Cours: Algorithmes et Structures des Données

Chapitre 8: Les algorithmes de tri et de recherche

Réalisé par:
Dr. Sakka Rouis Taoufik

1

I. Introduction

- Le problème de tri consiste à trier (où à ranger) dans un ordre croissant au sens large une suite d'éléments comparables (dotés d'une relation d'ordre total).
- Le problème de recherche consiste à examiner une suite d'éléments(élément par élément) et voir si **info** appartient ou non à la suite,
- Il existe plusieurs algorithmes permettant de réaliser l'opération de tri et de recherche.
- Ces algorithmes sont classés en deux catégories :
  - **Algorithmes élémentaires** : tri par sélection, tri par insertion, recherche séquentielle sans sentinelle, recherche séquentielle avec sentinelle, etc.
  - **Algorithmes évolués** : tri par tas, tri rapide, recherche dichotomique, etc.

2

II. Le tri par sélection

1) Présentation informelle (ou principe)

Cet algorithme consiste à trouver l'emplacement de l'élément le plus petit du tableau. C'est-à-dire un entier « m » telle que $a_i >= a_m$ pour tout i appartenant à $< a_0, a_1, a_2, \ldots, a_{n-1} >$, avec n est la taille du tableau à trier.

Une fois m est trouvé on échange a_i et a_m ;

On recommence ces deux opérations (emplacement de l'élément le plus petit et échange) sur la nouvelle suite jusqu'à la suite $<a_1, a_2, a_3, \ldots, a_{n-1} >$ soit formée d'un seul élément $<a_{n-1}>$

3

II. Le tri par sélection

2) Réalisation en C

Pour réaliser un algorithme en C, on fait appel au concept (ou construction) sous-programme.

Sous-programme ⇨ **procédure** ou **fonction**

On va fournir l'algorithme de tri sous forme procédure. Il reste à résoudre la communication entre le sous-programme (ici une procédure) et son environnement. On distingue les schémas suivant :

Schéma 1: **Communication implicite**	Schéma 2: **Communication explicite**

#define n 100
int a [n] ;
void tri_selection ( ) {
….
}
/* a est partagée par le sous programme et son environnement */
	void tri_selection (int a [ ], unsigned n) {
	….
	}
	/* a et n sont des paramètres formels */

4

II. Le tri par sélection

A) Solution en C : Schéma 1

```
# define n 100
 int a[n] ; /* tableau à trier*/
  void tri_selection ()
  {
      /* variables locales*/
        unsigned i ;    /*varie entre 0 et n-2. Elle indique l'avancement dans le tri*/
        unsigned j ;   /*varie entre i+1 et n-1. Elle permet de calculer m*/
        unsigned m ; /*emplacement de l'élément le plus petit entre i et n-1 */
        int t              /*pour l'échange a[i] et a[m]*/
    /*****partie executive*****/
    for (i=0 ; i<n-1 ; i++)   {   /*recherche de m*/
                m=i;
                for (j=i+1 ; j<n ; j++)
                        if (a[j] < a[m])
                                m=j;
                  /*échange entre a[i] et a[m]*/
                 t=a[i] ;
                a[i]=a[m] ;
                a[m]=t ;
     }/*fin for i*/
 }/*fin tri_selection*/
```

5

II. Le tri par sélection

B) Tester la procédure :

```
# define n 5
# include <stdio.h>
 int a [n] = {18,14,10,8,4} ;
  void main()  {
                int i ;
                tri_selection ();

                for(i=0;i<n;i++){
                        printf ("%3d  \t", a[i] );
                 }
   }
```

6

II. Le tri par sélection

C) Complexité:

Un algorithme a un coût

complexité → Spatiale : espace mémoire nécessaire

complexité → temporelle : temps exigé

La complexité est exprimée par rapport à la taille du problème résolu par l'algorithme concerné. Pour le problème de tri, la taille est le nombre d'éléments à trier : **n** éléments.

