Les algorithmes de tri

1. Introduction

Le tri est sans doute le problème fondamental de l'algorithmique

1. plus de 25% des CPU cycles sont dans les tri
2. le tri est fondamental à beaucoup d'autres problèmes, par exemple recherche binaire.

Ainsi donc, après le tri, beaucoup de problèmes deviennent faciles à résoudre. Par exemple :

1. Unicité d'éléments: après le tri tester les éléments adjacents

2. Une fois le tri fait, on peut déterminer le *k*ème plus grand élément en O(1)

Les problème de tri discutés dans ce chapitre sont ceux où l'ensemble des données à trier se trouvent en mémoire centrale. Les problèmes de tri dont les données sont en mémoire secondaire ne sont pas discutés dans ce chapitre.

2. Présentation du problème

Le tri consiste à réarranger une permutation of *n* objets de telle manière

$$X_1 \geq X_2 \geq X_3 \geq ... \geq X_n \quad \text{tri par ordre décroissant}$$

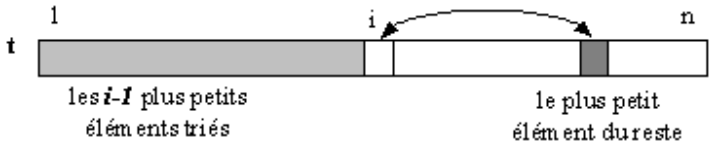
$$X_1 \leq X_2 \leq X_3 \leq ... \leq X_n \quad \text{tri par ordre croissant}$$

Comment trier ? Il existe plusieurs solutions:

3. Tri par sélection

Répéter

1. chercher le plus grand (le plus petit) élément
2. le mettre à la fin (au début)



Exemple

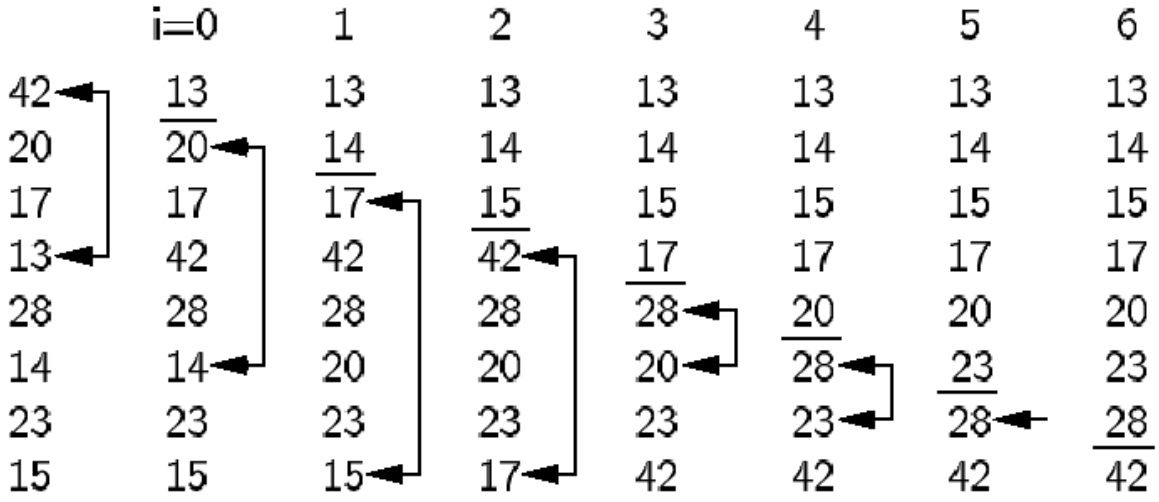


Figure: tri par sélection

Implémentation

```
void selsort(Elem* array, int n)
{
  for (int i=0; i<n-1; i++) { // Selectionner le ième element
    int lowindex = i; // mémoriser cet indice
    for (int j=n-1; j>i; j--) // trouver la plus petite valeur
       if (key(array[j]) < key(array[lowindex]))
            lowindex = j; // mettre à jour l'index
    swap(array, i, lowindex); échanger
  }
}
```

Complexité : Le pire cas, le plus mauvais cas et le cas moyen sont pareils (pourquoi?)

