Capitolul 5. Metoda Divide et impera

Puşi în faţa unei probleme pentru care trebuie să elaborăm un algoritm, de multe ori nu ştim cum să începem.

Aşa cum în matematică există metode specifice de rezolvare a unor probleme (cum ar fi, de exemplu, rezolvarea sistemelor de ecuaţii prin mai multe metode: metoda eliminării, metoda substituţiei etc.) şi în informatică există pentru foarte multe probleme metode deja standardizate, asemănătoare unor reţete, ce ne ajută să găsim mult mai repede algoritmii de rezolvare a lor dar şi să alegem cea mai adecvată metodă pentru a elabora un algoritm eficient. Prin urmare, ca şi în orice altă activitate, şi în elaborarea algoritmilor există câteva principii (metode, tehnici, proceduri etc.) generale care ne pot ajuta în astfel de situaţii.

Aşa cum vom vedea pe parcursul acestui capitol, sunt probleme care pot fi rezolvate prin mai multe metode, dar nu toate conduc în mod sigur la cea mai bună soluţie sau la un algoritm eficient din punct de vedere al timpului de execuţie sau al dimensiunii memoriei utilizate. Cunoscând însă foarte bine aceste metode de elaborare a algoritmilor şi, având o foarte bună experienţă în utilizarea lor, vom ştii s-o utilizăm pe cea mai bună şi, implicit, vom alege cel mai bun algoritm pentru a ajunge la soluţia problemei pe care o avem de rezolvat.

Există foarte multe metode (tehnici) de elaborare a algoritmilor, dintre care amintim: Greedy, Backtracking, Divide et impera, metoda programării dinamice, Branch and bound, metode euristice, algoritmi probabilistici, algoritmi genetici etc. Dintre acestea vom studia doar câteva dintre cele mai cunoscute.

5. 1. Descrierea tipurilor de probleme la care se aplică metoda Divide et impera

Metoda Divide et impera (***împarte şi stăpâneşte***) este o tehnică specială prin care se pot rezolva anumite probleme bazându-se pe un principiu extrem de simplu: se descompune (divide) problema iniţială în două sau mai multe subprobleme *de acelaşi tip* mult mai uşor de rezolvat (de stăpânit), care se rezolvă, iar soluţia pentru problema iniţială se obţine combinând soluţiile subproblemelor în care a fost descompusă. Bineînţeles că subproblemele în care a fost descompusă problema iniţială pot fi descompuse şi ele, la rândul lor, în alte subprobleme de acelaşi tip la fel cum a fost descompusă problema iniţială. Procedeul de descompunere poate continua până când se ajunge la subprobleme care admit o rezolvare imediată.

Datorită cerinţei ca problema iniţială să se poată descompune în subprobleme *de acelaşi tip* în mod repetat, numărul de probleme cărora li se poate aplica această metodă este relativ mic.

6.2. Descrierea metodei

Pentru a înţelege mult mai bine în ce constă metoda Divide et impera, să luăm în considerare o problemă simplă dar tipică pentru aplicarea acestei metode.

Problema 5.1. ***(Mergesort - sortarea prin interclasare).*** Fie **a** un vector cu **n** componente întregi. Să se sorteze crescător vectorul **a** utilizând sortarea prin interclasare.

Soluţie. Să studiem mai întâi problema interclasării care are enunţul ce urmează.

Se dau doi vectori **a** *şi* **b** *cu* **m** *şi* **n** *componente numere reale, sortaţi crescător (descrescător). Să se formeze din aceştia un al treilea vector* **c**, *cu* **m+n** *componente, sortate crescător (descrescător).*

Pentru rezolvare, simbolizăm cu **i** indicele elementului la care s-a ajuns în primul vector, cu **j** indicele la care s-a ajuns în al doilea vector şi cu **k** indicele elementului care urmează a fi scris în cel de-al treilea vector.

Atâta timp cât **i** este mai mic sau egal cu **m** şi **j** este mai mic sau egal cu **n**, comparăm **a[i]** cu **b[j]** şi în funcţie de rezultat procedăm astfel:

- dacă **a[i]>b[j]** atunci **c[k]** va lua valoarea lui **a[i]**, iar valorile variabilelor **i** şi **k** vor creşte cu **1**;
- altfel, **c[k]** va lua valoarea lui **b[j]**, în timp ce valorile variabilelor **j** şi **k** vor creşte cu **1**.

