

Algorithmes de tri

Tri : complexité minimale

- $n!$ résultats possibles
- si tests binaires, on obtient un arbre de hauteur minimale
 $\log_2 n! \sim \log_2 (n^n / e^n) \sim O(n \log n)$
- La complexité d'un tri sera donc au mieux en $O(n \log n)$

Tri par sélection (selection sort)

```
Ⓐ S O R T I N G E X A M P L E
A S O R T I N G E X Ⓐ M P L E
A A O R T I N G Ⓔ X S M P L E
A A E R T I N G O X S M P L Ⓔ
A A E E T I N Ⓖ O X S M P L R
A A E E G Ⓘ N T O X S M P L R
A A E E G I N T O X S M P Ⓕ R
A A E E G I L T O X S Ⓜ P N R
A A E E G I L M O X S T P Ⓝ R
A A E E G I L M N X S T P Ⓞ R
A A E E G I L M N O S T Ⓟ X R
A A E E G I L M N O P T S X Ⓡ
A A E E G I L M N O P R Ⓢ X T
A A E E G I L M N O P R S X Ⓣ
A A E E G I L M N O P R S T X
A A E E G I L M N O P R S T X
```

Récurrance sur le résultat (tableau de l à $i-1$ trié)

Tri par sélection (selection sort)

Pour chaque i de l à $r-1$, échanger $a[i]$ avec l'élément minimum de $a[i], \dots, a[r]$. Quand l'index i va de gauche à droite, les éléments à sa gauche sont à leur place finale dans le tableau (et ne seront plus modifiés), donc le tableau est complètement trié lorsque i arrive à l'extrémité droite.

```
static void selection(ITEM[] a, int l, int r)
{
    for (int i = l; i < r; i++)
    {
        int min = i;
        for (int j = i+1; j <= r; j++)
            if (less(a[j], a[min])) min = j;
        exch(a, i, min);
    }
}
```

Tri par insertion (insertion sort)

```
A S O R T I N G E X A M P L E
A (S) O R T I N G E X A M P L E
A (O) S R T I N G E X A M P L E
A O (R) S T I N G E X A M P L E
A O R S (T) I N G E X A M P L E
A (I) O R S T N G E X A M P L E
A I (N) O R S T G E X A M P L E
A (G) I N O R S T E X A M P L E
A (E) G I N O R S T X A M P L E
A E G I N O R S T (X) A M P L E
A (A) E G I N O R S T X M P L E
A A E G I (M) N O R S T X P L E
A A E G I M N O (P) R S T X L E
A A E G I (L) M N O P R S T X E
A A E (E) G I L M N O P R S T X
A A E E G I L M N O P R S T X
```

Récurrance sur les données (on insère l'élément i)

Tri par insertion (insertion sort)

(i) met le plus petit élément du tableau à la première place pour que cet élément serve de "sentinelle" ; (ii) une seule assignation, plutôt qu'un échange, dans la boucle principale ; et (iii) la boucle interne se termine quand l'élément inséré est à la bonne position. Pour chaque i , trie les éléments $a[1], \dots, a[i]$ en déplaçant d'une position vers la droite les éléments de la liste triée $a[1], \dots, a[i-1]$ qui sont plus grands que $a[i]$, place ensuite $a[i]$ à la bonne position.

```
static void insertion(ITEM[] a, int l, int r)
{ int i;
  for (i = r; i > l; i--)
    if (less(a[i], a[i-1])) exch (a, i-1, i);
  for (i = l+2; i <= r; i++)
    { int j = i; ITEM v = a[i];
      while (less(v, a[j-1]))
        { a[j] = a[j-1]; j--; }
      a[j] = v;
    }
}
```

Tri fusion (Merge sort)



Récurrence dichotomique sur les données

Tri fusion (Merge sort)

```
static void mergesort(ITEM[] a, int
l, int r)
{ if (r <= l) return;
  int m = (r+l)/2;
  mergesort(a, l, m);
  mergesort(a, m+1, r);
  merge(a, l, m, r);
}
```


Tri par segmentation (Quicksort, Hoare 1960)



