ALTE METODE DE PROIECTARE A ALGORITMILOR

CURS 13- 25.05.2021

Titular: Şef. Lucr. Dr. Mat. Cărbureanu Mădălina

Universitatea Petrol-Gaze din Ploieşti

OBIECTIVE:

- **METODA DIVIDE ET IMPERA (D&I):**
  - **PRINCIPIU METODĂ + MOD DE LUCRU;**
  - **PROBLEME REZOLVABILE PRIN D&I;**
  - **EXEMPLU.**
- **METODA GREEDY:**
  - **PRINCIPIU METODĂ + MOD DE LUCRU;**
  - **PROBLEME REZOLVABILE PRIN GREEDY;**
  - **EXEMPLU.**
- **METODA BACKTRACKING:**
  - **PRINCIPIU METODĂ + MOD DE LUCRU;**
  - **RUTINA BACKTRACKING;**
  - **PROBLEME REZOLVABILE PRIN BACKTRACKING;**
  - **EXEMPLU.**

Metoda Divide et Impera - "Împarte/dezbină și stăpânește"

- **Metoda Divide et Impera** ➡ cuprinde o clasă de tehnici algoritmice ce constau în descompunerea problemei de rezolvat (de obicei, o problemă de o dimensiune considerabilă), în două sau mai multe subprobleme mai simple, de același tip, rezolvarea recursivă a subproblemelor obținute și combinarea soluțiilor subproblemelor, în vederea obținerii soluției problemei inițiale;

➡ în multe dintre cazuri, D&I este o metodă simplă și puternică de rezolvare a problemelor considerate;

➡ tehnică ce admite implementarea recursivă;

➡ presupune parcurgerea următoarelor etape:

- împărțirea problemei inițiale în două sau mai multe subprobleme de același tip și dimensiune, subprobleme care de obicei se rezolvă pe cale recursivă (*Divide*);
- combinarea soluțiilor parțiale (soluțiilor subproblemelor) pentru a obține soluția problemei inițiale (*Impera*).

Metoda Divide et Impera - "Împarte/dezbină și stăpânește"

adhoc reprezintă subalgoritmul folosit pentru rezolvarea subcazurilor problemei

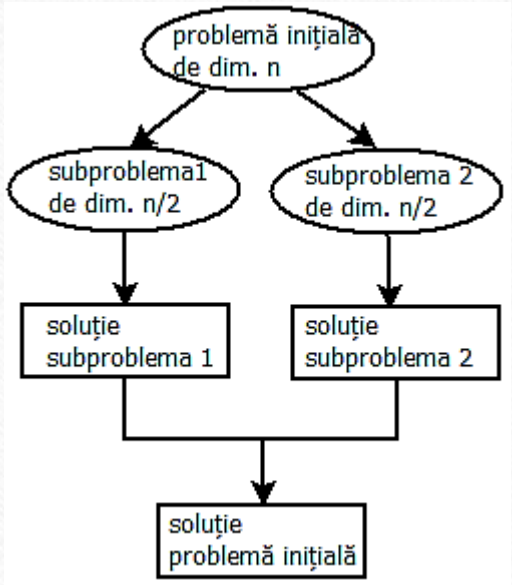

Fig. 1. Divide et Impera-mod de lucru

(1) funcţie divimp(x)
(2) {returnează o soluţie pentru cazul x;}
(3) dacă x este suficient de mic atunci divimp←adhoc(x);
(4) {descompune x în subcazurile x_1, x_2, ..., x_k;}
(5) pentru i←1,..., k execută y_i←divimp(x_i);
(6) {recompune y_1, y_2,..., y_k în scopul obţinerii soluţiei y pentru x;}
(7) divimp←y;

Fig. 2. Divide et Impera - descriere generală

Metoda Divide et Impera - "Împarte/dezbină și stăpânește"

- **Probleme rezolvabile prin D&I:**
  - maximul/minimul elementelor dintr-un vector;
  - căutarea binară;
  - sortarea prin interclasare (Merge Sort);
  - problema Turnurilor din Hanoi;
  - sortarea rapidă (Quick Sort);
  - calculul puterii unui număr;
  - problema dreptunghiului de arie maximă;
  - înmulțirea a două matrici pătratice, etc.

