Divizibilitate

În acest material ne ocupăm de problema determinării divizorilor unui număr și de găsirea soluțiilor pentru diverse probleme legate de divizori.

În acest material considerăm că divizorii unui număr natural `n` sunt acele numere naturale pentru care se obține restul `0` la împărțirea lui `n` la ele. Deducem că nivizorii sunt numere mai mici sau egale cu `n`. Așadar, prima soluție este să încercăm pe rând toate numerele de la `1` la `n` ca posibili divizori. Putem face asta ușor, având la dispoziție în limbaj atât structură repetitivă cât și operator de aflare a restului.

```
for (i=1;i<=n;i++)
    if (n%i == 0)
        cout<<i<<" ";
```

Fiind preocupați de a scrie programe care să ruleze cât mai repede, ne punem problema să reducem numărul de calcule. Observăm astfel că după `n/2` nu mai este alt divizor decât `n`. Reducem astfel la jumătate numărul de valori de încercat.

```
for (i=1;i<=n/2;i++)
    if (n%i == 0)
        cout<<i<<" ";
cout<<n;
```

Această optimizare face ca lucrurile să meargă doar puțin mai repede în practică dar nu reprezintă o îmbunătățire a complexității teoretice. De exemplu, pentru $n=10^9$, cu `n/2` rămânem în marja a câteva sute de mii de instrucțiuni.

Prezentăm în continuare observațiile care duc la obținerea unei soluții cu adevărat vaoroasă, timpul de executare la care vom ajunge fiind de ordinul $\sqrt{n}$.

Să analiăm divizorii lui `40`, scriși în ordine crescătoare: `1`, `2`, `4`, `5`, `8`, `10`, `20`, `40`. Observăm că produsul dintre primul și ultimul divizor este `40`, la fel produsul dintre al doilea și penultimul divizor. În general produsul divizorilor egal depărtați de extremitățile șirului divizorilor este acelasi.

Observația de mai sus este valabilă pentru orice număr natural `n` întrucât dacă `i` este divizor al lui `n` atunci și `n/i` este divizor al lui `n` și pe măsură ce `i` crește, `n/i` scade. Deci divizorii unui număr pot fi organizați în perechi cu proprietatea că produsul membrilor aceleiași perechi este chiar `n`.

Analizând cele de mai sus, deducem că nu se poate ca ambii membrii ai unei perechi să fie mai mari decât $\sqrt{n}$. Cum arătăm asta ? Prin reducere la absurd, iată:

Fie `d1` și `d2` doi divizori ai lui `n` cu proprietatea:

`d1*d2 = n;`

`d1 >` $\sqrt{n}$`;`

`d2 >` $\sqrt{n}$`;`

Din ultimele două relații, numerele fiind pozitive, înmulțind membru cu membru, rezultă că `d1*d2>`$\sqrt{n} * \sqrt{n}$, adică `d1*d2>n`. Ori noi avem de a face cu o pereche de divizori cu produsul `n`, contradicție.

Pe de altă parte, dacă `d1<`$\sqrt{n}$ resultă că `d2<`$\sqrt{n}$ (cu un raționament ca acela de mai sus) deduce că nu pot fi ambii divizori strict mai mici decât $\sqrt{n}$ (și cu produsul `n`).

Putem scrie un cod care variază divizorul posibil până la $\sqrt{n}$ și pentru un `i` găsit ca divizor, automat `n/i` este și el divizor. În acest fel găsim cu `i` membrul cel mic al perechii și `n/i` este membrul cel mare.

Divizorii fiind pereche înseamnă că numărul lor este par și în general așa este. Avem un caz particular de la această regulă: la pătratele perfecte numărul de divizori este impar, și doar la ele. Justificarea vine dacă ne gândim la divizorul $\sqrt{n}$ (dacă `n` este pătrat perfect această valoare este divizor al său). Dar dacă `i`=$\sqrt{n}$ este divizor, `n/i = n/`$\sqrt{n}$ este și el divizor, dar această valoare este egală tot cu $\sqrt{n}$. Așadar în cazul pătratelor perfecte perechea care conține radicalul are ambii membri egali. Și de aici justificarea. De exemplu, la `36`:

i	n/i
1	36
2	18
3	12
4	9
6	6

```
for (i=1;i<=sqrt(n);i++)
    if (n%i == 0){
        cout<<i<<" ";
        if (i!=n/i)
            cout<<n/i<<" ";
    }
```

Trebuie acordată atenție celui de-al doilea `if`. Rolul său este să rezolve cazul particular al pătratelor perfecte. Atenție la plasarea lui, trebuie între acoladele primului `if` întrucât ne punem problema perechii `n/i` a lui `i` doar dacă `i` este divizor.

