Tabăra de pregătire a lotului de juniori Vaslui

2014

Suport de curs

“Operaţii pe biţi”

Lot naţional Juniori

Prof. Dana Lica

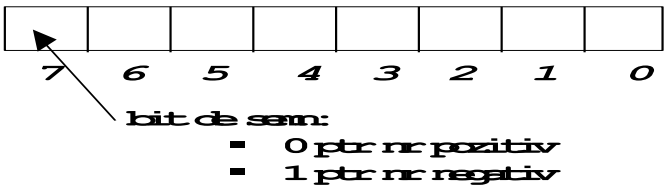
Operaţii pe biţi

1. Reprezentarea internă a numerelor întregi

Reprezentarea în memorie a numerelor întregi se face printr-o secvenţă de cifre de **0** şi **1**. Această secvenţă poate avea o lungime de **8**, **16** sau **32** de biţi.

Forma de memorare a întregilor se numeşte *cod complementar*. În funcţie de lungimea reprezentării se stabileşte domeniu valorilor care pot fi stocate.

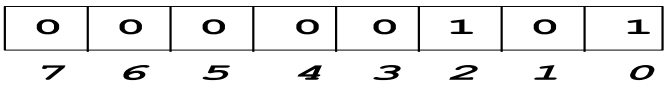
Modului de reprezentare (în cod complementar) va fi prezentat în cele ce urmează, folosind reprezentarea pe o lungime de **8** biţi, valabilă pentru tipul shortint în Pascal, respectiv char în C/C++.



Se observă că numerotarea poziţiilor se face de la dreapta la stânga (de la **0** la **7**), poziţia **7** fiind bit de semn (**0** pentru numerele pozitive şi **1** pentru numerele negative).
Rezultă că doar **7** biţi (poziţiile **0-6**) se folosesc pentru reprezentarea valorii absolute a numărului.

Numerele întregi pozitive se convertesc în baza **2**, şi se face completarea cu cifre **0** nesemnificative, până la completarea celor **7** biţi.

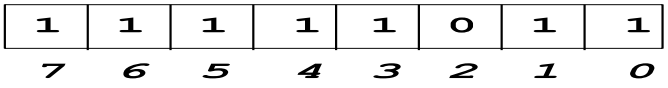
Exemplu:
Să determinăm forma de reprezentare a numărului întreg **5**.
$5_{10} = 101_2$
Vor fi necesare **4** cifre de **0** nesemnificative, pentru completarea primelor **7** biţii, iar poziţia **7** (bitul **8**) va fi **0** deoarece numărul este pozitiv.
Deci reprezentarea internă este următoarea:



Nu în acelaşi mod se face reprezentarea numerelor întregi negative. Pentru aceasta este necesară efectuarea următorilor paşi:

1) determinarea reprezentării interne a numărului ce repezintă valoarea absolută a numărului iniţial. Acesta are bitul de semn egal cu **0**.
2) se calculează complementul faţă de **1** a reprezentării obţinute la pasul anterior (bitul **1** devine **0**, iar bitul **0** devine **1**)
3) se adună **1** (adunarea se face în baza **2**) la valoarea obţinută.

Exemplu:
Pentru determinarea reprezentării numărului **-5** se procedează în felul următor:
Reprezentarea valorii absolute a numărului **-5** este **00000101.** Complementul faţă de **1** este:
11111010
Numărul obţinut după adunarea cu **1** este :
11111011
Deci reprezentarea valorii întregi **-5** pe **8** biţi este:



După cum observăm bitul de semn este **1** ceea ce ne indică faptul că avem de a face cu un număr negativ.

Putem trage concluzia că numerele întregi care se poate reprezenta pe **8** biţi sunt cuprinse între 10000000_2 şi 01111111_2, adică -128_{10}, 127_{10}.

Aşa cum am spus şi la începutul cursului, numerele întregi se pot reprezenta în cod complementar, având la dispoziţie **16, 32** sau chiar şi **64** de biţi. Mecanismul este acelaşi, însă valorile numerelor cresc.

2. Operatori la nivel de bit

Operatorii pe biţi se pot aplica datelor ce fac parte din tipurile întregi. Operaţiile se efectuează asupra biţiilor din reprezentarea internă a numerelor.

Operatorul de negaţie
not (Pascal) respectiv **~** (C/C++)

Este un operator unar care întoarce numărul întreg a cărui reprezentare internă se obţine din reprezentarea internă a numărului iniţial, prin complementarea faţă de **1** a fiecărui bit(**1→0** şi **0→1**).

Exemplu:

not 5=-6	**~ 5 == -6**

Reprezentarea lui **5** este **00000101**. Complementând acest număr obţin reprezentarea **11111010** care corespunde numărului întreg **-6**. Să verificăm:
Numărul **6** se reprezintă intern astfel:
6_2 **= 00000110**, complementând obţinem **11111001** şi adunând **1** (în baza **2**) se obţine în final **11111010** adică exact reprezentarea internă returnată a numărului returnat de operaţia **not 5** respectiv **~5**.

Operatorul de conjuncţie
and respectiv **&**

Este un operator binar care returnează numărul întreg a cărui reprezentare internă se obţine prin conjuncţia biţilor care apar în reprezentarea internă a operanzilor. Conjuncţia se face cu toate perechile de biţi situaţi pe aceeaşi poziţie.

Exemplu:

5 and 3 = 1	**5 & 3 == 1**

Să verificăm:
Reprezentarea internă a lui **5** este **00000101**, iar a lui **3** este **00000011**.

00000101 and	**00000101 &**
00000011	**00000011**
00000001	**00000001**

Această reprezentare este dată de numărul întreg **1**.

