Méthodes de tri

Les méthodes de tri

Les méthodes de tri permettent le réarrangement de données.

Par exemple, le fichier :

Philippe 21

Jean 33

Lineda 27

peut être trié par nom :

Jean 33

Lineda 27

Philippe 21

ou trié par âge :

Philippe 21

Lineda 27

Jean 33

Les méthodes de tri

Pour pouvoir comparer les méthodes de tris, on les appliquent à des tableaux d'entiers.

Tableau non trié :		20	6	1	3	1	7
	Indices :	0	1	2	3	4	5
Tableau trié :		1	1	3	6	7	20
	Indices :	0	1	2	3	4	5

```
Constante N <- ...
Programme de tri
Algorithme
Début
    Type tab = tableau[ N ]: entier
    Variable t: tab
    tri( t )
Fin
```

```
Procédure tri( t: tab en entrée sortie )
Algorithme
Début
    ...
Fin
```


Tri par sélection

Méthode :

on cherche l'élément de plus petite valeur pour l'échanger avec l'élément en première position ;

puis on cherche l'élément ayant la deuxième plus petite valeur pour l'échanger avec l'élément en deuxième position ;

et ainsi de suite.

20	6	1	3	1	7
1	6	20	3	1	7
1	1	20	3	6	7
1	1	3	20	6	7
1	1	3	6	20	7
1	1	3	6	7	20

Il faut :

- 1 boucle pour parcourir le tableau et sélectionner tous les éléments ;
- 1 boucle pour rechercher le minimum parmi les éléments non triés.

Algorithme du tri par sélection

```
Procédure triSélection( t: tab en entrée sortie )
Algorithme
Début
    Variables i, j, min, tmp : entier
    Pour i de 0 à N-2 répéter                  /* sélection d'un élément */
        min <- i
        Pour j de i+1 à N-1 répéter            /* recherche du minimum */
            Si t[ j ] < t[ min ] alors
                min <- j
            Fin si
        Fin pour
        tmp <- t[ i ]                          /* échange */
        t[ i ] <- t[ min ]
        t[ min ] <- tmp
    Fin pour
Fin
```


Tri par insertion

Méthode :

on considère les éléments les uns après les autres en insérant chacun à sa place parmi les éléments déjà triés.

20	6	1	3	1	7
6	20	1	3	1	7
1	6	20	3	1	7
1	3	6	20	1	7
1	1	3	6	20	7
1	1	3	6	7	20

Il faut :

- 1 boucle pour parcourir le tableau et sélectionner l'élément à insérer ;
- 1 boucle pour décaler les éléments plus grands que l'élément à insérer ;
- insérer l'élément.

Algorithme du tri par insertion

```
Procédure triInsertion( t: tab en entrée sortie )
Algorithme
Début
    Variables i, j, mem: entier
    Pour i de 1 à N-1 répéter                    /* sélection de l'élément à insérer*/
        mem <- t[ i ]
        j <- i
        Tant que j>0 et t[j-1]>mem répéter        /* décalage des éléments plus grands */
            t[ j ] <- t[ j-1 ]
            j <- j - 1
        Fin tant que
        t[ j ] <- mem                             /* insertion */
    Fin pour
Fin
```


Amélioration du tri par insertion

```
Procédure triInsertion( t: tab en entrée sortie )
Algorithme
Début
    ...
    Tant que j>0 et ... Répéter
      ...
      j <- j - 1
    Fin tant que
    ...
Fin
```

Le test j>0 qui évite les dépassements à gauche du tableau est presque toujours vrai !

