Metoda Divide et Impera

Curs 4

Tematică

1. Prezentare generală
2. Implementări ale metodei

1. Prezentare generală

- In informatică, această strategie reprezintă o metoda de rezolvare a problemelor.
- *Ideea de bază constă în împărțirea unei probleme în 2 sau mai multe subprobleme care se rezolvă separat, apoi se trece la combinarea rezultatelor problemelor rezolvate obținându-se, astfel, soluția finală.*
- La baza problemelor rezolvabile prin această metodă stă următorul enunț:
- Se da un șir de valori (secvență de valori) a1, a2, a3,…, an.
- Aceasta secvență trebuie prelucrată.

1. Prezentare generală

Prelucrearea secvenței se face astfel:

1. Șirul se împarte în 2 sau mai multe subșiruri
2. Fiecare subșir se va împărți, după aceiași metodă, în 2 sau mai multe subșiruri până când se ajunge la o problema rezolvabilă sau un rezultat cunoscut
3. Din aproape în aproape, prin combinarea rezultatelor obținute, se obține rezultatul final

1. Prezentare generală

- Presupunem că pentru orice p, q ∈ N, 1 ≤ p < q ≤ n, (∃) m din mulţimea { p,...,q-1 } astfel încât:

– prelucrarea secvenţei { a_p,...,a_q }, se face prelucrând secvenţele vecine următoare

{a_p,..., a_m}, { a_{m+1},...,a_q }

– şi apoi combinând rezultatele pentru a obţine prelucrarea întregii secvenţe

{ a_p,...,a_q }.

Tehnica DI – rezumat

1. Împarte problema în subprobleme de **același** tip

2. Rezolvă fiecare subproblemă independent

3. Combină soluțiile subproblemelor pentru a obține soluția problemei inițiale

4. DI admite implementare recursivă – problema permite descompunere succesivă în subprobleme

1. Prezentare generală

Următoarea procedură exemplifică metoda DI

```
procedure DivideEtImpera(p, q, a)
begin
        if q-p <= ε then
                Prelucreaza(p, q, a)
        else
                begin
                        Divide(p, q, m)
                        DivideEtImpera(p, m, b)
                        DivideEtImpera(m+1, q, c)
                        Combina(b, c, a)
                end
end
```

1. Prezentare generală

- Am notat prin ε lungimea maximă a unei secvenţe {ap,...,aq }
- Dacă dimensiunea problemei iniţiale sau a subproblemelor aparute este mai mică decât ε, atunci problema se rezolvă direct prin procedura Prelucreaza, soluţia fiind în vectorul a.
- Soluţia se pune în vectorul a, iar *soluţiile parţiale se pun în vectorii b, respectiv c*.
- Procedura DIVIDE împarte secvenţa în două subsecvenţe {ap,...,a_m} şi {a_{m+1},...,a_q}, obţinând rezultatul în m.
- Prin cele două autoapelări ale procedurii DivideEtImpera se rezolvă subproblemele punând rezultatele prelucrărilor în b şi respectiv în c.
- Procedura Combina obţine din soluţiile subproblemelor, soluţia problemei date.
- La început procedura DivideEtImpera se apelează cu p=1 şi q=n.

1. Prezentare generală

Exemple de probleme ce se rezolvă prin această metodă:

- Calculul sumei/produsului elementelor unui șir
- Determinarea minimului/maximului dintr-un șir
- Să se verifice anumiți termeni din șir care au o anumită proprietate dată (sunt pare / sunt prime / sunt pozitive, etc)
- Sortarea elementelor dintr-un șir de valori

1. Prezentare generală

Subprogram Divide_Et_Impera(inceput, sfarsit, rezultat)

/* inceput - pozitia primului termen din sir/subsir;

sfarsit - pozitia ultimului termen din sir/subsir */

daca (s-a terminat divizarea sirului in unu sau doi termeni) **atunci**

Prelucrez_secventa_divizata(inceput, sfarsit, rezultat)

altfel

Divizez_secventa(inceput, sfarsit, m) /* nu in toate problemele m reprezinta pozitia elementului de la mijloc */

