**Probleme care prelucrează șiruri de numere fără a folosi vectori**

Acest tip de probleme citesc valorile șirului într-o singură variabilă, orice valoare nou citita distrugând valoarea citită la pasul anterior.

Evident, NU orice problemă se pretează acestei abordări. Spre exemplu, pentru a sorta un șir de valori, acesta trebuie OBLIGATORIU memorat într-un vector.

Multe probleme însă se pot rezolva fără vectori, eventual, acolo unde cerința impune, e nevoie de memorarea ultimelor două sau trei valori introduse.

Iată câteva tipuri de enunțuri, cu tot cu implementarea lor:

A) Se citește n, se citesc apoi n valori. Să se ... (va fi luată în considerare fiecare dintre valorile introduse)

cin>>n; //n=câte valori se vor introduce

for(i=1;i<=n;i++)

{

cin>>x;//în variab. x citim valoare cu valoare

..prelucrăm x...

}

B) Se citesc valori în mod repetat, până la îndeplinirea unei condiții (gen până se bagă un 0, sau un nr. negativ). Să se ... (va fi luată în considerare fiecare dintre valorile introduse mai puțin 0ul final)

cin>>x;

while(x NU indeplinește condiția de terminare)

{

..prelucrăm x...

cin>>x;

}

A1) Se citește n, se citesc apoi n valori. Să se ... ținând cont de toate **perechile** de valori vecine în timpul introducerii. (se garantează că n>=2)

cin>>n; //n=câte valori se vor introduce

cin>>x;//se citește separat prima valoare

for(i=2;i<=n;i++)

{

cin>>y;//în variab. y citim următoarea valoare

..prelucrăm perechea (x,y)...

x=y;

}

B1) Se citesc valori în mod repetat, până la îndeplinirea unei condiții (gen până se bagă un 0, sau un nr. negativ). Să se ... ținând cont de toate perechile de valori vecine în timpul introducerii (obs: se garantează că se dau cel puțin 2 valori)

cin>>x>>y;

while(y NU indeplinește condiția de terminare)

{

..prelucrăm perechea (x,y)...

x=y;

cin>>y;

}

A2) Se citește n, se citesc apoi n valori. Să se ... ținând cont de toate **tripletele** de valori vecine în timpul introducerii. Obs: se garantează că n>=3

cin>>n; //n=câte valori se vor introduce

cin>>x>>y;//se citesc separat primele doua valori

for(i=3;i<=n;i++)

{

cin>>z;//în variab. z citim următoarea valoare

..prelucrăm tripletul (x,y,z)...

x=y;

y=z;

}

B2) Se citesc valori în mod repetat, până la îndeplinirea unei condiții (gen până se bagă un 0, sau un nr. negativ). Să se ... ținând cont de toate tripletele de valori vecine în timpul introducerii. Obs: se garantează că se dau cel puțin 3 valori

cin>>x>>y>>z;

while(z NU indeplinește condiția de terminare)

{

..prelucrăm tripletul (x,y,z)...

x=y;

y=z;

cin>>z;

}

**Descompunerea în factori primi**

Se face în mod asemănător celei de la matematică:

Ex: n=2100

|  |  |
| --- | --- |
| 2100 | **2** |
| 1050 | **2** |
| 525 | 2 |
| 525 | **3** |
| 175 | 3 |
| 175 | 4 |
| 175 | **5** |
| 35 | **5** |
| 7 | 5 |
| 7 | 6 |
| 7 | 7 |
| **1**  **STOP** | 7 |

Algoritmul presupune următoarele:

- luăm divizori începând de la valoarea d=2.

Pe o repetitivă "mare" (exterioară), ne ocupăm de fiecare divizor în parte, astfel:

- dacă numărul dat se divide la d, îl divizăm până nu se mai poate și numărăm câte împărțiri s-au efectuat (într-o variabilă p).

- după fiecare testare a unui divizor d, dacă acesta a fost un divizor de‑adevăratelea, afișăm informațiile despre el : valoarea (d) și puterea (p)

Iată algoritmul (mai puțin eficient):

fie n = nr. de descompus

d=2;

while(n>1)

{

p=0;//în p numărăm puterea la care apare factorul d în n

while(n%d==0)

{

n/=d;

p++;

}

if(p)

cout<<"Factor = "<<d<<" putere = "<<p<<"\n";

d++;

}

Obs: În urma algoritmului, valoarea lui n se distruge.

Ordinul de complexitate: O(n) - de fapt este pentru cazul cel mai defavorabil.

Ordinul de complexitate este variabil, în funcție de numărul pe care îl descompunem.

Ex: Dacă n=128 - se fac doar 7 pași.

The worst case este acela în care numărul este prim. Practic, în acel caz ni se iau

toți divizorii până la n, terminându-se algoritmul doar când n se împarte la el însuși.

Putem o țâră optimiza: în momentul în care divizorul d devine mai mare decât , pe ideea de la numerele prime, înseamnă că și numărul nostru, n (în fine, ceea ce-a rămas în n) este prim și putem să ne oprim din verificări.

La final, datorită acestei condiții, s-ar putea ca în n să NU fie 1, ci să rămânem chiar cu numărul prim.

Din acest motiv mai facem o verificare suplimentară la final.