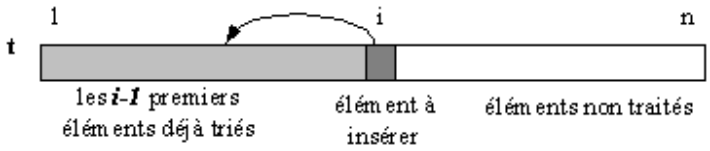
Pour trouver le plus petit éléments, (n-1) itérations sont nécessaires, pour le 2ème plus petit élément, (n-2) itérations sont effectuées, …. Pour trouver le dernier plus petit élément, 0 itération sont effectuées. Le nombre d'itérations que l'algorithme effectue est donc:

$$\sum_{i=1}^{n-1} i = n(n-1)/2 = n^2/2 + n/2 = O(n^2)$$

Si par contre, nous prenons comme mesure d'évaluations le nombre de mouvement de données, alors l,algorithme en effectue *n*-1, car il y a exactement un échange par itération.

4. Tri par Insertion

Dans ce cas, itérativement, nous insérons le prochain élément dans la partie qui est déjà triée précédemment. La partie de départ qui est triée est le premier élément.



En insérant un élément dans la partie triée, il se pourrait qu'on ait à déplacer plusieurs autres.

```
void inssort(Elem* array, int n)
{
   for (int i=1; i<n; i++) // insérer le ième element
       for (int j=i; (j>0) && (key(array[j])<key(array[j-1])); j--)
            swap(array, j, j-1);
}
```

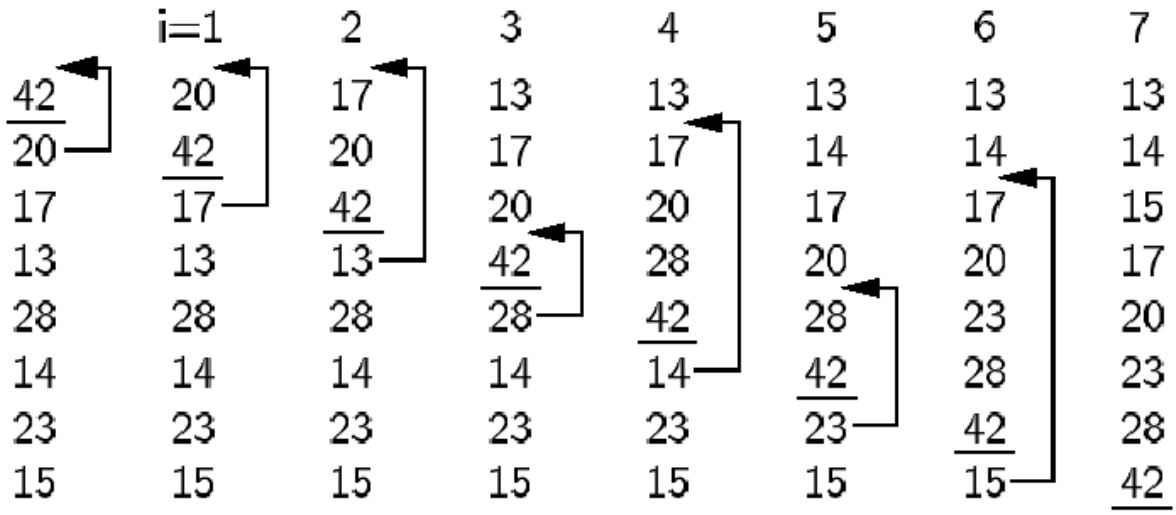


Figure: tri par insertion

Complexité : Comme nous n'avons pas nécessairement à scanner toute la partie déjà triée, le pire cas, le meilleur cas et le cas moyen peuvent différer entre eux.

Meilleur cas: Chaque élément est inséré à la fin de la partie triée. Dans ce cas, nous n'avons à déplacer aucun élément. Comme nous avons à insérer (n-1) éléments, chacun générant seulement une comparaison, la complexité est en $O(n)$.

Pire cas: Chaque élément est inséré au début de la partie trié. Dans ce cas, tous les éléments de la partie triée doivent être déplacés à chaque itération. La $i$ème itération génère (i-1) comparaisons et échanges de valeurs:

$$\sum_{i=1}^{n}(i-1) = n(n-1)/2 = O(n^2)$$

Note: C'est le même nombre de comparaison avec le tri par sélection, mais effectue plus d'échanges de valeurs. Si les valeurs à échanger sont importantes, ce nombre peut ralentir cet algorithme d'une manière significative.

