

# CHAP. VI ALGORITHMIQUE NON NUMÉRIQUE

## I Algorithmes de recherche

- I-1 Recherche dans une liste quelconque
- I-2 Recherche dans une liste ordonnée
- I-3 Algorithmes de mise à jour

## II Algorithmes de tri

- II-1 Méthodes simples de tri interne
- II-2 Méthode de tri fusion
- II-3 Méthode de tri rapide (quicksort)

# CHAP. VI ALGORITHMIQUE NON NUMÉRIQUE

**Les types structurés des langages: Chaînes, Tableau, Fichier**

**Structures de données et algorithmique: Listes linéaires ou chaînées, Piles et files d'attente, Tables, Arbres, Graphes**

**Algorithmes: Recherche, Mise à jour, Tri interne ou externe**

## **I Algorithmes de recherche**

- stockage d'ensembles de données et recherche d'éléments .
- recherche réalisée de manière à ne pas demander un temps trop long. =>représentation choisie pour les données, en mémoire centrale ou secondaire. L'ajout et la suppression d'un élément dépendent également de cette représentation.

## I-1 RECHERCHE DANS UNE LISTE QUELCONQUE

Recherche **séquentielle** ou **associative** car on associe une comparaison. L'algorithme serait :

*Pour  $i$  compris entre 1<sup>er</sup> et  $\text{long}(E)$  comparer  $c$  et  $i^{\text{ème}}$  de  $E$*

Si  $E$  est représenté par un tableau de type `Vec` avec  $C$  de type entier, on a alors la fonction *exist* qui renvoie 0 ou 1 :

```
int exist ( Vec e , int n, int c )  
    { int i = 0 ;  
      while ( i < n  &&  e [ i ] != c ) i ++;  
      return ( i < n ) ;  
    }
```

```
int exist ( Vec e , int n, int c )  
  {  
    int i = n - 1;  
    while ( i >= 0 && e [ i ] != c )  
        i - -;  
    return ( i >= 0 );  
  }
```

```
int exist ( Vec e , int n, int c )  
  {  
    if ( n == 0 ) return 0 ;  
        else if ( e [ n - 1 ] == c ) return 1;  
        else return exist (e, n-1, c);  
  }
```

Dans les 2 solutions, complexité proportionnelle au rang de l'élément avec 2 comparaisons à chaque fois.

Au mieux  $2$  soit  $\mathcal{O}(1)$

Au pire  $2n+1$  soit  $\mathcal{O}(n)$

Moyenne  $2(n+1)/2$  soit  $\mathcal{O}(n)$  si  $c \in E$  est sûr

$2*3(n+1)/4$   $\mathcal{O}(n)$  si  $c \in E$  1 fois sur 2

On peut améliorer la complexité en ne faisant qu'une comparaison à chaque fois : on ajoute une sentinelle en fin de liste ( par exemple on copie C à la fin du tableau).

la complexité est alors divisée par 2

Algorithmes auto-adaptatifs qui diminuent la complexité:

réarrangement, fréquence, progression

## I-2 RECHERCHE DANS UNE LISTE ORDONNÉE

Ordre total sur les éléments (liste triée en ordre croissant).

a) **Parcours en séquence** de la liste JQA trouver soit C, soit un élément >C (alors l'élément C non dans la liste).

```
int rech ( Vec e , int n , int c )  
    { int i=0;  
      while ( i <n && e [ i ] < c ) i ++;  
      return ( i < n && e [ i ] == c);  
    }
```

```
int rech ( Vec e , int n , int c )  
    { while ( n >= 0 && e[n] > c ) n --;  
      return ( n >= 0 && e [ n ] == c); }
```

```

int rech ( Vec e , int n , int c ){int i = 0;
        while ( i < n && e[i] < c ) i++;
        return ( i < n && e[i] == c );}

int rechr ( Vec e, int i, int c )
{ if ( e [ i ] < c )
    return rechr ( e, i+1, c );
  else    return ( e [ i ] == c );
}

```

Complexité au mieux  $\vartheta(1)$ , au pire  $\vartheta(k)$   
 en moyenne  $< n/2$

N-B : la complexité au pire est de  
 $n/2$  en moyenne soit  $\vartheta(n)$

## B) RECHERCHE DICHOTOMIQUE SI ACCÈS DIRECT POSSIBLE (TABLEAU)

Principe de la recherche dichotomique de C dans E :  
Comparer C avec l'élément M du milieu du tableau E :

- - si  $C=M$  on a trouvé une solution, la recherche s'arrête ;
- - si  $C>M$  alors C ne peut se trouver qu'à droite
- - sinon C ne peut se trouver que dans la moitié gauche

On continue en diminuant de moitié le nombre d'éléments de la liste restant à traiter, après chaque comparaison

