Tris

Deux pages web

Un premier lien www.toptal.com/developers/sorting-algorithm : comparaison de différents tris
Un autre lien http://lwh.free.fr/pages/algo/tri/tri.htm : animation de différents tris.

Petite motivation

Nous avons vu qu'il est plus efficace de chercher un élément dans une liste *triée*, plutôt que dans une liste « en désordre ». On rappelle que :

— l'algorithme de dichotomie (qui nécessite que la liste soit triée) possède une complexité en $O(\ln n)$.
— l'algorithme naïf de parcours séquentiel (qui ne nécessite pas que la liste soit triée) possède une complexité en $O(n)$.

Nous allons étudier divers algorithmes permettant de trier une liste.
Les tris de ce TP seront toujours dans l'ordre croissant, en gardant à l'esprit qu'il est immédiat de passer à la version décroissante.

Définition

Un algorithme de tri est un algorithme qui permet d'organiser une collection d'objets selon une relation d'ordre déterminée. Les objets à trier sont des éléments d'un ensemble muni d'un ordre total. Il est par exemple fréquent de trier des entiers selon la relation d'ordre usuelle « est inférieur ou égal à » ou des chaînes de caractère selon l'ordre lexicographique.
La collection à trier est souvent donnée sous forme de tableau ou de liste, afin de permettre l'accès direct aux différents éléments de la collection.

Un tri est dit **en place** s'il modifie directement la structure qu'il est en train de trier.
Ce caractère peut être très important si on ne dispose pas de beaucoup de mémoire.

Un tri est dit **stable** s'il préserve l'ordonnancement initial des éléments égaux.

La plupart des tris classiques agissent par *comparaison*, c'est-à-dire que l'on regardera deux à deux certains couples d'éléments, et on prendra une décision en fonction de l'élément qui est le plus grand des deux. En quelque sorte, parmi les $n!$ permutations d'une liste de n éléments distincts, une comparaison permet d'éliminer une moitié : celles qui ne donnent pas le même résultat pour la comparaison en question. Le meilleur tri par comparaison fait donc une sorte de dichotomie parmi l'ensemble des permutations candidates restantes, mais a priori jamais de manière explicite.

Le tri par sélection (selection sort)

Cliquer sur ce lien.
Le tri par sélection consiste à construire la liste triée en **sélectionnant** à chaque fois l'élément minimal parmi ceux qui n'ont pas encore été sélectionnés.

> Il s'agit de rechercher le plus petit élément de la liste et de l'échanger avec l'élément d'indice 0, puis de rechercher le second plus petit élément de la liste et de l'échanger avec l'élément d'indice 1, etc. et de continuer de cette façon jusqu'à ce que la liste soit entièrement triée.

Le tri par insertion (insertion sort)

Cliquer sur ce lien.
Le tri consiste à construire la liste triée en **insérant** à chaque fois au bon endroit dans la partie gauche de la liste le premier élément de la partie droite.

> On examine les deux premiers éléments de la liste : s'ils sont dans le mauvais sens, on les échange (les deux premiers éléments sont maintenant bien ordonnés entre eux).
> On examine les trois premiers éléments entre eux : on insère le troisième élément à la bonne place parmi ces trois éléments (les trois premiers éléments sont maintenant bien ordonnés entre eux).
> Etc.
> Pour insérer le dernier élément « d'un groupe » au bon endroit, on décale d'un rang vers la droite tous les éléments situés à sa gauche qui sont plus grands. On obtient un « trou » qui est la place de l'élément.

Le tri par insertion est l'algorithme que la plupart des personnes utilisent naturellement pour trier des cartes à jouer.

Le tri à bulles (bubble sort)

Cliquer sur ce lien.
Le tri à bulles est un algorithme de tri très simple dont le principe est de comparer répétitivement les éléments consécutifs d'une liste, et à les permuter lorsqu'ils sont mal ordonnés. Cela a pour effet de faire remonter à chaque étape le plus grand élément, comme les bulles d'air remontent à la surface de l'eau.