Divide_Et_Impera(inceput, m, rezultat1)

Divide_Et_Impera(m, sfarsit, rezultat2)

Combin_rezultate(rezultat1, rezultat2, rezultat)

Sfarsit subprogram

2. Implementări ale metodei

Sortarea unui vector A cu n elemente prin interclasare(Mergesort)

- Descompunerea unui vector în alţi doi vectori ce urmează a fi sortaţi, are loc până când avem de sortat vectori cu maxim două componente
- Algoritmul folosește următorele proceduri:

1. Procedura ***SORT*** ordonează crescător un vector de maxim două elemente şi corespunde procedurii **Prelucreaza** din schema generală.
2. Procedura **INTERCLASARE** interclasează rezultatele şi corespunde procedurii. **Combina** din schema general.
3. Procedura **DIVIDE_IMPERA** implementează strategia generala a metodei.
4. Procedura **DIVIDE** este realizată prin instrucţiunea:

$$m = [(p + q) / 2]$$

2. Implementări ale metodei

Procedura SORT

```
procedure SORT(p, q, A)  // A, adica vectorul a  ->  a[100]
begin
        if a_p > a_q then
                begin
                        t <- a_p
                        a_p <- a_q
                        a_q <- t
                end
end
```

2. Implementări ale metodei

Procedura DIVIDE_ET_IMPERA

```
procedure DIVIDE_IMPERA(p, q, A)
begin
        if q – p ≤ 1 then SORT(p, q, A)
        else
                begin
                        m=[( p + q ) / 2]
                        DIVIDE_ET_IMPERA(p, m, A)
                        DIVIDE_ET_IMPERA(m+1, q, A)
                        INTERCLASARE(p, q, m, A)
                end
end
```

2. Implementări ale metodei

Procedura INTERCLASARE

```
procedure INTERCLASARE(p, q, m, A)
vector A, B
/* B - vectorul in care se construieste vectorul ordonat prin interclasarea celor 2 subvectori */
/* introducem in B elementele din cei 2 subvectori in ordine crescatoare */
begin
        i<-   p,     j<-   m+1,        k<-   0
        while (i≤m) and (j≤q)
```

2. Implementări ale metodei

Procedura INTERCLASARE

begin

if $a_i \leq a_j$ then

/* dacă elementul curent din primul subvector este mai mare decat elementul curent din cel de-al doilea subvector, se va introduce în vectorul intermediar cel

din cel de-al doilea subvector, altfel se va introduce cel din primul subvector */

2. Implementări ale metodei

Procedura INTERCLASARE

begin

b_k<- a_i

i<- i+1

end

else

begin

b_k<- a_j

j<- j+1

end

k<- k+1

end // sfârșit while

2. Implementări ale metodei

Procedura INTERCLASARE

/*introducem in b elementele ramase in subvectorul din care nu s-au copiat toate elementele*/

if $i \leq m$ then

for j=i to m do

begin

b_k<- a_j

k<- k+1

end

else

for i=j to q do

begin

b_k<- a_i

k<- k+1

end

2. Implementări ale metodei

Procedura INTERCLASARE

/*copiem elementele vectorului intermediar in vectorul inițial - cel care va avea elementele sortate*/

```
        k<-  0
        for i=p to q do
        begin
                a_i<- b_k
                k<-  k+1
        end
end //sf. procedură
```

2. Implementări ale metodei

Funcția main ()

begin

for i=0 to n-1 do

citeste a_i

DIVIDE_ET_IMPERA(0, n-1, a)

for i=0 to n-1 do

scrie a_i

end

2. Implementări ale metodei

Algoritmul de la **Sortare rapida(quickSort)** se bazeaza pe metoda Divide et Impera.

• Imparte: Sirul de pe pozitiile p...u este impartit (rearanjat) in doua siruri nevide de elemente.

– Elementele celor 2 subsiruri se gasesc pe pozitiile p...m si respectiv m+1...u.

– Primul sir are proprietatea ca fiecare element din primul sir este mai mic decat orice element din al doilea sir.