- **Complexité spatiale** : La complexité spatiale est de l'ordre de n. On note O(n) : en effet le tri se fait dans le même tableau.
- **Complexité temporelle** : Le tri par sélection comporte deux types d'opérations élémentaires :
  - recherche de m entre a[i] et a[n]
  - l'échange entre a[i] et a[m]

7

II. Le tri par sélection

C) Complexité:

- **Pour l'échange:** L'échange a lieu systématiquement dans la boucle principale "for (i=0 ; i<n-1 ; i++)" qui s'exécute n-1 fois. Sachant que l'opération d'échange exige 3 affectations. On peut conclure que la complexité en transfert est **O(3n) = O(n)**

➔ **La complexité en nombre d'échange est de l'ordre de n, que l'on écrit O(n).**

8

II. Le tri par sélection

C) Complexité:

- **Pour la recherche de m :** On mesure cette complexité par rapport au nombre de comparaisons de l'opération élémentaire : a[j]<a[m] ? ;

Pour chaque itération (boucle externe) on démarre avec l'élément de position a_i et on le compare avec $a_{i+1}+ a_{i+2}+\ldots+ a_n$

Donc le nombre de comparaison est **(n-i)**

On commence avec i=1 et on termine avec i=n-1, ainsi, le nombre de comparaisons total est =

$(n-1)+ (n-2)+\ldots\ldots+2+1=$**$n*(n-1)/2$**

➔**La complexité en nombre de comparaison est de l'ordre de n^2, que l'on écrit $O(n^2)$.**

II. Le tri par sélection

C) Complexité:

- Toutefois cette complexité en nombre d'échanges de cellules n'apparaît pas comme significative du tri, outre le nombre de comparaison, c'est le nombre d'affectations d'indice qui représente une opération fondamentale.

Conclusion : la complexité est de l'ordre **$O(n^2)$.** On dit que son temps est quadratique par rapport à n (taille du problème).

III. Le tri par insertion

1) Présentation informelle (ou principe)

On suppose que les (i-1) premiers éléments sont triés.

On essaye de trouver la place de l'élément de position i par rapport aux (i-1) éléments déjà triés.

Et ainsi de suite jusqu'à que tous les éléments soient triés

➔En C, la condition d'arrêt est : (i==n).
➔En pseudocode ou Pascal, la condition d'arrêt est : (i=n+1)

III. Le tri par insertion

2) Réalisation en c

```
# define n 100
int a[n] ;/*a : tableau à trier*/
void tri_insertion() {
  unsigned i ; /* niveau d'avancement dans le tri*/
  unsigned j ;   /*pour le décalage*/
  int v ;  /*élément à insérer*/
  for (i=1 ; i<n ; i++) {
      /*insertion de a[i]*/
      v=a[i];
      j=i;
      while(j>0 && a[j-1]>v)   {
         a [ j ]=a [ j-1];
          j--;/*passer à l'élément précédent*/
      }
      a[ j ] = v ;
    }
}
```

III. Le tri par insertion

3) Complexité

Complexité spatiale : La complexité spatiale est de l'ordre de n. On note O(n) : en effet le tri se fait dans le même tableau.

Complexité temporelle :

On identifie l'opération atomique (ou élémentaire) à comptabiliser. On s'intéresse à la boucle interne (à while) et plus précisément, on s'intéresse à l'expression qui gouverne while à savoir j>0 && a[j-1]>v => ou va comptabiliser le nombre de comparaisons a[j-1]>v pour une itération donnée.

Le nombre de comparaisons (a[j-1]>v) n'est pas connu .

Il dépend de la configuration initiale du tableau a.

13

Chapitre 7 : Les algorithmes de tri et de recherche

III. Le tri par insertion

3) Complexité

Complexité temporelle : On distingue les 3 cas suivants :

A) Cas minimum

Ou favorable ou encore optimiste. Pour chaque élément à insérer on fait une comparaison : a[j-1]>v ?

Un tel cas traduit un tableau trié dans le bon ordre.

a

11	12	20	40	60

Ainsi, **C_{min} =n-1 comparaison. Elle est en O(n).**

B) Cas maximum ou défavorable ou encore pessimiste

Pour l'insertion d'un élément de position i, on fait i-1 comparaisons de l'expression a[j-1]>v. On commence par i=2 et on finit avec i=n.

Ainsi, **C_{max} =1+2+…+(n-1)=n(n-1)/2. Elle est en o(n^2).**

Un tel cas traduit un tableau trié dans l'ordre inverse.

a

60	40	20	12	11

14

III. Le tri par insertion

3) Complexité

Complexité temporelle : On distingue les 3 cas suivants :

Cas moyen :

Configuration aléatoires C_{moy} est compris entre o(n) et o(n^2). Hypothèse : pour insérer un élément de position i, on fait en moyenne i/2 comparaisons (a[j-1]>v). Il suffit de diviser C_{max} sur 2.

