Méthodes de tri d'un tableau à une dimension.

I. Introduction

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).

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.

NOTA : Pour trier des chaînes de caractères (mots), il suffit d'associer une valeur numérique à chaque caractère (code ASCII par exemple).

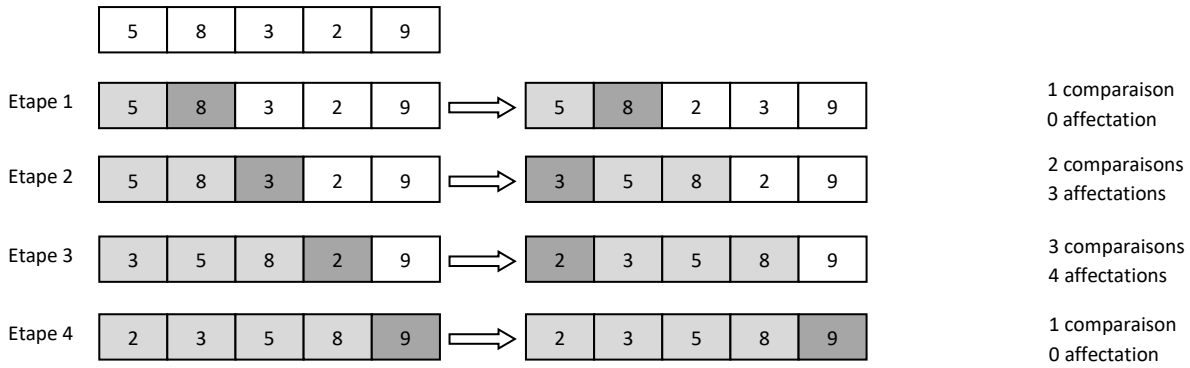
II. Le tri par insertion

Exemple 1 : Tri d'un jeu de cartes
Problème : Comment classer les cartes d'un jeu ?

Soit un paquet de « n » cartes.
On prend la première dans une main.
On saisie la seconde carte et on l'insère avant ou après la première selon le cas.
A l'étape « i », la $i^{ème}$ carte est insérée à sa place dans le paquet déjà trié.

Pour cela, on peut partir de la fin du tas déjà trié, et s'arrêter si l'on rencontre une carte plus petite que la $i^{ème}$
Le paquet contient alors « i » cartes triées.
On procède ainsi de suite jusqu'à la dernière carte.

Exemple 2 : Tri de valeurs numériques
Problème : Comment trier une liste de nombres ?



L'analyse de la complexité de l'algorithme peut se faire par l'étude du nombre de comparaisons à effectuer.

Meilleur des cas :

Dans le **pire des cas** :

Algorithme 1 Tri par insertion

Données : T : un tableau de valeurs numériques [1..n]
Résultat : le tableau T, trié par ordre croissant

NOTA : On peut aussi montrer que la complexité en moyenne est de classe quadratique lorsque les permutations sont équiprobables. L'efficacité du tri par insertion est excellente lorsque le tableau est déjà trié ou « presque trié » (C(n)=O(n)). Il surpasse alors toutes les autres méthodes de tri qui sont au mieux en O(n$\times$ln(n)).

III. Le tri rapide « quick sort »

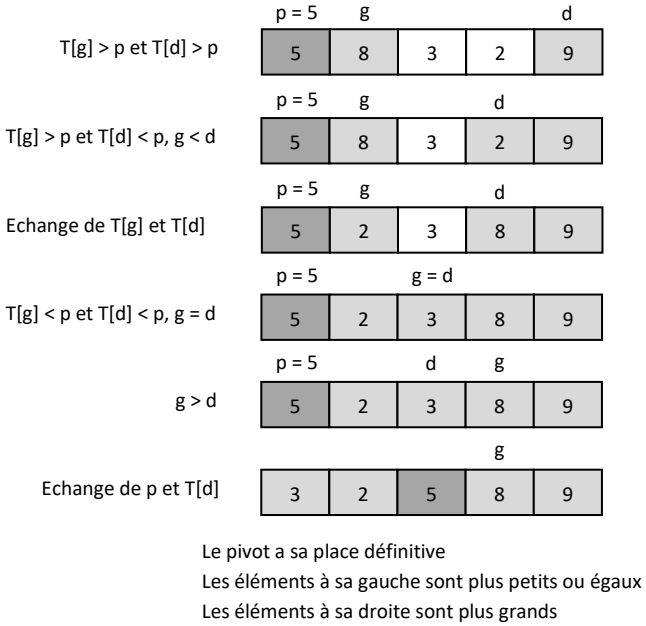
L'algorithme fait parti de la catégorie des algorithmes « diviser pour régner ».