Operatorul de disjuncţie
or respectiv **|**

Este un operator binar care returnează numărul întreg a cărui reprezentare internă se obţine prin disjuncţia biţilor care apar în reprezentarea internă a operanzilor. Disjuncţia se face între biţii situaţi pe aceeaşi poziţie.

Exemplu:

15 or 3=15	**15 | 3==15**

Să verificăm:
Reprezentarea internă a lui **15** este **00001111** iar a lui **3** este **00000011**.

00001111 or	**00001111 |**
00000011	**00000011**
00001111	**00001111**

Această reprezentare este dată de numărul întreg **15**.

Operatorul "sau exclusiv"
xor respectiv **^**

Este un operator binar care returnează numărul întreg a cărui reprezentare internă se obţine prin operaţia or exclusiv asupra biţilor care apar în reprezenarea internă a operanzilor. Operaţia se face între biţii situaţi pe aceeaşi poziţie.

Exemplu:

15 xor 3=12	**15 ^ 3==12**

Să verificăm:
Reprezentarea internă a lui **15** este **00001111** iar a lui **3** este **00000011**.

00001111 xor	**00001111 ^**
00000011	**00000011**
00001100	**00001100**

Această reprezentare este dată de numărul întreg **12**.

Operatorul shift left
shl respectiv **<<**

Este un operator binar care returnează numărul întreg a cărui reprezentare este obţinută din reprezentarea internă a primului operand prin deplasare la stânga cu un număr de biţi egal cu al doilea operand.

Exemplu:

4 shl 2=16	**4 << 2==16**

Să verificăm:
Reprezentarea internă a lui **4** este **00000100**. Prin deplasare la stânga cu doi biţi se obţine **00010000**, care este reprezentarea internă a lui **16**.
Acest operator poate fi folosit pentru calculul numerelor întregi de forma 2^n prin efectuarea operaţiilor de forma **1 shl n**. (**1<<n**)

Operatorul shift right
shr respectiv **>>**

Este un operator binar care returnează numărul întreg a cărui reprezentare este obţinută din reprezentarea internă a primului operand prin deplasare la dreapta cu un număr de biţi egal cu al doilea operand.
Prin deplasarea la dreapta, primii biţii din reprezentarea internă a numărului (poziţiile **1**,**2**, ş.a.m.d.) se pierd iar ultimii se completează cu zero.

Exemplu:

14 shr **2**=**3**	**14 >> 2==3**

Să verificăm:
Reprezentara internă a numărului **14** este **000001110**. Prin deplasare la dreapta cu doi biţi a reprezentării lui **14** se obţine **00000011**, care este reprezentarea internă a lui **3**.
Operaţia **n shr 1** (**n>>1**) este echivalentă cu împărţirea întreagă la **2**.

3. Operaţii la nivel de biţi

Transformarea unui bit în 1

Pornim de la valoarea întreagă **X=50**. Reprezentarea acestuia pe **8** biţi este **00110010**. Presupunem că dorim setarea bitului **2** la valoarea **1**. Pentru aceasta vom folosi o mască logică în care doar bitul **2** este **1** restul biţilor fiind **0**, adică **M**=**00000100**. Valoarea lui **M** este **1 shl 2** (2^2). Operaţia de disjuncţie **SAU** aplicată asupra lui **X** şi a lui **M**, conduce la obţinerea rezultatului dorit.

X 00110010 **OR** M 00000100 Rez 00110**1**10	X 00110010 **|** M 00000100 Rez 00110**1**10

Generalizând, dacă se doreşte ca valorii X, să i se seteze la valoarea **1**, bitul B (**0**≤**B**≤**7**), atunci masca logică este **1 shl B**.

X or (1 shl B)	**X | (1 << B)**

Transformarea unui bit în 0

Să luăm ca exemplu **X= 109**, pentru a vedea cum se setează un bit la valoarea **0**. Reprezentarea internă a lui este **01101101**. Se cere să se seteze bitul **5** la valoarea **0**. De data aceasta masca va conţine toţi biţii de **1**, excepţie bitul **5**. Aspura lui **X** şi **M** vom aplica **ŞI** logic.

X 01101101 **AND** M 11011111 Rez 01**0**01101	X 01101101 **&** M 11011111 Rez 01**0**01101

Presupunem că dorim să setăm la **0** valoarea bitului **B**(**0**≤**B**≤**7**). Pentru determinarea valorii măştii **M**, plecăm de la valoarea **255** din care scădem **1** shl **B**.

X and (255 – (1 shl B))	**X & (255 – (1 << B))**

Masca mai poate fi obţinută şi aplicând sau exclusiv în locul scăderii, **255 xor (1 shl B)**. Rescriem rezultatul sub forma:

X and (255 xor(1 shl B))	**X & (255 ^(1 << B))**

Testarea valorii unui bit

Plecăm de la valoarea **X=47**. Reprezentarea internă a lui este **00101111**. Presupunem că dorim să cunoaştem valoarea bitului **3** şi bitului **6**. Vom folosi măştile M_1=**00001000** şi M_2=**01000000**. Vom aplica de fiecare dată **ŞI** logic între **X** şi cele două măşti:

```
X   00101111 AND  |  X   00101111 &
M1  00001000      |  M1  00001000
Rez 00001000      |  Rez 00001000
```

Respectiv

```
X  00101111 AND   |  X  00101111 &
M2 01000000       |  M2 01000000
Rez 00000000      |  Rez 00000000
```

Generalizând, testarea se va realiza prin :

```
X and (1 shl B) <> 0  |  X & (1 << B) != 0
```