C_{moy} =n(n-1)/4 : elle est en O(n^2).

NOMBRE GLOBAL DE COMPARAISON

Cas minimum : Cmin=n-1=> O(n)

Cas maximum : Cmax=1+2+3+….+n-1=n(n-1)=>O(n^2)

Cas moyen : Cmoy= de l'ordre de O(n^2)

15

IV. Comparaison entre tri par insertion et tri par sélection

Complexité spatiale : elle est en o(n) pour les deux algorithmes.

Complexité temporelle :

- l'algorithme de tri par sélection ne prend pas en compte la configuration initiale du tableau à trier. Ceci explique **Cmin= =Cmoy= Cmax = O(n^2).**
- L'algorithme de tri par insertion est sensible à la configuration initiale du tableau à trier. On a tendance à comparer les algorithmes résolvant le même problème sur la complexité au pire des cas.
- Les deux algorithmes (tri par sélection et tri par insertion) présentent la même tendance O(n^2). Dans le cas pessimiste.
- En moyenne (en multipliant l'exécution sur des configurations différentes), le tri par insertion est plus efficace que le tri par sélection, car prend en compte la configuration initiale du tableau à trier.

16

V. Le tri rapide

1) Principe

Le tri rapide (quick sort) est fondé sur la méthode de conception diviser pour régner en utilisant les deux étapes suivantes :

A- placer un élément du tableau (appelé pivot) à sa place définitive, en permutant tous les éléments de telle sorte que tous ceux qui sont inférieurs au pivot soient à sa gauche et que tous ceux qui sont supérieurs au pivot soient à sa droite. Cette opération s'appelle le partitionnement.

B- Pour chacun des sous-tableaux (sous tableau gauche et sous tableau droite), on définit un nouveau pivot et on répète l'opération de partitionnement. Ce processus est répété récursivement, jusqu'à ce que l'ensemble des éléments soit trié.

17

V. Le tri rapide

2) Réalisation

```
void main() {
  int n;
  float  t [50];
  do{
        printf("Combien voulez-vous trier de données ?  ");
        scanf("%d",&n);
   } while(n<0 | | n>50);


  remplirTab (t, n);
  tri_rapide (t, 0, n-1);
  afficherTab (t, n);
}
```

18

V. Le tri rapide

2) Réalisation

```
void RemplirTab (float t [ ], int n) {
  int i;
  printf("\n Entrez les %d données réelles trier \n ", n);

  for (i=0;i<n;i++)
          scanf("%f", &t[i]);
}
```


V. Le tri rapide

2) Réalisation

```
int partition (float t [ ], int gauche, int droite) {
  float cle;
  int i,j;
  cle=t [gauche];
  i=gauche+1;
  j=droite;
  while (i<=j)     {
     while (t[i]<=cle)  i++;
     while (t[j]>cle)   j--;
     if (i<j) echanger (t, i++, j--);
    }
  echanger(t, gauche, j);
  return j;
}
```

V. Le tri rapide

2) Réalisation

```
void echanger (float t [ ], int i, int j) {
 float aux;
 aux=t[i];
 t [i]=t[j];
 t [j]= aux;
}

void afficherTab (float t [ ] ,int n) {
   int i;
   printf("\n Voici le tableau trié \n");
   for (i=0;i<n;i++)
     printf("%6.2f ",t [i]);
   printf("\n");
}
```


V. Le tri rapide

2) Réalisation

```
void tri_rapide(float t [ ], int gauche, int droite) {
  int pivot;

  if (gauche<droite)    {

      pivot=partition (t, gauche, droite);

      tri_rapide(t, gauche, pivot-1);

      tri_rapide(t, pivot+1, droite);

    }
}
```

V. Le tri rapide

3) Complexité

Rappelant qu'il il existe plusieurs algorithmes permettant de réaliser l'opération de tri.
Ces algorithmes sont classés en deux catégories :
Algorithmes élémentaires : tri par sélection, tri par insertion, etc. (complexité n^2)
Algorithmes évolués : tri par tas, tri rapide, etc. (complexité $n \log_2(n)$)
(demonstration par recurrence)

VI. La recherche séquentielle

1) Présentation informelle

Un problème de recherche est un problème ayant deux issues :
Echec : info se trouve dans la table
Succès : info ne se trouve pas dans la table

Soient une table a et une information « info » traduisant une caractéristique relative aux éléments stockés dans la table.

