Chapitre 2
Les algorithmes de tri

1- Présentation

Qu'est-ce qu'un tri ?

C'est l'opération qui consiste à ordonner une liste d'éléments en fonction d'une ou plusieurs **clés.**

Sur chaque clé est définie une **relation d'ordre**

Exemple :

tri de la liste des étudiants (définis par : le nom, le prénom, la date de naissance, l'année d'inscription, etc.)

en fonction : du nom **(clé primaire)** et du prénom **(clé secondaire)**

ou encore en fonction de l'année de naissance.

Pourquoi l'étude des algorithmes de tri ?

➢Les algorithmes de tri ont une grande importance pratique. Ils sont fondamentaux dans certains domaines, comme l'informatique de gestion où l'on tri de manière quasi-systématique des données avant de les utiliser.

➢L'étude du tri est également intéressante en elle-même car il s'agit sans doute du domaine de l'algorithmique qui a été le plus étudié et qui a conduit à des résultats remarquables sur la construction d'algorithmes.

➢Ainsi, pour un problème simple et unique, le tri d'un tableau de données, de nombreuses solutions existent.

➢L'étude du tri permet également d'aborder de façon concrète les problèmes de complexité des algorithmes et d'optimalité dans la classe des algorithmes de tri.

Évaluation de la complexité d'un algorithme de tri

1/ Taille des entrées :

C'est le nombre n de données à trier.

2/ Opérations fondamentales :

- Nombre de comparaisons
- Nombre d'échanges de valeur entre deux éléments du tableau

2- Les Algorithmes de tri

2-1 Tri par sélection: principe

•On commence par rechercher l'élément minimal du tableau, pour l'échanger avec celui en première position, puis on recommence avec le tableau privé de ce minimum, et on continue ainsi jusqu'à atteindre l'extrémité droite du tableau.

•A mesure que l'on progresse vers la droite, les éléments situés à gauche du tableau, occupent leur position définitive.

•Lorsque la partie non triée du tableau est réduite à une seule valeur celle ci est alors la plus grande et le tableau est entièrement trié.

•Cette méthode porte le nom de ***tri par sélection car elle procède*** à la sélection successive de l'élément minimal parmi ceux restants.

Tri par sélection : un exemple

Évolution du tableau au fil de l'algorithme. Sur fond rouge, les valeurs déjà traitées.

étape 0	4	3	1	2	6	5
étape 1	1	3	4	2	6	5
étape 2	1	2	4	3	6	5
étape 3	1	2	3	4	6	5
étape 4	1	2	3	4	6	5
étape 5	1	2	3	4	5	6

Procédure sélection_sort_itérative (↑↓V[n] : élément)
Début
i,j,k:entier ; temp:élément;
pour i de 1 à n-1 faire
« Recherche du rang k du minimum dans le tableau non encore trié »
k ←i ;
pour j de i+1 à n faire
si (V[j] <V[k]) alors k ←j fsi ;
fpr;
« Échange du minimum avec le premier élément du tableau non trié »
temp ←V[k] ; V[k] ←V[i] ; V[i] ← temp ;
fpr;
Fin.

Sélection_sort_itérative : complexité
À chaque itération, pour déterminer le minimum, on compare l'élément V[i] à V[i+1], ..., V[n].
On a ainsi (n-i) comparaisons par itération, soit un nombre total de comparaisons Tc égal à

$$Tc = (n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2}$$

Le nombre Te d'échanges réalisés est égal à (n-1).
Quel que soit l'ordre du tableau initial, le nombre d'opérations reste le même.
Le tri par sélection est donc en O**(n2), à la fois** dans le meilleur des cas, en moyenne et dans le pire des cas.

Procédure sélection_sort_récursiv(↓↑V[n]:élément,↓i:entier)
Début
j, k: entier ; temp: élément;
Si (i<n) alors
« Recherche du rang k du minimum dans le tableau non encore trié »
k ←i ;
pour j de i+1 à n faire
si (V[j] <V[k]) alors k ← j fsi ;
fpr;
« Échange du minimum avec le premier élément du tableau non trié »
temp ←V[k] ; V[k] ←V[i] ; V[i] ← temp ;
« tri de la fin du tableau »
sélection_sort_récursive (V, i+1) ;
fsi;
fin; « 1er appel : sélection_sort_récursive(V, 1) »

Sélection_sort_récursive : complexité
Soit Tc le nombre de comparaisons, on a n-1 comparaisons à Chaque appel.
Le point d'appui est atteint pour n=1. d'où :
Tc(1)=0 ; Tc(2)=1+Tc(1) ; Tc(3)=2+Tc(2) ; etc.
Tc est ainsi défini par la formule de récurrence :

$$Tc(n) = Tc(n-1) + n-1 \text{ et } Tc(1) = 0$$

Dont la solution est :

$$Tc(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Le nombre Te d'échanges réalisés est égal à (n-1).
Soit une complexité en O**(n²) comme pour la procédure itérative.**