Testarea valorilor ultimilor biţi

Pornim de la valoarea întreagă **X=50**. Reprezentarea acestuia pe **8** biţi este **00110010**. Presupunem că dorim să cunoaştem restul la împărţirea întreagă a lui **X** la **8**, adică **X mod 8** respectiv **x%8**. Valoarea ultimilor **3** biţi din reprezentarea internă a lui lui, reprezintă tocmai acest rest. Pentru aceasta vom folosi o mască în care ultimii trei biţii sunt **1** restul **0**. Aceasta mască are valoarea **7**, adică **00000111**. Vom aplica operaţia **ŞI** logic.

```
X   00110010 AND  |  X   00110010 &
M   00000111      |  M   00000111
Rez 00000010      |  Rez 00000010
```

Pe caz general, dacă dorim să cunoaştem valoarea ultimilor **B** biţi(care este egal cu restul împărţirii lui **X** la 2^B) vom exprima astfel:

```
X and (1 shl B - 1)  |  X & (1 << B - 1)
```

Aplicaţii

Problema 1.

Realizaţi un subprogram care determină numărul de cifre de **1** din reprezentarea binară a unui număr natural nenul **n** mai mic ca **2000000000**.

Soluţia 1:
Numărul îl vom reţine într-o variabilă **longint** respectiv **int** şi vom parcurge secvenţial biţii lui.

```
function nr( n:longint):byte;
var nm,i:byte;
begin
 nm:=0;
 for i:=0 to 31 do
  if n  and (1 shl i) <>0 then
    inc(nm);
 nr:=nm;
end;
```

```
int nr (int n){
  int nm=0;
  for ( int i=0; i<32; i++)
    if (n & (1 << i)) nm++;
  return nm;
}
```

Soluţia 2:
O altă soluţie, mai rapidă este folosirea operaţiei **ŞI** logic între valorile lui **n** şi a lui **n-1**.
Această operaţie anulează cel mai nesemnificativ bit de **1** a lui **n**. Să luam ca exemplu valoarea lui **n**:

n = (110011101000100)$_2$
n-1 = (110011101000011)$_2$
n and (n-1) = (110011101000000)$_2$

n = (110011101000100)$_2$
n-1 = (110011101000011)$_2$
n & (n-1) = (110011101000000)$_2$

De aici şi ideea algoritmului următor:

```
function nr( n:longint):byte;
var nm:byte;
begin
 nm:=0;
 repeat
   n:=n and(n-1); inc(nm);
 until n = 0;
 nr:=nm;
end;
```

```
int nr (int n){
  int nm=0;
  do {
    n &= n-1;
    nm++;}
  while (n);
  return nm;
}
```

Rezultatul este mai bun la cea de a doua metodă deoarece execută un număr de paşi egali cu numărul de cifre de **1** din reprezentare.

Problema 2.

Realizaţi un subprogram care verifică dacă un număr natural nenul **n** este o putere a lui **2**.
Soluţie:
Dacă **n** este o putere a lui **2**, va avea o singură cifră de **1** în reprezentarea binară. Dacă ne bazăm pe observaţia de la problema anterioară, obţinem:

```
function pow2( n:longint):boolean;
begin
 pow2 := n and(n-1) = 0;
end;
```

```
int pow2 (int n)
{
   return (n & (n-1)==0);
}
```

Problema 3.

Realizaţi un subprogram care identifică cea mai mare putere a lui **2** care îl divide pe **n**, număr natural nenul.

Soluţia 1:

Problema se reduce la determinarea celui mai nesemnificativ bit de **1** din reprezentarea binară a lui n. Prima soluţie caută poziţia, pornind cu bitul **0**.

```
function UB (n:longint):longint;
var i:integer;
begin
 i:=0;
 while n and (1 shl i)=0 do inc(i);
 UB := 1 shl i;
end;
```

```
int UB (int n){
  int i=0;
  while (!(n & (1<<i))) i++;
  return 1 << i ;
}
```

Soluţia 2

Se bazează pe observaţia că **n and (n-1)** are ca rezultat numărul din care lipseşte cel mai nesemnificativ bit de **0**. Atunci aplicând **XOR** intre valoarea iniţală a lui n şi ce a lui **n and(n-1)** vom obţine valoarea cerută.

Exemplu:

n $=(101000100)_2$
n and (n-1) $=(101000000)_2$
n xor(n and (n-1)) $=(000000100)_2$

n $=(101000100)_2$
n & (n-1) $=(101000000)_2$
n ^(n & (n-1)) $=(000000100)_2$

```
function UB ( n:longint):longint;
begin
 UB := n xor(n and(n-1));
end;
```

```
int UB (int n){

   return n^(n & (n-1)) ;
}
```

O altă variantă este descrisă în continuare:

```
function UB( n:longint):longint;
begin
 UB := (n and(-n));
end;
```

```
int UB(int n){

   return (n & (-n)) ;
}
```

Problema 4.

Realizaţi un subprogram care identifică numărul de ordine al celui mai semnificativ bit de **1** al lui **n** număr natural nenul.