C'est une méthode de résolution par partition encore appelée «**diviser pour régner**» qui se décrit par un algorithme récursif (fonction dir :  $C \in E$  ?):

```

int dir ( Vec e , int c , int g , int d ) ;
{   int m = ( g + d ) / 2 ;
    if ( g <= d )
        if ( c == e [ m ] ) return 1 ;
        else if ( c < e [ m ] ) return dir ( e , c , g , m
-1 ) ;
        else return dir ( e , c , m + 1 ,
d ) ;
    else return 0 ;
}

```

**Complexité** :

|              |                 |                    |
|--------------|-----------------|--------------------|
| - au mieux   | $\theta(1)$     |                    |
| - au pire    | $\log_2(n + 1)$ | $\theta(\log_2 n)$ |
| - en moyenne | $\log_2(n - 1)$ | $\theta(\log_2 n)$ |

## I-3 ALGORITHMES DE MISE À JOUR

Insertion ou suppression d'un élément C dans un tableau E ayant n éléments (  $n < \text{MAX}$ ) indicés de 0 à n-1 .

a) **Tableau non trié**

insertion à la fin

$E[n] \leftarrow C ; n \leftarrow n + 1 ;$

suppression

(i) trouver le rang K de C dans E

(ii) si  $E[K] = C \Rightarrow \{ E[K] \leftarrow E[n-1]; n \leftarrow n-1; \}$

Complexité

Ajout  $\vartheta ( 1 ) ;$

**suppression min  $\vartheta (1)$**

**et max  $\vartheta (n)$**

Insertion à sa place

- (i) trouver la place de l'élément à insérer, c-a-d  
trouver un indice  $p \in [0, n]$  tq  $E[0..p-1] < C \leq E[p..n]$
- (ii) insérer  $C$  à la place  $p$  par des décalages, à partir de la fin, de toutes les valeurs de  $p$  à  $n$ .

Suppression

- (i) trouver la place  $p$  de l'élément à supprimer ;
- (ii) supprimer physiquement par des décalages de la position  $p$  à la fin.

Complexité

au mieux

 $\mathcal{O}(1)$ 

au pire

 $\mathcal{O}(N)$

## II ALGORITHMES DE TRI

Liste de n éléments et à chaque élément est associée une **clé** dont la valeur appartient à un ensemble totalement ordonné. Le résultat est une liste dont les éléments sont une *permutation des éléments* de la liste d'origine telle que les valeurs des clés soient croissantes quand on parcourt la liste séquentiellement.

```
#define MAX 1000  
typedef int Element;  
typedef Element Tab [MAX];  
void perm ( Element * a , Element * b )  
    { Element c = *a; *a = *b; *b = c; }
```

**Complexité temporelle** : taille n

opération : des **comparaisons** entre clés

et des transferts ou **permutations**

**Complexité spatiale**: place mémoire nécessaire pour effectuer le tri d'une liste de n éléments.

# II-1 MÉTHODES SIMPLES DE TRI INTERNE

Ces algorithmes travaillent soit

- par sélection du plus petit élément, suivie du tri du reste de la liste,
- soit par tri du début de la liste, suivi de l'insertion des éléments non triés.

## II-1-1 MÉTHODES PAR SÉLECTION

- On recherche le minimum de la liste.
- On le met à la première place,
- On recommence sur la fin de la liste.

On peut noter qu'après le  $K^{\text{ième}}$  placement, les  $K$  plus petits éléments de la liste sont à leur place définitive.

```

void tournoi ( Tab t , int n , int i )
{  int j ;
   if ( i < n-1 )
       {  for ( j = i+1 ; j < n ; j++ )
           if ( t[i] > t[j] ) perm ( &t[i] , &t[j] ) ;
           tournoi ( t , n , i+1 ) ;
       }
}

```

### Complexité

–en nb de comparaisons =  $\theta ( N^2 )$

–en nb de permutations au mieux  $\theta ( 1 )$

au pire  $\theta ( N^2 )$



## II-1-1-3 TRI À BULLES

```
void bulle ( Tab t , int n , int i )
{ int j; if ( i < n-1)
    { for ( j = n-2; j > i; j - - )
        if ( t [ j ] > t [ j+1 ] )
            perm ( &t[j+1], &t[j] );
        bulle ( t, n, i+ 1 );
    }
}
```

### Complexité

en nb de comparaisons  $\theta ( N^2 )$

en nb de permutations au mieux  $\theta ( 1 )$

au pire  $\theta ( N^2 )$

## II-1-2 MÉTHODES PAR INSERTION

- On trie successivement les 1<sup>ers</sup> éléments de la liste et à la ième étape on insère le ième élément à son rang parmi les  $i-1$  éléments précédents qui sont déjà triés entre eux.