În practică se evită folosirea funcției radical întrucât are două mici dezavantaje: consumă ceva timp de calcul și implică lucru cu numere reale.

Alte două variante de a scrie condiția de la `for` sunt:

`i*i<=n`	Echivalența o deducem ușor aplicând radical ambilor membri.
`i<=n/i`	Justificarea acestei relații o facem pornind de la cea anterioară și împărțind la `i` în ambii membri.

Prima dintre cele două noi variante de mai sus are dezavantajul că implică un rezultat intermediar mai mare decât valorile ce intervin în calcule (`i*i`), apărând risc de overflow. Pesonal recomand folosirea ultimei variante, `i<=n/i`, pe care o veți întâlni în majoritatea exemplelor din acest material. O altă variantă bună și des întâlnită este de a calcula radicalul (cu funcția `sqrt` din bibliotecă - `cmath`) și de a-l stoca într-o variabilă `r` de la început iar condiția ce se scrie apoi în `for` va fi: `i<=r`.

Revenind la analiza timpului de executare pentru exemplul cu 10^9, observăm acum că acesta ajunge la sub `34000` de pași ($\sqrt{10^9}$), diferență colosală. Deci complexitatea teoretică în timp a algoritmului bazat pe optimizarea prezentată este de ordinul $\sqrt{n}$.

Iată alte rezultate importante care permit reducerea timpului de executare în problemele cu divizori.

- ***Verificarea dacă un număr este prim necesită căutarea vreunui divizor doar între 2 și radicalul numărului.*** Justificarea vine din observația că divizorii sunt perechi și în fiecare pereche unul dintre membri este mai mic decât radicalul iar celălalt este mai mare. Deci dacă e să existe divizor mai mare decât radicalul, va exista și perechea lui care este mai mic decât radicalul.

- ***Un număr poate avea cel mult un factor prim mai mare decât radicalul său.*** Justificarea se face prin reducere la absurd cu raționamentul de mai sus. Dacă am avea doi factori primi mai mari ca radicalul, produsul lor ar fi mai mare decât numărul. Trebuie să fim atenți că putem avea mai mulți divizori mai mari decât radicalul, dar aici vorbim de divizori primi. Divizorii fiind primi sunt și numere prime între ele, deci nu au divizori comuni și atunci produsul lor ar trebui să dividă și el numărul (produsul lor apare de fapt în descompunerea numărului în factori primi). Dar dacă ambii ar fi mai mari ca radical, produsul lor, care ar fi deci mai mare decât numărul, nu ar putea să dividă numărul dat și am ajunge la contradicție.
- ***Cel mai mic divizor propriu al unui număr (divizor propriu = diferit de 1 și de numărul însuși) este un număr prim.*** Și acest lucru se poate ușor justifica prin reducere la absurd: dacă `d` divide pe `n` și nu este prim rezultă că avem `d1` un divizor propriu al lui `d`. Adică `d1` divide pe `d` și cum divizibilitatea este tranzitivă, avem și că `d1` divide pe `n`. Dar `d1`, fiind divizor al lui `d`, este mai mic decât `d`. Deci `d` nu este cel mai mic divizor al lui `n`, așa cum am presupus.

Probleme rezolvare

Ca și în alte capitole, multe dintre aplicațiile de aici trebuie considerate semiteoretice, fiind cerințe clasice, utile la task-uri mai mari.

1. Dat fiind un număr natural `n`, să se afișeze `Da` sau `Nu` după cum acesta este sau nu prim.

Rezolvare

Este o problemă de verificare și, așa cum am arătat în suportul teoretic de mai sus, dacă găsim vreun divizor cuprins între `2` și radicalul numărului dat, atunci concluzionăm că data de intrare nu este număr prim. Din start afișăm `Nu` dacă este vorba de valorile `0` sau `1` întrucât acestea nu reprezintă numere prime iar dacă am aplica pentru ele aplica algoritmul descris sus ar reieși că sunt, negăsind divizori.

```
cin>>n;
if (n <= 1) {
    cout<<"Nu";
    return 0;
}
d = 0;
for(i=2; i<=n/i; i++)
    if(n%i==0)
        d++;
if(d==0)
    cout<<"Da";
else
    cout<<"Nu";
```