După ce unul din cei doi vectori a fost parcurs în totalitate şi scris în **c**, vectorul rămas se va copia în **c.**

Exemplu. Fie **a={1,3,5,9}** şi **b={2,4}**

- i=1, **j=1**, **k=1**;

- a[1]<b[1], deci **c[1]=1**, **i=2**, **k=2**;

- a[2]>b[1], deci **c[2]=2**, **j=2**, **k=3**;

- a[2]<b[2], deci **c[3]=3**, **i=3**, **k=4**;

- a[3]>b[2], deci **c[4]=4**, **j=3**, **k=5**;

- vectorul **b** a fost trecut în **c** în întregime, în continuare urmând a se copia ceea ce a rămas neparcurs din vectorul **a** în vectorul ***c*** (**c[5]=5**).

Vom utiliza acest algoritm de interclasare în vederea sortării unui vector.

Observaţii. 1) Un vector cu o singură componentă este un vector sortat.

2) Pentru a sorta (crescător) un vector cu două componente, acestea se compară între ele şi, dacă prima este mai mare decât cea de-a doua, locurile lor se inversează.

Algoritmul de sortare prin interclasare se bazează pe următoarea ideie: pentru a sorta un vector cu **n** componente îl împărţim în doi vectori care, odată sortaţi, se interclasează. Conform strategiei generale Divide et impera, problema este descompusă în alte două subprobleme de acelaşi tip şi, după rezolvarea lor, rezultatele se combină (în particular se interclasează). Descompunerea unui vector în alţi doi vectori care urmează a fi sortaţi are loc până când avem de sortat vectori de una sau două componente.

În programul ce urmează, procedura *prel* sortează un vector de maxim două componente; *comb* interclasează rezultatele; *divimp* implementează strategia generală a metodei studiate.

Varianta Pascal

```
program Sortint;
type     vector=array[1..10]     of
integer;
var  a:vector;
```

Varianta C/C++

```
#include <iostream.h>;
int a[10],n,i;
int prel(int p,int q,int a[10])
/*Sorteaza vectorul a[p],...,a[q],
```

```
    n,i:integer;
procedure  prel(p,q:integer;  var
a:vector);
{Sortează vectorul a[p],...,a[q],
furnizând rezultatul în a}
var t:integer;
begin
 if a[p]>a[q] then
 begin
  t:=a[p];
  a[p]:=a[q];
  a[q]:=t
 end
end;
procedure   combina(p,q,m:integer;
var a:vector);
{Interclasează subşirurile deja
sortate     a[p],...,a[m]     şi
a[m+1],...,a[q],         furnizând
rezultatul sortării în subşirul
a[p],...,a[q]}
var  b:vector;
    i,j,k:integer;
begin
 i:=p;
 j:=m+1;
 k:=1;
 while (i<=m) and (j<=q) do
  if a[i]<=a[j] then
  begin
   b[k] := a[i];
   i := i+1;
   k := k+1
  end
  else
  begin
   b[k]:=a[j];
   j:=j+1;
   k:=k+1
  end;
 if i<=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;
 k:=1;
 for i:=p to q do
 begin
```

```
furnizand rezultatul in a*/
{
 int t;
 if (a[p]>a[q])
 {
  t=a[p];
  a[p]=a[q];
  a[q]=t;
 }
};
int combina(int p,int q,int m,int
a[10])
/*Interclaseaza subsirurile deja
sortate a[p],...,a[m] si
a[m+1],...,a[q],
furnizand rezultatul sortarii in
subsirul a[p],...,a[q]*/
{
 int b[10],i,j,k;
 i=p;
 j=m+1;
 k=1;
 while ((i<=m)&&(j<=q))
  if (a[i]<=a[j])
  {
   b[k]=a[i];
   i=i+1;
   k=k+1;
  }
  else
  {
   b[k]=a[j];
   j=j+1;
   k=k+1;
  };
 if (i<=m)
  for (j=i;j<=m;i++)
  {
   b[k]=a[j];
   k=k+1;
  }
 else
  for (i=j;i<=q;i++)
  {
   b[k]=a[i];
   k=k+1;
  };
 k=1;
 for (i=p;i<=q;i++)
 {
  a[i]=b[k];
  k=k+1;
 }
};
int diviz(int p,int q)
```

```
  a[i]:=b[k];
  k:=k+1
 end
end;
procedure                    diviz(var
p,q,m:integer);
{Furnizează valoarea "m" ca partea
întreagă inferioară a lui (p+q)/2}
begin
 m:=(p+q) div 2
end;
procedure  divimp(p,q:integer;  var
a:vector);
{Procedura divide et impera}
var m:integer;
begin
 if (q-p)<=1 then prel(p,q,a)
 else
 begin
  diviz(p,q,m);
  divimp(p,m,a);
  divimp(m+1,q,a);
  combina(p,q,m,a)
 end
end;
Begin
 write('Dati lungimea vectorului
        n= '); readln(n);
 {Citirea vectorului}
 for i:=1 to n do
 begin
  write('a[',i,']= ');
  readln(a[i])
 end;
 divimp(1,n,a);
 {Afişarea vectorului sortat}
 for i:=1 to n do writeln(a[i])
End.
```

```
/* Furnizeaza valoarea "m" ca
partea intreaga inferioara a lui
(p+q)/2 */
{
 return (p+q)/2;
};
int divimp(int p,int q,int a[10])
/*Rutina divide et impera*/
{
 int m;
 if ((q-p)<=1) prel(p,q,a);
 else
 {
  m=diviz(p,q);
cout<<"m="<<m<<endl;
  divimp(p,m,a);
  divimp(m+1,q,a);
  combina(p,q,m,a);
 }
};
int main()
{
 cout<<"Dati lungimea vectorului
n= "; cin>>n;
 /*Citirea vectorului*/
 for (i=1;i<=n;i++)
 {
  cout<<"a["<<i<<"]= ";
  cin>>a[i];
 };
 divimp(1,n,a);
 /*Afisarea vectorului sortat*/
 for (i=1;i<=n;i++)
cout<<a[i]<<endl;
}
```