Metoda Divide et Impera - "Împarte/dezbină și stăpânește"

❑ **Exemplu ⟹ Căutarea binară**

- Fie şirul a[10] ordonat crescător cu *n*=6 elemente numere întregi, respectiv a[1]=1, a[2]=2, a[3]=3, a[4]=4, a[5]=5 şi a[6]=6, iar valoarea căutată este *val*=5.
- Aplicând metoda Divide et Impera, problema iniţială se împarte în trei subprobleme mai simple, de acelaşi tip:
  - o primă subproblemă este dată de o singură valoare, şi anume valoarea din mijlocul şirului (a[m]), unde m←$(ind_1 + ind_2)/2$, ind_1←1 şi ind_2←n;
  - a două subproblemă reprezentată de subşirul stâng obţinut prin împărţirea şirului iniţial în două subşiruri (subprobleme), respectiv a[1]=1,..., a[m-1]=2;
  - a treia subproblemă reprezentată de subşirul drept obţinut prin împărţirea şirului iniţial în două subşiruri (subprobleme), respectiv a[m+1]=4,... , a[n]=6.
- **Algoritmul constă în parcurgere următorilor paşi:**
  - dacă valoarea căutată este chiar mijlocul şirului a[m] (a[3]=3), atunci algoritmul se încheie;
  - dacă valoare căutată este mai mare decât jumătatea şirului, atunci căutarea se efectuează în subşirul drept a[m+1]=4,... ,a[n]=6;
  - dacă valoarea căutată este mai mică decât jumătatea şirului, atunci căutarea se efectuează în subşirul stâng a[1]=1,..., a[m-1]=2.

Metoda Divide et Impera - "Împarte/dezbină și stăpânește"

- aplicând Divide et Impera, problema inițială se descompune în următoarele subproblemele:
- subproblema reprezentată de subvectorul a[1]=1, a[2]=2, subproblema cu un singur element a[3]=3 şi subproblema reprezentată de subvectorul a[4]=4, a[5]=5 şi a[6]=6;
- deoarece valoarea căutată *val*=5 este mai mare strict decât jumătatea a[3]=3, căutarea continuă în subproblema reprezentată de subvectorul a[4]=4, a[5]=5 şi a[6]=6;
- la acest pas, m←(ind$_1$+ ind$_2$)/2=5, ind$_1$←4 şi ind$_2$←6;
- se descompune subproblema reprezentată de subvectorul a[4]=4, a[5]=5 şi a[6]=6 în următoarele subprobleme:
  - subproblema cu un singur element a[4]=4;
  - subproblema cu un singur element a[5]=5;
  - subproblema cu un singur element a[6]=6;
- pentru că valoarea căutată a[m]==*val* (a[5]=5), algoritmul se încheie deoarece a fost identificată valoarea căutată.

Metoda Divide et Impera - "Împarte/dezbină și stăpânește"

Căutare binară-Divide et Impera

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
int a[10],i,n,ind1,ind2,val, m;
void cautare_binara_DV(int ind1,int ind2)
{if(ind1>ind2) cout<<"Valoarea cautata"<<val<<"nu este
element al sirului!";
  else
    {m=(ind1+ind2)/2;
     if(val==a[m]) cout<<"Valoarea cautata"<<val<<"se afla
in sir pe pozitia"<<m<<".";
         else if(val>a[m]) cautare_binara_ DV(m+1,ind2);
            else cautare_binara_DV(ind1,m-1);
     }
}
```

