Metoda Greedy

Pentru rezolvarea următoarelor probleme vom utiliza o metodă de programare importantă, denumită Greedy. În general, metoda Greedy se aplică problemelor de optimizare. Specificul acestei metode constă în faptul că se construieşte soluţia optimă pas cu pas, la fiecare pas fiind selectat (sau „înghiţit") în soluţie elementul care pare „cel mai bun" la momentul respectiv, în speranţa că această alegere locală va conduce la optimul global.

Algoritmii Greedy sunt foarte eficienţi, dar nu conduc în mod necesar la o soluţie optimă. Şi nici nu este posibilă formularea unui criteriu general conform căruia să putem stabili exact dacă metoda Greedy rezolvă sau nu o anumită problemă de optimizare. Din acest motiv, orice algoritm Greedy trebuie însoţit de o demonstraţie a corectitudinii sale[1]*. Demonstraţia faptului că o anumită problemă are proprietatea alegerii Greedy se face de obicei prin inducţie matematică.*

Problema spectacolelor

Managerul artistic al unui festival trebuie să selecteze o mulţime cât mai amplă de spectacole ce pot fi jucate în singura sală pe care o are la dispoziţie. Ştiind că i s-au propus $n \leq 100$ spectacole şi pentru fiecare spectacol i i-a fost anunţat intervalul în care se poate desfăşura $[s_i, f_i)$ (s_i reprezintă ora şi minutul de început, iar f_i ora şi minutul de final al spectacolului i) scrieţi un program care să permită spectatorilor vizionarea unui număr cât mai mare de spectacole. De exemplu, dacă vom citi $n=5$ şi următorii timpi:

```
12 30 16 30
15 0 18 0
10 0 18 30
18 0 20 45
12 15 13 0
```

Spectacolele selectate sunt: `5 2 4`.

Soluţie

Ordonăm spectacolele crescător după ora de final. Selectăm iniţial primul spectacol (deci cel care se termină cel mai devreme). La fiecare pas selectăm primul spectacol neselectat, care nu se suprapune cu cele deja selectate (deci care începe după ce se termină ultimul spectacol selectat).

```
#include <iostream.h>
int inceput[100], sfarsit[100], nr[100];
int main()
{int n, i, h, m, schimb, ultim, aux;
 cout << "n= "; cin >> n;                          //citire
 cout<<"Introduceti inceputul si sfarsitul spectacolelor";
 for (i=0; i<n; i++)
```

1. Demonstraţia depăşeşte nivelul de cunoştinţe al unui elev de clasa a IX-a.

```
      {nr[i]=i+1;           //transform timpul in minute
       cin>>h>>m; inceput[i]=h*60+m;
       cin>>h>>m; sfarsit[i]=h*60+m;}
//ordonez spectacolele crescator dupa ora de final
do
 {schimb=0;
  for (i=0; i<n-1; i++)
       if (sfarsit[nr[i]]>sfarsit[nr[i+1]])
          {aux=nr[i];nr[i]=nr[i+1];nr[i+1]=aux; schimb=1;}
 }
while (schimb);
cout << "Spectacolele selectate sunt:\n"<<nr[0]<<' ';
for (ultim=0, i=1; i<n; i++)
     if (inceput[nr[i]]>=sfarsit[nr[ultim]])
         {cout <<nr[i]<<' '; ultim=i;}
cout<<endl;
return 0;}
```

Problema rucsacului

Un hoț neprevăzător are la dispoziție un singur rucsac cu care poate transporta o greutate maximă Gmax. Hoțul are de ales din $n \leq 50$ obiecte şi, evident, intenționează să obțină un câştig maxim în urma singurului transport pe care îl poate face. Cunoscând pentru fiecare obiect i greutatea g_i şi câştigul c_i pe care hoțul l-ar obține transportând obiectul respectiv în întregime, scrieți un program care să determine o încărcare optimală a rucsacului, în ipoteza că hoțul poate încărca în rucsac orice parte dintr-un obiect. De exemplu, pentru n=5, GMax=100 şi câştigurile-greutățile următoare:

(1000 120)(500 20)(400 200)(1000 100)(25 1) se va afişa pe ecran:

```
2              100.00%
4               79.00%
5              100.00%
```

Soluție

Vom reprezenta o soluție a problemei ca un vector x, cu n componente, în care reținem pentru fiecare obiect fracțiunea încărcată în rucsac de hoț. Deci vectorul x trebuie să îndeplinească următoarele condiții:

1. $x_i \in [0, 1], \forall i \in \{1, 2, \ldots, n\}$.
2. $g_1 \cdot x_1 + g_2 \cdot x_2 + \ldots + g_n \cdot x_n \leq Gmax$
3. $f(x) = c_1 \cdot x_1 + c_2 \cdot x_2 + \ldots + c_n \cdot x_n$ este maximă.