Prin urmare, algoritmul general pentru metoda Divide et impera poate fi sintetizat astfel:

divimp(p, q, a)
dacă q-p<=eps **atunci**
prel(p,q,a)
altfel
diviz(p,q,m)
divimp(p,m,b)
d*ivimp*(m+1,q,c)
comb(b,c,a)
■

Rutina *divimp* trebuie apelată prin:

divimp(1,n,a)

în **a** obţinându-se rezultatul final.

În acest algoritm, **eps** este lungimea maximă a unei secvenţe **{a_p, ..., a_q}** notată prescurtat **(p,q)**, pentru care prelucrarea se poate face direct, fără a mai fi necesară împărţirea în subprobleme.

Rutina *prel* realizează prelucrarea secvenţelor de acest tip, furnizând rezultatul în **a**.

Rutina *comb* realizează combinarea rezultatelor **b** şi **c** ale prelucrării a două secvenţe vecine **(p,m)** şi **(m+1,q)**, obţinându-se rezultatul **a** al prelucrării secvenţei **(p,q)**.

Valoarea **m** este obţinută apelând procedura *diviz*.

6.3. Probleme rezolvate

Problema 5.2. ***(Căutarea binară[1]) Fie*** **v** un vector cu **n** componente numere întregi ordonat crescător şi ***x*** un număr întreg oarecare. Problema constă în a căuta în vectorul **v** dacă **x** este printre componentele sale şi în caz afirmativ să se tipărească indicele componentei care conţine acea valoare.

Soluţie. O rezolvare în care **x** se compară pe rând cu cele **n** valori, este lipsită de valoare (nu exploatează faptul că cele **n** valori sunt în secvenţă crescătoare). Algoritmul propus (Divide et impera) este mult mai performant.