```
void main()
 {clrscr();
  cout<<"\n Nr. de elemente vector:";
  cin>>n;
  for(i=1;i<=n;i++)
  {cout<<"a["<<i<<"]=";
   cin>>a[i];}
  cout<<"\n Valoarea cautata este:";
  cin>>val;
  cautare_binara_DV(1,n);
  getch();
  }
```

Metoda Greedy (“avidă, lacomă”)

Metoda Greedy:

- tehnică de proiectare a algoritmilor care se aplică problemelor de optimizare (determinarea celor mai scurte drumuri într-un graf, determinarea arborelui parţial de cost minim al unui graf, etc.), metodă ce furnizează o singură soluţie şi nu mai multe;
- metoda se aplică problemelor în care pentru o mulţime dată A cu n elemente $(a_1, a_2, \ldots, a_n)$ se cere determinarea unei submulţimi (mulţimea soluţiilor) S ($S \subseteq A$) cu m elemente (m<=n) $(s1,s2,...,sm)$ submulţime ce trebuie să îndeplinească anumite condiţii interne (S acceptabilă) şi să fie optimală (să realizeze un minim sau un maxim);
- pentru obţinerea soluţiei finale a unei probleme, se determină pas cu pas soluţia optimă a problemei, soluţie obţinută după anumite criterii din mulţimea soluţiilor posibile;
- denumirea metodei provine din modul de lucru al acesteia, deoarece la fiecare pas este ales întotdeauna cel mai bun element la momentul respectiv, element ce rămâne în soluţie indiferent de ceea ce se întâmplă mai departe, iar atunci când un element este exclus din soluţie, acesta nu va mai fi luat în calcul niciodată.

Metoda Greedy (“avidă, lacomă”)

- **<u>Principiu metodă Greedy:</u>**
  - mulţimea soluţiilor S se iniţializează cu mulţimea vidă ($S \leftarrow \emptyset$);
  - la fiecare pas se alege un anumit element a_i $(i \leftarrow 1,...,n)$, neales la paşii precedenţi din A, care poate conduce la soluţie optimă;
  - se verifică dacă elementul considerat a_i $(i \leftarrow 1,...,n)$ poate fi adăugat în mulţimea soluţiilor S (dacă elementul curent prin compararea sa cu soluţia anterioară conduce la o soluţie optimă, acesta devine noua soluţie posibilă):
  - în caz afirmativ, elementul a_i $(i \leftarrow 1,...,n)$ este adăugat la mulţimea soluţiilor ($S \leftarrow S \cup \{a_i\}$), iar acest element odată introdus în mulţimea soluţiilor S, nu poate fi eliminat;
  - în caz contrar, elementul nu se mai testează ulterior;
  - procedeul continuă până când au fost determinate toate elementele din mulţimea soluţiilor S.

Metoda Greedy (“avidă, lacomă”)

