléments: L[:2]
    for i in range(n-1):
        p = pos_max(L[:n-i])
        L[p], L[n-i-1] = L[n-i-1], L[p]
```

3. Le tri n'est pas stable car on sélectionne la première occurrence du maximum au lieu de la dernière. Pour remédier à cela, il faut remplacer dans la fonction `pos_max` le test `L[i]>m` par le test `L[i]>=m`.

Solution 2

1.
```
def merge(L1,L2):
    """Renvoie la fusion des listes tiées L1 et L2"""
    n1, n2 = len(L1), len(L2)
    p1, p2 = 0, 0   #on se place sur le premier élément de \
    ↪ chaque liste
    L = []
    while p1 < n1 and p2 < n2:#tant que l'on a pas épuisé une \
    ↪ des 2 listes
        if L1[p1] <= L2[p2]:      #si l'élément courant de L1 \
        ↪ est le + petit
            L.append(L1[p1])    #on l'ajoute à L
            p1 += 1           #et on passe à l'élément suivant \
            ↪ de L1
        else:                   #sinon on inverse les rôles \
        ↪ des 2 listes
            L.append(L2[p2])
            p2 += 1
    if p1 == n1: #une des listes peut être non épuisée
        L += L2[p2:]
    else:
        L += L1[p1:]
    return L
```

```
>>> merge([0,2,2,4], [1,3,5])
[0, 1, 2, 2, 3, 4, 5]
```

2.
```
def tri_fusion(L):
    """Renvoie le tri fusion de la liste L"""
    if len(L) < 2:
        return L
    else:
        m = len(L)//2
        return merge(tri_fusion(L[:m]), tri_fusion(L[m:]))
```

```
>>> tri_fusion([-2, 1, 0, 4, 3])
[-2, 0, 1, 3, 4]
```

3. Pour que le tri soit stable, il faut s'assurer au moment de la fusion que, en cas d'égalité, les éléments de la moitié gauche soient placés avant ceux de la moitié droite.

Solution 3

1.
```
def split_list(L):
    """Renvoie le triplet Li,[p],Ls"""
    p, n= L[0], len(L)
    Li, Ls = [], []     #on initialise Li et Ls avec la liste \
    ↪ vide
    for i in range(1, n):        #on parcourt les éléments de \
    ↪ L[1:]
        if L[i] < p:              #et on les ajoute soit à Li
```

```
            Li.append(L[i])
        else:                   #soit à Ls
            Ls.append(L[i])
    return Li, [p], Ls
```

2.
```
def tri_rapide(L):
    """Renvoie le tri rapide de la liste L"""
    if len(L) < 2:
        return L
    else:
        Lg, Lm, Ld = split_list(L)
        return tri_rapide(Lg) + Lm + tri_rapide(Ld)
```

```
>>> tri_rapide([-2, 1, 0, 4, 3])
[-2, 0, 1, 3, 4]
```

3. Il est stable car, à chaque appel, les éléments de L égaux au pivot sont placés dans L*s* et seront placés après le pivot une fois la liste triée. Il n'est clairement pas en place.

4. **4.1)**
```
def Ranger(L,d,f):
 """ réordonne la liste L[d,f+1] de sorte à ce que les \
 ↪ éléments plus petit que le pivot soient situés avant \
 ↪ lui dans la liste finale"""
    p=L[d] # on mémorise le pivot
    m=d # place du dernier element connu <= au pivot
    for k in range(d+1,f+1):
        if L[k]<=p: # élément à placer avant le pivot
            m=m+1  # on se place sur un élément plus gd que p
            L[k],L[m]=L[m],L[k]
    L[m],L[d]=L[d],L[m] # on met le pivot en place
    return(m)
```

4.2)
```
def TriRapideAnnexe(L,d,f):
    if d>=f: return(L)
    else:
        m=ranger(L,d,f) # place du pivot
        TriRapideAnnexe(L,d,m-1) # on trie la sous liste \
        ↪ avant p
        TriRapideAnnexe(L,m+1,f) # on trie la sous liste \
        ↪ après p
```

4.3)
```
def TriRapide(L:list)->None:
    n=len(L)
    TriRapideAnnexe(0,n-1)
```

Solution 4

1.
```
def tri_comptage(L,a,b):
    """Renvoie le tri par comptage de la liste L dont les \
    ↪ éléments sont des entiers compris entre a et b"""
    C = (b-a+1)*[0]
    n = len(L)
    for i in range(n):  #parcours de L et remplissage de C
        C[L[i]-a] += 1
    Lt=[]
    for i in range(l):  #construction de la liste triée
        Lt += C[i]*[i+a]
    return Lt
```