Plus précisément, le principe est le suivant : on parcourt la liste et on compare les éléments consécutifs. Lorsque deux éléments consécutifs ne sont pas dans l'ordre, ils sont *permutés*.

- Après un premier parcours complet de la liste, le plus grand élément est ainsi en fin de la liste, à sa position définitive.

  En effet, aussitôt que le plus grand élément est rencontré durant le parcours, il est forcément mal trié par rapport à tous les éléments suivants, donc permuté avec le suivant jusqu'à arriver à la fin du parcours.
- Le reste de la liste est en revanche encore en désordre. Il faut donc le parcourir à nouveau (en s'arrêtant, si l'on veut, à l'avant-dernier élément).
- Après ce deuxième parcours, les deux plus grands éléments sont à leur position définitive.
- Il faut donc répéter les parcours de la liste, jusqu'à ce que les deux plus petits éléments soient placés à leur position définitive.

Tri fusion (merge sort)

Cliquer sur ce lien.
Cet algorithme a été inventé par John von Neumann en 1945.
Il s'agit d'un algorithme de type « *diviser pour régner* » c'est-à-dire qui consiste à découper un problème initial en sous-problèmes, résoudre les sous-problèmes (récursivement ou directement s'ils sont assez petits), puis calculer une solution au problème initial à partir des solutions des sous-problèmes.

Le principe du tri fusion repose sur l'observation suivante : à partir de deux listes triées, on peut facilement construire une liste triée comportant les éléments issus de ces deux listes (leur «fusion»).

L'algorithme est naturellement décrit de façon récursive.

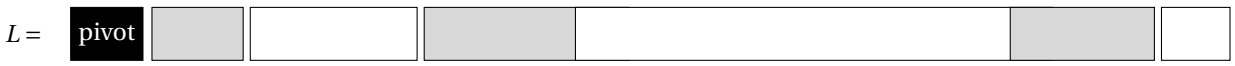
- Si le tableau n'a qu'un élément, il est déjà trié.
- Sinon, séparer le tableau en deux parties à peu près égales.
- Trier récursivement les deux parties avec l'algorithme du tri fusion.
- Fusionner les deux tableaux triés en un seul tableau trié.

Malheureusement, une difficulté se présente : comment fusionner deux demi-tableaux en place, c'est-à-dire en s'autorisant uniquement la permutation de deux éléments dans le tableau ?
Ce n'est pas impossible à réaliser mais la démarche est trop complexe pour être intéressante en pratique. C'est pourquoi nous allons déroger à nos exigences initiales en s'autorisant l'utilisation d'un tableau provisoire pour y stocker le résultat de la fusion.

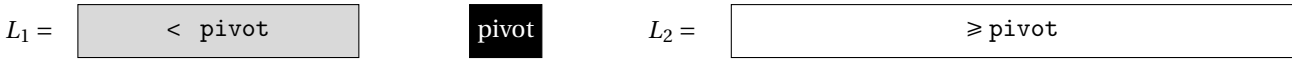
Tri rapide (quick sort)

Le tri rapide est aussi une méthode de type diviser pour régner.
Son principe consiste à séparer la liste en deux parties : pour cela, on sélectionne un terme de la liste appelé pivot et les valeurs sont réparties en deux ensembles, suivant qu'elles sont plus grandes ou plus petites que le pivot.
Après cette opération, le pivot est à sa place finale et on répète alors récursivement cette opération sur les deux sous-tableaux à droite et à gauche.

— **Sélection d'un pivot.** On sélectionne arbitrairement un élément `pivot` de la liste `L` à trier (par exemple ici `pivot=L[0]` mais le dernier élément ou un élément aléatoire est possible aussi).

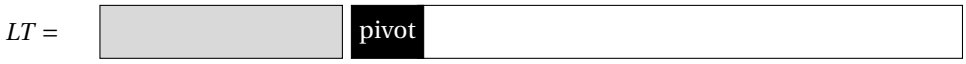


— **Diviser.** On crée alors deux tableaux `L1` et `L2` qui contiennent respectivement les éléments de `L` inférieurs et supérieurs à `pivot`.



— **Régner.** Par un appel récursif au tri rapide, les deux listes L_1 et L_2 sont triées.