2.2 Tri par insertion : principe
Le principe général du tri par insertion est de considérer que
les (i-1) premiers éléments du tableau sont triés et d'insérer le ième élément à sa place parmi les (i-1) déjà triés, et ce jusqu'à ce que i = n.
Pour insérer l'élément V[i], on conserve sa valeur dans une variable intermédiaire « temp », puis on compare temp successivement à chaque élément V[i-1], V[i-2], ... , qu'on déplace vers la droite tant que sa valeur est supérieure à celle de temp.
On affecte alors à l'emplacement dans le tableau laissé libre par ce décalage la valeur de temp

Tri par insertion : un exemple
Évolution du tableau au fil de l'algorithme. Sur fond rouge, les valeurs déjà traitées.

	Tableau						temp
étape 0	4	3	1	2	6	5	3
étape 1	3	4	1	2	6	5	1
étape 2	1	3	4	2	6	5	2
étape 3	1	2	3	4	6	5	6
étape 4	1	2	3	4	6	5	5
étape 5	1	2	3	4	5	6	

Procédure insertion_sort_itérative(↑↓V[n] : élément)
début
i,k:entier; temp:élément;
pour i de 2 à n faire
temp ← V[i] ;
« Recherche du rang k de V[i] dans la séquence triée V[1..i-1]»
K ←i-1 ;
tantque (k>0) et (V[k]>temp) faire
V[k+1] ← V[k] ; k← k-1;
ftq;
« placement de temp » V[k+1] ← temp;
fpr;
fin.

Procédure insertion_sort_itérative2(↓↑V[n] : élément)
début
i,k:entier; temp:élément;
pour i de 2 à n faire
temp ← V[i] ;
« Recherche du rang k de V[i] dans la séquence triée V[1..i-1] »
K ← i-1 ;
tantque (V[k]>temp) « avec V[0]<V[i] ∀i∈ [1,n] »
faire
V[k+1] ←V[k] ; k←k-1;
ftq;
« placement de temp » V[k+1] ←temp;
fpr;
Fin.

Insertion_sort_itérative : complexité
Nous comptons ici le nombre Tc de comparaisons (qui est le nombre de décalages à un près).
Dans le pire des cas, c'est-à-dire quand le tableau à trier de manière croissante est classé de façon décroissante, pour insérer le ième élément, il faut réaliser (i-1) comparaisons.
D'où : $Tc = 1 + 2 + \ldots + (n-1) = \frac{n(n-1)}{2}$

Cependant, en moyenne, seule la moitié des éléments est décalée.
Soit une complexité en moyenne de : $Tc = \frac{n(n-1)}{4}$

Là encore, il s'agit d'une complexité en O**(n^2)** analogue à la complexité du tri par sélection.

Tri par insertion : propriétés
La complexité de ce tri est fortement dépendante de l'ordre du tableau initial. Contrairement au tri par sélection qui nécessite un nombre constant de comparaisons, le tri par insertion ne conduit qu'à très peu de comparaisons lorsque le tableau initial est presque en ordre.
Dans le meilleur des cas, le tableau initial est trié et on effectue alors une comparaison à chaque insertion, soit un total de (n-1) comparaisons et une complexité en O**(n).**
Ce tri a donc de meilleures propriétés. C'est l'un des seuls tris dont la complexité est meilleure quand le tableau initial possède un "certain ordre".

Tri par insertion : avantages
Les propriétés du tri par insertion lui confèrent un avantage certain :
il peut être facilement mis en oeuvre pour insérer de nouvelles données dans un tableau déjà trié ou pour trier des valeurs au fur et à mesure de leur apparition (cas des algorithmes en temps réel où il faut parfois exploiter une série de valeurs triées qui s'enrichit, au fil du temps, de nouvelles valeurs).

Procédure insertion_sort_récursive(↓↑V[n]: élément,↓i:entier)
début
k:entier; temp:élément;
Si(i>1) alors
insertion_sort_récursive (V, i-1) ; « tri du début du tableau »
« Recherche du rang k de V[i] dans la séquence triée V[1..i-1] »
K← i-1 ; temp ←V[i] ;
tantque (k>0) et (V[k]>temp) faire
V[k+1] ← V[k] ; k←k-1;
ftq;
« placement de temp » V[k+1] ←temp;
fsi;
fin;
« pour trier tout le tableau : appel de insertion_sort_récursive(V,n) »

insertion_sort_récursive : complexité au pire

Nous comptons toujours le nombre Tc de comparaisons (égal au nombre de décalages à un près). Dans le pire des cas, c'est-à-dire quand le tableau à trier de manière croissante est classé de façon décroissante, on a n-1 comparaisons à chaque appel récursif et aucune comparaison au niveau du point d'appui (n=1), d'où :

Tc(1)=0 ;

Tc(2)=1+Tc(1) ;

Tc(3)=2+Tc(2) ; etc.