Cas moyen : Si on se donne une permutation aléatoire de nombre, la probabilité d'insérer l'élément à la $k$ème position parmi les positions *(0,1,2, ..., i-1)* est 1/i. Par conséquent, le nombre moyen de comparaisons à la $i$ème itération est:

$$\sum_{k=1}^{i}\frac{(k-1)}{i} = \frac{1}{i}\sum_{k=1}^{i-1}k = \frac{1}{i}\times\frac{i(i-1)}{2} = \frac{i-1}{2}$$

En sommant sur i, on obtient:

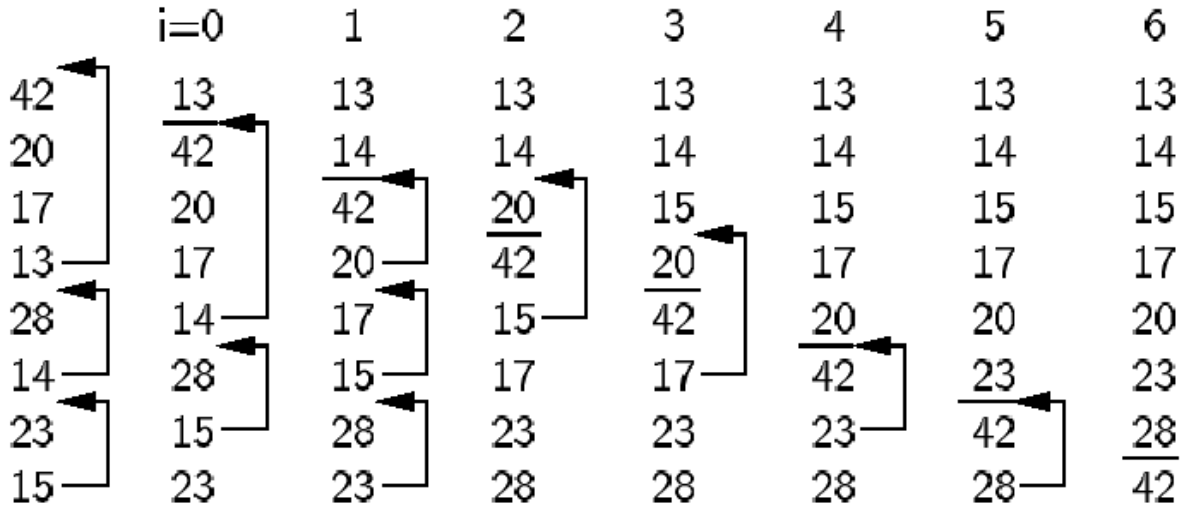
$$\sum_{i=1}^{n}\frac{i-1}{2} = \frac{1}{2}\left(\frac{n(n-1)}{2}+\frac{n}{2}\right) = \frac{n^2}{4}+O(n)$$

Tri par bulles.

La stratégie de cet algorithme est comme suit :

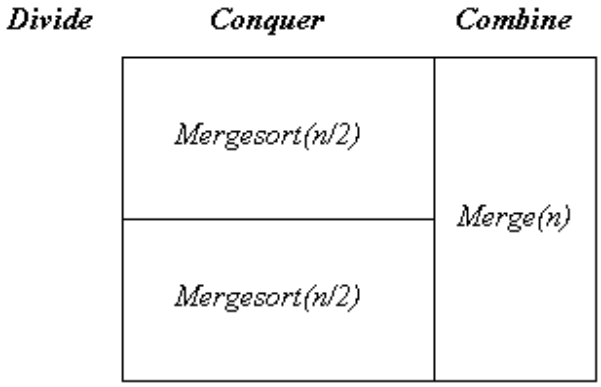
1. Parcourir le tableau en comparant deux à deux les éléments successifs, permuter s'ils ne sont pas dans l'ordre
2. Répéter tant que des permutation sont effectuées.