Ordonăm obiectele descrescător după câştigul pe unitatea de greutate (valoare care constituie o măsură a eficienței transportării obiectelor). Cât timp este posibil (încap în rucsac), selectăm obiectele în întregime. Completăm rucsacul cu un fragment din următorul obiect ce nu a fost selectat.

```
#include <iostream.h>
int o[50]; //ordinea obiectelor
float c[100], g[100], x[100], Gr, GMax;
```

```
int n;
int main()
{int i, schimb, aux;
 cout << "n= "; cin >> n;                    //citire
 cout<< "GMax= "; cin >> GMax;
 cout<<"Castigul si greutatea pt. fiecare obiect\n";
 for (i=0; i<n; i++)
     {o[i]=i; cin>>c[i]>>g[i];}
 //ordonez obiectele descrescator dupa castigul unitar
 do
  {schimb=0;
   for (i=0; i<n-1; i++)
       if (c[o[i]]/g[o[i]]<c[o[i+1]]/g[o[i+1]])
          {aux=o[i];o[i]=o[i+1]; o[i+1]=aux; schimb=1;}
  } while (schimb);
 for (i=0, Gr=GMax; i<n && Gr>g[o[i]]; i++)
     {x[o[i]]=1; Gr-=g[o[i]];}
 if (i<n) x[o[i]]= Gr/g[o[i]];
 cout << "Obiectele selectate sunt:\n";
 for (i=0; i<n; i++)
     if (x[i]) cout<<i+1<<' '<<x[i] * 100<<'%'<<endl;
 return 0;}
```

Observaţie

Am aplicat tot o strategie de tip *Greedy*. Se poate demonstra corectitudinea acestui algoritm în condiţiile în care putem încărca în rucsac orice parte a unui obiect. Pentru cazul în care hoţul poate încărca un obiect doar în întregime (varianta discretă a problemei rucsacului) algoritmul *Greedy* nu mai funcţionează. De exemplu, pentru `n=3`, `Gmax=10`, `g=(8, 6, 4)`, `c=(8, 6, 4)`, algoritmul *Greedy* va obţine soluţia `x=(1, 0, 0)` pentru care câştigul obţinut în urma transportului este `8`. Soluţia optimă este `x=(0, 1, 1)` pentru care se obţine câştigul `10`.

Reactivi

Într-un laborator de analize chimice se utilizează `N` reactivi. Se ştie că, pentru a evita accidentele sau deprecierea reactivilor, aceştia trebuie să fie stocaţi în condiţii de mediu speciale. Mai exact, pentru fiecare reactiv `x`, se precizează intervalul de temperatură [$\texttt{min}_x$, $\texttt{max}_x$] în care trebuie să se încadreze temperatura de stocare a acestuia. Reactivii vor fi plasaţi în frigidere. Orice frigider are un dispozitiv cu ajutorul căruia putem stabili temperatura (constantă) care va fi in interiorul acelui frigider (exprimată într-un număr întreg de grade Celsius).

Scrieţi un program care să determine numărul minim de frigidere necesare pentru stocarea reactivilor chimici.

Date de intrare

Fişierul de intrare `react.in` conţine:

– pe prima linie numărul natural `N`, care reprezintă numărul de reactivi;
– pe fiecare dintre următoarele `N` linii se află `min max` (două numere întregi separate printr-un spațiu); numerele de pe linia `x+1` reprezintă temperatura minimă, respectiv temperatura maximă de stocare a reactivului `x`.

Date de ieşire

Fişierul de ieşire `react.out` va conține o singură linie pe care este scris numărul minim de frigidere necesar.