```
(1) funcţie Greedy (A, n, S);
(2) S← ∅;
(3) pentru i←1, n execută
(4)   selectează (A, i, a_j); a_j∈{a_i, …, a_n}
(5)   acceptabil(S, a_j, t_1);
(6)     dacă t_1==1 atunci
(7)         sol_acceptabilă(S, a_j);
(8)         sol_posibilă(S, t_2);
(9)         dacă t_2==1 atunci exit;
(10)  sfârşit;
(11)sfârşit;
```

Fig. 3. Metoda Greedy-caz general

- mulţimea soluţiilor S iniţial este vidă ($S \leftarrow \emptyset$), deoarece este posibil ca problema să nu prezinte soluţie;
- se parcurg pe rând elementele mulţimii A şi la fiecare pas se alege, pe baza unui criteriu de selecţie, un element a_j neales la paşii anteriori (*selectează (A, i, a_j); $a_j \in \{a_i, \ldots, a_n\}$*) pentru a fi adăugat la soluţia acceptabilă S;
- dacă elementul curent a_j este acceptabil (*dacă t_1==1 atunci sol_acceptabilă(S, a_j);*) atunci se determină noua soluţie acceptabilă $S \leftarrow S \cup \{a_j\}$;
- se testează (*dacă t_2==1 atunci exit;*) dacă noua soluţie acceptabilă $S \leftarrow S \cup \{a_j\}$ este soluţie posibilă; în caz afirmativ, se încheie algoritmul pentru că a fost găsită soluţia problemei;
- funcţiile *selectează ( )* şi *acceptabil( )* sunt specifice fiecărei probleme în parte.

Metoda Greedy (“avidă, lacomă”)

- **Probleme rezolvabile prin Greedy (furnizează soluţie optimă):**

- problema restului de plată;
- problema submulţimii cu valoare maximă;
- problema continuă a rucsacului;
- problema spectacolelor;
- minimizarea timpului mediu de aşteptare;
- interclasarea optimă a şirurilor ordonate;
- probleme de determinare a arborelui parţial de cost minim într-un graf, probleme ce impun utilizarea algoritmilor Greedy de determinarea a arborelui parţial de cost minim, respectiv algoritmul lui Kruskal şi algoritmul lui Prim; aceşti algoritmi de tip Greedy au fost prezentaţi în secţiunea 8.1.6. (*Algoritmi pentru determinarea arborelui parţial de cost minim*);
- probleme de determinare a celor mai scurte drumuri într-un graf, probleme ce impun utilizarea algoritmului lui Dijkstra, acesta fiind un algoritm Greedy prezentat în secţiunea 8.1.5 (*Algoritmi pentru determinarea celor mai scurte drumuri într-un graf*).

Metoda Greedy (“avidă, lacomă”)

- **Exemplu: problema restului de plată;**
- **Enunţ problemă**: se cere achitarea unui rest de plată (R) cu un număr minim de bancnote, având la dispoziţie o mulţime de bancnote $A=\{ a_1, a_2, \ldots, a_n\}$ de diferite tipuri (din fiecare tip de bancnote se poate folosi un număr nelimitat de bancnote).
- **Obs:** Pentru ca problema să aibă soluţie, se consideră că există şi bancnote cu valoarea 1;
- **Elementele problemei sunt următoarele:**
  - mulţimea bancnotelor $A=\{ a_1, a_2, \ldots, a_n\}$;
  - o soluţie acceptabilă este o submulţime de bancnote S_j, astfel încât restul $R_j \leq R$;
  - o soluţie posibilă este o soluţie acceptabilă S_j, astfel încât $R_j = R$;
  - o soluţie optimă este o soluţie posibilă S_p, astfel încât fiecare element $s_p=\min\{s_1,\ldots, s_q\}$;
  - criteriul de selecţie a bancnotelor din mulţimea A este $a_j=\max\{a_1, a_2, \ldots, a_n\}$, unde a_j nu a fost selectat încă.

Metoda Greedy (“avidă, lacomă”)

```
(1) funcţie plată(A, n, S, N);
(2) sortare descrescătoare vector de bancnote A;
(3) S← ∅; N← ∅;
(4) pentru i←1,..., n execută
(5)     acceptabil(R, a_i, n_i, t_1);
(6)       dacă t_1==1 atunci
(7)           sol_acceptabilă(S, a_i, N, n_i);
(8)           sol_posibilă(R, t_2);
(9)           dacă t_2==1 atunci exit;
(10)      sfârşit;
(11) sfârşit;
```

Fig. 4. Problema restului de plată - Metoda Greedy

- funcţia *acceptabil*(R, a_i, n_i, t_1) calculează $n_i \leftarrow [R/a_i]$, $R \leftarrow R - n_i \times a_i$ şi valoarea lui $t_1=1$ dacă $n_i>0$;
- funcţia *sol_acceptabilă*(S, a_i, N, n_i) determină $S \leftarrow S \cup \{a_i\}$ şi $N \leftarrow N \cup \{n_i\}$;
- funcţia *sol_posibilă*(R, t_2) determină valoarea lui t_2 ($t_2=1$ dacă $R=0$);
- vectorul n_i este vectorul soluţiilor problemei furnizat de metoda Greedy.

Metoda Greedy (“avidă, lacomă”)

Problema restului de plată - Metoda Greedy

