TP9 - Méthodes de tri

Un algorithme de tri est un algorithme qui permet d'organiser une collection d'objets selon un ordre déterminé. Le tri permet notamment de faciliter les recherches ultérieures d'un élément dans une liste (recherche dichotomique, vu au TP6).

On s'intéresse ici à des méthodes de tri d'une liste de valeurs numériques. Celle-ci est implémentée sous la forme d'un tableau à une dimension. Les algorithmes de tri sont utilisés dans de très nombreuses situations. Ils sont en particulier utiles à de nombreux algorithmes plus complexes dont certains algorithmes de recherche, comme la recherche dichotomique. Ils peuvent également servir pour mettre des données sous forme canonique ou les rendre plus lisibles pour l'utilisateur.

Nous utiliserons donc une liste de 20 nombre entiers distincts, `T`, pour tous les exemples de ce TP.

La plupart des algorithmes de tri sont fondés sur des comparaisons successives entre les données pour déterminer la permutation correspondant à l'ordre croissant des données. Nous appellerons tri comparatif un tel tri.

La complexité de l'algorithme a alors le même ordre de grandeur que le nombre de comparaisons entre les données faites par l'algorithme.

Tous les algorithmes de ce TP sont des tris comparatifs, mis à part le dernier, qui est un tri basé sur la structure des données, nous l'expliquerons dans la dernière section.

I Algorithmes quadratiques

Soit une liste T de taille n. On peut utiliser le `shuffle` du module `random` pour mélanger la liste.

```
from random import *
T = [i for i in range(20)]
shuffle(T)
```

I.1 Tri par sélection (comparatif)

Le but est simple :

— parcourir la liste en plaçant le plus petit élément à la première position ;

— puis on recherche le second plus petit élément que l'on va placer en deuxième position ;

— etc.., jusqu'à ce que le tableau soit entièrement trié.

Il faut donc une première boucle pour parcourir les positions de la liste, et à l'intérieur, une algorithme de recherche du plus petit élément dans le **reste** de la liste. Pour échanger les positions, on peut utiliser une fonction *echange* utilisant le principe d'affectation parallèle :

```
def echange(l,i,j):
    # echange 2 valeurs d'une liste
    l[i],l[j] = l[j], l[i]
```

Q1. Proposer une fonction `tri_selection(L)` qui prend en argument une liste de nombre `L` et qui renvoie la liste `L` triée.