Problema este de a decide dacă valoarea căutată se găseşte printre numerele de indice cuprins între **i** şi **j** (iniţial **i=1** şi **j=n**). Pentru aceasta vom proceda astfel:

- dacă **x** coincide cu valoarea de indice **(i+j) div 2** (valoarea de la mijloc), se tipăreşte indicele şi se revine din apel (problema a fost rezolvată);
- în caz contrar, dacă **i<j** (nu s-a căutat peste tot) problema se descompune astfel:
- dacă numărul **x** este mai mic decât valoarea testată (din mijloc), înseamnă că avem şanse să-l găsim între componentele cu indicele între **i** şi **(i+j) div 2 - 1**, caz în care reapelăm procedura cu aceşti parametri;
- dacă numărul **x** este mai mare decât valoarea testată (din mijloc), înseamnă că avem şanse să-l găsim între componentele cu indicele între **(i+j) div 2 + 1** şi **j**, caz în care reapelăm procedura cu aceşti parametri.

Observaţie. De fapt problema nu se descompune în alte subprobleme care se rezolvă după care se combină soluţia, ci se reduce la o subproblemă sau la cealaltă (în funcţie de rezultatul testului **x<v[(i+j) div 2]**). Ne putem pune problema dacă un astfel de raţionament este specific tehnicii Divide et impera. Răspunsul este afirmativ şi putem spune că acest raţionament este de tip *Divide et impera degenerativ*. De altfel, această degenerare este benefică din punct de vedere al vitezei de execuţie (care este şi scopul nostru). Dar iată programul:

Varianta Pascal

```
program Cautbin;
var  v:array[1..100] of integer;
     n,i,x:integer;
procedure cautb(i,j:integer);
```

Varianta C/C++

```
#include <iostream.h>
int v[100],n,i,x;
int cautb(int i,int j)
{
```

[1] Căutarea binară este cea mai simplă aplicaţie a metodei Divide et impera, fiind cunoscută încă înainte de apariţia calculatoarelor. În esenţă, este algoritmul după care se caută un cuvânt într-un dicţionar, sau un nume în cartea de telefon.

```
begin
 if x=v[(i+j) div 2] then
  writeln('Gasit, indice ',
           (i+j) div 2)
 else
  if i<j then
   if x<v[(i+j) div 2] then
    cautb(i,(i+j) div 2-1)
   else
    cautb((i+j) div 2+1,j)
end;
Begin
 write('n= '); readln(n);
 for i:=1 to n do
 begin
  write('v[',i,']= ');
  readln(v[i])
 end;
 write('x= '); readln(x);
 cautb(1,n)
End.
```

```
  if (x==v[(i+j)/2])
   cout<<"Gasit, indice
"<<(i+j)/2;
  else
   if (i<j)
    if (x<v[(i+j)/2])
     cautb(i,(i+j)/2-1);
    else
     cautb((i+j)/2+1,j);
 };
 int main()
 {
  cout<<"n= "; cin>>n;
  for (i=1;i<=n;i++)
  {
   cout<<"v["<<i<<"]= ";
   cin>>(v[i]);
  };
  cout<<"x= "; cin>>x;
  cautb(1,n);
 }
```

Observaţie. La aplicarea metodei Divide et impera de obicei se apelează la recursivitate deoarece, în general, soluţiile recursive sunt mult mai clare decât cele nerecursive şi reduc lungimea textului sursă al unui program însă în majoritatea situaţiilor pot fi găsiţi şi algoritmi iterativi. Programul prezentat în continuare implementează un algoritm iterativ pentru problema căutării binare.

Varianta Pascal

```
Program Cautbin_iter;
const n = 5;
type tablou=array[1..n] of real;
var v:tablou;
    i,ind:integer;
    nr:real;
procedure cautb(var a:tablou;
                var x:real);
var i,j,k:integer;
begin
 i:=1;
 j:=n;
 while i<j do {a[i]<=x<a[j+1]}
 begin
  k:=(i+j) div 2;
  if x<a[k] then j:=k-1;
   if x>=a[k+1] then i:=k+1;
    if (x>=a[k]) and
       (x<a[k+1]) then
    begin i:=k; j:=k end
 end;
 ind:=i
end;
Begin
 writeln('Elementele tabloului:
');
```

