I Exercices

Q1 Écrire la fonction `est_trié(t:list)->bool` qui renvoie `True` si le tableau `t` de nombres est trié dans l'ordre croissant et `False` sinon.

Q2 *Tri pair-impair.* On se donne un tableau t.

- On parcourt tous les indices i **pairs** (à partir de 0 dans l'ordre croissant). Pour chacun d'eux, on compare $t[i]$ à $t[i+1]$. Si $t[i+1]$ est plus petit que $t[i]$, on permute $t[i]$ et $t[i+1]$. Sinon, on ne fait rien.
- Ensuite, on parcourt tous les indices i **impairs** (à partir de 1 dans l'ordre croissant). Pour chacun d'eux, on compare $t[i]$ à $t[i+1]$. Si $t[i+1]$ est plus petit que $t[i]$, on permute $t[i]$ et $t[i+1]$. Sinon, on ne fait rien.

On recommence ce qui précède tant que l'on fait des permutations.

(a) Écrire une fonction Python mettant en oeuvre le fonctionnement décrit ci-dessus.

(b) Évaluer sa complexité dans le meilleur et dans le pire des cas.

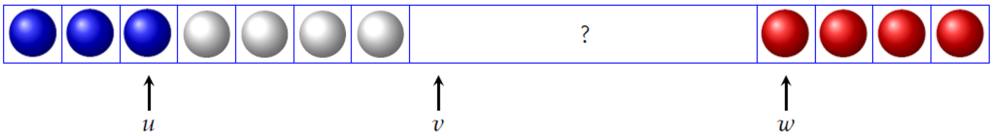
Q3 En se servant du tri par insertion, écrire une fonction `plus_frequent(t:list)->int` qui renvoie la valeur la plus fréquente de t. On notera qu'une fois triée, les valeurs les plus fréquentes sont côte à côte dans le tableau [1].

Q4 *Introduction à Timsort* [2].
L'algorithme *Timsort* est celui du `sort` de Python. Il n'est pas question ici de l'expliquer dans son exhaustivité, mais on en donne quelques bribes.

Soit t un tableau de longueur un multiple pair de 32 (on voit bien que l'on simplifie !).
Le principe consiste à "découper" (via un jeu d'indices) le tableau en blocs de longueur 32. Sur chacun d'eux, on établit un tri rapide. Ensuite, on fusionne deux à deux les tableaux par tri-fusion [3].

Écrire une procédure `Timsort(t:list)->int` qui fonctionne selon le principe décrit. On réutilisera telles quelles les fonctions déjà écrites `tri_rapide` et `fusion`.

Q5 *Segmentation et tri rapide : le drapeau tricolore* [4].
On considère un tableau t contenant des objets possédant une couleur qui ne peut être que "*bleu*", "*blanc*" ou "*rouge*". On souhaite trier ce tableau de sorte que les éléments bleus soient placés en tête, ensuite les éléments blancs, puis enfin les éléments rouges.



Écrire une procédure `drapeau(t:list)->None` qui trie le tableau selon les couleurs en utilisant les indices u, v, w comme sur la figure ci-dessus.

1. Il s'agit d'un sujet technique car un dictionnaire ferait mieux l'affaire.
2. établi en 2002 par Tim Peters, ingénieur software
3. Telle quelle la fusion n'est pas optimisée : l'algorithme réel établit un ordre entre les tableaux avant de les fusionner.
4. Le problème initial est attribué à Edsger Dijkstra : il s'agit donc à l'origine du drapeau néerlandais. On préférera cependant « bleu », « blanc » et « rouge ».

Corrigé

Q1

```
71 def est_trié(t:list)-> bool:
72     n = len(t)
73     for i in range(n-1):
74         if t[i] > t[i+1]:
75             return False
76     return True
```

Q2

(a) Tri pair-impair.

```
90 def tri_pair_impair(t:list)->None:
91     n = len(t)
92 
93     trié = False # cela permet d'entrer dans la boucle
94     while not trié:
95         trié = True
96         for i in range(0,n,2): # le pas de 2 permet de traiter les
       indices pairs
97             if i<= n-2 and t[i] > t[i+1]: # on veille à ce que t[i
       +1] soit défini
98                 t[i],t[i+1] = t[i+1],t[i]; trié = False
99         for i in range(1,n,2): # le pas de 2 permet de traiter les
       indices pairs
100             if i<= n-2 and t[i] > t[i+1]:
101                 t[i],t[i+1] = t[i+1],t[i]; trié = False
```

(b) Dans le meilleur des cas, le tableau est trié dans l'ordre croissant.
On fait n tests, que ce soit pour les indices pairs ou pour les indices impairs.
Donc la complexité temporelle dans le meilleur cas est en $\mathcal{O}(n)$.

Dans le pire des cas, le tableau est trié dans l'ordre décroissant.
On fait n boucles : en effet, $\lfloor n/2 \rfloor$ permutations amènent le plus petit nombre de droite situé à un indice pair en 0 ; de même $\lceil n/2 \rceil$ permutations amènent le plus petit nombre de droite situé à un indice impair en 1.
Pour chaque boucle, on effectue avec n tests.
Donc la complexité temporelle dans le pire des cas est en $\mathcal{O}(n^2)$.

Q3

```
155 def plus_frequent(t:list)->int:
156     assert len(t)>=1, 'le tableau est vide'
157 
158     tri_insertion(t)
159     v = t[0] # la valeur courante
160     c = 1    # et son nombre d'occurrences
161     mv = v   # la valeur la plus fréquente
162     m = 1    # et son nombre d'occurrences
163     for i in range(1,len(t)):
164         if t[i] == v:
165             c += 1
166             if c > m: # on vérifie si c est le maximum
167                 m = c; mv =v
168         else : # on passe à une autre valeur
169             v = t[i]; c = 1
170     return mv
171 
172 plus_frequent([1,2,4,4,3,1,2,4,1,4,5])
```

Q4

```
def Timsort(t:list)->None:
    n = len(t)

    N = 32
    k = n//N # on sait que n est un multiple de N=32
    # tri rapide par blocs de N = 32
    for i in range(1,k+1):
        tri_rapide(t[(i-1)*N: i*N])
    # tri fusion des blocs : n multiple pair de 32
    for i in range(1,k//2+1):
        fusion(t[(i-1)*N, i*N], t[i*N:(i+1)*N])

t=list(range(64))
Fisher_Yates(t);print(t);print()
Timsort(t); print(t)
```

Q5

```
def drapeau(t:list)->None:
    u,v,w = -1,0,len(t)
    while v < w:
        if t[v] == 'bleu':
            t[u+1],t[v] = t[v],t[u+1]
            u += 1; v += 1
        elif t[v] == 'rouge':
            t[w-1],t[v] = t[v],t[w-1]
            w -= 1
        else: # t[v] == 'blanc'
            v += 1
```