TD n° 3 : Tris

1 Exercices autour des tris

Exercice 1.

1. Écrire une fonction `est_triee` prenant en argument une liste et renvoyant le booléen `True` si la liste est triée par ordre croissant et `False` sinon.
2. En donner la complexité en fonction de la longueur n de la liste passée en argument.

Exercice 2 (d'après CCP TSI 2020)**.**

1. Écrire une fonction **récursive** `recherche` qui
   - prend en argument une liste `tab` d'entiers triés par ordre croissant et un entier `x` ;
   - renvoie le booléen `True` si `x` est dans `tab` et `False` sinon ;
   - procède par dichotomie.
2. Quel est l'intérêt de ce type d'algorithme (on parlera de complexité) ?

Exercice 3. *Trier des points du plan*
On dispose de points dans le plan muni d'un repère orthonormé d'origine O. Ces points sont donnés par leurs coordonnées (x, y) représentées par le tuple `(x, y)`. Nous allons trier ces points en fonction de leur distance à O, de la plus petite à la plus grande.

1. Écrire une fonction `distance` qui prend en paramètre un tuple `point` constitué de deux nombres (il représente les coordonnées d'un point P) et qui renvoie le carré de la distance de ce point à O.
2. Écrire une fonction `compare` qui prend en paramètres deux tuples `p1` et `p2` représentant deux points P_1 et P_2 et qui renvoie `-1` si P_1 est plus proche de O que P_2, `1` si P_2 est plus proche de O que P_1 et `0` si les deux points sont équidistants de O.
3. Écrire une fonction `tri_points` qui prend en paramètre une liste de tuples de deux nombres représentant des points du plan et qui trie cette liste selon la distance entre ces points et l'origine. Pour cela, on adaptera l'algorithme du tri par insertion.

⋆ **Exercice 4.** Adapter un algorithme de tri par insertion avec une insertion qui utilise une recherche dichotomique.

Exercice 5 (d'après EPITA 2020)**.**
Nous travaillons ici avec des listes qui ne contiennent que des nombres entiers de 0 à 9.

1. Écrire une fonction `hist(L)` qui retourne un histogramme (sous forme d'une liste) des nombres présents dans la liste `L`.
   Rappel : l'histogramme `H` est une liste telle que `H[i]` est le nombre d'occurrences de la valeur `i`.
   Par exemple, dans la première application ci-dessous, il y a 5 valeurs 0, 3 valeurs 1, 3 valeurs 2, etc.

```
>>> hist([1,5,9,3,0,1,2,0,1,0,2,5,0,5,2,0])
[5, 3, 3, 1, 0, 3, 0, 0, 0, 1]

>>> hist([1,0,1,0,1,0,1,0,1])
[4, 5, 0, 0, 0, 0, 0, 0, 0, 0]
```

2. Utiliser la fonction `hist` pour écrire une fonction `tri_comptage` qui trie la liste `L` par ordre croissant (la fonction construit une nouvelle liste qui doit être retournée).
   Par exemple :

```
>>> tri_comptage([1,5,9,3,0,1,2,0,1,0,2,5,0,5,2,0])
[0, 0, 0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 5, 5, 5, 9]
```

3. Quelle est la complexité de cette fonction `tri_comptage` en fonction de la longueur n de la liste à trier ?

⋆⋆ **Exercice 6** (d'après ENS sélection internationale 2017)**.**
Nous avons en notre possession n écrous et n vis de tailles différentes. Chaque vis correspond à un écrou et un seul (et réciproquement). Les écrous et les vis sont presque de la même taille et il est impossible de dire si un écrou est plus grand qu'un autre ou si une vis est plus grande qu'une autre. Cependant, si on essaie de faire correspondre un écrou à une vis, la vis sera soit trop grande, soit trop petite, soit de la bonne taille exactement. On peut donc faire des comparaisons écrou-vis, et chaque comparaison donne un résultat parmi les trois résultats possibles.

1. Proposer un algorithme qui, étant donné un écrou, retrouve la vis correspondante. Donner sa complexité en nombre de comparaisons écrou-vis (en fonction de n).
2. Proposer un algorithme probabiliste inspiré du tri rapide pour associer correctement chaque écrou à sa vis.

2 Quand les concours vous demandent de connaître votre cours

Exercice 7 (CCP TSI 2020)**.**

1. Écrire une fonction `triinsertion` de tri par insertion d'une liste de nombres.
2. Comment peut-on adapter la fonction `triinsertion` à une liste de chaînes de caractères ?

Exercice 8 (d'après CCS MP 2018)**.**

1. Écrire une fonction d'entête

```
insertion(q: [float], j: int) -> None:
```

   prenant en paramètres un entier `j` et une liste de flottants `q` avec `q[:j]` déjà triée par ordre croissant, et insérant `q[j]` au bon endroit afin qu'après exécution `q[:j+1]` soit triée par ordre croissant.
2. En déduire une fonction d'entête

```
tri_insertion(q: [float]) -> None:
```

   permettant de trier en place une liste dans l'ordre croissant.
3. Rappeler la complexité de ce tri dans le pire et le meilleur cas. Que peut-on dire de la complexité si dans la liste `q` tous les éléments excepté peut-être un sont dans l'ordre croissant ?