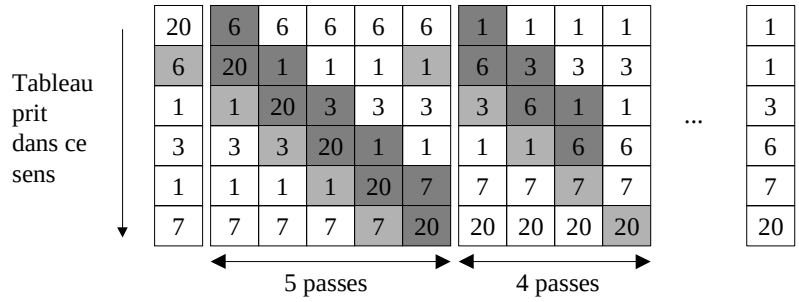
Placer une « sentinelle » : élément inférieur ou égale au minimum du tableau

-1	20	6	1	3	1	7

Tri à bulles

Méthode :

on parcours autant de fois le tableau en permutant 2 éléments adjacents mal classés qu'il le faut pour que le tableau soit trié.



20	6	6	6	6	6	1	1	1	1	1
6	20	1	1	1	1	6	3	3	3	1
1	1	20	3	3	3	3	6	1	1	3
3	3	3	20	1	1	1	1	6	6	6
1	1	1	1	20	7	7	7	7	7	7
7	7	7	7	7	20	20	20	20	20	20

Il faut :

- 1 boucle pour parcourir tout le tableau et sélectionner les éléments un à un ;
- 1 boucle pour permuter les éléments adjacents.

Algorithme du tri à bulles

```
Procédure triBulles( t: tab en entrée sortie )
Algorithme
Début
    Variables i, j, temp: entier
    Pour i de N-1 à 0 répéter          /* sélection de l'élément à insérer*/
        Pour j de 1 à i répéter        /* décalage des éléments plus grands */
            Si t[ j-1 ] > t[ j ] alors
                tmp <- t[ j-1 ]        /* échange */
                t[ j-1 ] <- t[ j ]
                t[j ] <- tmp
            Fin si
        Fin pour
    Fin pour
Fin
```

Comparaison des performances des tris élémentaires

```
Procédure triSélection( t: tab … )
Algorithme
Début
    Pour i de 0 à N-2 répéter
        Pour j de i+1 à N-1 répéter
        Fin pour
    Fin pour
Fin
```

```
Procédure triInsertion( t: tab … )
Algorithme
Début
    Pour i de 1 à N-1 répéter
        Tant que j>0 et t[j-1]>mem répéter
        Fin tant que
    Fin pour
Fin
```

```
Procédure triBulles( t: tab … )
Algorithme
Début
    Pour i de N-1 à 0 répéter
        Pour j de 1 à i répéter
        Fin pour
    Fin pour
Fin
```

Ces méthodes prennent de l'ordre de N^2 étapes pour trier N éléments :

- N petit ou éléments presque triés ;

- N grand et éléments dans un ordre aléatoire.

Performances du tri par sélection

Tri par sélection :
- nombre de comparaison de l'ordre de $N^2/2$;
- nombre d'échanges de l'ordre de N.

```
Procédure triSélection( t: tab en entrée sortie )
Algorithme
Début
  Pour i de 0 à N-2 répéter          /* boucle N-1 fois */
    Pour j de i+1 à N-1 répéter      /* boucle N-i fois */
      Si t[ j ] < t[ min ] alors     /* (N-1)+(N-2)+…+2+1=N(N-1)/2 comparaisons */
        min <- j                     /* seule partie qui dépend des données */
      Fin si
    Fin pour
    ...                              /* N-1 échanges */
  Fin pour
Fin
```

Performances du tri par insertion

Tri par insertion :
- nombre de comparaison de l'ordre de $N^2/4$;
- nombre d'échanges de l'ordre de $N^2/8$;
- linéaire si les données sont déjà triées.