Voici un exemple sur une liste mélangée : `[1, 0, 13, 4, 10, 9, 12, 2, 5, 18, 16, 11, 6, 19, 15, 8, 17, 14, 3, 7]` Et les étapes de l'algorithme :

```
[0, 1, 13, 4, 10, 9, 12, 2, 5, 18, 16, 11, 6, 19, 15, 8, 17, 14, 3, 7]
[0, 1, 13, 4, 10, 9, 12, 2, 5, 18, 16, 11, 6, 19, 15, 8, 17, 14, 3, 7]
[0, 1, 2, 4, 10, 9, 12, 13, 5, 18, 16, 11, 6, 19, 15, 8, 17, 14, 3, 7]
[0, 1, 2, 3, 10, 9, 12, 13, 5, 18, 16, 11, 6, 19, 15, 8, 17, 14, 4, 7]
[0, 1, 2, 3, 4, 9, 12, 13, 5, 18, 16, 11, 6, 19, 15, 8, 17, 14, 10, 7]
[0, 1, 2, 3, 4, 5, 12, 13, 9, 18, 16, 11, 6, 19, 15, 8, 17, 14, 10, 7]
[0, 1, 2, 3, 4, 5, 6, 13, 9, 18, 16, 11, 12, 19, 15, 8, 17, 14, 10, 7]
[0, 1, 2, 3, 4, 5, 6, 7, 9, 18, 16, 11, 12, 19, 15, 8, 17, 14, 10, 13]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 18, 16, 11, 12, 19, 15, 9, 17, 14, 10, 13]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 16, 11, 12, 19, 15, 18, 17, 14, 10, 13]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 19, 15, 18, 17, 14, 16, 13]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 19, 15, 18, 17, 14, 16, 13]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 19, 15, 18, 17, 14, 16, 13]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 18, 17, 14, 16, 19]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 18, 17, 15, 16, 19]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 18, 16, 19]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18, 17, 19]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Q2. Évaluer la complexité de ce tri, dans le pire des cas et dans le meilleur des cas.

Certains algorithmes de tri sont mis en illustration avec des danses folkloriques sur ce site : `http://www.laboiteverte.fr/algorithmes-tri-visualises-danses-folkloriques`
Q3. Observer la danse hispanique du *Select sort*, et modifier légèrement votre fonction de tri par sélection.

Une notion importante sur les algorithmes de tri est leur stabilité. Un tri est dit stable s'il préserve l'ordonnancement initial des éléments que l'ordre considère comme égaux. Pour définir cette notion, il est nécessaire que la collection à trier soit ordonnancée d'une certaine manière (ce qui est souvent le cas pour beaucoup de structures de données, par exemple pour les listes ou les tableaux).

Q4. Tester la stabilité du tri par sélection avec l'animation de ce site : `http://lwh.free.fr/pages/algo/tri/stabilite_tri.html`

I.2 Tri par insertion (comparatif)

Comme vous l'avez vu précédemment, le tri par sélection a une complexité quadratique quelque le soit le cas étudié. Le tri par insertion va permettre de ne pas faire de double boucle dans le cas d'une liste ordonnée. De plus, le tri par insertion est stable.
Voici le principe de l'algorithme :

- on commence par affecter en tant que *clef* le deuxième élément de la liste ;
- on la compare à la première valeur et on l'insère avant ou après la première selon le cas ;
- à l'étape i, le ième élément est inséré à sa place dans la partie de liste déjà triée. Il faut partir de la fin de cette partie de liste triée, c'est à dire l'élément $i-1$, comparer les valeurs précédentes à cette ième valeur et s'arrêter si l'on rencontre une valeur plus petite que la ième ;
- chaque élément dont la valeur est plus grande que la valeur i est affectée à la position suivante, ainsi toutes la valeurs plus grande que la ième « remontent »
- à la fin, on insère la ième valeur à sa place : la liste contient alors i valeurs triées.

— On procède ainsi de suite jusqu'à la dernière valeur.

Q5. Proposer une fonction `tri_insertion(L)` qui prend en argument une liste de nombre `L` et qui renvoie la liste `L` triée.

Voici un exemple sur une liste mélangée : `[15, 14, 4, 13, 16, 17, 3, 11, 7, 1, 8, 18, 10, 19, 9, 2, 12, 5, 0, 6]`

Avec quelques étapes de l'algorithme (on remarquera la « réaffectation » quand les valeurs remontent :

```
[15, 15, 4, 13, 16, 17, 3, 11, 7, 1, 8, 18, 10, 19, 9, 2, 12, 5, 0, 6]
[14, 15, 15, 13, 16, 17, 3, 11, 7, 1, 8, 18, 10, 19, 9, 2, 12, 5, 0, 6]
[14, 14, 15, 13, 16, 17, 3, 11, 7, 1, 8, 18, 10, 19, 9, 2, 12, 5, 0, 6]
[4, 14, 15, 15, 16, 17, 3, 11, 7, 1, 8, 18, 10, 19, 9, 2, 12, 5, 0, 6]
[4, 14, 14, 15, 16, 17, 3, 11, 7, 1, 8, 18, 10, 19, 9, 2, 12, 5, 0, 6]
[4, 13, 14, 15, 16, 17, 17, 11, 7, 1, 8, 18, 10, 19, 9, 2, 12, 5, 0, 6]
[4, 13, 14, 15, 16, 16, 17, 11, 7, 1, 8, 18, 10, 19, 9, 2, 12, 5, 0, 6]
[4, 13, 14, 15, 15, 16, 17, 11, 7, 1, 8, 18, 10, 19, 9, 2, 12, 5, 0, 6]
[4, 13, 14, 14, 15, 16, 17, 11, 7, 1, 8, 18, 10, 19, 9, 2, 12, 5, 0, 6]
[4, 13, 13, 14, 15, 16, 17, 11, 7, 1, 8, 18, 10, 19, 9, 2, 12, 5, 0, 6]
[4, 4, 13, 14, 15, 16, 17, 11, 7, 1, 8, 18, 10, 19, 9, 2, 12, 5, 0, 6]
[3, 4, 13, 14, 15, 16, 17, 17, 7, 1, 8, 18, 10, 19, 9, 2, 12, 5, 0, 6]

...
...