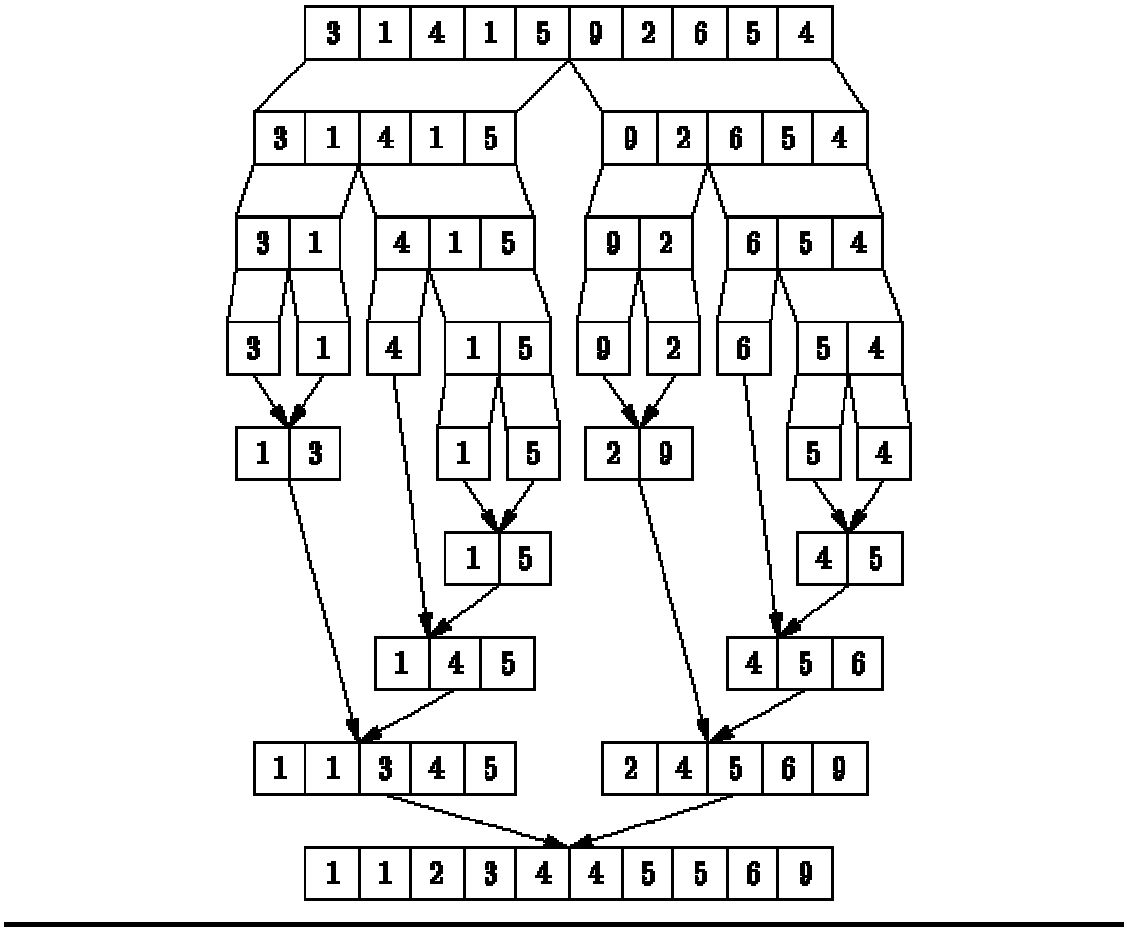
Exemple




```
void bubsort(Elem* array, int n) { // Bubble Sort
  for (int i=0; i<n-1; i++) // échanger
     for (int j=n-1; j>i; j--)
        if (key(array[j]) < key(array[j-1]))
           swap(array, j, j-1);
}
```

5. Tri far fusion Cet algorithme divise en deux parties égales le tableau de données en question. Après que ces deux parties soient triées d'une manière récursive, elle sont fusionnées pour le tri de l'ensemble des données. Remarquez cette fusion doit tenir compte du fait que ces parties soient déjà triées.





Implantation

```
void mergesort(Elem* array, Elem* temp, int left, int right)
{
   int i, j, k, mid = (left+right)/2;
   if (left == right) return;
   mergesort(array, temp, left, mid); // la première moitié
   mergesort(array, temp, mid+1, right);// Sort 2nd half

// l'opération de fusion. Premièrement,  copier les deux moitiés dans temp.
    for (i=left; i<=mid; i++)
          temp[i] = array[i];
    for (j=1; j<=right-mid; j++)
          temp[right-j+1] = array[j+mid];
// fusionner les deux moités dans array
    for (i=left,j=right,k=left; k<=right; k++)
        if (key(temp[i]) < key(temp[j]))
            array[k] = temp[i++];
        else array[k] = temp[j--];
}
```

Complexité:

La complexité de cet algorithme est donnée par la relation suivante:

$$T(n) = \begin{cases} O(1) & n = 1, \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & n > 1, \end{cases} \qquad (15.10)$$

n étant le nombre d'éléments dans le tableau.

Exercice: pourquoi les fonctions plancher et plafond comme paramètres dans T(.)?

Question : Peut-on parler des trois différentes complexités pour cet algorithme?