Dans le pire des cas :
- nombre de comparaisons -> $N^2/2$
- nombre d'échanges -> $N^2/4$

```
Procédure triInsertion( t: tab … )
Algorithme
Début
  Pour i de 1 à N-1 répéter          /* boucle N-1 fois */
    mem <- t[ i ]
    j <- i
    Tant que ... t[j-1]>mem répéter  /* boucle i fois ⇒ 1+2+…+(N-1)=N(N-1)/2 */
      t[ j ] <- t[ j-1 ]             /* N(N-1)/2 « demi échanges » */
    Fin tant que
  Fin pour
Fin
```


Performances du tri à bulles

Tri à bulles :
- nombre de comparaison de l'ordre de $N^2/2$;
- nombre d'échanges de l'ordre de $N^2/2$;
- 1 seule passe si le tableau est déjà trié.

```
Procédure triBulles( t: tab en entrée sortie )
Algorithme
Début
  Pour i de N-1 à 0 répéter   /* boucle N fois */
    Pour j de 1 à i répéter   /* boucle i fois */
      Si t[ j-1 ] > t[ j ] alors  /* 1+2+…+N ≈ N²/2 comparaisons au pire des cas */
        ...                      /* échanges = comparaisons ≈ N²/2 */
      Fin si
    Fin pour
  Fin pour
Fin
```

Tableau trié en ordre inverse

Comparaison des tris

Tri par sélection :
- nombre de comparaison de l'ordre de $N^2/2$;
- nombre d'échanges de l'ordre de N.

Tri par insertion :
- nombre de comparaison de l'ordre de $N^2/4$;
- nombre d'échanges de l'ordre de $N^2/2$;
- linéaire si les données sont déjà triées.

Tri à bulles :
- nombre de comparaison de l'ordre de $N^2/2$;
- nombre d'échanges de l'ordre de $N^2/2$;

Tableaux de grandes tailles et aléatoires :

- sélection ;
- insertion ;
- bulles.

Tableaux de petites tailles :

- sélection ;
- insertion ;
- bulles.

Tableaux presque triés :

- sélection ;

- insertion ;

- bulles.

Tableaux avec de grands enregistrements mais des clefs petites :

- sélection ;

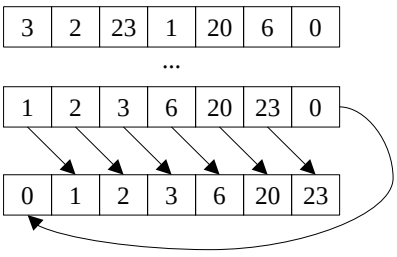
- insertion ;

- bulles.

Tri shell : amélioration du tri par insertion

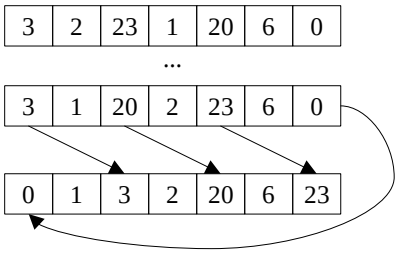
Le tri par insertion :

on considère les éléments les uns après les autres en insérant chacun à sa place parmi les éléments déjà triés.



Inconvénient : pour ramener un élément de la fin vers la tête, il faut décaler tous les éléments plus grands.

- réaliser un tri par insertion sur des éléments séparés par h cases => faire moins de décalage :



on obtient un tableau entrelacé partiellement trié ;

- faire décroître h jusqu'à 1 pour finir de trier le tableau.

Tri shell : amélioration du tri par insertion

Méthode :

- réaliser un tri par insertion sur des éléments distant de h cases ;
- faire décroître h.

	43	21	8	65	6	1	3	34	7	20	11	51	54	27	9
h = 13	27	21	8	65	6	1	3	34	7	20	11	51	54	43	9
	27	21	8	65	6	1	3	34	7	20	11	51	54	43	9
	6	21	8	65	7	1	3	34	27	20	11	51	54	43	9
h = 4	6	1	8	65	7	20	3	34	27	21	11	51	54	43	9
	6	1	3	65	7	20	8	34	27	21	9	51	54	43	11
							...								
	1	3	6	7	8	9	11	20	21	37	34	43	51	53	65

Le choix de h est empirique : le suite …, 364, 121, 40, 13, 4 et 1 est souvent utilisée.