[0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 19]
[0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 18, 19]
[0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 17, 18, 19]
[0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 16, 17, 18, 19]
[0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 15, 16, 17, 18, 19]
[0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 14, 15, 16, 17, 18, 19]
[0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 13, 14, 15, 16, 17, 18, 19]
[0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 12, 13, 14, 15, 16, 17, 18, 19]
[0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[0, 1, 2, 3, 4, 5, 7, 8, 9, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[0, 1, 2, 3, 4, 5, 7, 8, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[0, 1, 2, 3, 4, 5, 7, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Q6. Évaluer la complexité de ce tri, dans le pire des cas et dans le meilleur des cas.

Une deuxième version est proposée dans le corrigé, où l'on compare la ième carte au début de la liste triée et non en partant de la fin.

Q7. Vous pouvez observer la danse roumaine du *Insert sort* sur le site : `http://www.laboiteverte.fr/algorithmes-tri-visualises-danses-folkloriques`

Q8. Tester la stabilité du tri par insertion avec l'animation de ce site : `http://lwh.free.fr/pages/algo/tri/stabilite_tri.html`

Il existe une version du tri par insertion qui compare au début de la liste déjà triée plutôt que de remonter dans la liste. Le code est donné dans la correction.

II Algorithmes récursifs

II.1 Tri par partition-fusion (comparatif)

La méthode de tri fusion pour un tableau de données est la suivante :

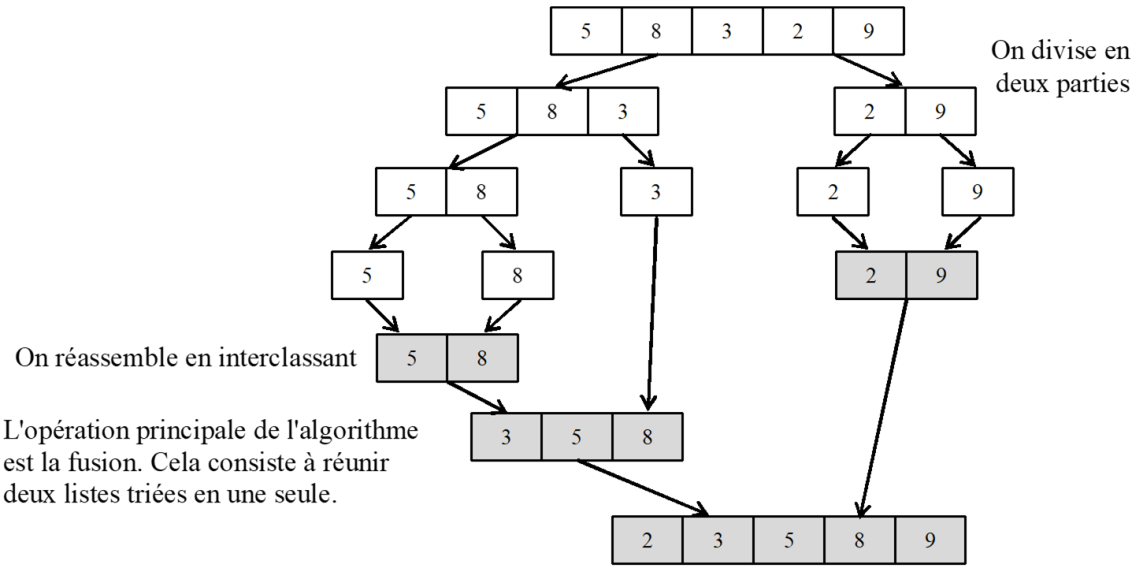
— On coupe en deux parties à peu près égales les données à trier.

— On trie les données de chaque partie par la méthode de tri fusion.

— On fusionne les deux parties en interclassant les données.

L'algorithme est donc récursif. Il fait partie des algorithmes « diviser pour régner » . La récursivité s'arrête car on finit par arriver à des listes composées d'un seul élément et le tri est alors trivial.