— **Fusionner.** Comme les deux listes sont triées, on fusionne juste L_1, `[pivot]` et L_2.



Des remarques

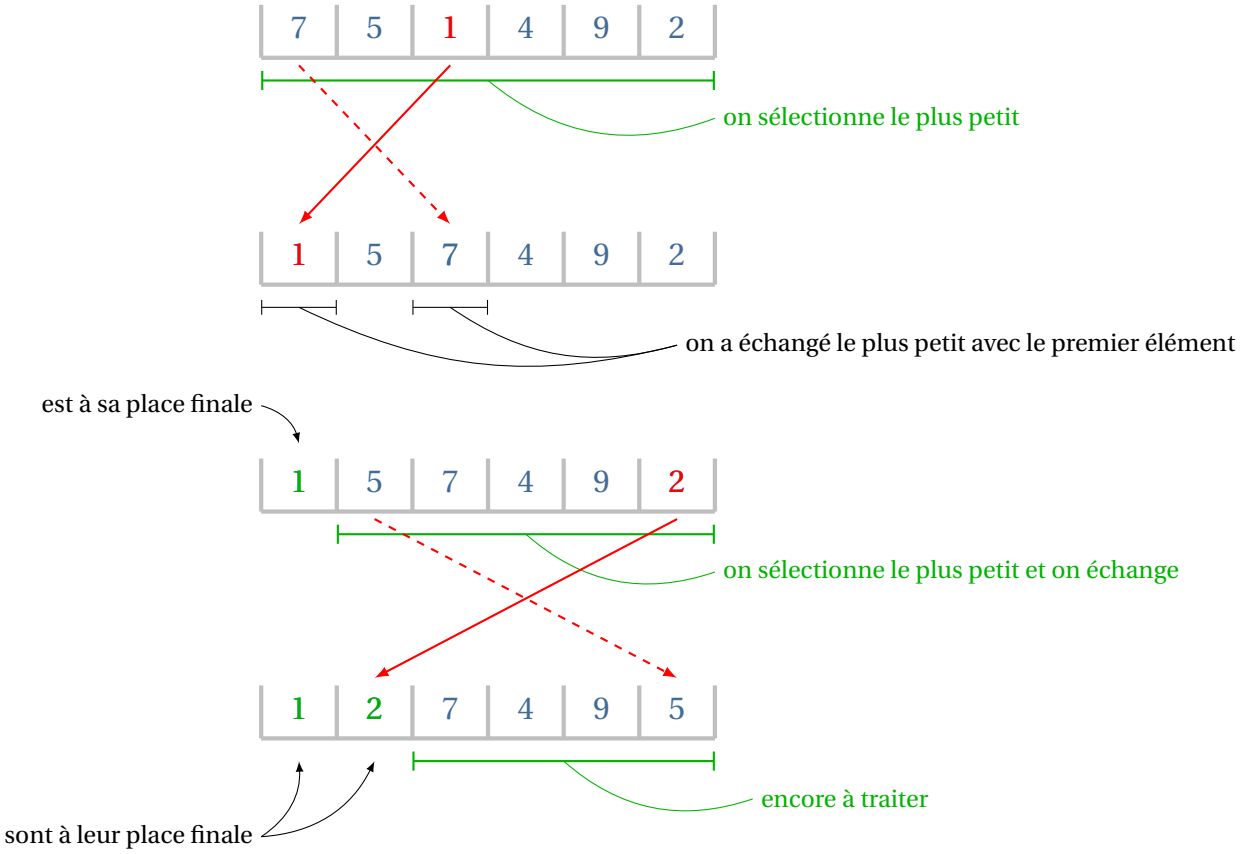
Les tris par sélection, par insertion, et à bulles sont des tris en place.
Les tris par insertion, et à bulles sont stables
Le tri fusion ne se fait généralement pas en place, et est stable.
Le tri rapide peut se faire en place et n'est pas stable.

Il existe bien d'autres algorithmes de tris : tri *par tas* (heap sort), tri *cocktail* (cocktail sort), tri *à peigne* (comb sort), etc, chacun pouvant être lui-même décliné selon différentes versions et variations.