Dans le but de simplifier la résolution de l'équation 15.10, nous supposons que $n = 2^k$ pour un entier $k \geq 0$. En remplaçant *O(n)* par *n*, on obtient (en principe, on doit la remplacer par *cn*):

$$T(n) = \begin{cases} 1 & n = 1, \\ 2T(n/2) + n & n > 1, \end{cases}$$

:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 4T(n/4) + 2n \\ &= 8T(n/8) + 3n \\ &\vdots \\ &= 2^k T(n/2^k) + kn \\ &\vdots \\ &= nT(1) + n\log_2 n \\ &= n + n\log_2 n \end{aligned}$$

La complexité temporelle de tri par fusion est donc en $O(n \log n)$.

6. Le tri rapide

La stratégie de l'algorithme de tri rapide (*quicksort*) consiste, dans un premier temps, à diviser le tableau en deux parties séparées par un élément (appelé pivot) de telle manière que les éléments de la partie de gauche soient tous inférieurs ou égaux à cet élément et ceux de la partie de droite soient tous supérieurs à ce pivot (dans l'algorithme donné ci-dessus, la partie qui effectue cette tâche est appelée partition) . Ensuite , d'une manière récursive, ce procédé est itéré sur les deux parties ainsi crées. Notez que qu'au départ, le pivot est choisi dans la version de ci-dessus, comme le dernier élément du tableau.

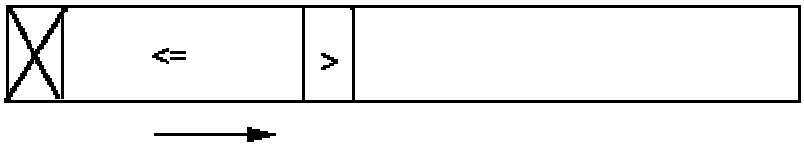
Implantation

```
void tri_rapide_bis(int tableau[],int debut,int fin){
 if (debut<fin){
    int pivot=partition(tableau,debut,fin);
    tri_rapide_bis(tableau,debut,pivot-1);
    tri_rapide_bis(tableau,pivot+1,fin);
}
}
void tri_rapide(int tableau[],int n)
     {
         tri_rapide_bis(tableau,0,n-1);
     }

void echanger(int tab[], int i, int j)
 {
     int memoire;
     memoire=tab[i];
     tab[i]=tab[j];
     tab[j]=memoire;
  }

int partition(int tableau[], int deb, int fin){
  int pivot = tableau[fin];
  int i = debut ; j = fin;
 do{
        do i++
        while (tableau[i] < pivot)

        do  j--
        while (tableau[j] > pivot)
        if (i < j)
          echanger (tableau,i,j)
      } while(i < j)
 tableau[deb] = a[j]; tableau[j] = pivot;
 return(j)
}
```

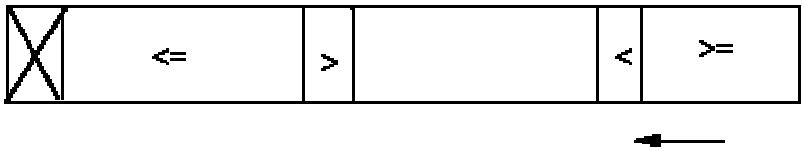
Choix du pivot : Le choix idéal serait que ça coupe le tableau exactement en deux parties égales. Mais cela n'est pas toujours possible. On peut prendre le premier élément. Mais il existe plusieurs autres stratégies!

Partitionnement :

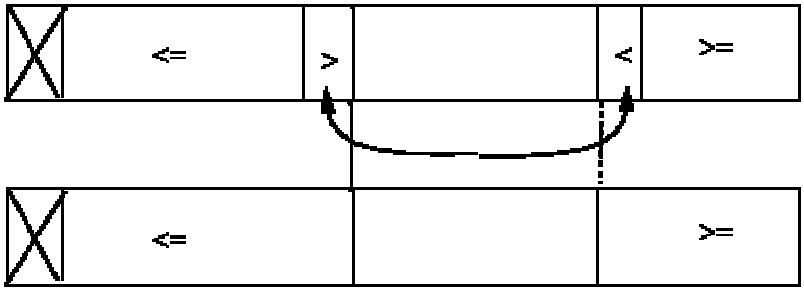
- on parcourt le tableau de gauche à droite jusqu'à rencontrer un élément supérieur au pivot