Varianta C/C++

```
 #include <iostream.h>
 #define N 5
 float v[N], nr;
 int i,ind;
 int cautb(float a[N], float x)
 {
  int i,j,k;
  i=0;
  j=N-1
  ;
  while (i<j) //* a[i]<=x<a[j+1]
*//
  {
   k=(i+j)/2;
   if (x<a[k]) j=k-1;
   if (x>=a[k+1]) i=k+1;
   if ((x>=a[k])&&(x<a[k+1]))
    {i=k; j=k;}
  };
  return i;
 };
 int main()
 {
  cout<<"Elementele tabloului: ";
  for (i=0;i<=N-1;i++)
  {
```

```
 for i := 1 to n do
 begin
  write('v[',i,']= ');
  readln(v[i])
 end;
 write('Dati numarul ce urmeaza
        a fi cautat: ');
 readln(nr);
 cautb(v, nr);
 if nr = v[ind] then
 writ]eln('Gasit, indice ', ind);
End.
```

```
    cout<<"v["<<i<<"]= ";
    cin>>v[i];
   };
   cout<<"Dati numarul ce urmeaza
a fi cautat: ";
   cin>>nr;
   ind=cautb(v,nr);
   if (nr==v[ind])
    cout<<"Gasit, indice "<<ind;
  }
```

Problema 5.3. (***Turnurile din Hanoi***) Se dau trei tije simbolizate prin **a**, **b**, **c**. Pe tija **a** se găsesc discuri de diametre diferite, aşezate în ordine descrescătoare a diametrelor privite de jos în sus. Se cere să se mute discurile pe o altă tijă respectând următoarele reguli:

- la fiecare pas se mută un singur disc;
- nu este permis să se aşeze un disc cu diametrul mai mare peste un disc cu diametrul mai mic.

Soluţie

Dacă **n=1** se face mutarea ***ab***, adică se mută discul de pe tija *a* pe tija *b*.

Dacă **n=2** se fac mutările ***ac***, ***ab***, ***cb***.

În cazul în care **n>2** problema se complică. Notăm cu **H(n,a,b,c)** şirul mutărilor celor **n** discuri de pe tija **a** pe tija **b**, utilizând ca tijă intermediară tija **c**.

Conform strategiei Divide et impera încercăm să descompunem problema în alte două subprobleme de acelaşi tip, urmând apoi combinarea soluţiilor. În acest sens, observăm că mutarea celor **n** discuri de pe tija **a** pe tija **b**, utilizând ca tijă intermediară tija **c**, este echivalentă cu:

- mutarea a **n-1** discuri de pe tija **a** pe tija **c**, utilizând ca tijă intermediară tija **b**;
- mutarea discului rămas pe tija **b**;
- mutarea a **n-1** discuri de pe tija **c** pe tija **b**, utilizând ca tijă intermediară tija **a**.

Parcurgerea celor trei etape permite definirea recursivă a şirului **H(n,a,b,c)** astfel:

$$H(n,a,b,c)=\begin{cases} ab, & dacă\ n=1 \\ H(n-1,a,c,b),ab,H(n-1,c,b,a), & dacă\ n>1 \end{cases}$$

Exemple.

Pentru **n=2** avem:

H(2,a,b,c)=H(1,a,c,b),ab,H(1,c,b,a)=ac,ab,cb.

Pentru **n=3** avem:

H(3,a,b,c)=H(2,a,c,b),ab,H(2,c,b,a)=
=H(1,a,b,c),ac,H(1,b,c,a),ab,H(1,c,a,b),cb,H(1,a,b,c)=
=ab,ac,bc,ab,ca,cb,ab.

Iată programul:

Varianta Pascal

```
program Hanoi;
var a,b,c:char;
    n:integer;
```

Varianta C/C++

```
#include <iostream.h>
char a,b,c;
int n;
int H(int n, char a,char b, char
```

```
procedure H(n:integer; a,b,c:char);
begin
 if n=1 then writeln(a,b)
 else
 begin
  H(n-1,a,c,b);
  writeln(a,b);
  H(n-1,c,b,a)
 End
end;
Begin
 write('n= '); readln(n);
 a:='a'; b:='b'; c:='c';
 H(n,a,b,c)
End.
```

```
c)
{
 if (n==1) cout<<a<<b<<endl;
 else
 {
  H(n-1,a,c,b);
  cout<<a<<b<<endl;
  H(n-1,c,b,a);
 }
};
int main()
{
 cout<<"n= "; cin>>n;
 a='a'; b='b'; c='c';
 H(n,a,b,c);
}
```