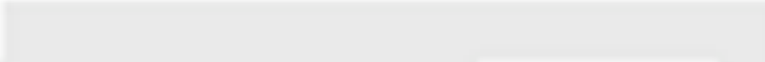
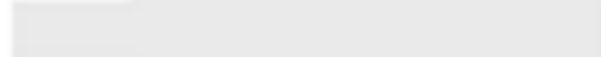

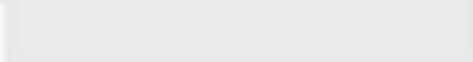
Tri par segmentation (Quicksort, Hoare 1960)


- Choisir un pivot v
- Réarranger le tableau pour que tous les éléments plus petits que v soient à gauche, tous les plus grands à droite
- Trier les deux sous-tableaux à droite et à gauche

```
static void quicksort(ITEM[] a, int l, int r)
{
    if (r <= l) return;
    int i = partition(a, l, r);
    quicksort(a, l, i-1);
    quicksort(a, i+1, r);
}
```

Tri par segmentation : partition (Quicksort, Hoare 1960)

A S O R T I N G E X A M P L E [ⓔ]

A S 
 A M P L 
A A  S M P L E

 O 
 E X 
A A E  O X S M P L E

 R 
 E R T I N G 

A A E [ⓔ] T I N G O X S M P L R

Tri par segmentation (Quicksort, Hoare 1960)

La variable `v` contient la valeur du pivot `a[r]`, et `i` et `j` sont respectivement les indices de parcours gauche et droit. La boucle de partition incrémente `i` et décrémente `j`, en maintenant la propriété invariante : aucun élément à gauche de `i` n'est plus grand que `v` et aucun élément à droite de `j` n'est plus petit que `v`. Quand les indices se rejoignent, on termine le partitionnement en échangeant `a[i]` et `a[r]`, ce qui met `v` dans `a[i]`, avec aucun élément plus grand que `v` à droite et aucun élément plus petit à gauche.

La boucle de partitionnement est codée comme une boucle infinie, avec un `break` quand les indices se croisent. Le test `j==l` protège du cas où l'élément de partitionnement est le plus petit du fichier.

```
static int partition(ITEM a[], int l, int r)
{ int i = l-1, j = r; ITEM v = a[r];
  for (;;)
  {
    while (less(a[++i], v)) ;
    while (less(v, a[--j])) if (j == l) break;
    if (i >= j) break;
    exch(a, i, j);
  }
  exch(a, i, r);
  return i;
}
```

Tri : complexité

- Tri par sélection : au plus $n^2/2$ comparaisons et n échanges
- Tri par insertion : au plus $n^2/4$ comparaisons et $n^2/4$ affectations
- Tri fusion : $O(n \log n)$ (démonstration à suivre)
- Quicksort : $n^2/2$ comparaisons au pire, $2n \ln n$ comparaisons en moyenne, donc $O(n \log n)$. Partitionnement en $O(n)$.
Situation optimale : pivot au milieu, donc choix du pivot crucial

Tri : temps d'exécution

Table 8.1. Empirical study of mergesort algorithms

These relative timings for various sorts on random files of integers indicate that standard quicksort is about twice as fast as standard mergesort; that adding a cutoff for small files lowers the running times of both bottom-up and top-down mergesort by about 15 percent; that top-down mergesort is about 10 percent faster than bottom-up mergesort for these file sizes; and that even eliminating the cost of the file copy (the method used in the Java system sort) leaves mergesort substantially slower than quicksort for randomly ordered files.

		<i>top-down</i>			<i>bottom-up</i>		
<i>N</i>	Q	T	T*	O	B	B*	S
12500	23	27	16	19	30	20	19
25000	20	43	34	27	42	36	28
50000	45	91	79	52	92	77	56
100000	95	199	164	111	204	175	117
200000	201	421	352	244	455	396	256
400000	449	904	764	520	992	873	529
800000	927	1910	1629	1104	2106	1871	1119

Key:

Q Quicksort, standard ([Program 7.1](#))

T Top-down mergesort, standard ([Program 8.1](#))

T* Top-down mergesort with cutoff for small files

O Top-down mergesort with cutoff and no array copy

B Bottom-up mergesort, standard ([Program 8.5](#))

B* Bottom-up mergesort with cutoff for small files

S `java.util.Arrays.sort`