Algorithme du tri shell

```
Procédure triShell( t: tab … )
Algorithme
Début
  Variables i, j, h, mem: entier
  h <- 1
  Tant que h <= N/9 répéter          /* génère le premier de la suite */
      h <- 3*h+1
  Fin tant que
  Tant que h > 0 répéter             /* pour toute la suite …364, 121, 40, 13, 4, 1*/
      /* tri par insertion d'éléments distant de h cases */
      h <- h/3
      Pour i de h+1 à N-1 répéter
          mem <- t[ i ]
          j <- i
          Tant que j>h et t[j-h]>mem répéter
          t[ j ] <- t[ j-h ]
          j <- j - h
          Fin tant que
          t[ j ] <- mem
      Fin pour
  Fin tant que
Fin
```

Performances du tri shell

Tri shell :

- $N^{3/2}$ comparaisons au maximum pour la suite 1, 4, 13, 40, … ;
- actuellement, on ne connaît pas la complexité en temps de cet algorithme.

Méthode de choix dans de nombreuses application :

- commencer par utiliser cette méthode même sur des tableaux relativement gros (- 5000 éléments) ;
- si les performances sont mauvaises, envisager une autre solution.

Tri rapide (quick sort)

Méthode diviser pour résoudre :

❶ choisir un seuil (pivot) ;

❷ mettre les éléments plus petit que le seuil à gauche du seuil et les éléments plus grands à droite ;

❸ recommencer séparément sur les parties droite et gauche.

❶	20	6	1	3	1	7
❷	1	6	1	3	7	20
❸	1	6	1	3		
❶	1	6	1	3		
❷	1	1	3	6		

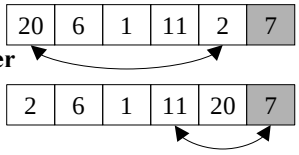
Il faut :

- partitionner selon un pivot ;
- recommencer sur les partitions (récursion).

Algorithme du tri rapide

```
Procédure triRapide( t: tab en entrée sortie, g: entier, d: entier )
Algorithme
Début
  Variable p: entier
  Si d > g alors
    p <- partitionnement( t, g, d )      /* p = pivot */
    triRapide( t, g, p-1 )               /* trier partie gauche */
    triRapide( t, p+1, d )               /* trier partie droite */
  Fin si
Fin
```


Algorithme du tri rapide




```
Fonction partitionnement( t: tab, g: entier, d: entier ): entier
Algorithme
Début
   Variable i, j, p, tmp: entier
   p <- t[d]                          /* pivot = élément de droite */
   i <- g-1                           /* i = curseur partant de la gauche */
   j <- d                             /* j = curseur partant de la droite */
   Répéter
      Répéter                         /* recherche à partir de la gauche ...*/
         i <- i+1
      Tant que t[ i ] < p             /* du premier élément > pivot */
      Répéter                         /* recherche à partir de la droite ...*/
         j <- j-1
      Tant que j>i et t[ j ] > p      /* du premier élément < pivot */
      Si i >= j alors                 /* si les curseurs se rejoignent … */
         break                        /* la partition est faite */
      Fin si
      tmp <- t[i]  t[i] <- t[j]  t[j] <- tmp  /* échange des éléments aux curseurs */
   Sans condition
      tmp <- t[i]  t[i] <- t[d]  t[d] <- tmp /* déplacement du pivot à sa place */
   retour i
Fin
```

Performance du tri rapide

Tri le plus employé :