5.4. Complexitatea algoritmilor Divide et impera

Să presupunem că avem un algoritm **A** cu timp pătratic ce permite rezolvarea unei subprobleme a problemei iniţiale. Fie **c** o constantă, astfel încât timpul pentru a rezolva o problemă de mărime **n** cu ajutorul algoritmului **A** este $\mathbf{t_A(n)\le cn^2}$. Să presupunem că este posibil să rezolvăm problema iniţială prin descompunerea sa în trei subprobleme, fiecare de mărime **[n/2]**. Fie **k** o constantă, astfel încât timpul necesar pentru descompunere şi recompunere este **t(n)≤kn**. Folosind vechiul algoritm **A** şi ideea de descompunere-recompunere a subproblemelor, obţinem un nou algoritm **B**, pentru care:

$$\mathbf{t_B(n)\le 3t_A([n/2])+t(n)\le 3c((n+1)/2)^2+kn=3/4cn^2+(3/2+d)n+3/4c}$$

Când **n** este suficient de mare, termenul domină pe ceilalţi, ceea ce înseamnă că algoritmul **B** este în esenţă cu **25%** mai rapid decât algoritmul **A**. Cu toate acestea, prin algoritmul **B** nu am reuşit să schimbăm ordinul de complexitate al algoritmului **A**, algoritmul **B** fiind tot pătratic.

Dar procedeul de descompunere-recompunere continuă în mod recursiv aşa că se obţine un nou algoritm recursiv **C** cu ajutorul căruia se ajunge la subprobleme care nu sunt mai mari decât un prag $\mathbf{n_0}$ pentru care bineînţeles că folosim spre rezolvare algoritmul **A**. Algoritmul **C** va avea timpul:

$$\mathbf{t_C(n)}=\begin{cases}\mathbf{t_A(n)} & \textbf{pentru } \mathbf{n\le n_0}\\ \mathbf{3t_A(\lceil n/2\rceil)+t(n)} & \textbf{pentru } \mathbf{n>n_0}\end{cases}$$

Se poate demonstra că $\mathbf{t_C(n)}$ este în ordinul lui $\mathbf{O(n^{lg3})}$. Cum $\mathbf{lg3\approx 1,59}$ înseamnă că de această dată am reuşit să îmbunătăţim ordinul timpului.

În concluzie, prin tehnica Divide et impera se reuşeşte o reducere importantă a timpului de execuţie.

5.5. Probleme propuse

Problema 5.4. ***(Maxim).*** Se citeşte un vector cu **n** componente, numere naturale. Se cere să se tipărească valoarea maximă utilizând strategia Divide et impera.

Problema 5.5. ***(Cel mai mare divizior comun).*** Fie **n** valori naturale $\mathbf{a_1,a_2,\ldots,a_n}$. Să se determine cel mai mare divizor comun (**cmmdc**) al lor utilizând metoda Divide et impera.

Indicaţie. Pentru aceasta, vom împărţi şirul iniţial de numere în două, adică $\mathbf{a_1,a_2,\ldots,a_m}$ şi $\mathbf{a_{m+1},a_{m+2},\ldots,a_n}$, unde **m=(1+n)/2** (împărţind în acest fel problema în două subprobleme) ş.a.m.d. pânăcând se obţin subşiruri $\mathbf{a_p..a_q}$ de cel mult două elemente (**p-q≤1**) pentru care apelăm o subrutină ce utilizează algoritmul lui Euclid pentru calculul celui mai mare divizor comun, *Euclid(a[p],a[q])*.

Problema 5.5. ***(Quicksort[2]).*** Fie vectorul **v** cu **n** componente numere întregi reale. Se cere ca vectorul să fie sortat crescător utilizând metoda sortării rapide.

Indicaţie. Spre deosebire de Mergesort, partea nerecursivă a algoritmului este dedicată construirii subcazurilor şi nu combinării soluţiilor lor. Se alege din vector un element *pivot* după care vectorul este partiţionat în două subtablouri, alcătuit de-o parte şi de alta a acestui pivot astfel: elementele mai mari decât pivotul sunt mutate în dreapta pivotului, iar celelalte elemente sunt mutate în stânga pivotului. Acest mod de partiţionare se numeşte *pivotare*. În continuare, cele două subtablouri sunt sortate în mod independent prin apeluri recursive ale algoritmului. Rezultatul este tabloul complet sortat; nu mai este necesară nici o interclasare.