A chaque appel de la fonction tri on choisit une valeur "pivot", par exemple le premier élément. On effectue une partition des éléments à trier. Un premier groupe est constitué de valeurs inférieures au pivot et un deuxième avec les valeurs supérieures. Le pivot est alors placé définitivement dans le tableau.

On traite alors chacun des groupes de façon indépendante. On peut les traiter avec le même algorithme (méthode récursive).

Tableau T initial : 5 est lepivot	**5**	**14**	**11**	**8**	**17**	**7**
On partitionne pour trouver une partie droite et une partie gauche. Sauf qu'ici il n'y aura pas de partie gauche car 5 est la plus petite valeur. 5 est donc à sa place définitive	**5**	**14**	**11**	**8**	**17**	**7**
On traite de façon indépendante les partie gauche et droite. Ici pas de 5 partie gauche. On cherche alors les nouveaux pivots (ici 14)	**5**					
		14	**11**	**8**	**17**	**7**
On segmente alors les parties droite et gauche autour du pivot 14 dont on a ainsi déterminé la position définitive.	**5**					
		7	**8**	**11**	**14**	**17**
On traite de façon indépendante les partie gauche et droite. On recherche pour chacune le pivot (ici 7 et 17).	**5**				**14**	
		7	**11**	**8**		**17**
Pour la chaine de pivot 7 on segmente mais il n'y a pas de partie gauche. Le pivot 17 est seul dans sa chaine il est donc directement à la bonne position.	**5**				**14**	
		7	**11**	**8**		**17**
On traite de façon indépendante les partie gauche et droite. Ici pas de partie gauche. On cherche alors les nouveaux pivots (ici 8).	**5**	**7**			**14**	**17**
			8	**11**		
On classe le seul élément restant par rapport au pivot.	**5**	**7**			**14**	**17**
			8	**11**		

On écrit tout d'abord l'algorithme effectuant la **segmentation du tableau** :
- Le pivot (1er élément du tableau) est mis à sa place définitive,
- Pour des indices inférieurs, toutes les valeurs sont plus petites ou égales,
- Pour des indices supérieurs, toutes les valeurs sont plus grandes.



Le pivot a sa place définitive
Les éléments à sa gauche sont plus petits ou égaux
Les éléments à sa droite sont plus grands

Algorithme 2 Segmentation
Données : T : un tableau de valeurs numériques [1..n]
i, j : les indices de début et de fin de la segmentation à effectuer
Résultat : le tableau T « segmenté » avec le pivot à sa place définitive, l'indice de la place du pivot

```
Segmente(T,i,j)
    g ← i+1
    d ← j
    p ← T[i]
    Tant que g <= d faire
        Tant que d >= 0 ∧ T[d] > p faire
            d ← d−1
        Fin Tant que
        Tant que g <= j ∧ T[g] <= p faire
            g ← g+1
        Fin Tant que
        Si g < d alors
            Echange(T,g,d)          // Echange de T[g] et T[d]
            d ← d−1
            g ← g+1
        Fin Si
    Fin Tant que
    k ← d
    Echange(T,i,d)
    Retourner k
```

Le nombre de comparaisons du type $T[d] > p$ et $T[g] <= p$ est égal dans meilleur des cas à n – 1.

La complexité de cet algorithme est donc au mieux de classe linéaire : C(n)=O(n).

Dans le pire des cas, tous les éléments sont identiques. La complexité est alors quadratique : $C(n)=O(n^2)$.

L'algorithme récursif de tri rapide est le suivant :

Algorithme 3 Tri rapide

Données : T : un tableau de valeurs numériques [1..n]
i, j : les indices de début et de fin de tri à effectuer
Résultat : le tableau T trié entre les indices i et j compris

```
Tri_rapide(T,i,j)
    Si i < j alors
        k = Segmente(T,i,j)
        Tri_rapide(T,i,k-1)
        Tri_rapide(T,k+1,j)
    Fin Si
```

Le « coût » temporel de l'algorithme de tri est principalement donné par des opérations de comparaison sur les éléments à trier. On raisonne donc sur le nombre de données à traiter pour l'analyse de la complexité de l'algorithme.