• Stapaneste: Cele doua siruri sunt ordonate prin apeluri succesive ale algoritmului de sortare rapida (quickSort)

• Combina: Deoarece cele doua siruri sunt sortate pe loc, nu este nevoie de nicio ordonare.

2. Implementări ale metodei

```
#include <iostream.h>
int a[50], n, i;
void quickSort (int a[ ], int
p, int u)
{
int i = p, j = u;
int m;
int pivot = a[(p + u) / 2];
while (i <= j)
{
while (a[i] < pivot) i++;
while (a[j] > pivot) j--;
```

2. Implementări ale metodei

```
if (i <= j)
{
m=a[i];
a[i] =a[j];
a[j] = m;
i++;
j--;
}
}
if (p < j) quickSort(a, p, j);
if (i < u) quickSort(a, i, u);
}
```

2. Implementări ale metodei

```
int main()
{
int n, i;
cout<<"n="; cin>>n;
for(i=1;i<=n;i++)
{
cout<<"a["<<i<<"]=";
cin>>a[i] ;
}
quickSort(a,1,n);
cout<<"Sirul sortat : "<<endl;
for(i=1;i<=n;i++)
cout<<a[i]<<" ";
}
```

2. Implementări ale metodei

Algoritmul de **căutarea binară**

Fie un vector v cu n elemente ordonate crescător şi un număr nr.

Să se caute acest număr în vector şi în caz că este găsit să se afiseze indicele pe care se găseşte.

Paşi:

1.verificam dacă x =$v\left[\frac{n}{2}\right]$, adică dacă este egal cu valoarea din mijlocul vectorului.

2. Dacă da atunci funcţia se opreşte cu succes.

Am găsit elementul x pe poziţia n/2

2. Implementări ale metodei

3. Dacă nu atunci

• dacă x < $v\left[\frac{n}{2}\right]$ cautam în jumătatea $\left[v[0], v\left[\frac{n}{2}-1\right]\right]$

• dacă x > $v\left[\frac{n}{2}\right]$ cautam în jumătatea $\left[v\left[\frac{n}{2}+1\right], v[n-1]\right]$

• căutarea în unul dintre cele două intervale se face în mod similar: comparam cu jumătatea intervalului.

Dacă elementul este egal cu x, ne oprim, dacă nu verificam în care parte se poate afla x.

2. Implementări ale metodei

Recursivitatea: de fapt, la fiecare pas căutăm în intervalul cuprins între poziţia i şi poziţia j (iniţial i = 0 şi j = n).

(1) Dacă i>j

algoritmul se încheie cu insucces (s-a căutat în tot vectorul şi nu s-a găsit x)

(2) Dacă i<=j atunci

- dacă x = $v\left[\frac{i+j}{2}\right]$ - funcţia se opreşte cu succes

- altfel compar $v\left[\frac{i+j}{2}\right]$ cu x. Dacă x e mai mare căutam la dreapta lui $v\left[\frac{i+j}{2}\right]$, altfel caut la stânga lui $v\left[\frac{i+j}{2}\right]$.

2. Implementări ale metodei

```
#include<iostream.h>
int n,i,nr,v[100];
int caut(int i,int j)
{
        if(nr==v[(i+j)/2]) return ((i+j)/2);
        else
                if(i<j)
                        if(nr < v[(i+j)/2]) return(caut(i, (i+j)/2  - 1));
                        else return(caut((i+j)/2 + 1, j));
}
```

2. Implementări ale metodei

```
int main(void)
{
cout<<"Dati numarul de elemente ";   cin>>n;
cout<<"\n Dati elementele vectorului \n";
for(i=1;i<=n;i++)
{
        cout<<"v["<<i<<"]= ";
        cin>>v[i];
}
cout<<"Dati numarul care se cauta ";
cin>>nr;
cout<<"Am gasit elementul pe pozitia "<<caut(1,n);
}
```

2. Implementări ale metodei

Algoritmul Turnurile din Hanoi

Se dau trei tije numerotate cu A, B, C şi n discuri perforate având diametre diferite.

Iniţial toate discurile sunt asezate pe tija A, în ordinea crescătoare a diametrelor lor, considerând sensul de la vârful tijei la baza ei.