2. Dat fiind un număr `n`, cel mult egal cu $2*10^9$, să se determine suma divizorilor săi (pbinfo.ro, #376).

Rezolvare

```
#include <iostream>
using namespace std;
long long suma;
int n, i;
int main () {
    cin>>n;
    suma = 0;
    for (i=1;i<=n/i;i++)
        if (n % i == 0) {
            suma = suma + i + n/i;
```

```
            if (i == n/i)
                suma = suma - i;
        }
    cout<<suma;
    return 0;
}
```

Pentru a obține un timp de calcul foarte mic valoarea lui `n` necesita un algoritm cu timp de executare de ordin $\sqrt{n}$. Am modificat puțin abordarea la al doilea `if`, noi anterior am adaugat oricum doi divizori, deci cu acest `if` il scădem pe cel adunat în plus în cazul în care numărul dat este pătrat perfect. Altă observație: valoarea dată fiind aproape de limita maximă de la tipul `int`, s-a impus folosirea tipului `long long` pentru `suma`.

3. Dat fiind un număr `n`, să se determine numărul de divizori pari ai săi. Valoarea dată poate fi până la $2*10^9$ (pbinfo.ro, #388).

Rezolvare

```
#include <iostream>
using namespace std;
int n, i, sol;
int main()
{
    cin>>n;
    for (i=1; i<=n/i; i++) {
        if (n % i == 0) {
            if (i%2 == 0)
                sol++;
            if (n/i != i && n/i % 2 == 0)
                sol++;
        }
    }
    cout<<sol;
}
```

Aici doresc să ne concentrăm atenția pe primul `if`. Fiind vorba de divizori pari, dacă nu suntem atenți vom căuta divizorii pari de până la radical (adică să scriem primul `if` asa: `if(n%i == 0 && i%2 == 0)`, de exemplu). Se vede din codul de mai sus că nu facem asta ci găsim divizori oarecare până la radical și tratăm separat, în interiorul `if`-ului paritatea atât pentru `i` cât și pentru `n/i`. Un exemplu care lămurește lucrurile este: pentru `n=12`, găsim `3` ca divizor până în radical, care nu este par, dar perechea lui, `4`, este par și trebuie contorizat. Însă, dacă nu intram în primul `if` pentru `3` nu l-am fi obținut nici pe `4`.

4. Dar fiind un număr `n`, natural, cuprins între `2` și 10^9, să se determine două numere `a` și `b`, cu diferența în modul minimă și al căror produs să fie `n`. De exemplu, dacă numărul `n` este `40`, valorile determinate sunt `5` și `8` (pbinfo.ro, #377).

Rezolvare

Observăm că în soluție este vorba despre una dintre perechile de divizori, `i` și `n/i` despre care am vorbit. Astfel, căutăm divizorii până la radical și pe noi ne interesează de fapt ultimul găsit cu perechea lui (cu cât el este mai mare cu atât perechea este mai mic).

```
#include <iostream>
using namespace std;
int n;
int a, b, i, j;
int main () {
    cin>>n;
    for (i=1; i<=n/i; i++)
        if (n%i == 0) {
```

```
            a = i;
            b = n/i;
        }
    cout << a <<" "<< b;
    return 0;
}
```

5. Pentru un număr natural dat n, cel mult egal cu $2*10^9$, se cere să verificăm dacă poate fi scris ca produs a două numere prime distincte, caz în care vom afișa DA iar în caz contrar vom afișa NU (pbinfo.ro, #379).

Rezolvare

Observația esențială este că primul divizor propriu găsit pentru un număr, la o căutare în ordine crescătoare, este prim (lucru justificat anterior în material). Așadar odată cu găsirea primului divizor d rămâne doar să verificăm cum este n/d. Dacă acesta este prim (și diferit de d, conform enunțului) rezultatul final este DA, altfel este NU. Nu are sens să continuăm căutarea pentru că sigur unul dintre divizorii scrierii cerute trebuie să fie chiar primul găsit (de fapt problema este echivalentă cu a verifica dacă numărul are exact doi divizori proprii și nu este pătrat perfect).

```
#include<iostream>
using namespace std;
int n,d,k=0, e, nrd, i;
int main(){
    cin>>n;
    for(d=2; d<=n/d; d++)
        /// cel mai mic divizor al unui numar este prim
        if(n%d==0) {
            /// dupa ce gasesc cel mai mic divizor d, mai verific doar daca
            /// n/d este prim

            e = n/d;
            nrd = 0;
            for (i=2; i<=e/i; i++)
                if (e%i == 0)
                    nrd++;
            if (nrd == 0 && e != d)
                cout<<"DA";
            else
                cout<<"NU";
            return 0;
        }
    cout<<"NU";
}
```

Instrucțiunea cout de la final este pentru situația în care numărul dar este prim, caz în care forul nu ar detecta niciun divizor.