Restricții

- $1 \leq N \leq 8000$
- $-100 \leq min_x \leq max_x \leq 100$ (numere întregi, reprezentând grade Celsius), pentru orice x de la 1 la N
- un frigider poate conține un număr nelimitat de reactivi

Exemple

react.in	**react.out**	**react.in**	**react.out**	**react.in**	**react.out**
3 −10 10 −2 5 20 50	2	4 2 5 5 7 10 20 30 40	3	5 −10 10 10 12 −20 10 7 10 7 8	2

(Olimpiada Județeană de Informatică 2004, clasa a IX-a)

Soluție

Intervalele de temperatură pot fi considerate segmente de dreaptă. Problema cere, de fapt, determinarea unui număr minim de puncte astfel încât orice segment să conțină cel puțin unul dintre punctele determinate.

Vom sorta în primul rând intervalele de temperatură crescător după temperatura minimă şi descrescător după temperatura maximă.

Deschidem un frigider şi plasăm primul reactiv în acest frigider. Pentru frigiderul curent reţin temperatura minimă şi temperatura maximă (intervalul de temperatură în care poate fi setat).

Parcurg succesiv reactivii (în ordine) şi pentru fiecare reactiv verific dacă el poate fi plasat în frigiderul curent (pentru aceasta, trebuie ca intersecția dintre intervalul de temperatură al frigiderului şi intervalul de temperatură al reactivului să fie nevidă). Dacă da, plasăm acest reactiv în frigiderul curent (actualizând corespunzător intervalul de temperatură al frigiderului). În caz contrar, deschidem un nou frigider (intervalul de temperatură al acestui frigider va fi intervalul reactivului plasat în el).

```
#include <fstream.h>
ifstream fin("react.in");
ofstream fout("react.out");
int N;
int ti[8000], tf[8000];
/*ti[i]=temperatura minima pentru reactivul i
```

```
  tf[i]=temperatura maxima pentru reactivul i */
int main ()
{int nf, poz, miny, maxx, max, i, j, icx, icy;
 //citire
 fin>>N;
 for (i=0;i<N;i++) fin>>ti[i]>>tf[i];
 fin.close();
 //sortare prin selectia maximului
 for (i=N-1; i>0; i--)
     {for (poz=i, j=0; j<i; j++)
          if (ti[j]>ti[poz] ||
              ti[j]==ti[poz] && tf[j]<tf[poz]) poz=j;
      max=ti[poz];ti[poz]=ti[i]; ti[i]=max;
      max=tf[poz];tf[poz]=tf[i]; tf[i]=max;  }
/* deschid un frigider si plasez primul reactiv
icx=temperatura minima de functionare a frigiderului curent
icy=temperatura maxima de functionare a frigiderului curent
nf=numarul de frigidere deschise */
 nf=1; icx=ti[0];icy=tf[0];
//parcurg ceilalti reactivi in ordine
 for (i=1; i<N; i++)
     {
/* verific daca intervalul curent icx, icy se intersecteaza
cu intervalul de temperatura al reactivului i */
      miny=icy;if (miny>tf[i]) miny=tf[i];
      maxx=icx;if (maxx<ti[i]) maxx=ti[i];
      if (maxx <= miny)
         //actualizez intervalul de temperatura
         {icx=maxx;icy=miny;}
         else //deschid un nou frigider
         {nf++;
          icx=ti[i];icy=tf[i]; }
     }
 fout<<nf<<'\n';
 fout.close(); return 0;}
```

Problema instructorului de schi

Un instructor de schi are la dispoziție `n` perechi de schiuri (`n≤50`) pe care trebuie să le distribuie la `n` elevi începători. Scrieți un program care să distribuie cele `n` perechi de schiuri astfel încât suma diferențelor absolute dintre înălțimea elevului şi lungimea schiurilor atribuite să fie minimă.

Zig-zag

Din fişierul `ZigZag.in` se citesc `N` numere întregi (`N≤1000`). Afişați pe prima linie a fişierului `ZigZag.out` cel mai lung MZigZag care se poate construi

din numerele citite. Numim MZigZag o secvență de numere a_1, a_2, ..., a_m astfel încât $a_1 \le a_2 \ge a_3 \le a_4 \ge a_5 \le ... \ a_{m-1} \ge a_m$. De exemplu:

ZigZag.in	ZigZag.out
7 7 5 0 1 4 9 3	0 9 3 7 1 5 4

Sortare

Fie n ($n \in \mathbf{N}^*$) elemente a_0, a_1, …, a_{n-1} dintr-o mulțime total ordonată. Ordonați crescător elementele a_0, a_1, …, a_{n-1}.