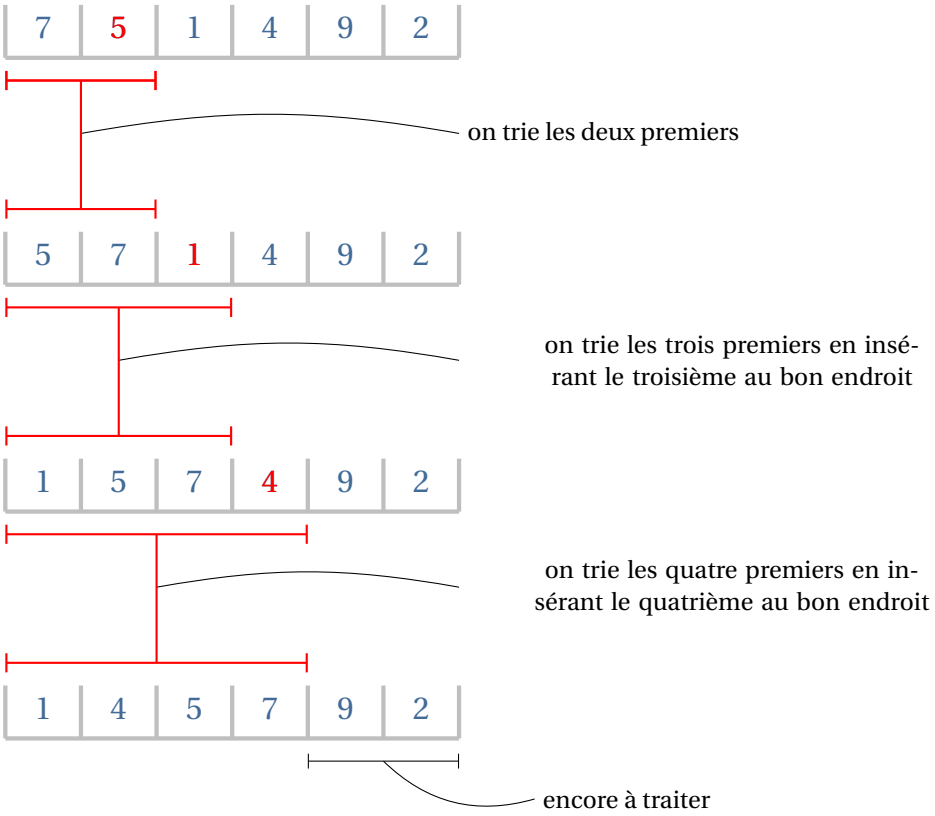
Certains sont bien plus utilisés que d'autres en pratique. Le tri par insertion est souvent plébiscité pour des données de petite taille, tandis que des algorithmes asymptotiquement efficaces, comme le tri fusion, le tri par tas ou quicksort, seront utilisés pour des données de plus grande taille. Il existe des implémentations finement optimisées, qui sont souvent des algorithmes hybrides.

On peut montrer (mais cela dépasse le cadre de ce cours) que la complexité en moyenne d'un tri par comparaison ne peut être inférieure à $O(n \ln n)$.