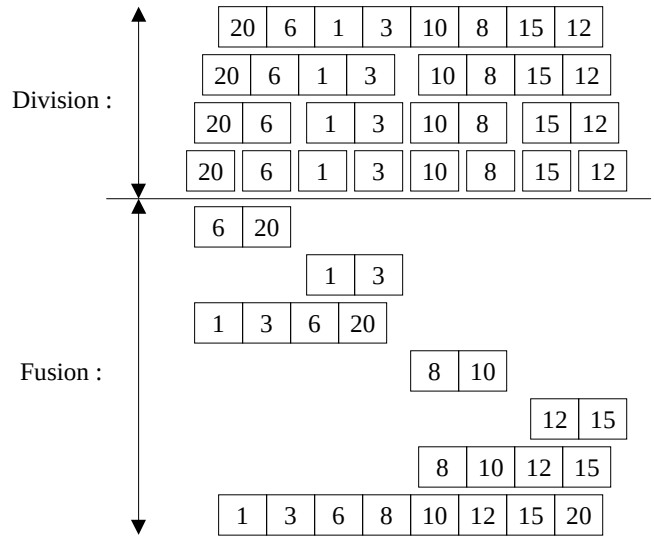
- relativement simple ;
- de l'ordre de NlogN opérations en moyenne pour trier N éléments ;

- de l'ordre de N^2 opérations dans le pire des cas : tableaux simples (déjà triés par exemple) ;

1	2	6	7	11	20
1	2	6	7	11	
1	2	6	7		
1	2	6			
1	2				
1					

- ∃ méthodes pour améliorer les performances (partitionnement par la médiane).

Tri par fusion (merge sort)

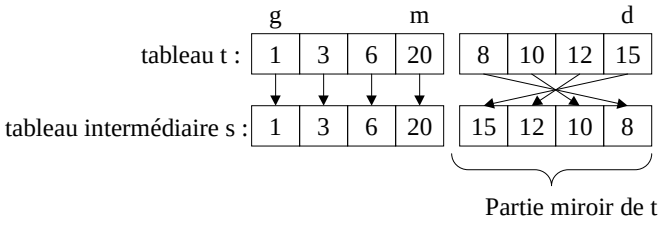


Diviser

```
Procédure triFusion( t: tab en entrée sortie, g: entier, d: entier )
Algorithme
Début
  Variable m: entier
  Si d > g alors
    m <- (g+d) / 2
    triFusion( t, g, m )          /* trier partie gauche */
    triFusion( t, m+1, d )        /* trier partie droite */
    …                             /* fusionner */
  Fin si
Fin
```


Fusionner à l'aide d'un tableau intermédiaire 1/2

- Remplir le tableau intermédiaire s.

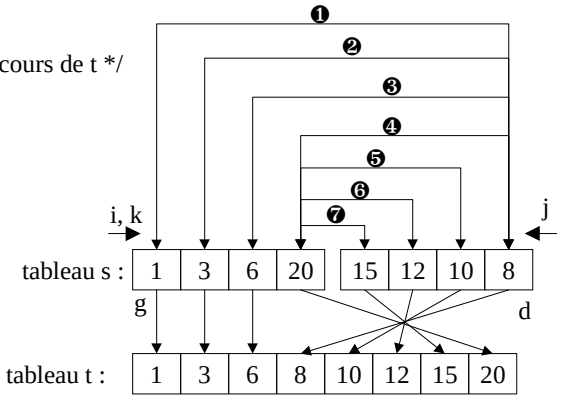



```
...
Pour i de m à g répéter
        s[ i ] <- t[ i ]
Fin pour
Pour j de m+1 à d répéter
        s[ d+m+1-j ] <- t[ j ]
Fin pour
...
```

Fusionner à l'aide d'un tableau intermédiaire

- Comparer les éléments de s deux à deux pour les fusionner dans t.

```
…
k<-g   i<-g   j<-d
Tant que k <= d répéter /* parcours de t */
    Si s[i] < s[j] alors
        t[k] <- s[i]
        i <- i+1
    Sinon
        t[k] <- s[j]
        j <- j-1
    Fin si
    k <- k+1
Fin tant que
```