Un exemple de la méthode est donné sur la figure ci-dessous avec une liste simple :



Voici la fonction `fusion_IT(T,g,d,m)` qui permet de fusionner la partie gauche et droite d'une liste de manière itérative.

```
def fusion_IT(T,g,d,m):
    G=T[g:m+1]
    D=T[m+1:d+1]
    for k in range(g,d+1): #on replace dans T avec le bon ordre
        if len(G)>0 and len(D)>0 :
            if G[0]<=D[0]:
                T[k]=G.pop(0)
            else:
                T[k]=D.pop(0)
        elif len(G)==0 and len(D)>0: T[k]=D.pop(0)
        elif len(D)==0 and len(G)>0 : T[k]=G.pop(0)
```

Q9. Écrire la fonction **récursive** `tri_fusion(T)` qui prend en argument une liste quelconque de nombre `T` et qui renvoie la liste triée. On utilisera bien évidemment la fusion des moitiés de listes, jusqu'à obtenir des listes à 1 seul élément.
Un exemple plus complet est donné sur la liste suivante : `T=[12, 3, 19, 5, 10, 1, 17, 4, 8, 15, 14, 7, 6, 18, 11, 2, 9, 0, 13, 16]`

```
[12, 3, 19, 5, 10, 1, 17, 4, 8, 15, 14, 7, 6, 18, 11, 2, 9, 0, 13, 16]
# divisions successives "gauche" et "droite"
[12, 3, 19, 5, 10, 1, 17, 4, 8, 15]
[12, 3, 19, 5, 10]
[12, 3]
[19, 5, 10]
[5, 10]
[1, 17, 4, 8, 15]
[1, 17]
[4, 8, 15]
[8, 15]

[14, 7, 6, 18, 11, 2, 9, 0, 13, 16]
[14, 7, 6, 18, 11]
[14, 7]
[6, 18, 11]
[18, 11]
[2, 9, 0, 13, 16]
[2, 9]
[0, 13, 16]
[13, 16]
#fusion finale
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Q10. Exprimer la complexité de ce tri avec une relation de récurrence.

Q11. On peut également écrire la fonction `fusion` de manière **récursive**. Écrire la fonction **récursive** `fusion(L1, L2)` qui prend en argument deux listes **triées** de nombre `L1,L2` et qui renvoie : `L1` s'il n'y a plus d'éléments dans `L2` et inversement, et qui renvoie la liste fusionnée de `L1` et `L2` en comparant les premiers éléments successifs des deux listes. L'intérêt de cette fonction est surtout de ne pas introduire de variables locales `G` et `D` et donc de diminuer l'espace mémoire utilisé.
Q12. Écrire ensuite la fonction **récursive** `tri_fusion(L)` qui prend en argument une liste quelconque de nombre `L` et qui renvoie la liste triée. On utilisera bien évidemment la fusion des moitiés de liste, jusqu'à obtenir des listes à 1 seul élément.

Q13. Vous pouvez observer la danse du *Merge sort* sur le site : `http://www.laboiteverte.fr/algorithmes-tri-visualises-danses-folkloriques`

Q14. Tester la stabilité du tri fusion avec l'animation de ce site : `http://lwh.free.fr/pages/algo/tri/stabilite_tri.html`

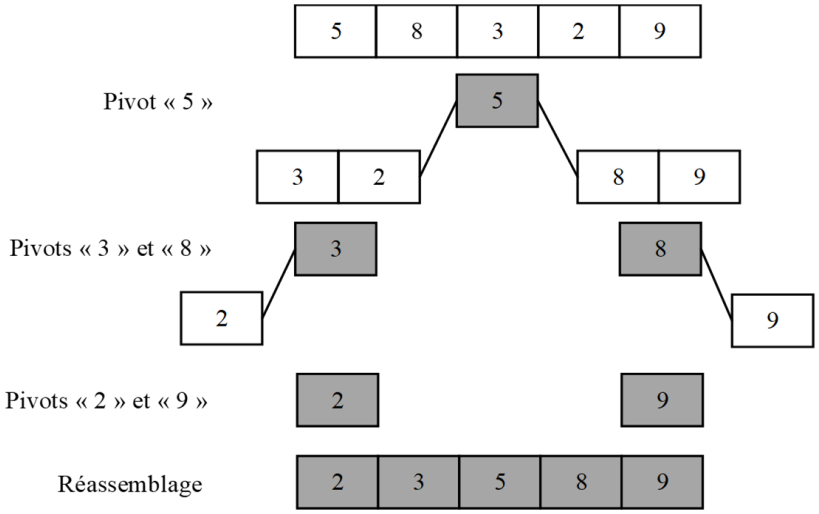
II.2 Tri rapide (comparatif)

L'algorithme fait aussi parti de la catégorie des algorithmes « diviser pour régner ». A chaque appel de la fonction de tri, le nombre de données à traiter est diminué de un. C'est-à-dire que l'on ne traite plus l'élément appelé « pivot » dans les appels de fonction ultérieurs, il est placé à sa place définitive dans le tableau. Le tableau de valeurs est ensuite segmenté en deux parties :

— dans un premier tableau, toutes les valeurs numériques sont inférieures au « pivot »,

— dans un second tableau, toutes valeurs numériques sont supérieures au « pivot ». L'appel de la fonction de tri est récursif sur les tableaux segmentés.

Un exemple de la méthode est donné sur la figure ci-dessous avec une liste simple :



Q15. Écrire la fonction **récursive** `tri_rapide(T)` qui prend en argument une liste quelconque de nombre `T` et qui renvoie la liste triée : si la liste est vide, on renvoie une liste vide, sinon il faut construire la liste des éléments plus petits et celle des éléments plus grands que le pivot et replacer ces deux listes de part et d'autre du pivot.
Un exemple plus complet est donné sur la liste suivante : `[4, 11, 15, 13, 14, 9, 6, 0, 12, 17, 18, 3, 1, 10, 7, 16, 2, 19, 8, 5]`