Problema 5.6. ***(Problema tăieturilor).*** Se dă o bucată de tablă de formă dreptunghiulară cu lungimea **l** şi înălţimea **h**, având pe suprafaţa ei **n** găuri de coordonate numere întregi. Se cere să se decupeze din ea o bucată de arie maximă care nu prezintă găuri. Sunt permise numai tăieturi orizontale şi verticale.

Indicaţie. Pentru a identifica o zonă dreptunghiulară cu laturile paralele cu laturile bucăţii de tablă, este suficient să reţinem coordonatele colţului din stânga-jos **(xs,ys)** şi cele două dimensiuni **l** şi **h**. Subrutina *taie(xs,ys,l,h)* va determina zonele dreptunghiulare fără găuri ce se pot obţine prin tăieturi realizate prin găurile bucăţii de tablă cu colţul stânga-jos (**xs,ys)** şi dimensiunile **l,h** pe direcţii paralele cu laturile bucăţii de tablă în nişte variabile globale aria maximă (s-o notăm cu **amax**), respectiv coordonatele bucăţii de tablă dreptunghiulară fără găuri de arie maximă. Pentru aceasta, procedura va căuta în interiorul bucăţii de tablă dacă există vreo gaură, iar dacă nu există, atunci o compară cu aria reţinută în variabila **amax** şi dacă e mai mare o reţine. Dacă a fost găsită o gaură de coordonate (xi,yi) în interiorul plăcii (xs,ys,l,h), atunci împărţim problema în 4 subprobleme apelând subrutina taie pentru cele 4 dreptunghiuri formate tăind bucata de tablă prin două tăieturi paralele cu laturile şi care trec prin gaură.

Problema 5.7. Scrieţi un program în care calculatorul să ghicească un număr natural ales de dumneavoastră (numărul este cuprins între **1** şi **30.000**). Atunci când calculatorul vă propune un număr îi veţi răspunde prin:

2, dacă numărul este prea mare;

1, dacă numărul este prea mic;

0, dacă numărul a fost ghicit.

Problema 5.8. ***(Plieri[3]).*** Se consideră un vector cu **n** componente, numere naturale. Definim *plierea vectorului* ca fiind suprafaţa unei jumătăţi, numită *donatoare*, peste o

---

[2]Algoritmul **Quicksort** a fost inventat de **C.A.R. Hoare** în 1962 şi este unul dintre cei mai utilizaţi algoritmi de sortare, fiind implementat în bibliotecile multor limbaje de programare.

[3]Acesta este un enunţ modificat al unei probleme date la faza finală a Olimpiadei Naţionale de Informatică, clasa a X-a, Arad 1992.

alta, numită *receptoare*. În cazul în care vectorul are un număr impar de componente, cea din mijloc este eliminată. În acest fel se ajunge la un vector ale cărui elemente au numerotarea jumătăţii receptoare.

Exemplu. Vectorul **1,2,3,4,5** se poate plia în două moduri: **1,2** şi **4,5.**

Plierea se aplică în mod repetat până se ajunge la un vector cu o singură componentă, iar conţinutul ei se numeşte *element final*. Să se precizeze care sunt elementele finale şi care este şirul de plieri prin care se poate ajunge la ele.

Indicaţie. Pentru a determina toate elementele finale şi succesiunile de plieri corespunzătoare pentru un vector cu numerotarea **p..q**, vom utiliza o procedură *pliaza(p,q)*. Dacă **p=q**, atunci vectorul este format dintr-un sigur element, deci final, iar dacă **p<q**, atunci, în cazul în care numărul de elemente din vector **(p-q+1)** este impar, elementul din mijloc **(p+q)/2** este eliminat, plierea din stânga se face la poziţia **(p+q)+2-1**, iar plierea din dreapta la poziţia **(p+q)+2+1**.Vom codifica plierea din stânga reţinând în şirul mutărilor simbolul ‚**S**' urmat de poziţia **Ls**, de la care se face plierea spre stânga, iar o pliere spre dreapta prin simbolul ‚**D**' urmat de poziţia **Ld**, de la care se face plierea spre dreapta. Pentru a determina toate elementele finale din intervalul **p..q**, apelăm recursiv procedura *pliaza* pentru prima jumătate a intervalului, precedând toate succesiunile de plieri cu o pliere spre stânga, apoi apelăm recursiv procedura *pliaza* pentru cea de-a doua jumătate a intervalului, precedând toate succesiunile de plieri corespunzătoare cu o pliere spre dreapta

Problema 5.9. Se dă un vector cu **n** componente la început nule. O secţiune pe poziţia **k** va incrementa toate elementele aflate în zona de secţionare anterioară situate între poziţia **1** şi **k**.