Soluţie

Să luăm ca exemplu următorul şir de operaţii:

n=(10000000)$_2$
n=n or (n shr 1)
n=(11000000)$_2$
n=n or (n shr 2)
n=(11110000)$_2$
n=n or (n shr 4)
n=(11111111)$_2$

n=(10000000)$_2$
n=n | (n >> 1)
n=(11000000)$_2$
n=n | (n >> 2)
n=(11110000)$_2$
n=n | (n >> 4)
n=(11111111)$_2$

Se observă că aplicând o secvenţă asemănătoare de instrucţiuni putem transforma un număr **n** în alt număr în care numărul de biţi de **1** este egal cu **1** + indexul celui mai semnificativ bit de **1**.

```
function index( n:longint):byte;
begin
 n := n or (n shr 1);
 n := n or (n shr 2);
 n := n or (n shr 4);
 n := n or (n shr 8);
 n := n or (n shr 16);
 index:= nr(n)-1;
end;
```

```
int index(int n){
 n = n | (n >> 1);
 n = n | (n >> 2);
 n = n | (n >> 4);
 n = n | (n >> 8);
 n = n | (n >> 16);
 return nr(n) - 1;
}
```

Problema 5

Gigel trebuie să cumpere n medicamente, numerotate de la 1 la n. Doctorul i-a dat m reţete de două tipuri, codificate cu numerele 1, 2 astfel:

1 - reţetă necompensată, adică preţul medicamentelor de pe reţetă se achită integral de către cumpărător;

2 - reţetă compensată **50**%, adică preţul medicamentelor înscrise pe reţetă se înjumătăţeşte.

Se ştie că pe reţete nu există un alt medicament decât cele numeroatete de la 1 la n şi o reţetă nu conţine două medicamente identice.

Dacă o reţetă este folosită atunci se vor cumpăra toate medicamentele înscrise pe ea.

Cerinţă

Scrieţi un program care să determine suma minimă de bani necesară pentru a cumpăra exact câte unul din fiecare dintre cele n medicamente, folosindu-se de reţetele avute la dispoziţie.

Date de intrare

Fişierul de intrare reteta.in are următorul format :

- pe prima linie sunt scrise numerele naturale n şi m;
- pe următoarele m linii sunt descrise cele m reţete, câte o reţetă pe o linie. Linia care descrie o reţetă conţine tipul reţetei (1 necompensată sau 2 compensată), urmat de un număr natural q reprezentând numărul de medicamente de pe reţetă, apoi de q numere distincte din mulţimea {1, 2, ..., n} reprezentând medicamentele înscrise pe acea reţetă;
- pe ultima linie a fişierului de intrare sunt scrise n numere naturale separate prin câte un spaţiu, reprezentând în ordinea de la 1 la n, preţul medicamentelor.

Toate numerele de pe aceeaşi linie sunt separate prin câte un spaţiu.

Date de ieşire

Fişierul de ieşire reteta.out va conţine o singură linie pe care va fi scris un număr real cu o singură zecimală, reprezentând suma minimă determinată.

Restricţii

$1 \le N \le 20$

$1 \le M \le 15$

1 ≤ preţul oricărui medicament ≤ 200; Pentru datele de test există întotdeauna soluţie.

Exemplu

reteta.in	reteta.out	**Explicaţie**
4 5 2 1 3 2 2 2 3 1 1 1 1 3 4 1 2 1 1 3 8 20 2 16	45.0	Soluţia s-a obţinut prin folosirea primei şi celei de a patra reţete. O altă soluţie, dar de cost mai mare, s-ar fi obţinut dacă se folosea reţeta a patra şi cea de a cincea.

Timp maxim de execuţie/test: **1** secundă

Soluţie

Problema admite o soluţie de complexitate exponenţială O(2^m). Se generează fiecare submulţime a mulţimii reţetelor. Se identifică acele submulţimi pentru care medicamentele înscrise pe reţete includ toate cele **n** medicamente, fără să repete vreunul.

Se va păstra ca soluţie submulţimea de reţete care are cost total minim. În continuare vă este prezentată o rezolvare folosind lucru pe biţi.

O reţetă se va codifica cu ajutorului unui întreg (**longint-long**). Fiecare bit **x** (**0≤x≤19**)va codifica prezenta-absenţa medicamentului **x+1** de pe reţetă. Cum există **m** reţete, vectorul **L** va codifica prin elementul **L[i]** reţeta **i+1**, **0** ≤ **i**≤ **m-1** .

Plecând de la exemplu din enunţ, a doua reţetă ce vonţine medicamentele **2** şi **3** se va codifica prin elementul **L[1]** şi va acea valoarea **6**.

.....				1	1	
31 ..6	*5*	*4*	*3*	*2*	*1*	*0*

În acelaşi mod, pentru generarea submulţimilor de reţete ce vor fi folosite, ne vom folosi de vectorii caracteristici asociaţi acestora. Pentru aceasta vom utiliza reprezentarea binară a tuturor numerelor de la **0** la $\mathbf{2^m-1}$.

Dacă variabila **s** are valoarea **11** atunci ea cuprinde prima, a doua şi a patra reţetă. Medicamentele pe care acestea le conţin sunt codificate în **L[0]**, **L[1]** şi **L[3]**.