Problema ordonării unor elemente (cunoscută şi sub denumirea de sortare) este frecvent întâlnită în practică şi din acest motiv a fost studiată intens. Ca urmare, au fost elaboraţi numeroşi algoritmi de sortare. Cum de obicei numărul de elemente care trebuie să fie ordonate este mare, s-a studiat şi eficienţa acestor algoritmi, în scopul elaborării unor algoritmi de sortare performanţi. Pentru început, vom studia algoritmi de sortare simpli, nu performanţi, urmând ulterior să învăţăm să evaluăm eficienţa acestor algoritmi şi să elaborăm algoritmi eficienţi.

Sortare prin selecție

Sortarea prin selecție are două variante: sortarea prin selecția elementului maxim şi sortarea prin selecția elementului minim. În ambele variante, ideea de bază este aceeaşi: se selectează cel mai mare element din vector (sau cel mai mic) şi se plasează pe ultima poziție în vector (respectiv, pe prima poziție). Apoi se calculează cel mai mare dintre elementele rămase şi se plasează pe penultima poziție în vector (sau cel mai mic dintre elementele rămase şi se plasează pe a doua poziție), ş.a.m.d. Acest procedeu se repetă de n−1 ori.

```
for (dr=n-1; dr>0; dr--)   //calculez maximul de la 0 la dr
    {for (max=a[0],pozmax=0,i=1; i<=dr; i++)
          if (a[i] > max)  max=a[i], pozmax=i;
     a[pozmax]=a[dr];      //plasez maximul pe pozitia dr
     a[dr] = max; }
```

Sortare prin compararea vecinilor (bubblesort)

O altă metodă de sortare este de a parcurge vectorul, comparând fiecare două elemente vecine şi (dacă este cazul) interschimbându-le. Cum într-o singură trecere nu se poate realiza sortarea vectorului, acest procedeu se repetă până când vectorul devine sortat (la ultima trecere nu am mai efectuat nici o interschimbare).

```
do
  {schimb=0;         //initial nu am facut nici o schimbare
   for (i=0; i<n-1; i++)               //parcurg vectorul
        if (a[i]>a[i+1])     //compar doua elemente vecine
           { //nu sunt in ordine, le interschimb
             aux=a[i]; a[i]=a[i+1]; a[i+1]=aux;
             schimb=1;}      //retin ca am facut schimbari
  }    //procedeul se repeta cat timp se executa schimbari
while (schimb);
```

Sortare prin inserție

Metoda de sortare prin inserție, este de asemenea o metodă simplă, pe care o utilizăm adesea când ordonăm cărțile la jocuri de cărți: de fiecare dată când tragem o carte o plasăm pe poziția sa corectă, astfel încât în mână cărțile să fie ordonate.

Utilizând această idee, sortăm vectorul astfel: parcurgem vectorul, element cu element; la fiecare pas `i`, căutăm poziția corectă a elementului curent `a[i]`, astfel încât secvența `a[0]`, `a[1]`, ..., `a[i]` să fie ordonată:

```
for (i=1; i<n; i++)
    {v=a[i];                  //caut pozitia lui v
     for (poz=i; poz && a[poz-1]>v; poz--)
         a[poz]=a[poz-1];   //mut la dreapta elementele > v
     a[poz] = v;  }     //poz este pozitia corecta pentru v
```

Sortare prin numărarea aparițiilor

În anumite probleme, elementele vectorului au ca valori numere naturale dintr-un interval [`0`, `Max`) de dimensiune redusă (sau au valori care pot fi asociate numerelor naturale dintr-un astfel de interval). Prin dimensiune redusă înțelegem că se poate declara un vector `V` cu `Max` componente întregi.

În acest caz particular cea mai bună metodă de sortare este sortarea prin numărarea aparițiilor: se numără pentru fiecare valoare din intervalul `[0, Max)` de câte ori apare în vector.

```
#include <iostream.h>
#define Max 1000
#define NrMax 10000

int n, V[Max], a[NrMax];
int main()
{int x, i, j, nr;
 cout<<"n="; cin>>n;
 for (i=0; i<n; i++)
     {cin>>x;  //citesc o noua valoare
     V[x]++;} //marim numarul de aparitii ale valorii x
//plasam in ordine elementele in vectorul a
for (nr=i=0; i<Max; i++)
    for (j=0; j<V[i]; j++)
        a[nr++]=i;
for (i=0; i<n; i++) cout<<a[i]<<' ';
cout<<endl; return 0;}
```

Observație

Numărul de operații efectuate de acest algoritm de sortare este de ordinul lui `n+Max`.

