

Tabăra de pregătire a lotului de juniori
Ramnicu Valcea, 2015

Suport de curs

“Operații pe biți”

Lot
național
Juniori

Prof. Dana Lica

Revista
Infobits
<https://revista.infobits.ro>

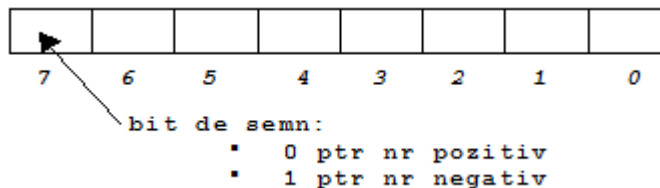
Operații pe biți

1. Reprezentarea internă a numerelor întregi

Reprezentarea în memorie a numerelor întregi se face printr-o secvență de cifre de 0 și 1. Această secvență poate avea o lungime de 8, 16 sau 32 de biți.

Forma de memorare a întregilor se numește *cod complementar*. În funcție de lungimea reprezentării se stabilește 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 biți, valabilă pentru tipul shortint în Pascal, respectiv char în C/C++.



Se observă că numerotarea pozițiilor se face de la dreapta la stânga (de la 0 la 7), poziția 7 fiind bit de semn (0 pentru numerele pozitive și 1 pentru numerele negative).

Rezultă că doar 7 biți (pozițiile 0–6) se folosesc pentru reprezentarea valorii absolute a numărului.

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

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 biți, iar poziția 7 (bitul 8) va fi 0 deoarece numărul este pozitiv.

Deci reprezentarea internă este următoarea:

0	0	0	0	0	1	0	1
7	6	5	4	3	2	1	0

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

- 1) determinarea reprezentării interne a numărului ce reprezintă valoarea absolută a numărului inițial. Acesta are bitul de semn egal cu 0.
- 2) se calculează complementul față de 1 a reprezentării obținute la pasul anterior (bitul 1 devine 0, iar bitul 0 devine 1)
- 3) se adună 1 (adunarea se face în baza 2) la valoarea obținută.

Exemplu:

Pentru determinarea reprezentării numărului -5 se procedează în felul următor:

Reprezentarea valorii absolute a numărului -5 este 00000101. Complementul față de 1 este:

11111010

Numărul obținut după adunarea cu 1 este :

11111011

Deci reprezentarea valorii întregi **-5** pe **8** biți este:

1	1	1	1	1	0	1	1
7	6	5	4	3	2	1	0

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** biți sunt cuprinse între 10000000_2 și 01111111_2 , adică -128_{10} , 127_{10} .

Așa cum am spus și la începutul cursului, numerele întregi se pot reprezenta în cod complementar, având la dispoziție **16**, **32** sau chiar și **64** de biți. Mecanismul este același, însă valorile numerelor cresc.

2. Operatori la nivel de bit

Operatorii pe biți se pot aplica datelor ce fac parte din tipurile întregi. Operațiile se efectuează asupra biților din reprezentarea internă a numerelor.

Operatorul de negație

not (Pascal) respectiv **~** (C/C++)

Este un operator unar care întoarce numărul întreg a cărui reprezentare internă se obține din reprezentarea internă a numărului inițial, prin complementarea față de **1** a fiecărui bit (**1**→**0** și **0**→**1**).

Exemplu:

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

Reprezentarea lui **5** este **00000101**. Complementând acest număr obținem reprezentarea **11111010** care corespunde numărului întreg **-6**. Să verificăm:

Numărul **6** se reprezintă intern astfel:

$6_2 = 00000110$, complementând obținem **11111001** și adunând **1** (în baza **2**) se obține în final **11111010** adică exact reprezentarea internă returnată a numărului returnat de operația **not 5** respectiv **~5**.

Operatorul de conjuncție

and respectiv **&**

Este un operator binar care returnează numărul întreg a cărui reprezentare internă se obține prin conjuncția biților care apar în reprezentarea internă a operanzilor. Conjuncția se face cu toate perechile de biți situați pe aceeași poziție.