.....			1		1	1
31 ..6	*5*	*4*	*3*	*2*	*1*	*0*

Variabila **u** va reprezenta o mască logică care indică medicamentele prinse pe reţetele ce aparţin submulţimii curente.

```
const INF=2000000000;
var L,T,C:array[0..20]of longint;
    P:array[0..20]of real;
    n,m,i,j,k,nr_med,x,s,u:longint;
    ok:boolean;
    sum,min:real;
    f:text;
begin

 assign( f, 'reteta.in');reset(f);
 read(f,n,m);
 for  i:=0 to m-1 do begin
   read(f, T[i] , nr_med  );
   for j:=1 to nr_med do begin
      read(f, x);
      L[i]:=L[i] or (1 shl(x-1));
    end;
 end;
 for i:=0 to n-1 do read(f, C[i]);
 close ( f );


 for i:=0 to m-1 do begin
  sum := 0;
```

```
#include <stdio.h>
#include <string.h>

int L[20],T[20],C[20],n,m,i,j;
int k,nr_med,x,s,u,corect;
double P[20],sum,min;

int main (){

 freopen ( "reteta.in" , "r" , stdin );
 scanf ( "%d %d" , &n , &m );
 for ( i=0 ; i<m ; i++ ) {
  scanf ( "%d %d" , &T[i] , &nr_med  );
  for ( j=0 ; j<nr_med ; j++ ) {
    scanf ( "%d" , &x );
    L[i] |= 1<<(x-1);
  }
 }
 for (i=0;i<n;i++) scanf ("%d",&C[i]);
fclose ( stdin );


for ( i=0 ; i<m ; i++ ) {
  sum = 0;
```

```
  for  j:=0 to n-1 do
   sum:=sum+((L[i] shr j) and 1)*C[j];
   P[i]:=sum/T[i];
 end;


 min:=INF;
 for s:=0 to 1 shl m -1 do begin
  sum:=0; u := 0; ok:=true; i:=0;
  while (i<m) and ok do begin
   if (s shr i) and 1 = 1 then
      if (u and L[i])<>0 then
          ok:=false
      else begin
          sum:=sum+P[i];
          u:=u or L[i];
      end;
      inc(i);
  end;
  if ok and (sum<min) and
  ( u=( 1 shl n)-1) then min:=sum;
 end;


 assign(f,'reteta.out'); rewrite(f);
  if (INF-min>0.1)then
    writeln (f, min:0:1)
  else
    writeln(f,'imposibil');
 close ( f );
end.
```

```
  for ( j=0 ; j<n ; j++ )
    sum += ((L[i]>>j)&1) * C[j];
    P[i]=sum/T[i];
  }
  int pp=0;

  min=2000000000;
  for (s=0 ; s<=(1<<m)-1 ; s++)
   {
    sum=0; u = 0; corect=1;
    for (i=0 ; i<m && corect ; i++ )
      if ((s>>i)&1)
         if (u&L[i]) corect=0;
         else
         {
          sum += P[i];
           u |= L[i];
         }

if (corect && sum<min && u==(1<<n)-1)
           min=sum;
    }

freopen ( "reteta.out","w",stdout);
 if (min!=2000000000)
  printf ( "%.1lf\n" , min );
 else printf ( "imposibil\n" );
 fclose ( stdout );
return 0;
}
```

Problema 6 - Codul Gray

Se consideră şirul tuturor cuvintelor de lungime n formate din cifre binare. Să se aranjeze aceste cuvinte astfel încît oricare două alăturate să difere printr-o singură poziţie.

Exemplu: Pentru n=2 şirul conţine:

```
00
10
11
01
```

Soluţie

Dacă asociem fiecărui element din şirul soluţie numerele de la 1 la 1 shl n (1 << n) se observă că pentru un număr x bitul schimbat în elementul următor din şirul Gray este egal cu 1 + puterea maximă lui 2, care îl divide pe x.

```
var a:Array[1..20] of integer;
    f,g:text;
    n,i,x,nr:longint;

procedure afis(nr:longint);
var i:longint;
begin
      for i:=1 to n do
                write(g,a[i]);
      write(g,'  --->  ',nr);
      writeln(g);
end;

begin
 assign(f,'Gray.in'); reset(f);
 assign(g,'Gray.out'); rewrite(g);
 readln(f,n);
 for i:=1 to (1 shl n) do begin
       afis(i);
       x:=i; nr:=1;
```

```
#include <stdio.h>
int a[21];
int  n,i,x,nr;

void afis(int nr){
 int i;
 for (i=1;i<=n;i++)
  printf("%d",a[i]);
  printf("  --->  %ld", nr);
  printf("\n");
 }

int main(){
 freopen("gray.in","r",stdin);
 freopen("gray.out","w",stdout);

 scanf("%ld",&n);

 for (i=1; i<= (1<<n); i++){
            afis(i);
```

```
        while x and 1=0 do begin
                nr:=nr+1;
                x:=x shr 1;
        end;
        a[nr]:=a[nr] xor 1;
 end;
 close(f); close(g);
end.
```

```
            x=i; nr=1;
            while (!(x & 1)) {
                        nr++;
                        x >>= 1;
            }
            a[nr] ^= 1;
 } fclose(stdin); fclose(stdout);
 return 0;}
```

Problema 7

Se dau v1 , v2 ... vk numere prime k<=20. Să se determine câte numere mai mici ca un N dat (N<=2000000000) sunt divizibile cu cel putin unul din cele K numere prime.
Exemplu
Pentru n=20 , k=4 si şirul 2, 3, 5, 7 se va afişa 15.

Soluţie:

Se foloseste principiul includerii si al excluderii. Astfel numărul de numere divizibile cu cel puţin un număr dintre cel K este egal cu:

$$Nr(V1,V2,..Vk) = \sum_{1<=i<=n} Vi - \sum_{1<=i<j<=ni} Vi*Vj + ...(-1)^k \sum V1*V2*..Vk$$

```
var v,mask:array[0..20]of longint;
    f,g:text;
i,x,numar,nr,produs,actual,suma,k,n:
                                longint;

begin
 assign(f,'P1.in'); reset(f);
 assign(g,'P1.out'); rewrite(g);
 read(f,k,n);
 for i:=1 to k do
        read(f,v[i]);
 suma:=0;
 produs:=1; nr:=0;
 for i:=1 to (1 shl k)-1 do begin
        x:=i; numar:=1;
        while x and 1=0 do begin
                x:=x shr 1;
                numar:=numar+1;
        end;
        if mask[numar]=1 then begin
          mask[numar]:=0;
          nr:=nr-1;
          produs:=produs div v[numar];
        end
        else begin
          mask[numar]:=1;
          nr:=nr+1;
          produs:=produs * v[numar];
        end;
        actual:=n div produs;
        if nr and 1=1 then
                suma:=suma + actual
        else
                suma:=suma - actual;
 end;
 writeln(g,suma);
 close(f); close(g);end.
```

```
#include <fstream.h>
int v[21],mask[21];
int i,x,numar,nr,produs,actual,suma,k,n;

int main(){
 ifstream f("P1.in");
 ofstream g("P1.out");
 f>>k>>n;
 for(i=1;i<=k;i++)
  f>>v[i];
 suma=0;
 produs=1; nr=0;
 for (i=1; i< (1 << k);i++) {
            x=i; numar=1;
            while ((x & 1)==0) {
                        x >>= 1;
                        numar++;
            }
            if (mask[numar]==1) {
                  mask[numar]=0;
                  nr--;
                  produs /= v[numar];
            }
            else{
                  mask[numar]=1;
                  nr++;
                  produs *= v[numar];
            }
            actual = n/produs;
            if ((nr & 1)==1)
                        suma +=actual;
            else
                        suma -=actual;
 }
 g<<suma;
 f.close();g.close(); return 0;
}
```

2. Arbori indexati binar

Arborii indexaţi binar reprezintă o structură de date care permite efectuarea următoarelor operaţii în complexitata O(*log N*) pentru un vector de *N* elemente:

- ADD(*x, y*) – adună la elementul de pe poziţia *x* din vector, valoarea *y;*
- SUM (*x*) – calculează suma primelor *x* elemente din vector (se presupune că vectorul este indexat de la 1 la *N*).

Operaţiile descrise mai sus pot fi efectuate în complexitatea menţionată folosind un arbore de intervale (descris în capitolul anterior), dar folosirea unui arbore indexat binar oferă anumite avantaje:

- implementarea este mai uşoară;
- se folosesc fix *N* locaţii de memorie pentru stocarea arborelui;
- operaţiile se pot extinde uşor pentru a rezolva aceeaşi problemă în mai multe dimensiuni.

Din păcate, această structură de date nu oferă aceeaşi flexibilitate precum un arbore de intervale, existând seturi de operaţii care pot fi efectuate în complexitate logaritmică folosind doar arbori de intervale.

Eficienţa acestei structuri de date nu are legătură cu echilibrarea arborelui precum alte structuri de date convenţionale, ci din modul de indexare a elementelor, ţinând cont de reprezentarea binară (de unde şi *numele de arbore indexat binar*).

Ideea de bază este că, precum un număr *n* poate fi exprimat ca sumă de puteri ale lui 2, aşa şi un interval [1…*n*] poate fi exprimat ca reuniunea unor subintervale de lungimi egale cu puteri ale lui 2.

Vom lua în continuare, ca exemplu, intervalul [1…11]:

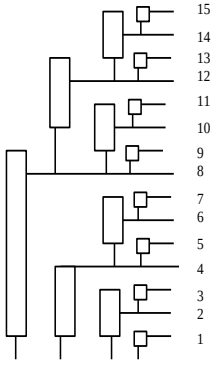
$11 = 2^3 + 2^1 + 2^0 = 1011_{(2)}$.

Eliminând cel mai puţin semnificativ bit de 1 din reprezentarea lui 11 în baza 2, constatăm că suma pentru intervalul [1…11] poate fi calculată astfel:

[1…11] ($11 = 1011_{(2)}$)= [1…10] + [11…11]
[1…10] ($10 = 1010_{(2)}$)= [1…8] + [9…10]
[1…8] ($8 = 1000_{(2)}$)

Astfel, dacă se asociază fiecărui element *x* din arbore un interval [x-2^k+1…x], unde *k* reprezintă numărul zerourilor terminale din reprezentarea binară a lui *x*, fiecare interval de forma [1…*x*] poate fi exprimat ca reuniunea a cel mult $log_2 N$ intervale, folosind procedeul (descris mai sus) de eliminare succesivă a celui mai nesemnificativ bit de 1. Având stabilită această structură, operaţia SUMĂ(*x*) se efectuează însumând intervalele din descompunerea lui [1…*x*].

Considerăm un arbore indexat binar cu 15 elemente (distribuţia elementelor este reprezentată în figura alăturată). Pentru a actualiza valoarea unui element *x* (operaţia ADAUGĂ(*x, y*)) se actualizează întâi intervalul asociat lui *x* ([x-2^k+1…x]) cât şi restul intervalelor care-l conţin pe *x*. Acest lucru se efectuează adăugând succesiv cel mai nesemnificativ bit de 1. Spre exemplu, pentru a actualiza elementul de pe poziţia 9, se modifică următoarele elemente (se consideră că sunt 15 elemente):



9 ($9 = 1001_{(2)}$) - [9…9]
10 ($10 = 1010_{(2)}$) - [9…10]
12 ($10 = 1100_{(2)}$) - [9…12]

Pentru a determina în O(1) cel mai nesemnificativ bit de 1 al unui număr, se poate folosi expresiaŞ **`(x and (-x))`** (Pascal) / **`(x & (-x))`** (C/C++).

Sau
`(x and (x-1)) xor x` (Pascal) / **`(x & (x-1)) ^ x`** (C/C++).