Să se mute toate discurile pe tija B în aceeaşi ordine, folosind tija C şi respectând următoarele reguli:

– la fiecare pas se mută un singur disc

– în permanenţă pe fiecare tijă deasupra fiecărui disc pot apare numai discuri de diametre mai mici.

2. Implementări ale metodei

Rezolvarea acestei probleme se bazează pe urmatoarele considerente logice:

• daca n=1, atunci mutarea este imediata A -> B

(mutam discul de pe A pe B)

• daca n=2, atunci sirul mutarilor este: A -> C, A -> B, C -> B

• daca n>2 procedam astfel:

- mutam (n-1) discuri A -> C
- mutam un disc A -> B
- mutam cele (n-1) discuri C -> B

2. Implementări ale metodei

Observam ca problema initiala se descompune in trei subprobleme mai simple, similare problemei initiale:

- mutam (n-1) discuri A -> C
- mutam ultimul disc pe B
- mutam cele (n-1) discuri C -> B

Dimensiunile acestor subprobleme sunt: n-1, 1, n-1.

- Aceste subprobleme sunt independente, deoarece tija initiala (pe care sunt dispuse discurile), tija finala si tija intermediare sunt diferite.

2. Implementări ale metodei

Notam:

H(n,A,B,C) = sirul mutarilor a n discuri de pe A pe B, folosind C.

PENTRU

n=1 A -> B

n>1 H(n,A,B,C) = H(n-1,A,C,B), A -> B, H(n-1,C,B,A)

2. Implementări ale metodei

Implementarea solutiei:

```
#include <iostream.h>
int n;
void hanoi(int n, char a, char b, char c)
{
	if(n==1) cout<<a<<" -> "<<b<<endl;
/* daca avem un singur disc pe tija A il mutam pe tija B */
```

2. Implementări ale metodei

```
else {
        hanoi(n-1, a, c, b);
/* mutam n-1 discuri de pe tija A, pe tija C folosind ca tija intermediara tija B */
        cout<<a<<" -> "<<b<<endl;
/* mutam discul ramas de pe tija A pe tija B */
        hanoi(n-1, c, b, a);
/* mutam cele n-1 discuri de pe tija C, pe tija B folosind ca tija intermediara tija B */
        }
} // sf procedură hanoi
```

2. Implementări ale metodei

```
int main()
{
cout<<"Numarul de discuri pe tija A = ";
cin>>n;
cout<<"Mutarile sunt urmatoarele:\n";
hanoi(n, 'A', 'B', 'C');
/* mutam n discuri de pe tija A, pe tija B folosind tija intermediara C, - A,B,C sunt numele discurilor */
}
```

2. Implementări ale metodei

Maximul în vectorul neordonat

Problema: Se dă un vector neordonat cu n elemente. Să se găsească maximul folosind metoda DI.

Rezolvare:

• Se pornește de la următoarea observație:

$\text{Max(v}[0], \dots, v[n-1]) = \max(Max(v[0], \dots, v[k]), Max(v[k+1], \dots, v[n-1]))$

$$\max(a,b) = \begin{cases} a, a > b \\ b, b \geq a \end{cases}$$

Astfel, problema admite descompunerea în subprobleme de același tip

Problema/subproblemele se descompun succesiv până nu mai este necesară descompunerea și rezolvarea este simplă – căutarea maximului între 1 sau 2 elemente

2. Implementări ale metodei

Maximul în vectorul neordonat

```
int DI(int a[10], int ic, int sf){
        int m1, m2;
        if (ic<fs){
                m1 = DI(ic,(ic+sf)/2);
                m2 = DI((ic+sf)/2 + 1,sf);
                if (m1<m2) return m2;
                else return m1;
        }else
                return a[ic];
}
```

```
void main(){
{
        cout << DI(a, 0,n-1));
}
```

2. Implementări ale metodei

Minimul în vector neordonat

Tema de seminar!

Considerați cazul în care problema este direct rezolvabilă dacă numărul de elemente este 2, adică avem doar ic și sf, deci diferența dintre cele două este 1.

Pentru asta comparați valorile de pe cele două poziții în vectorul a.