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 disjuncție

or respectiv **|**

Este un operator binar care returnează numărul întreg a cărui reprezentare internă se obține prin disjuncția biților care apar în reprezentarea internă a operanzilor. Disjuncția se face între biții situați pe aceeași poziție.

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
00000011
00001111

00001111 |
00000011
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 obține prin operația or exclusiv asupra biților care apar în reprezentarea internă a operanzilor. Operația se face între biții situați pe aceeași poziție.

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
00000011
00001100

00001111 ^
00000011
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 obținută din reprezentarea internă a primului operand prin deplasare la stânga cu un număr de biți 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 biți se obține **00010000**, care este reprezentarea internă a lui **16**.

Acest operator poate fi folosit pentru calculul numerelor întregi de forma **2ⁿ** prin efectuarea operațiilor de forma **1 shl n**. (**1<<n**)

Operatorul shift right

shr respectiv **>>**

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

Prin deplasarea la dreapta, primii biți din reprezentarea internă a numărului (pozițiile 1,2, ș.a.m.d.) se pierd iar ultimii se completează cu zero.

Exemplu:

14 shr 2=3

14 >> 2==3

Să verificăm:

Reprezentarea internă a numărului **14** este **000001110**. Prin deplasare la dreapta cu doi biți a reprezentării lui **14** se obține **00000011**, care este reprezentarea internă a lui **3**.

Operația **n shr 1 (n>>1)** este echivalentă cu împărțirea întreagă la **2**.

3. Operații la nivel de biți

Transformarea unui bit în 1

Pornim de la valoarea întreagă **x=50**. Reprezentarea acestuia pe **8** biți 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 biților fiind **0**, adică **M=00000100**. Valoarea lui **M** este **1 shl 2 (2²)**. Operația de disjuncție **SAU** aplicată asupra lui **X** și a lui **M**, conduce la obținerea rezultatului dorit.

X	00110010	OR
M	00000100	
Rez	00110 <u>1</u> 10	

X	00110010	
M	00000100	
Rez	00110 <u>1</u> 10	

Generalizând, dacă se dorește 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 conține toți biții de **1**, excepție bitul **5**. Asupra lui **X** și **M** vom aplica **ȘI** logic.

X	01101101	AND
M	11011111	
Rez	01 <u>0</u> 01101	

X	01101101	&
M	11011111	
Rez	01 <u>0</u> 01101	

Presupunem că dorim să setăm la **0** valoarea bitului **B(0≤B≤7)**. Pentru determinarea valorii măștii **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 obținută și 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 cunoaștem valoarea bitului **3** și bitului **6**. Vom folosi măștile $M_1=00001000$ și $M_2=01000000$. Vom aplica de fiecare dată **ȘI** logic între X și cele două măști:

Respectiv	$\begin{array}{r} X \quad 00101111 \text{ AND} \\ M1 \quad 00001000 \\ \hline Rez \quad 0000\underline{1}000 \end{array}$	$\begin{array}{r} X \quad 00101111 \text{ \&} \\ M1 \quad 00001000 \\ \hline Rez \quad 0000\underline{1}000 \end{array}$
	$\begin{array}{r} X \quad 00101111 \text{ AND} \\ M2 \quad 01000000 \\ \hline Rez \quad 0\underline{0}000000 \end{array}$	$\begin{array}{r} X \quad 00101111 \text{ \&} \\ M2 \quad 01000000 \\ \hline Rez \quad 0\underline{0}000000 \end{array}$

Generalizând, testarea se va realiza prin :

$$X \text{ and } (1 \text{ shl } B) \neq 0 \quad | \quad X \text{ \& } (1 \ll B) \neq 0$$

Testarea valorilor ultimilor biți

Pornim de la valoarea întreagă $X=50$. Reprezentarea acestuia pe **8** biți este **00110010**. Presupunem că dorim să cunoaștem restul la împărțirea întreagă a lui X la **8**, adică $X \bmod 8$ respectiv $x\%8$. Valoarea ultimilor **3** biți din reprezentarea internă a lui X , reprezintă tocmai acest rest. Pentru aceasta vom folosi o mască în care ultimii trei biți sunt **1** restul **0**. Aceasta mască are valoarea **7**, adică **00000111**. Vom aplica operația **ȘI** logic.