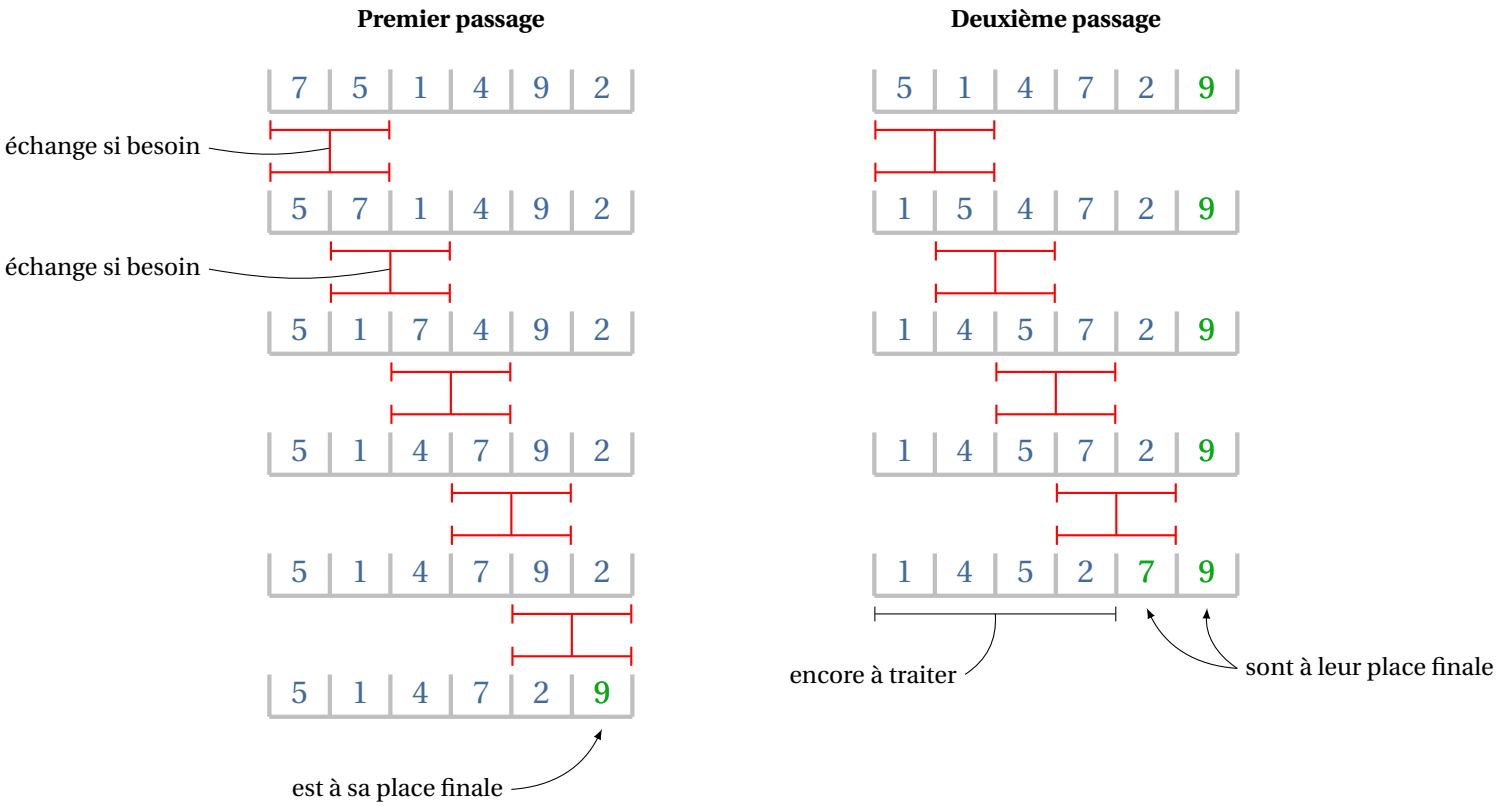
Tri par sélection



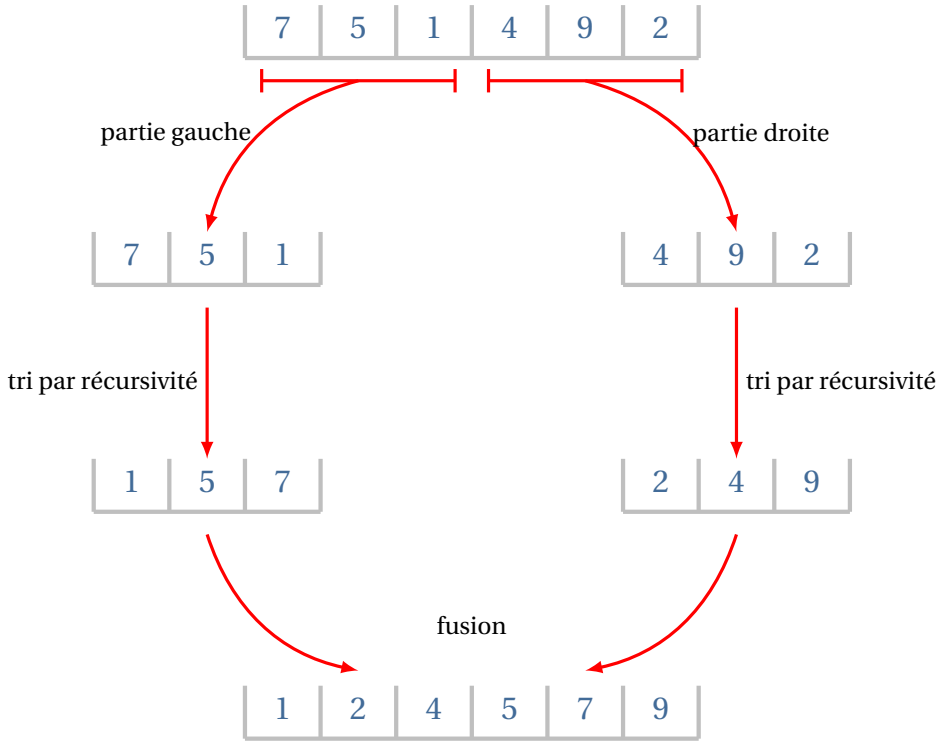
Tri par insertion



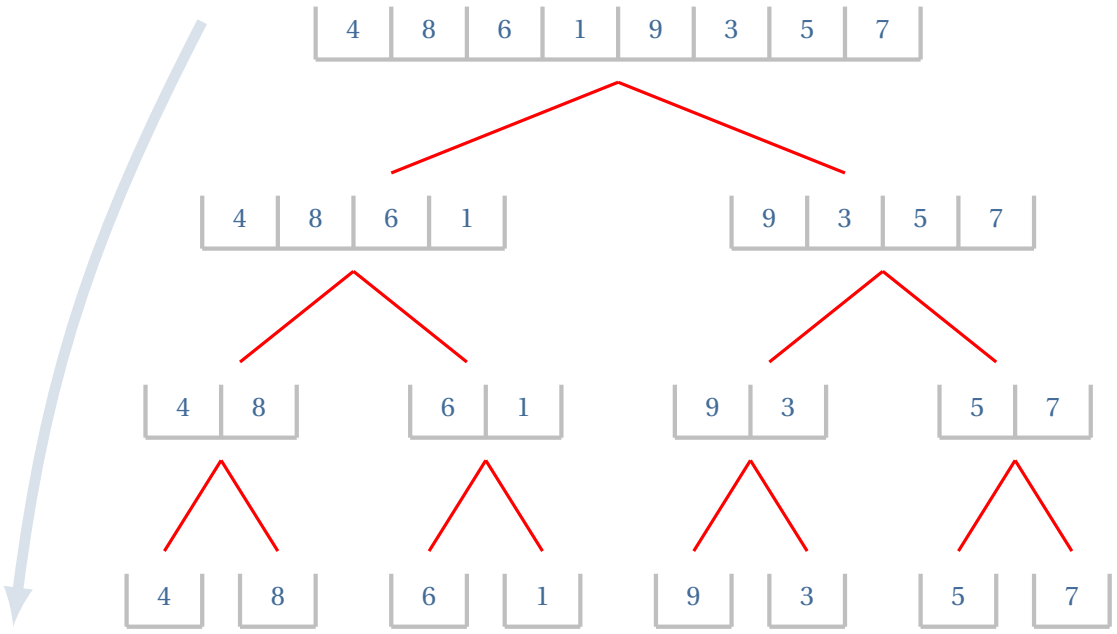
Tri à bulles



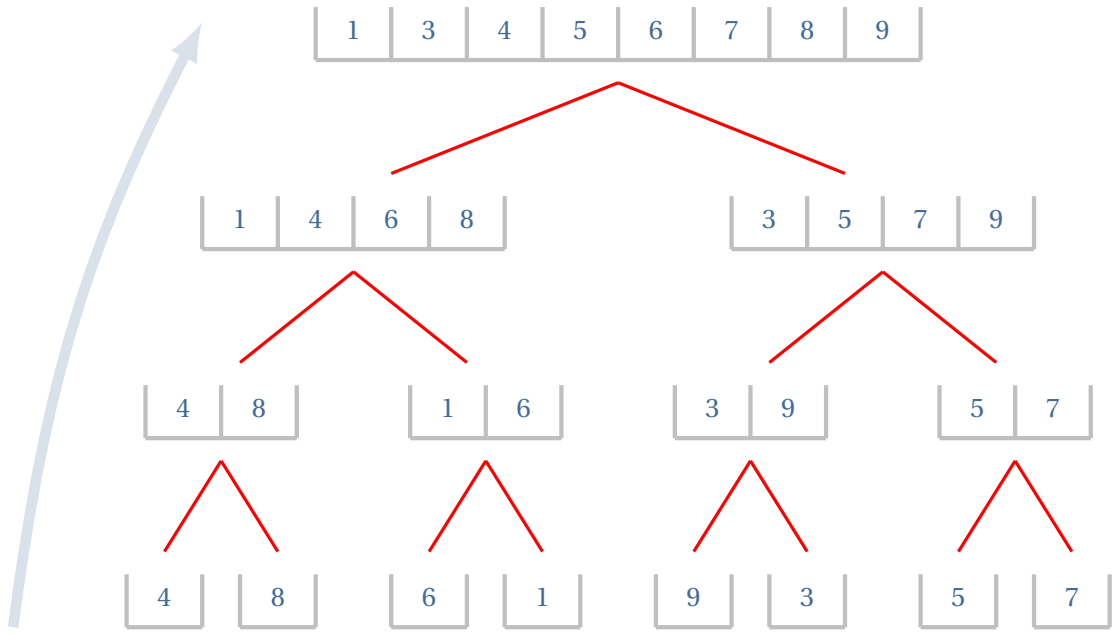
Tri fusion



Tri fusion : descente



Tri fusion : remontée



TP Tris

Pour tout ce TP, on créera une liste de longueur 10 (pour les tests de validité), les autres de longueur 100, 1000 et 10000 (pour les tests d'efficacité).
A la fin du TP, on pourra comparer les temps d'exécution des différents tris à l'aide la fonction `time` du module `time`.

1. Taper ceci dans la console :

```
import numpy.random as rd
L = list(rd.randint(0,100,10))
```

   Quel est l'effet de `L.sort()` ?

   Que vaut `L` une fois cette commande effectuée ?