```
#include<stdio.h>
#include<conio.h>
#include<iostream.h>
int a[100],nr,i,n[100],r, aux;
void citire_vector_bancnote(int a[],int nr)
{for(int i=1;i<=nr;i++)
  {cout<<"a["<<i<<"]=";
   cin>>a[i];}}
void afisare_vector_bancnote(int a[] ,int nr)
{for(int i=1;i<=nr;i++)
 cout<<a[i]<<" ";}
void sortare_vector_bancnote(int a[] ,int nr)
{for(int i=1;i<=nr-1;i++)
 for(int j=i+1;j<=nr;j++)
    if(a[i]<a[j])
          {aux=a[i]; a[i]=a[j];  a[j]=aux;}
}
void main()
{clrscr();
cout<<"\n Numarul de tipuri de bancnote este:";
cin>>nr;
citire_vector_bancnote(a, nr);
cout<<”\n Vectorul bancnotelor este:”;
afisare_vector_bancnote(a, nr);
cout<<"\n Restul care trebuie platit este:";
cin>>r;
sortare_vector_bancnote(a,nr);
cout<<"\n Vectorul bancnotelor ordonat descrescator este:";
cout<<endl;
afisare_vector_bancnote(a, nr);
for(i=1;i<=nr;i++)
{n[i]=r/a[i];
 r=r-n[i]*a[i];
}
cout<<”Solutie:”;
for(i=1;i<=nr;i++)
 if(n[i]>0)
 cout<<"\n"<<n[i]<<"bancnote de"<<a[i]<<"lei";
 getch();}
```

Metoda Backtracking (“căutare cu revenire”)

- se aplică problemelor în care soluţia se poate reprezenta sub forma unui vector $s=(x_1, x_2, \ldots, x_n) \in S$, unde $S=S_1 \times S_2 \times \ldots \times S_n$ ($x_1 \in S_1$, $x_2 \in S_2, \ldots, x_n \in S_n$) se numeşte **spaţiul soluţiilor posibile**, iar o soluţie $s \in S$ se numeşte **soluţie posibilă**;
- în cadrul unei probleme, între elementele $x_1, x_2, \ldots, x_n$ există anumite condiţii (relaţii), numite **condiţii (restricţii) interne**;
- soluţiile unei probleme prin această metodă, numite **soluţii rezultat (soluţii finale)** se obţin din acele soluţii posibile care îndeplinesc condiţiile (restricţiile) interne;
- **Principiu tehnică Backtracking:**
  - se construieşte soluţia pas cu pas, respectiv $x_1, x_2, \ldots, x_n$;
  - dacă pentru o valoarea aleasă se constată că nu se poate ajunge la soluţie, atunci se renunţă la respectiva valoare, iar căutarea se reia din punctul în care s-a ajuns (de la aceste întoarceri provine şi denumirea acestei metodei).

Metoda Backtracking (“căutare cu revenire”)

- **Metoda Backtracking** se poate descrie astfel:
  - elementul x_k $(k=1,\ldots, n)$ primește pe rând valori din S_k $(k=1,\ldots, n)$, numai dacă elementele precedente $x_1, x_2, \ldots, x_{k-1}$ au primit valorile corespunzătoare;
  - după atribuirea valorii elementului x_k și înainte de a se atribui o valoare elementului x_{k+1}, se verifică îndeplinirea unor **condiții de continuare** derivate din condițiile interne, prin care se verifică dacă se poate realiza sau nu trecerea la pasul $k+1$;
  - dacă aceste condiții de continuare nu sunt îndeplinite, valorii x_k i se asociază o altă valoare;
  - dacă valorile lui S_k $(k=1,\ldots, n)$ au fost epuizate, se revine la pasul $k-1$ unde se asociază o altă valoare elementului x_{k-1}.

Metoda Backtracking (“căutare cu revenire”)