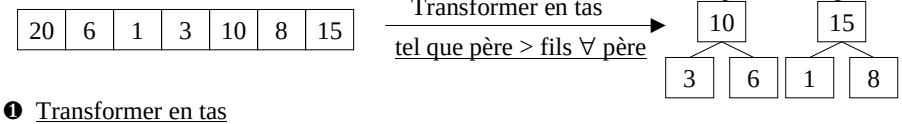
Performance du tri par fusion

Tri par fusion :

- nombre de comparaison de l'ordre NlogN pour trier N éléments ;
- place mémoire supplémentaire proportionnelle à N ; 

- tri idéal pour pour les listes chaînées (accès séquentiel) en changeant la partie fusion de l'algorithme.

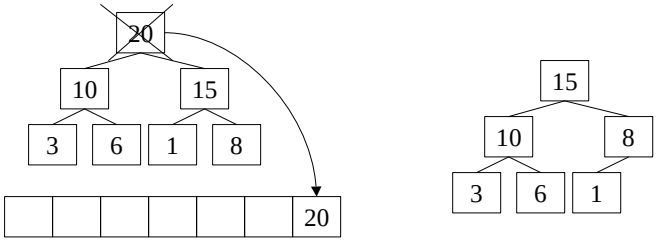
Tri par tas (heap sort)



❶ Transformer en tas

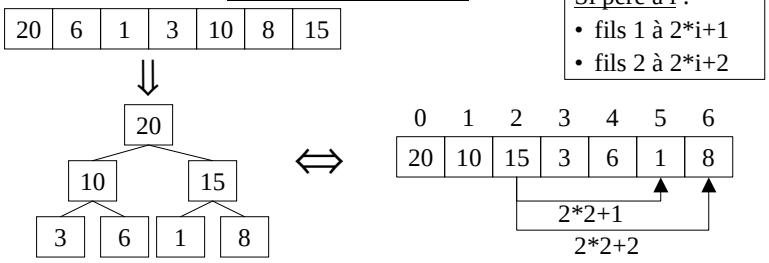
❷ Trier :

- supprimer le maximum et rétablir la condition père > fils ∀ père ;

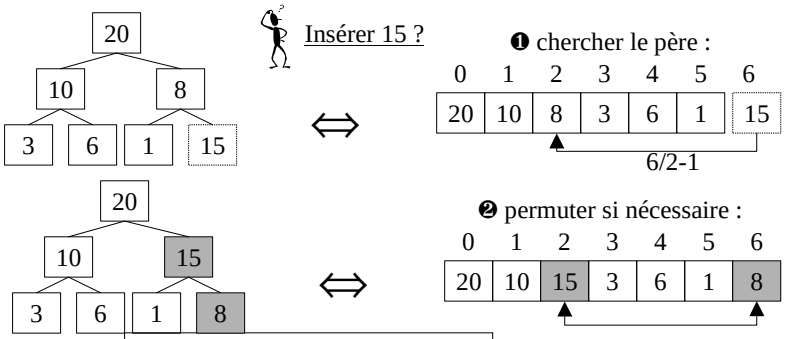


- itérer le processus tant que le tableau n'est pas trié.

Transformer en tas

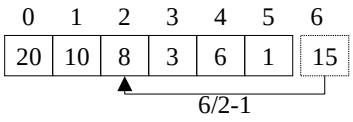


Si père à i :
- fils 1 à 2*i+1
- fils 2 à 2*i+2



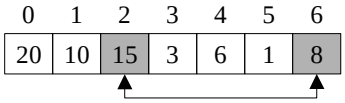
Transformer en tas

❶ chercher le père :