2. Écrire une fonction `tri_selection` qui prend en argument une liste et la trie à l'aide de l'algorithme du tri par sélection.

   On pourra commencer par écrire une fonction `minimum(L, i)` qui renvoie l'indice du minimum dans la liste $[L[i],\dots,L[n-1]]$ où n est la longueur de la liste `L`.

3. Écrire une fonction `tri_insertion` qui prend en argument une liste et la trie à l'aide de l'algorithme du tri par insertion.

4. Écrire une fonction `tri_bulles` qui prend en argument une liste et la trie à l'aide de l'algorithme du tri à bulles.

5. Écrire une fonction `fusion(A,B)` qui prend en argument deux listes A et B (de longueur a et b) ordonnées de manière croissante et qui réalise la fusion de A et B, c'est-à-dire qui renvoie une liste C de longueur $a+b$ contenant les éléments de $A\cup B$ rangés de manière croissante.

6. Écrire une fonction `tri_fusion` qui prend en argument une liste et la trie à l'aide de l'algorithme du tri fusion.

   Pour cela, on pourra commencer par écrire une fonction `fusion` qui prend en argument deux listes triées et qui les «fusionne».

7. Écrire une fonction `tri_rapide` qui prend en argument une liste et la trie à l'aide de l'algorithme du tri rapide.

8. Créer une liste de longueur 10000 «presque triée». Tester les différentes fonctions de tri sur celle-ci. Commentaires ?

9. Créer une liste de longueur 10000 triée dans l'ordre décroissant. Tester les différentes fonctions de tri sur celle-ci. Commentaires ?