Ciurul lui Eratostene

Fie `n` un număr natural (`n≤10000`). Să se genereze toate numerele prime mai mici decât `n`.

Soluție

O primă idee ar fi să parcurgem toate numerele naturale din intervalul [`2`, `n`], pentru fiecare număr să verificăm dacă este prim şi să afişăm numerele prime determinate.

O idee mai eficientă provine in antichitate şi poartă numele ciurul lui Eratostene[2]. Ideea este de a „pune în ciur" toate numerele mai mici decât `n`, apoi de a „cerne" aceste numere până rămân în ciur numai numerele prime. Mai întâi „cernem" (eliminăm din ciur) toți multiplii lui 2, apoi cernem multiplii lui 3, ş.a.m.d.

Vom reprezenta ciurul ca un vector cu `10000` de componente care pot fi `0` sau `1`, cu semnificația `ciur[i]=1` dacă numărul `i` este în ciur şi `0` altfel.

Vom parcurge vectorul ciur de la `2` (cel mai mic număr prim), până la $\sqrt{n}$).

Dacă `ciur[i]` este `1` deducem că `i` este prim, dar toți multiplii lui nu vor fi (deci eliminăm din ciur toți multiplii lui `i`).

În final în vectorul `ciur` vor avea valoarea `1` doar componentele de pe poziții numere prime.

```
#include <iostream.h>
#define NMax 10000
int main()
{int ciur[NMax], n, i, j;
 cout<<"n= "; cin>>n;
 //initial toate numerele sunt in ciur
 for (i=2; i<n; i++) ciur[i]=1;
 for (i=2; i*i<=n; i++)
     if (ciur[i])//i este prim
        //elimin toti multiplii lui i
        for (j=2; j*i<n; j++)
            ciur[i*j]=0;
 for (i=2; i<n; i++)
     if (ciur[i]) cout<<i<<' ';
 return 0; }
```

Subsecvență de sumă maximă

Fie $x=(x_0, x_1, ..., x_{n-1})$ o secvență de numere întregi, dintre care cel puțin un element pozitiv. Să se determine o subsecvență de elemente consecutive x_i, x_{i+1},

2. Eratostene (276–196 î.e.n) a fost un important matematician şi filosof al antichității. Deşi puține dintre lucrările lui s-au păstrat, a rămas celebru prin metoda rapidă de determinare a numerelor prime şi prin faptul că a fost primul care a estimat cu acuratețe diametrul Pământului.

..., x_j, astfel încât suma elementelor din subsecvență să fie maximă. De exemplu, pentru x=(1, 10, 3, −5, 4, 2, −100, 30, 15, 25, −10, 40, 1000) subsecvența de sumă maximă începe la poziția 7, are lungimea 6, iar suma elementelor este 1100.

Soluția 1

Considerăm toate subsecvențele de elemente consecutive, calculăm suma elementelor din subsecvență şi reținem poziția de început şi lungimea subsecvenței de sumă maximă.

```
#include <iostream.h>
int main ()
{int x[50], n, i, st, dr, poz, Lg, Sum, SMax;
 cout <<"n="; cin >> n;
 for (i=0; i<n; i++) {cout <<"x["<<i<<"]="; cin>>x[i];}
 for (SMax=x[0], st=0; st<n; st++)
     for (dr=st; dr<n; dr++)
         {for (Sum=0,i=st; i<=dr; i++) Sum += x[i];
              if (SMax<Sum) SMax=Sum, Lg=dr-st+1, poz=st; }
cout <<"Poz= "<<poz<<" ;Lg= "<<Lg<< " ; Smax= "<<SMax;
return 0;}
```

Datorită celor trei cicluri `for` imbricate, algoritmul realizează un număr de operații comparabil (ca ordin de mărime) cu n^3.

Soluția 2

Pentru a calcula suma elementelor din subsecvența de la `st` la `dr`, vom folosi suma deja calculată a elementelor de la `st` la `dr-1`. Secvența de instrucțiuni care rezolvă cerințele problemei devine:

```
for (SMax=x[0], st=0; st<n; st++)
     for (Sum=0,dr=st; dr<n; dr++)
          {Sum += x[dr];
           if (SMax < Sum) SMax=Sum, poz=st, Lg=dr-st+1;}
```

În acest caz, numărul de operații efectuate de algoritm este comparabil (ca ordin de mărime) cu n^2. Spunem că algoritmul este *pătratic*.