```
#liste des petits liste des grands
[0, 3, 1, 2]       [11, 15, 13, 14, 9, 6, 12, 17, 18, 10, 7, 16, 19, 8, 5]
[]       [3, 1, 2]
[1, 2]       []
[]       [2]
[]       []
[9, 6, 10, 7, 8, 5]       [15, 13, 14, 12, 17, 18, 16, 19]
[6, 7, 8, 5]       [10]
[5]       [7, 8]
[]       []
[]       [8]
[]       []
[]       []
[13, 14, 12]       [17, 18, 16, 19]
[12]       [14]
[]       []
[]       []
[16]       [18, 19]
[]       []
[]       [19]
[]       []
#liste finale
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Q16. Exprimer la complexité de ce tri avec une relation de récurrence, dans le pire des cas et dans le meilleur des cas.

Q17. Vous pouvez observer la danse du *Quick sort* sur le site :

`http://www.laboiteverte.fr/algorithmes-tri-visualises-danses-folkloriques`

Q18. Tester la stabilité du tri rapide avec l'animation de ce site : `http://lwh.free.fr/pages/algo/tri/stabilite_tri.html`

Cette version du tri rapide est une version condensée, qui n'est pas en place (définition dans la dernière section). Il existe une solution pour améliorer ce tri, une fonction annexe permet de partitionner et de replacer le pivot à sa place dans une partition de liste. La fonction *segmente*, proposée, ci-dessous, permet de positionner le pivot entre les valeurs plus petites et les valeurs plus grandes que lui, et elle renvoie la position finale du pivot dans la partie de liste `[i,j]`. Pour ceci, on échange les positions des valeurs plus grandes et plus petites jusqu'à avoir tester toutes les valeurs.

```
def segmente(T,i,j):
    p=T[i] # le pivot est la première valeur de cette partie de liste
    g=i+1  # g est l'indice des valeurs plus grandes que le pivot
    d=j    # d est l'indice des valeurs plus petites que le pivot
    while g<=d :
        while d>=i and T[d]>p:
            d=d-1
        while g<=j and T[g]<=p:
            g=g+1
        if g<d: # on échange jusqu'à ce que les indices de droite et de gauche se croisent
            echange(T,g,d)
            d=d-1
            g=g+1
    echange(T,i,d) # la dernière position de d est la position finale du pivot
    return d
```

Q19. Écrire alors une nouvelle fonction **récursive** *tri_rapide2(L,i,j)* qui permet de faire le tri rapide en utilisant la fonction *segmente*.

III Tri par comptage

Le tri comptage (*Counting sort* en anglais), appelé aussi tri casier, est un algorithme de tri par dénombrement qui s'applique sur des valeurs entières. A la différence des précédents algorithmes de tri comparatifs, celui-ci n'utilise aucunes comparaisons, c'est son intérêt.

Le principe repose sur la construction de l'histogramme des données, puis le balayage de celui-ci de façon croissante, afin de reconstruire les données triées. Ici, la notion de stabilité n'a pas réellement de sens, puisque l'histogramme factorise les données, plusieurs éléments identiques seront représentés par un unique élément quantifié. Ce tri ne peut donc pas être appliqué sur des structures complexes, et il convient exclusivement aux données constituées de nombres entiers compris entre une borne min et une borne max connues. Dans un souci d'efficacité, celles-ci doivent être relativement proches l'une de l'autre, ainsi que le nombre d'éléments doit être relativement grand.