Tc est ainsi défini par la formule de récurrence :

Tc(n) = Tc(n-1) + n-1 et

Tc(1) = 0

Dont la solution est :

TC(n)=n(n-1)/2

Soit une complexité en O **(n^2), comme pour la procédure itérative.**

insertion_sort_récursive : complexité au mieux

Si le tableau à trier est initialement ordonné, on fait une comparaison à chaque appel récursif.

On a donc la formule de récurrence :

Tc(n) = Tc(n-1) + 1 et

Tc(1) = 0

Dont la solution est :

Tc(n) = n-1

Soit une complexité en O **(n).**

2-3 Tri rapide: Motivation

➢Les algorithmes de tri en O(n^2) sont dits naïfs car ils ne sont pas efficaces pour trier de grands ensembles d'éléments.

➢Le tri est une tâche importante et fréquente en programmation, il est essentiel qu'il soit réalisé avec **efficacité.**

➢Le tri rapide (Hoare 1960) vise à atteindre cet objectif en se basant sur la stratégie **"diviser pour régner" dont le** principe consiste à appliquer récursivement une méthode de résolution d'un problème de taille donnée à des sous problèmes similaires de taille inférieure.

➢Les algorithmes, basés sur cette stratégie permettent souvent d'importantes réductions de complexité.

Tri rapide : définition

Un tri :

⊲ sans tableau intermédiaire et

⊲ de complexité moyenne en O(nlog2(n))

Approche "diviser pour régner" :

⊲ Diviser le tableau en deux sous–tableaux

⊲ Trier chacun des sous–tableaux

⊲ Fusionner le tout

Principe :

⊲ On prend le premier élément du tableau, T[1], appelé pivot.

⊲ On met tous les éléments >pivot à droite

⊲ On met tous les éléments ≤pivot à gauche

⊲ On met le pivot au milieu

⊲ On applique ce principe récursivement aux deux sous–tableaux obtenus

Tri rapide : le principe

•L'algorithme de tri rapide est de type dichotomique.

•Le principe de cet algorithme, consiste à réaliser une partition du tableau à trier en deux zones telles que tous les éléments de la première soient inférieurs à tous les éléments de la seconde.

•Ce processus est répété récursivement, jusqu'à ce que les partitions soient réduites à un seul élément.

Tri rapide : la méthode de partitionnement

Pour réaliser la partition du tableau en deux zones, on choisit un élément du tableau appelé **pivot** puis, en procédant par permutations on classe tous les éléments inférieurs ou égaux au pivot dans la zone de gauche et tous ceux qui lui sont supérieurs dans la zone de droite.

On range ensuite le pivot entre les deux zones.

Tri rapide : le choix du pivot

✓Le choix du pivot est le problème central de cet algorithme.
✓L'idéal serait de pouvoir répartir l'ensemble des éléments en deux parties de taille à peu prés égales.
✓Cependant, la recherche du pivot qui permettrait une telle partition aurait un coût trop important.
✓Le pivot est donc choisi de façon aléatoire parmi les valeurs de l'ensemble.
✓Dans l'algorithme de partitionnement qu'on va proposé le pivot est le premier élément du tableau.

Tri rapide : l'algorithme de partitionnement

On utilise deux indices **g** et **d** qui partent des extrémités gauche et droite du tableau et avancent l'un vers l'autre.

1. L'indice **g** : part du début du tableau et avance tant que l'élément en cours de traitement est inférieur ou égal au pivot.
2. L'indice **d** : part de la fin du tableau et avance tant que l'élément en cours de traitement est supérieur au pivot.
3. Suite à ce parcours,

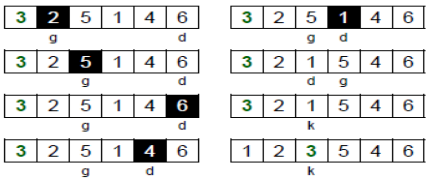
si **g<d**, on échange V[**d**] et V[**g**] puis on continue à faire avancer les indices.

4-Lorsque **d** et **g** se croisent (**d<g**), l'indice k=**d** donne la position définitive du pivot.

5-L'algorithme se termine après échange de V[k] avec le pivot.