Soluția 3

O dată cu parcurgerea vectorului `x` de la stânga la dreapta, se va memora suma maximă calculată până la momentul curent, în variabila `SMax`. în variabila `Sum` este reținută suma elementelor subsecvenței curente. Elementul `x[i]` va fi înglobat în subsecvența curentă dacă `Sum+x[i]`≥0. Dacă `Sum+x[i]`<0, înglobarea lui `x[i]` în orice subsecvență ar conduce la o subsecvență cu suma elementelor mai mică decât cea maximă până la momentul curent, prin urmare `x[i]` trebuie ignorat, iar noua subsecvență curentă va începe la poziția următoare. Mai exact, la pasul `i`: `Sum`=max{`Sum`, 0}+`x[i]`; `SMax`=max{`SMax`, `Sum`}. Secvența de instrucțiuni care rezolvă cerințele problemei devine:

```
for (SMax=Sum=x[0],st=poz=0, Lg=i=1; i<n; i++)
     if (Sum < 0)    //noua subsecventa incepe pe pozitia i
         Sum = x[i], st=i;
         else        //inglobez x[i] in subsecventa curenta
         {Sum += x[i];
          if (SMax<Sum) //subsecventa curenta este maximala
             SMax=Sum, poz=st, Lg=i-st+1;}
```

În acest caz, numărul de operații efectuate de algoritm este comparabil (ca ordin de mărime) cu n. Spunem că algoritmul a devenit *liniar*.

Depozit

Considerăm un depozit cu n ($n \leq 1000$) camere, care conțin respectiv cantitățile de marfă C_1, C_2, …, C_n, (numere naturale). Scrieți un program care să determine un grup de camere cu proprietatea că suma cantităților de marfă pe care le conțin se poate împărți exact la cele n camioane care o transportă.

(Olimpiada Națională de Informatică, Iaşi, 1993)

Soluție

Problema este particulară: solicită determinarea unei submulțimi a unei mulțimi cu n elemente, divizibilă tot cu n. Vom construi sumele S1, S2, ..., Sn şi resturile pe care acestea le dau prin împărțire la n astfel:

```
S1=C1               ⇒ R1=S1 % n
S2=C1+C2            ⇒ R2=S2 % n
...
Si=C1+C2+...+Ci     ⇒ Ri=Si % n
...
Sn=C1+C2+...+Cn     ⇒ Rn=Sn % n
```

Cazul I: Există un rest Ri=0. În acest caz suma Si este divizibilă cu n, prin urmare camerele solicitate sunt 1, 2, ..., i.

Cazul II: Toate resturile sunt diferite de 0. Prin urmare R1, R2, ..., Rn sunt n resturi care iau valori în mulțimea {1, 2, ..., n−1}. În mod obligatoriu există cel puțin două resturi egale: **Ri=Rj** (i<j), adică Si şi Sj produc același rest la împărțirea cu n ⇒ suma Sj−Si este divizibilă cu n, deci camerele solicitate sunt i+1, i+2, ..., j.

Observații

1. Soluția nu este unică. Procedeul prezentat produce o soluție posibilă.
2. Rezolvarea se bazează pe *Principiul cutiei al lui Dirichlet*[3].

```
#include <iostream.h>
int main ()
{ int SR[100], n, i, j, k, c;
  cout <<"n="; cin >> n;
```

3. Matematicianul Peter Gustav Lejeune Dirichlet a enunțat următorul principiu: „*Se consideră n obiecte care trebuie plasate în p cutii, unde n>p*k, k∈N*. Oricum am plasa obiectele, există o cutie care va conține k+1 obiecte.*”

```
 SR[0]=0;                  //calculez sumele de la citire
 for (i=1; i<=n; i++)
     {cout<<"C"<<i<<"="; cin>>c;
      SR[i]=SR[i-1]+c;}
 for (i=1; i<=n; i++) SR[i]%=n;       //calculez resturile
 cout << "Solutia este " << endl;
 for (i=1; i<=n; i++) //caut un rest = 0
     if (!SR[i])
        {for (k=1;k<=i; k++) cout <<k<<' '; return 0;}
 //cazul II, caut doua resturi egale
 for (i=1; i<n; i++)
     for (j=i+1; j<=n; j++)
          if (SR[i]==SR[j])
             {for (k=i+1;k<=j;k++) cout<<k<<' ';return 0;}
return 0;}
```