Dans cette configuration, et avec une distribution de données suivant une loi uniforme discrète, ce tri est le plus rapide (on troque, en quelque sorte, du temps de calcul contre de la mémoire). La restriction très particulière imposée à ses valeurs d'entrée en fait un tri en temps linéaire, alors qu'un tri par comparaisons optimal nécessite un nombre d'opérations de l'ordre de $n.ln(n)$.
L'algorithme présenté ici n'est pas la seule solution au problème mais elle permet de bien comprendre son fonctionnement.

— Il est nécessaire de connaitre la borne supérieure des valeurs entières de la liste à triée `T`. Il faut alors créer une liste de comptage, un tableau contenant n éléments, n étant la valeur maximale dans la liste à triée, qui permet de compter les occurrences de chacune des valeurs dans la liste `T`.

— Pour chaque indice de `T`, on place ensuite les valeurs contenues dans la liste de comptage.

Un exemple est présenté sur la liste suivante :`T = [2, 3, 4, 4, 5, 0, 1, 2, 3, 6, 5, 4, 1, 8, 9, 9, 7, 8, 8, 4, 6, 4, 5, 2, 3, 4]`

```
tabComptage [1, 2, 3, 3, 6, 3, 2, 1, 3, 2]
[0, 3, 4, 4, 5, 0, 1, 2, 3, 6, 5, 4, 1, 8, 9, 9, 7, 8, 8, 4, 6, 4, 5, 2, 3, 4]
[0, 1, 1, 4, 5, 0, 1, 2, 3, 6, 5, 4, 1, 8, 9, 9, 7, 8, 8, 4, 6, 4, 5, 2, 3, 4]
[0, 1, 1, 2, 2, 2, 1, 2, 3, 6, 5, 4, 1, 8, 9, 9, 7, 8, 8, 4, 6, 4, 5, 2, 3, 4]
[0, 1, 1, 2, 2, 2, 3, 3, 3, 6, 5, 4, 1, 8, 9, 9, 7, 8, 8, 4, 6, 4, 5, 2, 3, 4]
[0, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 4, 4, 4, 9, 7, 8, 8, 4, 6, 4, 5, 2, 3, 4]
[0, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 4, 4, 4, 5, 5, 5, 8, 4, 6, 4, 5, 2, 3, 4]
[0, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 4, 4, 4, 5, 5, 5, 6, 6, 6, 4, 5, 2, 3, 4]
[0, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 4, 4, 4, 5, 5, 5, 6, 6, 7, 4, 5, 2, 3, 4]
[0, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 4, 4, 4, 5, 5, 5, 6, 6, 7, 8, 8, 8, 3, 4]
[0, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 4, 4, 4, 5, 5, 5, 6, 6, 7, 8, 8, 8, 9, 9]
[0, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 4, 4, 4, 5, 5, 5, 6, 6, 7, 8, 8, 8, 9, 9]
```

Q20. Écrire la fonction `tri_comptage(T,borneSup)` qui prend en argument une liste quelconque de nombre `T` compris entre `0` et `borneSup` et qui renvoie la liste triée, en utilisant la table de comptage.

Q21. Évaluer la complexité de votre algorithme dans le pire et dans le meilleur des cas.

IV Synthèse

Un tri est dit en place s'il n'utilise qu'un nombre très limité de variables et qu'il modifie directement la structure qu'il est en train de trier. Ceci nécessite l'utilisation d'une structure de donnée adaptée (un tableau par exemple). Ce caractère peut être très important si on ne dispose pas de beaucoup de mémoire.

Toutefois, on ne déplace pas, en général, les données elles-mêmes, mais on modifie seulement des références (ou pointeurs) vers ces dernières.

Voici un tableau de synthèse des algorithmes vus dans ce TP :

Nom	Pire des cas	Meilleur	(En moyenne)	Mémoire (en place)	Stable
Tri par sélection	$O(n^2)$	$O(n^2)$	$O(n^2)$	1 (oui)	non
Tri par insertion	$O(n^2)$	$O(n)$	$O(n^2)$	1 (oui)	oui
Tri fusion	$O(n.ln(n))$	$O(n.ln(n))$	$O(n.ln(n))$	n (non)	oui
Tri rapide	$O(n^2)$	$O(n.ln(n))$	$O(n.ln(n))$	$ln(n)$ (non)	non
Tri par comptage	$O(n+k)$	$O(n)$	$O(n+k)$	n (non)	/