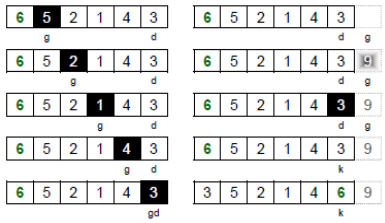
Tri rapide : premier exemple de partitionnement

pivot : première valeur du tableau
valeur en cours de traitement : sur fond **noir**



Tri rapide : deuxième exemple de partitionnement

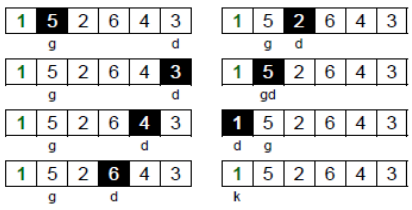
Cas limite où le pivot est l'élément maximal du tableau



La valeur 9 (supérieure à toutes celles du tableau) est une sentinelle qui arrête la progression de g

Tri rapide : troisième exemple de partitionnement

Cas limite où le pivot est l'élément minimal du tableau

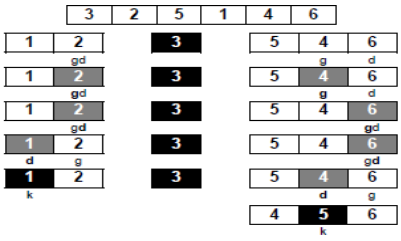


La progression de l'indice d est arrêtée par le pivot

procédure Partitionner (↓↑V[n+1]:élément, ↓g, d:entier, ↑k: entier)

début
i:entier ; temp, pivot : élément ;
pivot ←V[g] ; i ←g ; g← g+1;
tantque (g≤d) faire
tantque (V[g] ≤ pivot) faire g ← g+1 ftq ;
tantque (V[d] > pivot) faire d ← d-1 ftq ;
si (g<d) alors
temp ←V[g] ; V[g] ←V[d] ; V[d] ← temp;
g ←g+1 ; d ←d-1 ;
fsi ;
ftq ;
k ←d ; V[i] ←V[k] ; V[k] ← pivot ;
Fin.

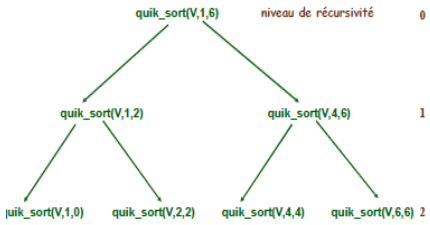
Tri rapide : Exécution sur le 1er exemple
valeurs en cours de traitement (fond gris) classées (fond noir)



Procédure quick_sort (↑↓V[n+1]:élément,↓g, d:entier)
début
k:entier ;
Si (g<d) « si le tableau contient au moins 2 éléments »
alors
Partitionner(V, g, d, k) ;
quick_sort (V, g , k-1) ;
quick_sort (V, k+1, d) ;
fsi ;
Fin.
« le tri du tableau complet est réalisé par l'appel de quik_sort(V,1,n) »

Tri rapide : déroulement de l'algorithme

Le déroulement de quick_sort sur le premier exemple génère la suite des appels suivants :



Tri rapide : complexité temporelle
1. par niveau de récursivité
A chaque niveau de récursivité on parcourt des sous-tableaux dont la réunion est de taille inférieure ou égale à la taille n du tableau initial.
On peut majorer le nombre de comparaisons pour chaque niveau par **n+1.**
Soit une complexité par niveau en O **(n).**
2. Profondeur de la récursivité
Si chaque partition génère des sous-tableaux de tailles à peu prés égales, la profondeur de la récursivité est de l'ordre de **$\log_2 n$.**
En effet, si **$n=2^K$ la profondeur est donnée par $k=\log_2 n$. Par** exemple, pour $n=8=2^3$ la profondeur est de l'ordre de 3.
Si la taille des sous-tableaux est déséquilibrée, la méthode de partitionnement dégénère. Dans le pire des cas, la position finale du pivot correspond toujours à une extrémité d'un sous-tableau, la profondeur est alors de l'ordre de **n.**

Tri rapide : complexité temporelle
3. Temps global d'exécution:
En moyenne et dans le meilleur des cas le tri rapide à une complexité en O **($n\log_2 n$).**
Dans le pire des cas, tri rapide dégénère en tri par sélection avec une complexité en O **(n^2).**

Conclusion:
Grâce à ses bonnes performances et à son implémentation facile, quick-sort est un des algorithmes de tri les plus répandus.
Le problème le plus important est le choix du pivot, car il détermine la profondeur de la récursivité dont dépend l'efficacité de l'algorithme.
Dans la pratique, pour les partitions avec un faible nombre d'éléments (10 ou moins), on a souvent recours à un tri par insertion qui se révèle plus efficace que le tri rapide.