Exemplu.

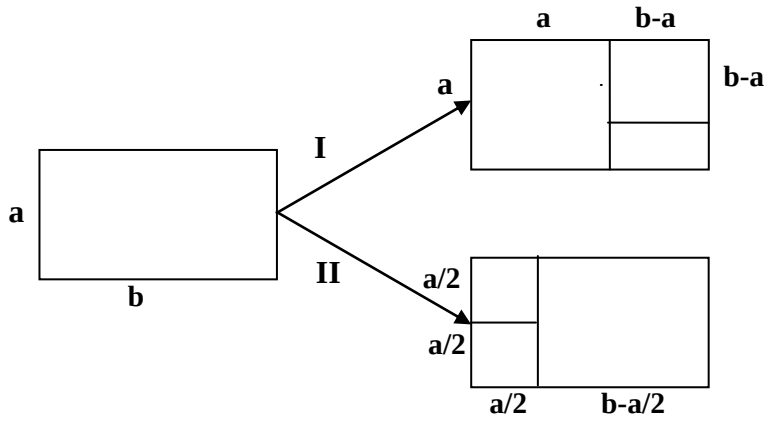
0 0 0 0 0 0 0 0 se secţionează pe poziţia 4;
1 1 1 1 0 0 0 0 se secţionează pe poziţia 1;
2 1 1 1 0 0 0 0 se secţionează pe poziţia 3;
3 2 2 1 0 0 0 0 etc.

Să se determine o ordine de secţionare a unui vector cu **n** elemente astfel încât suma elementelor sale să fie **s**. Valorile **n** şi **s** se citesc.

Problema 5.10. (***Descompunere***[4]) Se consideră un dreptunghi de dimensiuni **a**x**b**, cu **a, b** numere naturale ce satisfac relaţia **b-a<a<b**.

Există două moduri de a *descompune* un dreptunghi în două pătrate şi un dreptunghi, aşa cum sunt prezentate în figura următoare.

[4] Problemă dată la Olimpiada judeţeană de informatică, Iaşi, 1996.



Procesul de descompunere se aplică apoi dreptunghiului rezultat în urma descompunerii şi continuă în acelaşi mod până se obţine un dreptunghi care nu mai satisface relaţia de mai sus.(numit *dreptunghi nedecompozabil*).

Problema constă în a determina un şir de descompuneri în urma căruia să rezulte un număr minim de figuri nedecompozabile.

Să se scrie un program care citeşte de la tastatură dimensiunile **a** şi **b** ale dreptunghiului iniţial şi scrie în fişierul desc.out descompunerea cu număr minim de figuri componente. O descompunere este scrisă în fişierul de ieşire ca o secvenţă de numere întregi $\mathbf{p_1,p_2,...,p_k,d_1,d_2}$ (unde $\mathbf{d_1}$ şi $\mathbf{d_2}$ sunt dimensiunile ultimului dreptunghi).

De exemplu, pentru **a=21**, **b=34** obţinem minim **7** figuri, dimensiunile acestora fiind **21,13,4,8,1,1,7**.

Indicaţii. Pentru a determina o descompunere **d** a unui dreptunghi având laturile **x,y** cu un număr minim **n** de figuri componente, vom utiliza o subrutină *descomp(x,y,d,n)*. Dacă dreptunghiul este decompozabil, descompunem dreptunghiul în cele două variante posibile, pentru dreptunghiurile obţinute determinăm o descompunere cu număr minim de figuri componente, reţinând pentru fiecare variantă de descompunere numărul minim de figuri obţinute şi dimensiunile acestora. Se alege varianta de descompunere cu număr minim de figuri şi se adaugă dimensiunile celor două pătrate corespunzătoare variantei alese.