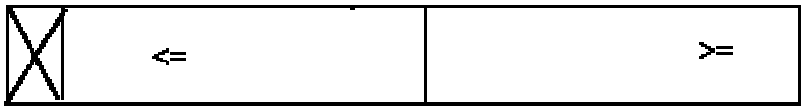
- on parcourt le tableau de droite à gauche jusqu'à rencontrer un élément inférieur au pivot



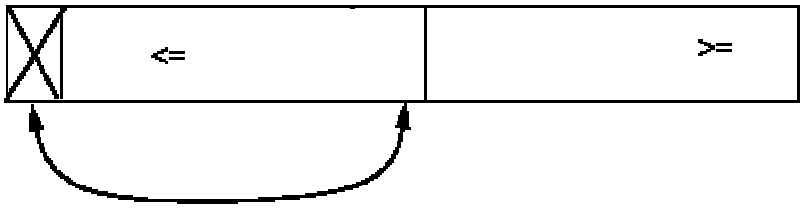
- on échange ces deux éléments



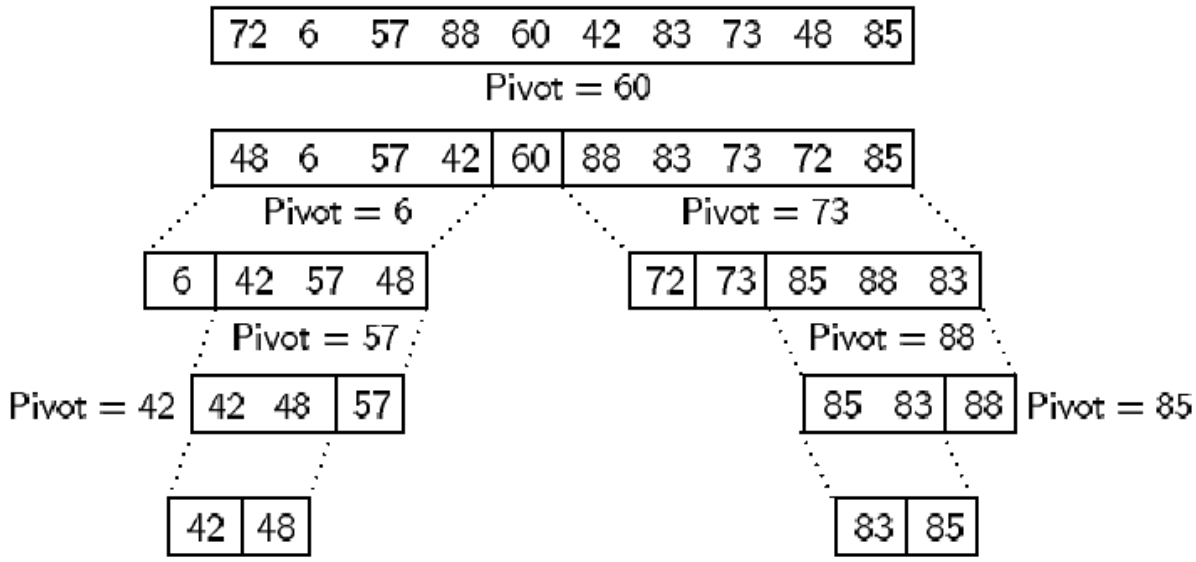
et on recommence les parcours gauche-droite et droite-gauche jusqu'à a avoir :



- il suffit alors de mettre le pivot à la frontière (par un échange)



Exemple



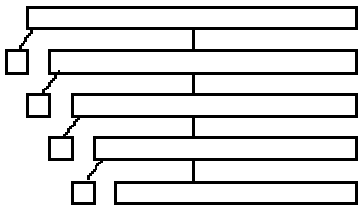
Complexité

À l'appel de QSORT (1,*n*), le pivot se place en position *i*. Ceci nous laisse avec un problème de tri de deux sous parties de taille *i-1* et *n-i*.. L'algorithme de partition clairement a une complexité au plus de cn pour une constante c.

$$T(n) = T(i-1) + T(n-i) + cn$$
$$T(1) = 1$$

Voyons les trois cas possibles de complexité:

Cas défavorable



Le pivot est à chaque fois le plus petit élément. La relation de récurrence devient

$T(n) = T(n-1) + cn$ (pourquoi ?)