Exercice 9 (d'après CCP PSI 2019)**.**
On souhaite trier une liste de listes à deux éléments selon la première valeur de chacune de ces listes. Par exemple, si au départ on a `T = [[1, 3], [2, 4], [0, 4]`, on doit obtenir `[[0, 4], [1, 3], [2, 4]]`. (On ne précise pas le comportement en cas d'égalité de la première valeur, les listes `[[0, 1], [0, 2]]` et `[[0, 2], [0, 1]]` sont considérées comme triées.)
On retient l'algorithme suivant :

```
1  def tri(T):
2      if len(T) <= 1:
3          return T
4      else:
5          m = len(T)//2
6          tmp1 = []
7          for x in range(m):
8              tmp1.append(T[x])
9          tmp2 = []
10         for x in range(m, len(T)):
11             tmp2.append(T[x])
12         return fct(tri(tmp1), tri(tmp2))
13
14 def fct(T1, T2):
15     if T1 == []:
16         ...
17     if T2 == []:
18         ...
19     if T1[0][0] < T2[0][0]:
20         return [T1[0]] + fct(T1[1:], T2)
21     else:
22         ...
```

1. Donner le nom de ce tri ainsi que son intérêt par rapport à un tri par insertion. Préciser un inconvénient du programme proposé par rapport à ce tri.
2. Préciser les lignes 16, 18 et 22 de la fonction `fct` pour que l'algorithme de tri soit fonctionnel.

Exercice 10 (d'après CCP TSI 2018)**.**
On considère la fonction :

```
1 def tri(L):
2     for i in range(1, len(L)):
3         if L[i] < L[i-1]:
4             p = i
5             while p > 0 and L[i] < L[p-1]:
6                 p = p - 1
7             x = L.pop(i)
8             L.insert(p, x)
9     return L
```

Quelle est l'algorithme de tri utilisé dans cette fonction ? Donner sa complexité dans le meilleur et le pire cas.

Exercice 11 (d'après Mines-Ponts 2016)**.**
Dans le but ultérieur de réaliser des études statistiques, on souhaite se doter d'une fonction tri. On se donne la fonction `tri` suivante :

```
1 def tri(L):
2     n = len(L)
3     for i in range(1, n):
4         j = i
5         x = L[i]
6         while 0 < j and x < L[j-1]:
7             L[j] = L[j-1]
8             j = j - 1
9         L[j] = x
```

1. Lors de l'appel `tri(L)` lorsque `L` est la liste `[5, 2, 3, 1, 4]`, donner le contenu de la liste `L` à la fin de chaque itération de la boucle `for`.
2. Soit `L` une liste non vide d'entiers ou de flottants. Montrer que « la liste `L[0:i+1]` est triée par ordre croissant à l'issue de l'itération `i` » est un invariant de boucle. En déduire que `tri(L)` trie la liste `L`.
3. Évaluer la complexité dans le meilleur et le pire des cas de l'appel `tri(L)` en fonction du nombre n d'éléments de `L`.
4. Citer un algorithme de tri plus efficace dans le pire des cas. Quelle en est la complexité dans le meilleur et dans le pire des cas ?
5. On souhaite, partant d'une liste constituée de couples (chaîne, entier), trier la liste par ordre croissant de l'entier associé suivant le fonctionnement suivant :

```
>>> L = [('Bresil', 76), ('Kenya', 26017), ('Ouganda', 8431)]
>>> tri_chaine(L)
>>> L
[('Bresil', 76), ('Ouganda', 8431), ('Kenya', 26017)]
```

   Écrire une fonction `tri_chaine` réalisant cette opération.

Exercice 12 (CCP TSI 2021)**.**
On considère la fonction suivante :

```
def tri(L):
    """ L est une liste de nombre réels. """
    for i in range(1, len(L)):
        x = L[i]
        j = i
        while j > 0 and x < L[j-1]:
            L[j] = L[j-1]
            j = j - 1
        L[j] = x
    return L
```

1. On exécute `tri(L)` avec `L = [3, 5, 2, 1]`. Combien y a-t-il d'itérations de la boucle `for` ? Donner la valeur de `L` à la fin de chaque itération de la boucle `for`.
2. Ce tri fonctionnerait-il pour une chaîne de caractères dont chaque caractère est un entier ? Justifier.
3. Quelle est la méthode de tri utilisée dans la fonction `tri` ? Donner sa complexité (en nombre de comparaison) dans le pire des cas et dans le meilleur des cas. *On justifiera soigneusement les complexités données.*

Exercice 13 (d'après CCP TSI 2017)**.**

1. Corriger les deux erreurs de l'algorithme de tri suivant.

```
def echange(L, i, j):
    """ échange deux valeurs d'une liste """
    L[i], L[j] = L[j], L[i]

def tri_1(liste):
    for i in range(liste):
        mini = i
        for j in range(i+1, len(liste)):
            if liste[j] > liste[mini]:
                mini = j
        echange(liste, i, mini)
```

2. En appelant n la taille de la liste, donner la complexité de la fonction `tri_1` en fonction de n dans le pire cas. Critiquer le résultat.
3. Commenter et expliquer chaque ligne d'instruction de la fonction `tri_2` suivante (on ne commentera pas `segmenter`).

```
1  def segmenter(T, i, j):
2      g = i+1
3      d = j
4      p = T[i]
5      while g <= d:
6          while d >= 0 and T[d] > p:
7              d = d-1
8          while g <= j and T[g] <= p:
9              g = g+1
10         if g < d:
11             echange(T, g, d)
12             d = d - 1
13             g = g + 1
14     k = d
15     echange(T, i, d)
16     return k
17
18 def tri_2(L, i, j):
19     if  i < j:
20         k = segmenter(L, i, j)
21         tri_2(L, i, k-1)
22         tri_2(L, k+1, j)
```

4. Donner l'instruction qui utilise cette fonction `tri_2` pour trier une liste donnée `tab`.
5. Quelle est la particularité de cette fonction `tri_2` ?

⋆ Modifier la fonction `tri_2` afin qu'elle retourne comme résultat le nombre d'appels récursifs de la fonction `tri_2`.