L'algorithme de recherche séquentielle (ou linéaire) consiste à examiner la table éléments par éléments et voir si « info » appartient ou non à la table a.

VI. La recherche séquentielle

2) Réalisation

/*Soit un annuaire téléphonique comportant un ensemble de paires <nom, tel> avec nom est le nom de l'abonné et tel son numéro. On vous demande de trouver le numéro de téléphone d'un abonné donné. représentation de l'annuaire en deux tableau indicées en parallèle*/

```
# define n 100
char * nom[n] ;
unsigned tel[n] ;
```


IV. La recherche séquentielle

2) Réalisation (Solution 1:sans sentinelle)

```
int recherche (char *x) {
/* rend -1 si x n'appartient pas à nom sinon tel
correspondant*/
unsigned i ;/*pour parcourir la table nom*/

for(i=0 ;i<n ;i++) {
        if (strcmp (nom[i],x)==0) /*issue positive*/
                return(tel[i]);
}
/*issue négative*/
return -1 ;
}
```

VI. La recherche séquentielle

2) Réalisation (Solution 2 : basée schéma while)

```
int recherche (char * x) {
/*même rôle*/
unsigned i ; /* pour parcourir le tableaux nom*/
i=0 ;
while (i<n && (strcmp(x, nom[i]) !=0)
        i++;  /*pour passer à l'élément suivant*/

/*à la sortie de la  boucle while :  i>=n =>échec  ou
        strcmp(x, nom[i]==0) => succès et    forcement i<n*/
if(i<n)
        return tel[i] ;
else
        return -1 ;
}
```


VI. La recherche séquentielle

2) Réalisation (Solution 3 : avec sentinelle)

```
/*on utilise n= le nombre d'éléments+1 cases et on insère
dans la dernière case l'élément à recherché*/
int recherche (char *x)      { /* rend -1 si x n'appartient pas à
nom sinon tel correspondant*/
  unsigned i ;  /*pour parcourir nom*/
  /*garnir la sentinelle*/
   nom[n-1]=(char*) malloc (sizeof(strlen(x)+1)) ;
   strcpy(nom[n-1], x);
   tel[n-1]=-1 ;
i=0;
/*algorithme de recherche*/
while(strcmp(x,nom[i]!=0))
        i++;
return(tel[i] );  }
```

VI. La recherche séquentielle

3) Comparaison solution2/solution3 :

complexité spatiale :

solution 2→n éléments ➔ O(n)

solution 3→(n+1) éléments ➔ O(n)

complexité temporelle :

solution 2 : à chaque itération, 3 évaluations

solution 3 : à chaque itération, 1 évaluation

On a amélioré le temps de la solution 2 moyennant un espace supplémentaire (1seul élément)

On ne recommande pas la solution 1.

Remarque : Dans une structure linéaire ici les tableaux on peut placer moyennement un espace supplémentaire soit une sentinelle à droite soit à gauche.

VI. La recherche séquentielle

3) Comparaison solution2/solution3 :

-complexité temporelle : l'opération à comptabiliser est la comparaison entre x et l'élément courant : strcmp (x, nom [i])

On distingue :

-Cas minimum : une seule comparaison, un tel cas traduit que x coïncide avec le premier élément du tableau.

-Cas maximum : n comparaisons x coïncide avec le dernier élément ou x n'appartient pas à nom [i] :

- sans sentinelle ➔ n comparaisons
- avec sentinelle ➔ n+1 comparaisons

Elle est en o(n)

-Cas moyen : n/2 comparaisons.

VII. La recherche dichotomique

1) Présentation informelle

La table est triée par ordre croissant sur une caractéristique. Par exemple, l'annuaire téléphonique est trié sur le nom (par ordre alphabétique).

L'algorithme de recherche séquentielle est applicable sur une table triée. Mais on peut faire mieux, en proposant l'algorithme suivant :

On compare « x » par rapport au milieu de notre espace de recherche, si sa coïncidence avec cet élément, alors on s s'arrête avec un résultat positif sinon on s'intéresse soit au sous tableau de gauche (si x < à l'élément du milieu), soit au sous tableau de droite (si x > à l'élément du milieu). Et on respecte ces opérations jusqu'à le sous tableau de recherche soit vide.