Cela s'appelle aussi la méthode du joueur de cartes.

```
void triinsert ( Tab t , int n )
```

```
    { if ( n > 1 )           /* n=nombre de valeurs */
      { triinsert ( t, n-1 );
        inserer ( t , n-1 );
      }
    }
```

# INSERTION SÉQUENTIELLE OU DICHOTOMIQUE

## II-1-2-1 Insertion séquentielle

- Complexité - en comparaisons au mieux  $\theta ( N )$   
au pire  $\theta ( N^2 )$   
- en permutations au mieux  $\theta ( 1 )$   
au pire  $\theta ( N^2 )$

## II-1-2-2 Insertion dichotomique

- Complexité - en comparaisons au mieux  $\theta ( N )$   
au pire  $\theta ( N \log_2 N )$   
en permutations au mieux  $\theta ( 1 )$   
au pire  $\theta ( N^2 )$

## II-2 MÉTHODE DE TRI FUSION

### II-2-1 PRINCIPE ET PROGRAMMATION

- On partage la liste en 2 sous-listes que l'on trie et on interclasse ou fusionne ces 2 sous-listes :
- Cas particulier : une liste d'un seul élément est triée.
- Définition récurrente : «diviser pour régner»

*Procédure trier ( liste  $t$  ,  $n$  )*

*couper la liste en 2 sous-listes de taille =*

*trier ( sous-liste 1 ,  $n/2$  )*

*trier ( sous-liste 2 ,  $n/2$  )*

*fusionner les 2 sous-listes triées*

### 3 OPÉRATIONS : COUPER, TRIER, FUSIONNER

- Couper c-a-d calculer l'indice de l'élément milieu du tableau  $t[p,r]$ , ce qui donne 2 sous-listes  $t[p,q]$  et  $t[q+1,r]$

$$q = (p + r) / 2$$

- Trier le tableau  $t[p, r]$  si  $p < r$ , sinon (au + 1 élément) il est trié.

*void trifusion ( Tab t, int p, int r )*

- Fusionner les sous-tableaux  $t[p,q]$   
et  $t[q+1,r]$  dans  $t[p,r]$ .

*void merge ( Tab t, int p, int q, int r )*

```
void trifusion ( Tab t , int p ,  
int r )  
{ if ( p < r )  
  {  
    int q = ( p + r ) / 2;  
    trifusion ( t , p , q );  
    trifusion ( t , q + 1 , r );  
    merge ( t , p , q , r );  
  }  
}
```

```
void merge ( Tab t, int p, int q, int r )  
{ int i = p, j = q+1, k = 0; Tab b;  
while ( i <= q && j <= r )  
    if ( t[i] <= t[j] ) b[ k++ ] = t [ i++];  
        else b[ k++ ] = t [ j++];  
while ( j <= r ) b [ k ++ ] = t [ j ++ ];  
while ( i <= q ) b [ k ++ ] = t [ i ++ ];  
for ( i = 0 ; i < k ; i++ ) t [ p+i ] = b  
    [ i ];  
}
```

## II-2-2 COMPLEXITÉ TEMPORELLE

- mesure de la taille des données =  $n$ 
  - opération fond = comparaison de 2 éléments
  - config des données : joue un rôle pour la fusion !

- $T(n)$  = nombre de comparaisons avec  $n = r-p+1$
- Tableau de taille  $n$ , partagé en 2 tableaux de taille  $n/2$ ; on les trie récursivement et on les fusionne. On arrête lorsque le tableau est de taille 1 ( $p=r$ ).

$$T(1) = 0$$

$$T(n) = T(n/2) + T(n/2) + g(n) \quad n > 1$$

## COMPLEXITÉ DE MERGE (A, P, Q, R) = G(N)

- Le nombre de comparaisons pour fusionner 2 tableaux de taille k et m est en  $(k + m)$ , puisque on accède une fois et une seule à chaque case des 2 tableaux et donc ici  $(r - p + 1) = (n \text{ à la } 1^{\text{ère}} \text{ itération})$

en fait  $n/2$  dans le meilleur des cas

$n-1$  dans le pire des cas

Ainsi donc  $g(n) = \theta(n)$ , la configuration des données n'influe pas.

## Complexité de la fusion:

- Fonction  $T(n)$  monotone croissante strictement.