```
(1) void Backtracking()
(2) k←1; init(k);
(3)   cât timp(k>0) execută
(4)         cât timp(succesor(k)) execută
(5)               dacă (valid(k)) atunci
(6)                     dacă(soluţie(k)) atunci tipar();
(7)                           altfel k←k+1; init(k); sfârşit;
(8)         sfârşit;
(9)     k←k-1;
(10)  sfârşit;
(11) sfârşit;
```

Fig. 5. Rutina Backtracking

- soluţia problemei se generează sub forma unui vector, iar generarea soluţiilor se realizează într-o stivă în care pe primul nivel se regăseşte elementul $x_1 \in S_1$, pe al doilea nivel elementul $x_2 \in S_2$, pe nivelul k se regăseşte elementul $x_k \in S_k$;
- nivelul $k+1$ şi oricare alt nivel al stivei trebuie iniţializat cu valoarea 0, în acest sens fiind utilizată funcţia *init(k)*, unde k reprezintă nivelul care trebuie iniţializat;
- pentru identificarea următorului element netestat din mulţimea S_k $(k=1,..., n)$, se utilizează funcţia *succesor(k)* prin care acest element (dacă există) este depus în stivă, situaţie în care funcţia întoarce valoarea 1, altfel (nu există niciun element netestat) întoarce valoarea 0;
- din moment ce elementul a fost identificat, trebuie verificat dacă acesta îndeplineşte condiţiile de continuare (se verifică de fapt dacă elementul este valid), verificare care se realizează cu ajutorul funcţiei *valid(k);*
- pentru a verifica dacă s-a ajuns sau nu la o soluţie rezultat, se utilizează funcţia *soluţie(k)*, iar afişarea (tipărirea) soluţiei se realizează prin intermediul funcţiei *tipar()*.

Metoda Backtracking (“căutare cu revenire”)

Probleme rezolvabile prin metoda Backtracking:

- **generarea permutărilor unei mulţimi cu n elemente** ➡ exemplificare metodă la lab. pentru n=3 elem.;
- problema celor n dame (regine);
- produsul cartezian a n mulţimi;
- generarea aranjamentelor, combinărilor primelor n numere naturale luate câte m;
- generarea partiţiilor unei mulţimi;
- problema colorării hărţilor;
- problema comis voiajorului;
- problema discretă a rucsacului, etc.

Metoda Backtracking (“căutare cu revenire”)

- Exemplu: generarea permutărilor unei mulțimi cu *n* elemente *A={1, 2, …, n}*

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
int st[50],n,k;
void init(int k)
{st[k]=0;}
int succesor(int k)
{if(st[k]<n)
   {st[k]=st[k]+1;
    return 1;}
 else return 0;
}
int valid(int k)
{int i;
 for(i=1;i<k;i++)
  {if(st[k]==st[i])  return 0;}
 return 1;
}
int solutie(int k)
{if(k==n) return 1;
return 0;
}
```

```
void tipar()
{for(int i=1;i<=n;i++)
 cout<<st[i]<<" ";
 cout<<endl;}
void Backtracking()
{k=1;
init(k);
while(k>0)
{while(succesor(k))
   {if(valid(k))
    if(solutie(k)) tipar();
    else {k++; init(k);}
    }
  k--;}}
void main()
{clrscr();
cout<<"Numarul de elemente=";
cin>>n;
cout<<"\nPermutarile  sunt:";
cout<<endl;
Backtracking();
getch();
}
```

- Testarea aplicaţiilor prezentate în cadrul cursului;
- Testarea aplicaţiilor 10.2.1, 10.2.2 și 10.2.3, pag. 243-248;
- Aplicaţii propuse: aplicaţiile din cadrul lucrării de laborator nr. 14, pag. 248;
- **Obs: se va utiliza cartea "*Elemente de proiectarea algoritmilor. Ghid teoretic și practic*", Șef. lucr. dr. mat. Cărbureanu Mădălina, Editura Universităţii Petrol-Gaze din Ploieşti, 2021.**

Să vă fie de folos și spor la lucru!