VII. La recherche dichotomique

2) Réalisation

```
#include <string.h>
#define n 100
char * nom[n];/*nom est trié par ordre alphabétique*/
unsinged tel[n] ;
int recherche_dichotomique(char *x) {
/*-1 si x n'appartient pas à nom sinon tel correspondant*/
  int g,d ;/*sous tableau de recherche*/
  int m ;/*indice de l'élément au milieu*/
  int comp ;/*résultat de comparaison  x et nom[m]*/
/*initialisation*/
  g=0 ;
  d=n-1 ;
```

VII. La recherche dichotomique

2) Réalisation

```
do {    m=(g+d)/2 ;/*division entière*/
        cmp=strcmp(x,nom[m]) ;
        if(cmp==0)
                return tel[m] ; /*issue positive*/
        /*soit déplacer g soit déplacer d jamais déplacer les
deux à la fois*/
        if(cmp<0) /*x<nom[m]*/
                d=m-1;
        else /*(cmp>0)=> x>nom[m]*/
                g=m+1 ;
}while(g<=d) ;
/*g>d sous tableau vide*/
return -1 ; /*issue négative*/
}
```


VII. La recherche dichotomique

3) Complexité

-Complexité en temps :

On va comptabiliser l'opération de comparaison strcmp(x,nom[m]) ?

On distingue les cas suivant :

-Cas minimum ou optimiste: une seul comparaison, ceci traduit que x coïncide avec nom[0+(N-1)/2].

-Cas maximum ou pessimiste : (dans le pire des cas) un tel cas traduit x n'appartient pas à nom. À chaque itération, on part d'un problème de taille N et moyennant une comparaison (strcmp), on tombe sur un problème de taille N/2. L'algorithme de recherche dichotomique applique le principe « diviser pour résoudre » ou encore « diviser pour régner » : le problème initiale est divisé en deux sous problème **disjoints** et de taille plus ou moins égale.

VII. La recherche dichotomique

3) Complexité

Supposant que N est le nombre d'éléments

On note C_N : le nombre de fois où strcmp est effectuée.

➔$C_N=C_{N/2}+1$

avec $C_1=1$

On pose $N=2^n$ ou $n=\log_2 N$ ➔ $C_N=C_{2^n}$

$C_{2^n}=C_{2^{n-1}}+1$

$= C_{2^{n-2}}+1+1$

$= C_{2^{n-3}}+3$

…

$=C_1+n=1+n=1+\log_2 N$

Ainsi, cet algorithme est $O(\log_2 N)$

-Cas moyen : entre 1 et $\log_2 N$

VII. La recherche dichotomique

3) Complexité

Remarque : le gain apporté par l'application de l'algorithme de recherche dichotomique sur un tableau **trié** peut être illustré par l'exemple suivant :

On souhaite effectuer une recherche sur un tableau trié T de taille N=10000.

Si on applique

- l'algorithme de recherche séquentielle la complexité dans le cas moyen est 5000
- l'algorithme de recherche dichotomique, la complexité au pire des cas est $O(\log_2 10000)\approx 14$

VIII. Exercices d'application

Exercice 1:

On donne un tableau appelé redondant contenant des entiers redondants et dans un ordre quelconque.
Présenter d'une façon informelle puis écrire un programme C permettant de compter la fréquence de chaque élément figurant dans le tableau redondant dans une table.

Exercice 2 :

Écrire un programme C qui permet de calculer le nombre de sous-séquences et la plus longue sous-séquences strictement croissante dans un tableau de n entiers.

VIII. Exercices d'application

Exercice 3:

On possède un tableau de nombres entiers, classés par valeur croissante, chaque entier pouvant être répété plusieurs fois : 3 5 5 5 7 9 9 9 9 10 10 12 12 12 13

On appelle plateau la suite de tous les entiers consécutifs de même valeur : dans l'exemple ci-dessous le plateau de valeur 5 comporte 3 nombre, celui de valeur 10 comporte 2 etc. On appelle la longueur d'un plateau la valeur commune à tous ses termes. Un tableau T de n termes étant donné, trouver la longueur et la valeur de son plus long plateau.