- Dans le **pire des cas** :

- Dans le **meilleur des cas** :

La complexité est donc de classe quasi linéaire :

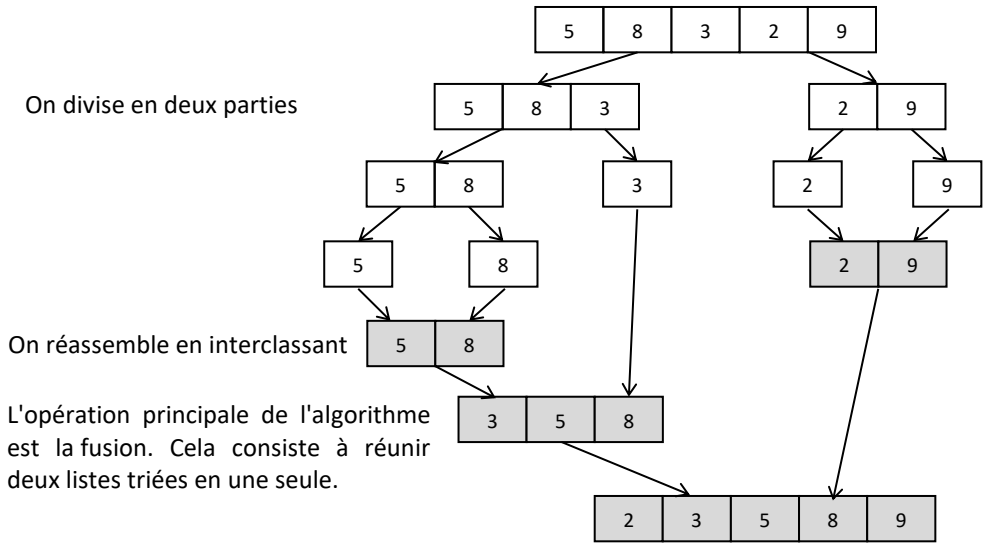
NOTA : Cette méthode de tri est très efficace lorsque les données sont distinctes et non ordonnées. La complexité est alors globalement en $O(n \times ln(n))$. Lorsque le nombre de données devient petit (<15) lors des appels récursifs de la fonction de tri, on peut avantageusement le remplacer par un tri par insertion dont la complexité est linéaire lorsque les données sont triées ou presque.

IV. Le tri par fusion

Cet algorithme fait aussi partie des algorithmes « diviser pour régner ».

Le principe consiste à couper le tableau de départ en deux. On trie chacun des groupes indépendamment. Puis, on fusionne les deux groupes en utilisant le fait que chacun des groupes est déjà ordonné. Il est possible pour réaliser l'ordonnancement de chacun des groupes d'utiliser à nouveau l'algorithme de tri de façon récursive.

La récursivité s'arrête car on finit par arriver à des listes composées d'un seul élément et le tri est alors trivial.



Algorithme 4 : Fusion de deux listes triées

Données : T : un tableau de valeurs numériques [g..d]
m : un indice tel que $g \leq m < d$ et que les sous-tableaux T[g..m] et T[m+1..d] soient ordonnés

Résultat : le tableau T de valeurs numériques [g..d] ordonnées

Cet algorithme a une complexité en temps de classe linéaire : C(n)=O(n).
Par contre, il oblige à utiliser un espace supplémentaire égal à la taille du tableau original T.

L'algorithme récursif de tri fusion est le suivant :

Algorithme 5 : Tri fusion

Données : T : un tableau de valeurs numériques non triées [g..d]
g, d : les indices de début et de fin de tableau

Résultat : le tableau T avec les mêmes valeurs mais triées

Si l'on s'intéresse au nombre de données à traiterà chaque appel de fonction, la relation de récurrence est du type :

$C(n) = 2 \times C(n / 2) + n.$

La méthode de tri fusion a donc une efficacité temporelle comparable au tri rapide en $O(n \times \ln(n))$. Par contre, elle n'opère pas en place : une zone temporaire de données supplémentaire de taille égale à celle de l'entrée est nécessaire. Des versions plus complexes peuvent être effectuées sur place mais sont moins rapides.

NOTA : D'autres méthodes de tri existent : tri par sélection $O(n)^2$, tri à bulles $O(n)^2$, etc.

V. Synthèse

Exemple : Tri de 10000 valeurs numériques générées aléatoirement
Problème : Quelle est l'efficacité des différentes méthodes de tri ?

La simulation a été effectuée pour un tableau de 10000 valeurs numériques générées aléatoirement. Les temps sont donnés en secondes.

Tri Bulle	37.03597636012455
Tri par Insertion	16.26099099187195
Tri par Sélection	15.775444085769777
Tri Fusion	0.157514432699827
Tri Rapide	0.09675168212743301
Tri Natif Python (méthode sort) :	0.0075889533436566126

NOTA : On s'aperçoit que le tri Python (méthode sort) est le plus efficace.