CORRIGÉ - TP : TRIS

1 Tri par sélection

1.

```
def minimum1(L):
    p, m =  0, L[0]
    for i in range(1, len(L)):
        if L[i]<m:
            p, m = i, L[i]
    return p, m
```

2.

```
def minimum2(L,deb):
    p, m = deb, L[deb]
    for i in range(deb+1, len(L)):
        if L[i]<m:
            p, m = i, L[i]
    return p,m
```

3.

```
def tri_selection(L):
    n = len(L)
    for i in range(0,n-1):
      #inutile de trier le dernier element de la liste
      p, m =minimum2(L,i)
      L[i], L[p] = L[p], L[i]
```

4. Le tri par sélection est tri comparatif et en place.
   Il n'est pas stable : appliquons le à la liste $[\boxed{4}, 4, 1]$, où j'ai encadré un 4 pour les distinguer de l'autre. Lors du premier passage, on échange le premier 4 avec le 1 et la liste devient $[1, 4, \boxed{4}]$ ce qui ne changera plus. Les deux 4 ne sont donc pas dans le même ordre qu'au départ, le tri n'est pas stable.
5. On note n la taille de la liste. Pour chaque passage dans la boucle, `minimum2` fait $n-i-1$ comparaisons. Le nombre maximal de comparaisons est donc :

$$N = \sum_{i=0}^{n-2}(n-i-1) = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2}.$$

2 Tri par insertion

1.

```
def tri_insertion(L):
    n = len(L)
    for k in range(1, n):
      i = k #indice de l'element a inserer
        # les elements entre 0 et k-1 sont tries
        while i>0 and L[i-1] > L[i]:
            L[i], L[i-1] = L[i-1], L[i] # decale les elements de la liste
            i = i-1
```

Variante :

2.

```
def tri_insertion(L):
    n = len(L)
    for k in range(1, n):
      x = L[k]
        i = k-1 #indice du predecesseur de x
        # les elements entre 0 et k-1 sont tries
        while i>=0 and L[i]>x:
            L[i+1] = L[i] # decale les elements de la liste
            i = i-1
        L[i+1] = x # on met la valeur a inserer dans le trou
```

3. Le tri par insertion est un tri comparatif, stable et en place.

3 Tri par partition-fusion

1. On propose deux versions : l'une itérative et l'autre récursive :

```
def fusion_ite(L1, L2):
    n1,n2=len(L1),len(L2)
    i1,i2=0,0
    L=[]
    while i1<n1 and i2<n2:
        if L1[i1] <= L2[i2]:
            L.append(L1[i1])
            i1+=1
        else:
            L.append(L2[i2])
            i2+=1
    L = L + L1[i1:n1] + L2[i2:n2]
    return L
```

et

```python
def fusion_rec(L1, L2):
    if L1==[]:
        return L2
    elif L2==[]:
        return L1
    x , y = L1[-1] , L2[-1]
    if x < y:
        L2.pop()
        L = fusion_rec(L1,L2)
        L.append(y)
        return L
    else:
        L1.pop()
        L = fusion_rec(L1,L2)
        L.append(x)
        return L
```

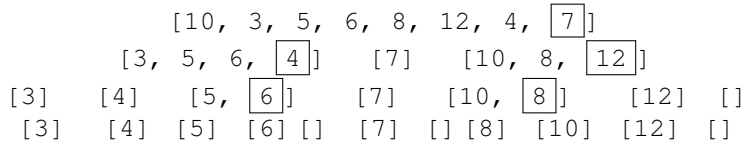
2. On écrit une fonction récursive sur le principe « diviser pour régner », faisant appel à la fonction `fusion` précédente :

```python
def tri_fusion(L):
    n = len(L)
    if n <=1:
        return L
    else:
       m = n//2
       L1 = tri_fusion(L[0:m])
       L2 = tri_fusion(L[m:n])
    return fusion(L1,L2)
```

3. C'est un tri comparatif, stable mais qui n'est pas en place.

4 Tri rapide

1. On applique l'algorithme à la liste `L = [10, 3, 5, 6, 8, 12, 4, 7]`, le pivot à chaque étape est encadré :



2.

```python
def repartition(L):
    G = []
    M = []
    D = []
    pivot = L[-1]
    for i in range(len(L)):
        x = L[i]
        if x < pivot:
            G.append(x)
        elif x > pivot:
            D.append(x)
        else:
            M.append(x)
    return G, M, D
```

3. On en déduit :

```python
def tri_rapide(L):
    if len(L) <= 1:
        return L
    G, M, D = repartition(L)
    return tri_rapide(G) + M + tri_rapide(D)
```

4. Si on choisit toujours le dernier élément de la liste pour pivot, pour une liste triée, la première sous-listes contiendra tous les éléments de la liste sauf ceux qui sont égaux au dernier élément, et la troisième sera vide. Cela entraine une dissymétrie qui rend notre algorithme moins efficace, l'idéal étant d'avoir deux sous-listes à trier de tailles comparables. Pour remédier à cela, on pourrait choisir comme pivot l'élément au milieu, ou bien un élément aléatoire de la liste.

5. On souhaite à présent écrire une implémentation en place du tri rapide.

   (a)

```python
def repartition_en_place(L, a, b):
    pivot = L[b]
    ind_p = a
    for i in range(a,b):
        if L[i]<=pivot:
            L[i], L[ind_p] = L[ind_p], L[i]
            ind_p +=1
    L[b], L[ind_p] = L[ind_p], L[b]
    return ind_p
```

   (b)

```python
def tri_sous_liste(L, a, b):
    if b > a:
        ind_p = pivot(L,a,b)
        repartition_en_place(L, a, ind_p-1)
        repartition_en_place(L, ind_p+1, b)
    # si a=b (liste a un element), ou b<a (liste vide) la sous-liste
    # est triee : condition d'arret.
```

   (c)

```python
def tri_rapide_en_place(L):
    tri_sous_liste(L,0,len(L)-1)
```

5 Tri par comptage

1.

```
def tri_comptage(L):
    n = len(L)
    L_triee = []
    #recherche des minimum et maximum de la liste
    mini, maxi = L[0], L[0]
    for i in range(1,n):
        if L[i] < mini:
            mini = L[i]
        if L[i] > maxi:
            maxi = L[i]

    D = [0]*(maxi-mini+1) #cree un tableau initialise a zero pour les
    #occurences
    for i in range(n): # on parcourt L pour incrementer D
        D[L[i]-mini]+=1

    for i in range(maxi-mini+1): # on parcourt D pour creer la liste triee
        if D[i] > 0:
            for j in range(int(D[i])):
                L_triee.append(i+mini)
    return L_triee
```

2. Le tri par comptage n'est pas un tri comparatif, il n'est pas en place. Il n'est pas stable.

Pour visualiser les différents tri en dansant :
`https://www.laboiteverte.fr/algorithmes-tri-visualises-danses-folkloriques/`