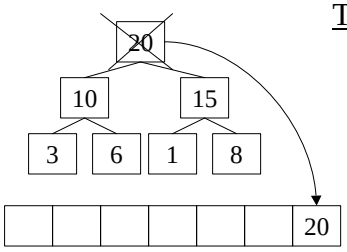


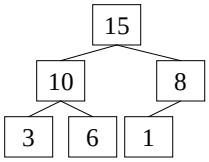

```
Procédure insertionTas( t: tab ..., v: entier )
Algorithme
Début
  Variable p: entier
  N <- N + 1
  t[ N ] <- v                                  /* ajout nouveau en fin de tableau */
  i <- N
  Tant que i<=1 et t[ i/2-1 ] <= v répéter     /* tant que père <= nouveau */
    t[ i ] <- t[ i/2-1 ]                       /* fait descendre le père */
    i <- i/2-1                                 /* père suivant */
  Fin tant que
  t[ i ] <- v            /* positionne nouveau */
Fin
```

❷ permuter si nécessaire :



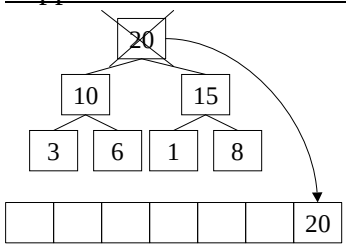
Trier



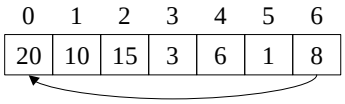



```
Procédure triParTas( t: tab en entrée sortie )
Algorithme
Début
  Variable i: entier
  Pour i de 1 à N répéter
        insertionTas( t, t[ i ] )          /* transformer en tas */
  Fin pour
  Pour i de N à 0 répéter
        t[ i ] <- supprimerMax( t )        /* trier */
  Fin pour
Fin
```

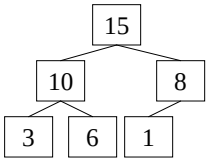
Supprimer le maximum 1/2

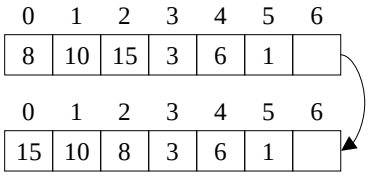


Fonction supprimerMax(t: tab en entrée sortie) : **entier**
Algorithme
Début
  Variable i, j, v, max: **entier**
  max <- t[1]
  t[1] <- t| N] /* place le dernier élément en tête */
  N <- N-1 /* supprime la dernière case (le max) */
  … /* reconstruire le tas */
  retour max
Fin



Supprimer le maximum 2/2






```
/* reconstruire le tas */
v <- t[ 1 ]
Tant que i <= N/2 répéter
        j <- i+i                        /* j est le premier fils de i */
        Si j<N et t[j]<t[j+1] alors     /* si le deuxième fils est plus grand … */
                j <- j+1                /* on le garde */
        Fin si
        Si v >= t[ j ] alors
                break
        Fin si
        t[ i ] <- t[ j ]                /* échange avec le plus grand des fils */
        i <- j                          /* y a-t'il d'autres fils ? */
Fin tant que
t[ i ] <- v
```

Performance du tri par tas

Tri par tas :

- nécessite de l ’ordre de NlogN étapes pour trier N éléments ;
- pas de place mémoire supplémentaire ;
- en moyenne deux fois plus lent que le tri rapide.

Comparaison des tris

Tri shell :
- de l’ordre de $N^{3/2}$ comparaisons.

Tri rapide :
- de l’ordre de NlogN opérations ;
- de l’ordre de N^2 opérations dans le pire des cas.

Tri par fusion :
- nombre de comparaison de l’ordre NlogN ;
- place mémoire supplémentaire proportionnelle à N.

Tri par tas :
- nécessite de l’ordre de NlogN étapes ;
- en moyenne deux fois plus lent que le tri rapide.

Le plus employé :
- tri rapide

Le plus simple sur des fichiers relativement gros (-5000 éléments) :
- tri shell

Tri de listes chaînées :
- tri par fusion

Tri de fichiers partiellement triés :
- tri par tas