$T(n-1) = T(n-2) + c(n-1)$

$T(n - 2) = T(n - 3) + c(n - 2)$

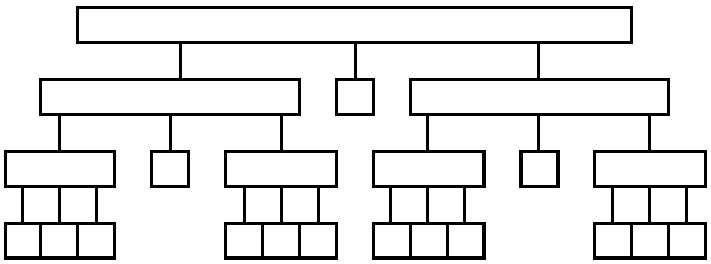
... …….

$T(2) = T(1) + c(2)$

En ajoutant membre à membre, on obtient :

$T(n) = O(n^2)$

Cas favorable :



Dans le meilleur des cas, le pivot est, à chaque fois, situé au milieu de la parti à trier.

$T(n) = 2T(n/2) + cn$

Ce développement s'arrête dès qu'on atteint T(1). Autrement dit, dès que

$$n / 2^k = 1$$

$$n = 2^k \Rightarrow k = \log n$$

Solution finale: $T(n) = O(n \log n)$

Question : déterminer la relation de récurrence exacte de la complexité dans ce cas de figure?

Cas moyen

Nous savons que $T(n) = T(i-1) + T(n - i) + cn$ (voir plus haut). Supposons que toutes les permutations des éléments du tableau soient équiprobables. Autrement dit, la probabilité que la case i soit choisie comme pivot est $1/n$. Comme i peut varier entre 1 et n, alors on obtient facilement la récurrence suivante:

$$t(n) = cn + \frac{1}{n}\sum_{i=1}^{n} t(i-1) + t(n-i)$$

Comme $\sum_{i=1}^{n} t(i-1) = \sum_{i=1}^{n} t(n-i)$, on obtient la relation: $t(n) = cn + \frac{2}{n}\sum_{i=1}^{n} t(i)$ qui peut s'écrire aussi:

$$(1) \qquad nt(n) = cn^2 + 2\sum_{i=1}^{n} t(i)$$

qui est aussi vraie pour n+1. Autrement dit,

$$(2) \qquad (n+1)t(n+1) = c(n+1)^2 + 2\sum_{i=1}^{n+1} t(i)$$

en soustrayant (1) de (2), on obtient:

$$(n+1)t(n+1) - nt(n) = c(2n+1) + 2t(n+1)$$
$$(n-1)t(n+1) - nt(n) = c(2n+1)$$
$$\frac{t(n+1)}{n} - \frac{t(n)}{n-1} = c\frac{(2n+1)}{n(n-1)} = c(\frac{2}{n-1} - \frac{1}{n})$$

Cette équation est aussi vraie pour *n, n-1, n-2, ..., 4*

$$\frac{t(n)}{n-1} - \frac{t(n-1)}{n-2} = c(\frac{2}{n-2} - \frac{1}{n-1})$$

$$\frac{t(n-1)}{n-2} - \frac{t(n-2)}{n-3} = c(\frac{2}{n-3} - \frac{1}{n-2})$$

$$\frac{t(n-2)}{n-3} - \frac{t(n-3)}{n-4} = c(\frac{2}{n-4} - \frac{1}{n-3})$$

……

$$\frac{t(3)}{2} - \frac{t(2)}{1} = c(\frac{2}{1} - \frac{1}{2})$$

En additionnant membre à membre pour n, n-1,…., on obtient après simplifications:

$$\frac{t(n)}{n-1} - \frac{t(2)}{1} = c\left(\frac{1}{n-2} + \frac{1}{n-3} + ... + 1\right) + c\left(1 - \frac{1}{n-1}\right)$$

Or, on sait que :

$$\log(n-2) < 1 + \frac{1}{2} + ... + \frac{1}{n-2} < \log(n-1)$$

on obtient donc :