Să luam ca exemplu un vector V cu 15 elemente, care suferă modificări succesive. Acestuia îi asociem un arbore indexat binar, reprezentat sub forma unui vector AIB în care elementele reţin informaţii referitoare la intervale de elemente din V, după cum urmează:

[1..1]	[1..2]	[3..3]	[1..4]	[5..5]	[5..6]	[7..7]	[1..8]	[9..9]	[9..10]	[11..11]	[9..12]	[13..13]	[13..14]	[15..15]
AIB[1]	AIB[2]	AIB[3]	AIB[4]	AIB[5]	AIB[6]	AIB[7]	AIB[8]	AIB[9]	AIB[10]	AIB[11]	AIB[12]	AIB[13]	AIB[14]	AIB[15]

Implementare functiilor ADD si SUM se poate realiza astfel:

```
1  ...
2  function UB(x:integer):integer;
3  begin UB:=(x and (-x)) end;
4
5  procedure ADD(x,val:integer);
6  var i,j:integer;
7  begin
8  i:=x;
9   while i<=n do begin
10    inc(AIB[i],val);
11    inc(i,UB(i));
12  end;
13 end;
14
15 function SUM(x,y:integer):integer;
16 var i,j,S:integer;
17 begin
18 i:=x;
19  while i>0 do begin
20    inc(S,AIB[i]);
21    dec(i,UB(i));
22  end;
23 SUM:=S;
24 end;
```

```
...
#define UB(x) (x&(-x))

...
int AIB[NMAX],V[NMAX];


void ADD(int x,int val)
{
    int i;
    for(i=x;i<=N;i+=UB(i)) AIB[i]+=val;
}


int SUM(int x)
{
    int i,S=0;

    for(i=x;i>0;i-=UB(i)) S+=AIB[i];

    return S;
}
```

Deşi această structură de date nu este la fel de flexibilă ca un arbore de intervale, ea poate fi extinsă să suporte şi alte operaţii:

- SUMA(x, y) – suma elementelor cu poziţii între x şi y: se poate rescrie ca suma elementelor din intervalul [1..x] minus suma elementelor din intervalul [1..y-1];
- MODIFICĂ(x, y) – actualizează valoarea elementului x la y, dacă aceasta era mai mică decât y;
  MAXIM(x) – determină maximul din primele x elemente.

Aceste operaţii pot fi implementate înlocuind operaţia de adunare ("+") cu cea de maxim.

- ADAUGĂ(x, y, z) – adaugă la toate elementele cu poziţii între x şi y valoarea z;
- AFLĂ(x) – determină valoarea elementului de pe poziţia x.

Aplicaţii - Arbori indexaţi binar

Problema – Inv (www.infoarena.ro)

Se dă un şir S de lungime n cu numere întregi. Numim o inversiune o pereche de indici `(i, j)` astfel încât `1 ≤ i < j ≤ n` şi `S[i] > S[j]`.

Cerinţă

Să se determine câte inversiuni sunt în şirul dat.

Date de intrare

Fişierul de intrare **inv.in** conţine pe prima linie numărul natural **N**. Pe următoarea linie se găsesc **N** numere întregi, reprezentând în ordine elementele şirului S.

Date de ieşire

În fişierul de ieşire **inv.out** se va afişa un singur număr, reprezentând restul împărţirii numărului de inversiuni din şir la valoarea 9917.

Restricţii

- $2 \leq N \leq 100\ 000$

Exemplu

inv.in	**inv.out**
5 3 4 1 2 5	4

Soluţie:

Problema se rezolvă plecând de la numărul maxim de inversiuni dintr-un vector. Şirul S preluat din **inv.in** va fi sortat crescător, pentru fiecare element reținâdu-se și poziția inițială în S(normalizare). Din numărul maxim se inversiuni se vor scade numărul de inversiuni ce nu s-au format, adică numărul de elemente mai mici decât cel curent, situate înaintea lui, în şirul S.

Într-un arbore indexat binar, elementul AIB[x] va reține numărul de elemente introduse până în acel moment și care sunt situate în șirul S pe poziții între $[x-2^k+1,..,x]$. Evident ele au valori mai mici decât elementul curent și deci nu formează cu el inversiuni.

```
#include <cstdio>
#include <algorithm>
#include <vector>

#define ub(x) (x&(-x))

#define NMAX 100001
#define MOD 9917

using namespace std;

vector < pair< int,int > > V;
int A[NMAX];
int N,i,x,k;
long long sol;

void add(int x)
{
    int i=0;

    for (i=x;i<=N;i+=ub(i)) A[i]++;
}

int Sum(int x)
{
    int i=0,S=0;

    for (i=x;i>0;i-=ub(i)) S+=A[i];
    return S;
}

int main()
{
    freopen("inv.in","r",stdin);
    freopen("inv.out","w",stdout);

    scanf("%d",&N);

    for (i=0;i<N;i++)
```

```
    {
        scanf("%d",&x);
        V.push_back(make_pair(x,i+1));
    }
    sort(V.begin(),V.end());

    sol=1LL*N*(N-1)/2;
    for (i=0;i<N;i++)
    {
        x=V[i].second;
        sol-=1LL*Sum(x);
        add(x);
    }

    printf("%d\n",sol%MOD);
    return 0;
}
```

Problema – Schi (www.infoarena.ro)

La un concurs de schi, are loc o probă contra-cronometru, concurenții trebuind să termine traseul în cel mai scurt timp cu putință. Acestia vor evolua consecutiv și, după finish, li se va comunica locul ocupat în clasamentul intermediar. Mai exact, dupa evolutia celui de-al **p**-lea concurent, acesta va ști locul său în clasamentul format de primii **p** concurenti. Cunoscând pozițiile intermediare, se cere să se determine clasamentul final al competiției.

Date de intrare

Pe prima linie a fișierului de intrare **schi.in** se află **N**, reprezentând numărul de concurenți aliniați la startul concursului. Pe următoarele **N** linii se află câte o valoare întreagă, indicând locul ocupat de fiecare concurent în clasamentul intermediar, actualizat după evoluția sa.

Date de ieşire

Fisierul de iesire **schi.out** va avea **N** linii. Pe linia **i** se va afișa numărul de ordine al concurentului care ocupa locul **i** în clasamentul final.