On pose donc  $n = 2^i$ .

$$T(n) = T(2^i) = 2T(2^{i-1}) + \alpha 2^i$$

- Sommons, après simplification on obtient

$$\begin{aligned} T(n) = T(2^i) &= [ \alpha 2^i + \alpha 2^i + \dots + \alpha 2^i ] + 2^i \cdot T(1) \\ &\quad \left\{ \begin{array}{l} i \text{ termes} \end{array} \right\} \quad \left\{ 0 \right\} \\ &= \alpha i 2^i \end{aligned}$$

- Puisque  $i = \log_2 n$ ,  $T(n) = \alpha n \log_2(n)$  et donc

$$\mathbf{Min(n) = Moy(n) = Max(n) = \Theta(n \log_2 n)}$$

## II-3 MÉTHODE DE TRI RAPIDE (QUICKSORT)

Encore appelée tri par segmentation  
ou tri des bijoutiers  
ou tri de Hoare,  
c'est un exemple de tri par dichotomie

On partage la liste à trier en 2 sous-listes, telles que tous les éléments de la première liste soient inférieurs à tous les éléments de la seconde. On recommence jusqu'à avoir des sous-listes réduites à un élément.



8 12 5 9 17 10 16 20 11 14

pivot = 14

8 12 5 9 11 10 14 20 17 16

pivot = 10

8 9 5 10 11 12

pivot = 17

16 17 20

pivot = 8

5 8 9

pivot = 11

11 12

et donc finalement:

**5 8 9 10 11 12 14 16 17 20**

1- Trouver indice du pivot entre les indices  $i$  et  $j$ .

*int trouverpivot ( Tab t, int i, int j )*

soit  $i$  soit  $j$  soit  $(i+j)/2$  soit indice valeur non extrémale  
entre  $t[i]$ ,  $t[j]$ ,  $t[(i+j)/2]$ , soit ...

2- Partitionner = réorganiser le tableau entre  $i$  et  $j$  tq les valeurs  
< pivot soient dans la ss-liste gauche et les autres dans l'autre.  
On retourne l'indice pivot ou point de césure.

*int partition ( Tab t, int i, int j, element pivot)*

3- Trier le tableau entre  $i$  et  $j$  par segmentation. Si  
 $i \geq j$  alors le tableau est trié (au + 1 élément).

*void trirapide ( Tab t, int i, int j )*

```
void trirapide ( Tab t, int i, int j )  
  { int m;  
    Element pivot ;  
    if ( i < j )  
      { pivot = t [ trouverpivot ( t, i,  
j ) ];  
        m=partition ( t, i, j, pivot );  
        trirapide ( t, i, m - 1 );  
        trirapide ( t, m+1, j );  
      }  
    }
```

```
int partition ( Tab t, int i, int j,
Element pivot )
{ int g = i, d = j;
  do
  { perm ( & t [ g ], & t [ d ] );
    while ( t [ g ] < pivot )    g = g
+1;
    while ( t [ d ] >= pivot )  d --;
  }
  while ( g <= d );
return g;
}
```

○ Complexité spatiale  $\theta(N)$  ou  $\theta(\text{MAX})$

• Complexité temporelle

- Nb d'opérations de *trouverpivot* est en  $\theta(1)$  ou  $\theta(N)$

- Nb de comparaisons de *partition* est en  $(j-i+1)$  ou  $(j-i)$  ou encore  $\theta(n)$ , en effet on accède 1 fois et 1 seule à chaque case du tableau t.

Le nombre d'échanges est au mieux de 1,  
au pire de  $(j-i)/2 = n/2 = \theta(n)$

- Nb d'opérations de *trirapide* dépend de la taille des sous-tableaux  $t [i, m-1]$  et  $t[m+1, j]$ . Si on appelle  $C(i, j)$  la complexité temporelle de la procédure *trirapide* ( $t, i, j$ ), on aura alors :

$$C(i, j) = C(i, m-1) + C(m+1, j) + K |i - j|$$

- Le cas le plus défavorable:  $m=i$  ou  $m=j$  (sous-tableau vide), si c'est toujours ainsi (tableau trié) alors :

$$C(1, N) = C(1, N-1) + (N-1) = C(N) = C(N-1) + K(N-1)$$

$$C(N) = \theta(N^2)$$

- Le cas le plus favorable et en moyenne : ( $m = (i+j)/2$ )

$$C(N) = 2 C(N/2) + (N-1)$$

## IV COMPLEXITÉ SPATIALE

C'est la taille mémoire nécessaire pour l'exécution d'un programme ou d'un algorithme.

### IV-1 Taille du programme

Le programme source est exprimé dans un langage puis est compilé,...

Le module exécutable occupe une certaine place mémoire :

$$\sum_{\text{toutes Instr}} (\text{Instructions\_Machine}) * (\text{taille\_instructions})$$

## IV-2 TAILLE DES DONNÉES

Les objets de type entier, réel, caractère, booléen ont une représentation en mémoire exprimée en nombre d'octets, et la place mémoire est alors :

$$\Sigma$$

Données \* ( taille\_représentation )

Toutes les données

Avec la même mesure de la taille des données ( c-a-d  $n$  ), on peut alors avoir une complexité spatiale  $f(n)$  comme pour la complexité temporelle.