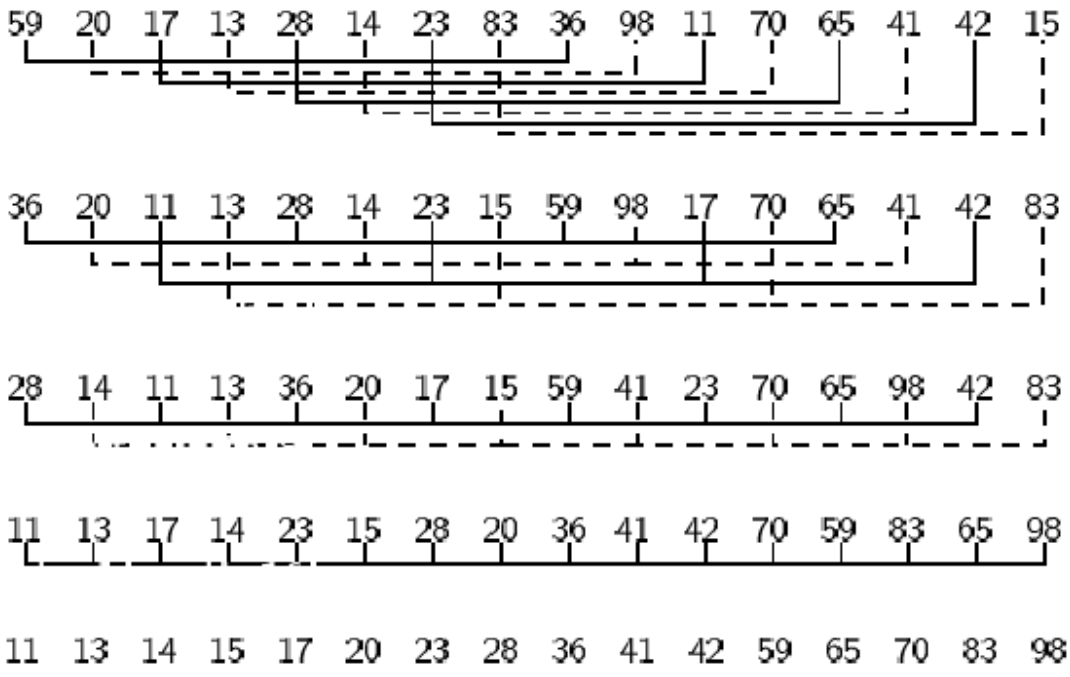
$$t(n) < c(n-1)\left(\log(n-1) + 1 - \frac{1}{n-1}\right) + t(2)$$

t(2) étant une constante, on obtient : $t(n) = O(n \log n)$.

7. Le tri de Shell

La stratégie de cette méthode consiste à subdiviser à subdiviser la liste clés à trier en plusieurs sous-listes de telle manière que les éléments de chacune de ces listes sont à des positions à distance fixée, appelé incrément. Chacune de sous-listes est triée en utilisant l'algorithme de tri par insertion. Ensuite, un autre groupe de sous-liste est choisi avec un incrément plus petit que le précédent. On répète ce processus jusqu'à ce que l'incrément soit égal à 1.

Par exemple, supposons que le nombre n d'éléments à trier soit une puissance de 2. Si on choisit un incrément de départ de n/2, ensuite de n/4, … , jusqu'à 1, alors pour n =16, on aura 8 sous-listes de 2 éléments, séparé de 8 positions dans le tableau; ensuite 4 sous-listes de 4 éléments séparés de 4 postions dans le tableau; ensuite 2 sous-listes de 8 éléments séparés 2 positions dans le tableau; ensuite une seule de 16 éléments. Par exemple :



```
void shellsort(Elem* array, int n)
{
  for (int i=n/2; i>2; i/=2) // pour chaque incrément
  for (int j=0; j<i; j++) // trier chque sous-liste
  inssort2(&array[j], n-j, i);
  inssort2(array, n, 1);
}

// Version de tri par insertion avec des incréments qui varient

void inssort2(Elem* A, int n, int incr) {
   for (int i=incr; i<n; i+=incr)
   for (int j=i; (j>=incr) &&
   (key(A[j])<key(A[j-incr])); j-=incr)
   swap(A, j, j-incr);
}
```

La complexité de cet algorithme est de $O(n^{3/2})$

Résultats expérimentaux sur pentium III windows 98

Algorithm	10	100	1,000	10K	30K
Insert. Sort	.00200	.1833	18.13	1847.0	16544
Bubble Sort	.00233	.2267	22.47	2274.0	20452
Selec. Sort	.00167	.0967	2.17	900.3	8142
Shellsort	.00233	.0600	1.00	17.0	59
Shellsort/O	.00233	.0500	.93	16.3	65
QSort	.00367	.0500	.63	7.3	24
QSort/O	.00200	.0300	.43	5.7	18
Merge	.00700	.0700	.87	10.7	35
Merge/O	.00133	.0267	.37	5.0	16
Heapsort	.00900	.1767	2.67	36.3	122