Les algorithmes de tris

Pour chaque algorithme :

- Prouver la terminaison en donnant un variant de boucle.
- Prouver la correction en donnant un invariant de boucle.
- Déterminer la complexité.
- Le tri est-il en place ?
- Le tri est-il stable ?

Le tri à bulles

```
1  def triBulles(L):
2      n = len(L)
3      for i in range(n):
4          for j in range(n-1):   ### mieux for j in range(......)
5              if L[j] > L[j+1]:
6                  L[j], L[j+1] = L[j+1], L[j]
7      return L
```

Terminaison. .....

Invariant de boucle. En fin d'itération i, la liste ..... est triée.

Complexité.

En place ?
Stable ?

Le tri par insertion

```
8  def triInsertion(L):
9      n = len(L)
10     for i in range(........):
11         valeur = L[i]
12         j = i
13         while (j ..... ) and (valeur < L[.....]):
14             L[.....] = L[.....]
15             j = .....
16         L[.....] = valeur
17     return L
```

Terminaison. .....

Invariant de boucle. En fin d'itération i, la liste ..... est triée.

Complexité.

En place ?
Stable ?

Le tri par sélection

```
def triSelection(L):
    n = len(L)
    for i in range(.....):
        m = ...
        ind = ...
        for j in range(.....):
            if L[j] < m:
                m = ...
                ind = ...
        L[i], L[ind] = L[ind], L[i]
    return L
```

Terminaison. .....

Invariant de boucle. En fin d'itération i, la liste ..... est triée.

Complexité.

En place ?
Stable ?

Le tri fusion

```
def fusion(L1,L2):
    L = []
    i, j = 0, 0
    while i < len(L1) .... j < len(L2):
        if L1[i] <= L2[j]:
            L.append(......)
            i = i+1
        else:
            L.append(......)
            j = j+1
    for k in range(...........):
        L.append(L1[k])
    for ...
        ...
    return L


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

Le tri rapide

En relisant la définition du tri rapide donné page 3, compléter le code :

```
55 def triRapide(L):
56     if len(L) <= 1:
57         return ....
58     L1, L2 = .....
59     cle = L.pop()
60     for x in L:
61         if ...
62             L1.append(x)
63         else:
64             L2.append(x)
65     return ...
```


Les algorithmes de tris

corrigés

Le tri par insertion

```
66 def triInsertion(L):
67     n = len(L)
68     for i in range(1,n):
69         valeur = L[i]
70         j = i
71         while (j > 0) and (valeur < L[j-1]):
72             L[j] = L[j-1]
73             j = j-1
74         L[j] = valeur
75     return L
```


Le tri par sélection

```
76 def triSelection(L):
77     n = len(L)
78     for i in range(n):
79         m = L[i]
80         ind = i  ## valeur et indice du min de la liste L[i+1:]
81         for j in range(i+1, n):
82             if L[j] < m:
83                 m = L[j]
84                 ind = j
85         L[i], L[ind] = L[ind], L[i]
86     return L
```


Le tri fusion

```
88  def fusion(L1,L2):
89      L = []
90      i, j = 0, 0
91      while i < len(L1) and j < len(L2):
92          if L1[i] <= L2[j]:
93              L.append(L1[i])
94              i = i+1
95          else:
96              L.append(L2[j])
97              j = j+1
98      for k in range(i, len(L1)):
99          L.append(L1[k])
100     for k in range(j, len(L2)):
101         L.append(L2[k])
102     return L
103
104
105 def triFusion(L):
106     n = len(L)
107     if n <= 1:
108         return L
109     else:
110         m = n//2
111         L1 = triFusion(L[:m])
112         L2 = triFusion(L[m:])
113         return fusion(L1,L2)
```

Le tri rapide

```
115 def triRapide(L):
116     if len(L) <= 1:
117         return L
118     L1, L2 = [], []
119     cle = L.pop()
120     for x in L:
121         if x < cle:
122             L1.append(x)
123         else:
124             L2.append(x)
125     return triRapide(L1) + [cle] + triRapide(L2)
```