Support de cours
Module : ASD1

Chapitre 7: Les algorithmes de tri et de recherche
Partie ½ : Les algorithmes élémentaires

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

Chapitre 7 : Les algorithmes de tri et de recherche

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 < $a_1, a_2, \dots, a_{n-1}, a_n$ >, 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 <$a_2, \dots, a_{n-1}, a_n$ > soit formée d'un seul élément<a_n>

3

Chapitre 7 : Les algorithmes de tri et de recherche

II. Le tri par sélection

2) Réalisation:

```
Const n100
Type Tab : Tableau [1..n] d'entiers
proc tri_selection ( var a: Tab; n: entier )
var
      /* variables locales*/
       i: entier     /*varie entre 1 et n-1. Elle indique l'avancement dans le tri*/
       j: entier     /*varie entre i+1 et n. Elle permet de calculer m*/
        m: entier   /*emplacement de l'élément le plus petit entre i et n */
       aux: entier /*pour l'échange a[i] et a[m]*/
    /*****partie executive*****/
debut
pour i de 1 à n-1 faire
  /*recherche de m*/
              m ← i;
            pour j de i+1 à n faire
                 si (a[j] < a[m]) alors
                   m ← j;
            fin pour
             /*échange entre a[i] et a[m]*/
             aux=a[i] ;
             a[i]=a[m] ;
             a[m]=aux ;
    fin pour
 fin proc.
```

4

Chapitre 7 : Les algorithmes de tri et de recherche

II. Le tri par sélection

3) Réalisation en C:

```
void tri_selection ( int a [ ], int n)  {

      /* 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

Chapitre 7 : Les algorithmes de tri et de recherche

II. Le tri par sélection

4) Complexité:

Un algorithme a un coût → 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]

6

II. Le tri par sélection

4) Complexité:

- **Pour l'échange:** L'échange a lieu systématiquement dans la boucle principale " pour i de 1 à n-1 faire" 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).

Chapitre 7 : Les algorithmes de tri et de recherche

II. Le tri par sélection

4) 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}+…+ 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)+……+2+1=**(n-1)*(n-1 + 1)/2 = (n-1)*n /2**

➔La complexité en nombre de comparaison est de l'ordre de n², que l'on écrit O(n²).

II. Le tri par sélection

4) 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 i=n.

Chapitre 7 : Les algorithmes de tri et de recherche

III. Le tri par insertion

2) Réalisation: Proposer un algorithme pour cette procédure

Chapitre 7 : Les algorithmes de tri et de recherche

III. Le tri par insertion

3) Réalisation en c

```
void tri_insertion( int a[ ], int n) {
   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

4) 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 (**tant que**) et plus précisément, on s'intéresse à l'expression qui gouverne **tant que** à savoir **j>1 et 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.

Chapitre 7 : Les algorithmes de tri et de recherche

III. Le tri par insertion

4) 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

➔ 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.

➔ 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

III. Le tri par insertion

4) Complexité

Cas moyen :

Configuration aléatoires $\mathbf{C_{moy}}$ est compris entre **o(n)** et **o(n²)**.

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.

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

Conclusion: NOMBRE GLOBAL DE COMPARAISON
Cas minimum : $\mathbf{C_{min}}$=n-1=> O(n)
Cas maximum : $\mathbf{C_{max}}$=1+2+3+….+n-1=n(n-1)=>$O(n^2)$
Cas moyen : $\mathbf{C_{moy}}$= de l'ordre de $O(n^2)$

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 = $\mathbf{O(n^2)}$.**
- 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.

V. 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.

V. La recherche séquentielle

2) Solution 1 (sans sentinelle):

proposer un algorithme pour cette procédure

V. La recherche séquentielle

3) Solution 1 en C:

```
int recherche (int T[], int n, int info) {
   /* rend -1 si info n'appartient pas à T
   sinon la position correspondante*/

unsigned i ;/*pour parcourir le tableau*/

for(i=0 ;i<n ;i++) {
         if (T[i] == info) /*issue positive*/
                  return( i );
}
*issue négative*/
return -1 ;
}
```


VI. La recherche séquentielle

4) Solution 2 : proposer un algorithme basé sur le schéma Tant Que

VI. La recherche séquentielle

5) Solution 2 en C : (basée sur le schéma while)

```
int recherche (int t [ ], int n, int info) {
unsigned i ; /* pour parcourir le tableau*/
i=0 ;
while (i<n && T[i] !=info)
          i++;  /*pour passer à l'élément suivant*/

 /*à la sortie de la  boucle while :
        i>=n            => échec  ou
        T[i][==info   => succès et    forcement i<n   */
if(i<n)
  return i ;
else
  return -1 ;
}
```


VI. La recherche séquentielle

6) Solution 3 : (en utilisant une sentinelle)

L'idée consiste à utiliser n+1 cases et on insère dans la dernière case l'élément à recherché.

VI. La recherche séquentielle

6) Solution 3 en C : (en utilisant une sentinelle)

```
/*on utilise n+1 cases et on insère dans la dernière case
l'élément à recherché*/
int recherche (int t[], int n1; int info)     {
/* rend -1 si info n'appartient pas au tableau sinon la position
correspondante*/
unsigned i ;/*pour parcourir nom*/
T[n1]=info ; /*garantir le non dépassement du tableau */
while(T[i] != info)
        i++;
if( i==n1)
        return( -1 ) ;
else
        return( i ) ;
 }
```


Chapitre 7 : Les algorithmes de tri et de recherche

VI. La recherche séquentielle

7) 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

7) Comparaison solution2/solution3 :

-complexité temporelle : l'opération à comptabiliser est la comparaison entre x et l'élément courant

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. Exercices d'application

Soit T un tableau contenant n éléments de type entier et x un entier quelconque.

Exercice 1:

Écrire une fonction qui calcule le nombre d'apparitions de x dans le tableau T.

Exercice 2:

Écrire une fonction qui calculer le nombre d'éléments distincts de T.

Exercice 3 :

Présenter d'une façon informelle et réaliser un algorithme permettant de compter la fréquence des éléments stockés dans un tableau. Ces éléments sont des entiers compris entre 1 et 100.