$\begin{array}{r} X \quad 00110010 \text{ AND} \\ M \quad 00000111 \\ \hline Rez \quad 00000\underline{010} \end{array}$	$\begin{array}{r} X \quad 00110010 \text{ \&} \\ M \quad 00000111 \\ \hline Rez \quad 00000\underline{010} \end{array}$
--	---

Pe caz general, dacă dorim să cunoaștem valoarea ultimilor B biți (care este egal cu restul împărțirii lui X la 2^B) vom exprima astfel:

$$X \text{ and } (1 \text{ shl } B - 1) \quad | \quad X \text{ \& } (1 \ll B - 1)$$

Aplicații

Problema 1.

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

Soluția 1:

Numărul îl vom reține într-o variabilă **longint** respectiv **int** și vom parcurge secvențial biții lui.

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

Soluția 2:

O altă soluție, mai rapidă este folosirea operației **SI** logic între valorile lui **n** și a lui **n-1**.

Această operație anulează cel mai nesemnificativ bit de **1** a lui **n**. Să luăm ca exemplu valoarea lui **n**:

n	= (110011101000100)₂	n	= (110011101000100)₂
n-1	= (110011101000011)₂	n-1	= (110011101000011)₂
n and (n-1)	= (110011101000000)₂	n & (n-1)	= (110011101000000)₂

De aici și ideea algoritmului următor:

<pre>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;</pre>	<pre>int nr (int n){ int nm=0; do { n &= n-1; nm++;} while (n); return nm; }</pre>
---	--

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

Problema 2.

Realizați un subprogram care verifică dacă un număr natural nenul **n** este o putere a lui **2**.

Soluție:

Dacă **n** este o putere a lui **2**, va avea o singură cifră de **1** în reprezentarea binară. Dacă ne bazăm pe observația de la problema anterioară, obținem:

<pre>function pow2(n:longint):boolean; begin pow2 := n and(n-1) = 0; end;</pre>	<pre>int pow2 (int n) { return (n & (n-1)==0); }</pre>
--	--

Problema 3.

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

Soluția 1:

Problema se reduce la determinarea celui mai nesemnificativ bit de 1 din reprezentarea binară a lui n . Prima soluție caută poziția, 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 ;  
}
```

Soluția 2

Se bazează pe observația că $n \text{ and } (n-1)$ are ca rezultat numărul din care lipsește cel mai nesemnificativ bit de 0. Atunci aplicând XOR între valoarea inițială a lui n și ce a lui $n \text{ and } (n-1)$ vom obține 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.

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

Soluție

Să luăm ca exemplu următorul șir de operații:

```
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 secvență asemănătoare de instrucțiuni putem transforma un număr n în alt număr în care numărul de biți de 1 este egal cu $1 + \text{indexul celui mai semnificativ bit de } 1$.

```
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 rețete de două tipuri, codificate cu numerele 1, 2 astfel:

1 - rețetă necompensată, adică prețul medicamentelor de pe rețetă se achită integral de către cumpărător;

2 - rețetă compensată 50%, adică prețul medicamentelor înscrise pe rețetă se înjumătățește.

Se știe că pe rețete nu există un alt medicament decât cele numeroatete de la 1 la n și o rețetă nu conține două medicamente identice.

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

Cerință

Scrieți 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 rețetele avute la dispoziție.

Date de intrare

Fișierul de intrare `reteta.in` are următorul format :

- pe prima linie sunt scrise numerele naturale n și m ;
- pe următoarele m linii sunt descrise cele m rețete, câte o rețetă pe o linie. Linia care descrie o rețetă conține tipul rețetei (1 necompensată sau 2 compensată), urmat de un număr natural q reprezentând numărul de medicamente de pe rețetă, apoi de q numere distincte din mulțimea $\{1, 2, \dots, n\}$ reprezentând medicamentele înscrise pe acea rețetă;
- pe ultima linie a fișierului de intrare sunt scrise n numere naturale separate prin câte un spațiu, reprezentând în ordinea de la 1 la n , prețul medicamentelor.

Toate numerele de pe aceeași linie sunt separate prin câte un spațiu.

Date de ieșire

Fișierul de ieșire `reteta.out` va conține o singură linie pe care va fi scris un număr real cu o singură zecimală, reprezentând suma minimă determinată.

Restricții

$$1 \leq N \leq 20$$

$$1 \leq M \leq 15$$

$1 \leq \text{prețului oricărui medicament} \leq 200$; Pentru datele de test există întotdeauna soluție.

Exemplu

reteta.in	reteta.out	Explicație
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	Soluția s-a obținut prin folosirea primei și celei de a patra rețete. O altă soluție, dar de cost mai mare, s-ar fi obținut dacă se folosea rețeta a patra și cea de a cincea.

Timp maxim de execuție/test: 1 secundă

Soluție

Problema admite o soluție de complexitate exponențială $O(2^m)$. Se generează fiecare submulțime a mulțimii rețetelor. Se identifică acele submulțimi pentru care medicamentele înscrise pe rețete includ toate cele n medicamente, fără să repete vreunul.

Se va păstra ca soluție submulțimea de rețete care are cost total minim. În continuare vă este prezentată o rezolvare folosind lucru pe biți.

O rețetă se va codifica cu ajutorul unui întreg (**longint-long**). Fiecare bit x ($0 \leq x \leq 19$) va codifica prezenta-absența medicamentului $x+1$ de pe rețetă. Cum există m rețete, vectorul L va codifica prin elementul $L[i]$ rețeta $i+1$, $0 \leq i \leq m-1$.

Plecând de la exemplu din enunț, a doua rețetă ce conține medicamentele 2 și 3 se va codifica prin elementul $L[1]$ și va avea valoarea 6.

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

În același mod, pentru generarea submulțimilor de rețete ce vor fi folosite, ne vom folosi de vectorii caracteristici asociați acestora. Pentru aceasta vom utiliza reprezentarea binară a tuturor numerelor de la 0 la 2^m-1 .

Dacă variabila s are valoarea 11 atunci ea cuprinde prima, a doua și a patra rețetă. Medicamentele pe care acestea le conțin sunt codificate în $L[0]$, $L[1]$ și $L[3]$.

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

Variabila u va reprezenta o mască logică care indică medicamentele prinse pe rețetele ce aparțin submulțimii 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 ( "%.11f\n" , min );
else printf ( "imposibil\n" );
fclose ( stdout );
return 0;
}

```

Problema 6 - Codul Gray

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

Exemplu: Pentru $n=2$ șirul conține:

```

00
10
11
01

```

Soluție

Dacă asociem fiecărui element din șirul soluție numerele de la 1 la $1 \text{ shl } n$ ($1 \leq n$) se observă că pentru un număr x bitul schimbat în elementul următor din șirul 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 $v_1, v_2 \dots v_k$ numere prime $k \leq 20$. Să se determine câte numere mai mici ca un N dat ($N \leq 2000000000$) sunt divizibile cu cel puțin unul din cele K numere prime.

Exemplu

Pentru $n=20$, $k=4$ și șirul 2, 3, 5, 7 se va afișa 15.

Soluție:

Se folosește principiul includerii și al excluderii. Astfel numărul de numere divizibile cu cel puțin un număr dintre cele K este egal cu:

$$Nr(V_1, V_2, \dots, V_k) = \sum_{1 \leq i \leq n} V_i - \sum_{1 \leq i < j \leq n} V_i * V_j + \dots + (-1)^k \sum V_1 * V_2 * \dots * V_k$$

```

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 indexați binar reprezintă o structură de date care permite efectuarea următoarelor operații în complexitatea $O(\log N)$ pentru un vector de N elemente:

- $ADD(x, y)$ – adună la elementul de pe poziția 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).

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

- implementarea este mai ușoară;
- se folosesc fix N locații de memorie pentru stocarea arborelui;
- operațiile se pot extinde ușor pentru a rezolva aceeași problemă în mai multe dimensiuni.

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

Eficiența acestei structuri de date nu are legătură cu echilibrarea arborelui precum alte structuri de date convenționale, ci din modul de indexare a elementelor, ținând cont de reprezentarea binară (de unde și *numele de arbore indexat binar*).

Ideea de bază este că, precum un număr n poate fi exprimat ca sumă de puteri ale lui 2, așa și un interval $[1 \dots n]$ poate fi exprimat ca reuniunea unor subintervale de lungimi egale cu puteri ale lui 2.

Vom lua în continuare, ca exemplu, intervalul $[1 \dots 11]$:

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

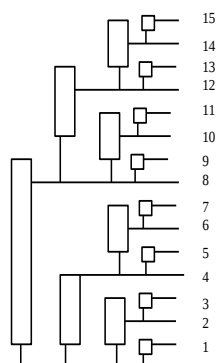
Eliminând cel mai puțin semnificativ bit de 1 din reprezentarea lui 11 în baza 2, constatăm că suma pentru intervalul $[1 \dots 11]$ poate fi calculată astfel:

$$\begin{aligned} [1 \dots 11] \ (11 = 1011_{(2)}) &= [1 \dots 10] + [11 \dots 11] \\ [1 \dots 10] \ (10 = 1010_{(2)}) &= [1 \dots 8] + [9 \dots 10] \\ [1 \dots 8] \ (8 = 1000_{(2)}) & \end{aligned}$$

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

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

$$\begin{aligned} 9 \ (9 = 1001_{(2)}) &- [9 \dots 9] \\ 10 \ (10 = 1010_{(2)}) &- [9 \dots 10] \\ 12 \ (12 = 1100_{(2)}) &- [9 \dots 12] \end{aligned}$$



Pentru a determina în $O(1)$ cel mai nesemnificativ bit de 1 al unui număr, se poate folosi expresia $(x \text{ and } (-x))$ (Pascal) / $(x \ \& \ (-x))$ (C/C++).

Sau

$(x \text{ and } (x-1)) \text{ xor } x$ (Pascal) / $(x \& (x-1)) \wedge x$ (C/C++).

Să luăm ca exemplu un vector V cu 15 elemente, care suferă modificări succesive. Aceștia îi asociem un arbore indexat binar, reprezentat sub forma unui vector AIB în care elementele rețin informații 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 funcțiilor ADD și SUM se poate realiza astfel:

<pre>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;</pre>	<pre>... #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; }</pre>
--	---

Deși această structură de date nu este la fel de flexibilă ca un arbore de intervale, ea poate fi extinsă să suporte și alte operații:

- SUMA(x, y) – suma elementelor cu poziții între x și 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 maximumul din primele x elemente.

Aceste operații pot fi implementate înlocuind operația de adunare ("+") cu cea de maxim.

- ADAUGĂ(x, y, z) – adaugă la toate elementele cu poziții între x și y valoarea z ;
- AFLĂ(x) – determină valoarea elementului de pe poziția x .

Aplicații - Arbori indexați binar

Problema – Inv (www.infoarena.ro)

Se dă un șir S de lungime n cu numere întregi. Numim o inversiune o pereche de indici (i, j) astfel încât $1 \leq i < j \leq n$ și $S[i] > S[j]$.

Cerință

Să se determine câte inversiuni sunt în șirul dat.

Date de intrare

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

Date de ieșire

În fișierul de ieșire **inv.out** se va afișa un singur număr, reprezentând restul împărțirii numărului de inversiuni din șir la valoarea 9917.

Restricții

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

Exemplu

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

Soluție:

Problema se rezolvă plecând de la numărul maxim de inversiuni dintr-un vector. Șirul **S** preluat din **inv.in** va fi sortat crescător, pentru fiecare element reținându-se și poziția inițială în **S** (normalizare). Din numărul maxim de 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 șirul **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 $[x-2^k+1, \dots, x]$. Evident ele au valori mai mici decât elementul curent și deci nu formează cu el inversiuni.