Algoritmul astfel obținut va avea în cel mai rău caz ordinul de complexitate O()

Iote-l:

d=2;

while(d\*d<=n)

{

p=0;//în p numărăm puterea la care apare factorul d în n

while(n%d==0)

{

n/=d;

p++;

}

if(p)

cout<<"Factor = "<<d<<" putere = "<<p<<"\n";

d++;

}

if(n!=1) cout<<"Factor = "<<n<<" putere = 1\n";

Dacă vrem să ne rupem în skeme și să folosim for-ul așa cum este el în C++, cu toate avantajele sale:

for(d=2;d\*d<=n; d++)

{

for(p=0;n%d==0;p++)

n/=d;

if(p)

cout<<"Factor = "<<d<<" putere = "<<p<<"\n";

}

if(n!=1) cout<<"Factor = "<<n<<" putere = 1\n";

Ex: n=28

|  |  |
| --- | --- |
| 28 | **2** |
| 14 | **2** |
| 7 | 2 |
| STOP | **3** |

ne-am oprit pt. că d\*d>n (d=3 și n=7) ⇒ ceea ce a rămas în n este factorul prim la puterea 1

Într-adevăr 28=22⋅71

Obs: Numărul total de divizori ai unui număr natural se poate obține ca produsul tuturor puterilor factorilor săi primi adunate cu 1.

Ex: 12 = 2**2** ⋅ 3**1** Deci puterile sunt **2** și **1**. Produsul lor adunate cu 1: (**2**+1)⋅(**1**+1)=6

(Explicație: orice divizor al lui 12, dată fiind descompunerea sa în factori primi, este de forma: div=2x⋅3y unde 0≤x≤2 iar 0≤y≤1 deci câte 3 valori posibile pt. x respectiv câte 2 posibile pt. y, deci în total 2\*3=6

Cei 6 divizori sunt: 20⋅30=1, 20⋅31=3, 21⋅30=2, 21⋅31=6, 22⋅30=4, 22⋅31=12

**Șirul lui Fibonacci**

Este un șir definit astfel (recurent):

- primii doi termeni sunt egali cu 0 1 (în anumite probleme se consideră 1 1)

- orice alt termen este egal cu suma celor 2 de dinaintea sa.

Iată primii 15 termeni ai săi:

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

El poate fi foarte ușor generat "jonglând" cu 3 variabile în care reținem:

t1, t2 = ultimii doi termeni calculați

tc = termenul curent, care la fiecare pas se obține ca suma celor 2 de dinainte.

Iată cum se pot genera și afișa primii n termeni:

t1=0;

t2=1;

for(i=1;i<=n;i++)

{

cout<<t1<<" ";

tc=t1+t2;

t1=t2;

t2=tc;

}

Obs: Pe tipul int merg calculați primii 46 de termeni.

Obs: Raportul t2/t1, când n→ ∞ poartă numele de "raportul de aur" sau "phi" și are valoare aproximativă de 1,61803...

**Determinarea minimului dintr-un șir de valori**

A) Se ia o variabilă "min" care se inițializează cu:

- cu prima dintre valorile dintre care calculăm minimul

- cu o valoare SIGUR mai mare decât cele dintre care calculăm minimul

- cu cea mai mare valoare acceptată de int, și anume constanta INT\_MAX

(care e definită în biblioteca <climits>) - în cazul în care lucrăm pe int.

(INT\_MAX este de fapt egală cu 2147483647)

B) se compară rând pe rând valorile introduse (fie o astfel de valoare "x") și se actualizează min dacă este cazul:

if(x<min)

min=x;

**Determinarea celor mai mici două minime printr-o singură parcurgere**

A) Se procedează analog, considerând două variabile min1 și min2, în care vom determina, în ordine, cele mai mici două minime (min1 < min2)

Inițializăm ca mai înainte, preferabil cu ceva f. mare:

min1 = INT\_MAX;

min2 = INT\_MAX;

B) se compară rând pe rând valorile introduse prin situarea lor între min1 și min2 respectiv actualizările convenabile:

if(x<min1)

{

min2=min1;

min1=x;

}

else

if(x<min2)

min2=x;

**Afișare cu format - biblioteca <iomanip>**

Se folosește preponderent în următoarele cazuri:

- avem de afișat numere naturale pe care dorim să le aliniem la dreapta, pe un câmp total de "p" poziții. Pentru asta, folosim:

**cout<<setw(p)<<kestii\_de\_afișat;**

- avem de afișat numere reale pe care dorim să le afișăm în format fix (fără "e") cu un număr de exact **z** zecimale:

**cout<<fixed<<setprecision(z)<<kestii\_de\_afișat;**

Obs: putem combina setw cu fixed.

Obs: afișarea cu setprecision produce, dacă este cazul, rotunjire la ultima zecimală afișată.

Ex:

**cout<<fixed<<setprecision(2)<<1.462; //afișează 1.46**

**cout<<fixed<<setprecision(2)<<1.469; //afișează 1.47**

**cout<<fixed<<setprecision(2)<<1.465; //afișează 1.47**

**cout<<fixed<<setprecision(2)<<1.999; //afișează 2.00**

Obs: Dacă avem o variabilă reală **x** pe care vrem s-o trunchiem la un număr dat de zecimale (adică să nu se mai facă rotunjire la ultima zecimală afișată) - facem ca în exemplu, pe care l-am considerat pt. 2 zecimale, de-aia am folosit valoarea **100**. Dacă vrem la 3 zecimale, folosim 1000, etc.

**x=(int)(x\*100)/100.0;**

Obs: Numerele reale (atât float cât și double) sunt reprezentate în memoria calculatorului într-un standard numit "virgulă mobilă" (floating point) - adică numărul dat se reprezintă ca număr real în baza 2, iar rezultatul este normalizat, adică se înmulțește cu o putere a lui 2 astfel ca în fața virgulei să fie o singură cifră (care în baza 2 va fi 1).

Din nefericire trecerea între bazele de numerație 10 și 2 face ca destul de multe valori care în baza 10 au reprezentare finită (de exemplu 0.1 sau 0.15) în baza 2 să devină periodice.