6. **Factorizarea unui număr.** Dat fiind un număr n, maxim egal cu $2*10^9$, să se afișeze descompunerea sa în factori primi. Factorii primi se afișează în ordine crescătoare urmați pe același rând de exponentul la care apar în descompunere.

Exemplu

Date de intrare	Date de ieșire
90	2 1 3 2

	5 1

Rezolvare

Soluția propusă se bazează pe faptul că odată găsit un factor prim `d`, pentru a afla la ce putere apare acesta în număr putem folosi codul:

```
e = 0;
while (n%d == 0) {
   e++;
   n /= d;
}
if (e != 0)
   cout<<d<<" "<<e<<"\n";
```

Adică vom consuma factorul `d` din număr și prin contorizarea repetițiilor obținem exponentul.

Pentru a găsi primul factor prim căutăm de fapt primul divizor (despre care am arătat că este prim). Obervăm un lucru foarte interesant: după consumarea din număr a factorului găsit, următorul factor prim este iarăși primul divizor al numărului rămas etc. Deci ce avem defăcut este să luăm pe rând numerele naturale începând cu `2` și pentru cele găsite drept divizori aplicăm codul de mai sus. Altă justificare a faptului că vom găsi în acest fel doar divizori numere prime este aceea că noi consumăm un divizor din număr, deci la întâlnirea următorului divizor (care este mai mare decât cei găsiți anterior), nu e posibil ca acesta să mai aibă și alți divizori pentru că ar fi fost găsiți înainte și consumați.

Algoritmul de încercare pe rând a tuturor numerelor ca posibili divizori primi se oprește când `n` devine `1`.

```
#include <iostream>
using namespace std;
int n, d, e;
int main () {
    cin>>n;
    for (d = 2; n!=1; d++) {
        e = 0;
        while (n%d == 0) {
            e++;
            n /= d;
        }
        if (e != 0) {
            cout<<d<<" "<<e<<"\n";
        }
    }
    return 0;
}
```

În cazul în care `n` este un număr prim timpul de executare este de ordin `n` (singurul divizor care se va găsi este chiar `n` și contorul `d` merge din `1` în `1` până la `n`). Așadar algoritmul nostru ar fi lent. Aceeași problemă apare când numărul are un singur divizor prim foarte mare, fără să ca numărul însuși să fie neapărat prim. Dar noi am arătat că un număr poate avea maxim un factor prim mai mare decât radicalul său. Optimizarea vine din observația că dacă aplicăm codul de mai sus cel mult până când `d` ajunge la $\sqrt{n}$ și dacă totuși `n` nu a ajuns `1`, deoarece noi am consumat din el factorii primi găsiți, atunci valoarea rămasă a lui `n` este chiar singurul său factor prim mai mare ca radicalul său. Obținem astfel un algoritm cu timp de executare de ordin $\sqrt{n}$.

```
#include <iostream>
using namespace std;
int n, d, e;
int main () {
    cin>>n;
```

```
    for (d = 2; n!=1 && d <= n/d; d++) {
        e = 0;
        while (n%d == 0) {
            e++;
            n /= d;
        }
        if (e != 0) {
            cout<<d<<" "<<e<<"\n";
        }
    }
    if (n!=1) {
        cout<<n<<" "<<1<<"\n";
    }
    return 0;
}
```

7. Pentru un număr natural dar, notat cu `n`, maxim egal cu $2*10^9$, să se determine cel mai mic număr prim strict mai mare decât `n`. De exemplu, pentru numărul `20` ar trebui obținută valoarea `23`.

Rezolvare

Strategia abordată este una brut, care iterează crescător valorile mai mari ca `n` și la prima care trece testul de primalitate ne oprim.

```
#include <iostream>
using namespace std;
int n, i, j, k;
int main () {
    cin>>n;
    for (i=n+1; ;i++) {
        int divizori = 0;
        for (j=2; j<=i/j; j++)
            if (i%j == 0) {
                divizori++;
            }
        if (divizori == 0)
            break;
    }
    cout<<i;
    return 0;
}
```

Remarcă

Pentru a testa pentru **mai multe numere** dintr-un set dat dacă sunt sau nu prime, putem face pentru fiecare dintre ele verificarea așa cum am arătat mai sus, obținând timp de calcul de ordinul `numar_valori*` $\sqrt{valoarea\ maximă}$.

La capitolul dedicat vectorilor de frecvență vom studia abordarea numită "***Ciurul lui Eratostene***" care ne oferă un timp de calcul mai bun.