```
1 #include <cstdio>
2 #include <algorithm>
3 #include <vector>
4
5 #define ub(x) (x&(-x))
6
7 #define NMAX 100001
8 #define MOD 9917
9
10 using namespace std;
11
12 vector < pair< int,int > > V;
13 int A[NMAX];
14 int N,i,x,k;
15 long long sol;
16
17 void add(int x)
18 {
19     int i=0;
20
21     for (i=x;i<=N;i+=ub(i)) A[i]++;
22 }
23
24 int Sum(int x)
25 {
26     int i=0,S=0;
27
28     for (i=x;i>0;i-=ub(i)) S+=A[i];
29     return S;
30 }
31
32 int main()
33 {
34     freopen("inv.in","r",stdin);
35     freopen("inv.out","w",stdout);
36
37     scanf("%d",&N);
38
```

```

39     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, după evoluția celui de-al p -lea concurent, acesta va ști locul său în clasamentul format de primii p concurenți. Cunoscând pozițiile intermediare, se cere să se determine clasamentul final al competiției.

Date de intrare

Pe prima linie a fișierului de intrare **sch_i.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 ieșire

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

Restricții

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

Exemplu

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

Soluție:

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 $[x-2^k+1, \dots, x]$. Se va parcurge invers șirul din fișierul de intrare, și pentru fiecare valoare $X[i]$, se va determina cea mai mică 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 numărul 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  int A[NMAX],X[NMAX],sol[NMAX];
8  int N,i,x;
9
10 void Decrease(int x)
11 {
12     int i;
13     for(i=x;i<=N;i+=ub(i)) A[i]--;
14 }
15
16 int Sum(int poz)
17 {
18     int i=0,T=0;
19     for(i=poz; i>0; i-=ub(i)) T+=A[i];
20     return T;
21 }
22
23 int binary_search(int x)
24 {
25     int S=1,D=N,M=0,T=0,Minim=INT_MAX;
26     while(S<=D)
27     {
28         M=(S+D)/2;
29         T=Sum(M);
30         if (T==x && M<Minim) Minim=M;
31         else if (T>x) D=M-1;
32             else S=M+1;
33     }
34     return Minim;
35 }
36
37 int main()
38 {
39     freopen("schi.in","r",stdin);
40     freopen("schi.out","w",stdout);
41
42     scanf("%d",&N);
43     for (i=1; i<=N; ++i) scanf("%d",&X[i]);
44     for (i=1; i<=N; ++i) A[i]=ub(i);
45
46     for (i=N; i>=1; --i)
47     {
48         x=binary_search(X[i]);
49         sol[x]=i;
50         Decrease(x);
51     }
52
53     for (i=1;i<=N;++i) printf("%d\n",sol[i]);
54
55     return 0;
56 }

```

Problema – Order (www.infoarena.ro)

Se consideră N copii așezați în cerc și numerotați 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 numără i copii în sens trigonometric și este eliminat copilul la care se ajunge; la pasul următor număratoarea începe de la copilul care urmează după cel eliminat.

Așadar, 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 șaptelea, apoi la al patrulea pas al unsprezecelea și așa mai departe.

Cerință

Va trebui să determinați ordinea în care vor fi eliminați copiii.

Date de intrare

Fișierul de intrare order.in conține pe prima linie un număr întreg N , care reprezintă numărul de copii.

Date de ieşire

Fişierul de ieşire order.out trebuie să conţină o singură linie pe care se vor afla **N** numere distincte cuprinse între 1 şi **N** care reprezintă numerele de ordine ale copiilor în ordinea în care au fost eliminaţi.

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 reţine la fiecare pas, copiii rămaşi 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.

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

```

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

```

Probleme suplimentare:

Pe campion.edu.ro

- Concurs1:

Pe infoarena.ro:

- Datorii
- AIB
- Ben
- Evantai
- Cutii
- Distincte