Restricţii

- $1 \leq N \leq 30\ 000$

Exemplu

inv.in	**inv.out**
8 1 1 3 4 4 2 1 3	7 2 8 6 1 3 5 4

Soluţie:

Pentru a rezolva problema vom reține, într-un arbore indexat binar numărul de poziții libere în clasamentul final. Astfel, elementul AIB[x] va reține câte poziții libere există între $[x-2^k+1,..,x]$. Se va parcurge invers șirul din fișierul de intrare, și pentru fiecare valoare X[i], se va determina cea mai mica poziție x pentru care suma elementelor din AIB pe intervalul [1..x] este egală cu X[i].
Poziția x este poziția din clasamentul final a concurentului cu mumărul de ordine i.

```
1  #include <cstdio>
2  #include <climits>
3  #define ub(x) (x&(-x))
4  #define NMAX 30001
5
6  using namespace std;
7
```

```
 8  int A[NMAX],X[NMAX],sol[NMAX];
 9  int N,i,x;
10
11  void Decrease(int x)
12  {
13      int i;
14      for(i=x;i<=N;i+=ub(i)) A[i]--;
15  }
16
17  int Sum(int poz)
18  {
19      int i=0,T=0;
20      for(i=poz; i>0; i-=ub(i)) T+=A[i];
21      return T;
22  }
23
24  int binary_search(int x)
25  {
26      int S=1,D=N,M=0,T=0,Minim=INT_MAX;
27      while(S<=D)
28      {
29          M=(S+D)/2;
30          T=Sum(M);
31          if (T==x && M<Minim) Minim=M;
32                else if (T>=x) D=M-1;
33                             else S=M+1;
34      }
35      return Minim;
36  }
37
38  int main()
39  {
40      freopen("schi.in","r",stdin);
41      freopen("schi.out","w",stdout);
42
43      scanf("%d",&N);
44      for (i=1; i<=N; ++i) scanf("%d",&X[i]);
45      for (i=1; i<=N; ++i) A[i]=ub(i);
44
45      for (i=N; i>=1; --i)
46      {
47          x=binary_search(X[i]);
48          sol[x]=i;
49          Decrease(x);
50      }
51
52      for (i=1;i<=N;++i) printf("%d\n",sol[i]);
53
54      return 0;
55  }
```

Problema – Order (www.infoarena.ro)

Se consideră **N** copii aşezaţi în cerc şi numerotaţi de la 1 la **N** în sens trigonometric. Copiii joacă următorul joc: jocul începe de la primul copil (cel al cărui număr de ordine este 1); la fiecare al **i**-lea pas al jocului se numară **i** copii in sens trigonometric şi este eliminat copilul la care se ajunge; la pasul următor numărătoarea începe de la copilul care urmează după cel eliminat.
Aşadar, dacă numărul copiilor este suficient de mare, la primul pas este eliminat al doilea copil, la al doilea pas al patrulea, la al treilea pas al şaptelea, apoi la al patrulea pas al unsprezecelea şi aşa mai departe.

Cerinţă
Va trebui să determinaţi ordinea în care vor fi eliminaţi copiii.

Date de intrare
Fişierul de intrare order.in conţine pe prima linie un numar intreg **N**, care reprezintă numărul de copii.

Date de ieşire

Fișierul de ieșire order.out trebuie să conțină o singura linie pe care se vor afla **N** numere distincte cuprinse între 1 și **N** care reprezintă numerele de ordine ale copiilor în ordinea în care au fost eliminați.

Restricţii

- $2 \leq N \leq 30\ 000$

Exemplu

order.in	**order.out**
6	2 4 1 3 5 6

Soluţie:

În mod asemănător, în AIB vom reține la fiecare pas, copiii rămași în cerc. Pentru fiecare extragere se va identifica, printr-o căutare binară, cel mai mic indice *x* din AIB, pentru care suma [1..*x*] este egală cu cel a copilului ce urmează la extragere.

```
#include <cstdio>
#include <climits>
#define ub(x) (x&(-x))
#define NMAX 30001

using namespace std;

int A[NMAX],sol[NMAX];
int x,N,i,ind,P=2;

void add(int x,int val)
{
    int i=0;
    for(i=x;i<=N;i+=ub(i)) A[i]+=val;
}

int Sum(int x)
{
    int i=0,S=0;
    for(i=x;i>0;i-=ub(i)) S+=A[i];
    return S;
}

int binary_search(int x)
{
    int D=N,S=1,M=0,Sm=0,Minim=INT_MAX;

    while(S<=D){
        M=(S+D)/2;
        Sm=Sum(M);
        if( Sm==x && M<Minim) Minim=M;
             else if (Sm>=x) D=M-1;
                  else S=M+1;
    }
    return Minim;
}

int mod(int x,int MOD)
{
    if (!(x%MOD)) return MOD;

    return x%MOD;
}

int main()
{
    freopen("order.in","r",stdin);
    freopen("order.out","w",stdout);
    scanf("%d",&N);
    for(i=1;i<=N;++i) add(i,1);
    ind=N;

```

```
51      for(i=1;i<=N;++i)
52      {
53          --ind;
54
55          x=binary_search(P);
56          add(x,-1);
57
58          sol[i]=x;
59          P+=i;
60
61          if (ind!=0) P=mod(P,ind);
62      }
63
64      for(i=1;i<=N;++i) printf("%d ",sol[i]);
65      return 0;
66  }
```

Probleme suplimentare:

Pe campion.edu.ro

- Concurs1:

Pe infoarena.ro:

- Datorii
- AIB
- Ben
- Evantai