**Cel mai mare divizor comun (CMMDC) a două numere în C++**

Se dau două numere naturale a și b și se cere să aflăm CMMDC-ul celor două numere (**c**el **m**ai **m**are **d**ivizor **c**omun).

**Exemplu:** pentru a = 14, b = 6, cmmdc(a, b) = 2.

**Ce este CMMDC-ul a două numere?**

CMMDC-ul, sau Cel Mai Mare Divizor Comun a două numere naturale a și b este un alt număr natural d cu proprietățile:

* d | a și d | b (d divide a și d divide b);
* d este cel mai mare număr posibil.

Cmmdc-ul a două numere poate avea diverse notații: cmmdc(a, b), gcd(a, b) — *greatest common divisor*, sau chiar mai scurt, (a, b).

**Determinarea celui mai mare divizor comun în C++**

Există câteva metode de a determina cel mai mare divizor comun a două numere naturale nenule a și b. Vom prezenta două variante, ambele bazate pe același algoritm, **algoritmul lui Euclid**.

**Metoda 1. Algoritmul lui Euclid cu scăderi**

Algoritmul lui Euclid cu scăderi este o metodă mai puțin eficientă, însă mai ușor de înțeles. Acest algoritm se bazează pe principiul următor: dacă d | a și d | b, atunci d | a - b (unde a ≥ b) — dacă d divide ambele numere, atunci divide și diferența lor.

Iată algoritmul propriu zis:

* cât timp numerele sunt diferite, se scade numărul mai mic din cel mai mare;
* când numerele devin egale, atunci CMMDC-ul lor este chiar numărul lor.

Iată implementarea algoritmului în C++:

#include <iostream>

using namespace std;

int main()

{

//Declarăm și citim cele două numere

int a, b;

cin >> a >> b;

while(a != b) {

if(a > b) a = a - b;

else b = b - a;

}

cout << "cmmdc-ul este " << a;

return 0;

}

**Metoda 2. Algoritmul lui Euclid clasic (cu împărțiri)**

Algoritmul lui Euclid cu împărțiri este mai dificil de înțeles, însă este mai eficient (acesta este algoritmul recomandat pentru rezolvarea problemelor).

Pe scurt, observăm că în codul anterior facem scăderi repetate. Aceste scăderi se pot înlocui cu împărțiri, astfel că algoritmul este următorul:

* cât timp b este diferit de 0, a primește b, iar b primește restul împărțirii lui a la b;
* răspunsul în final se află în a.

#include <iostream>

using namespace std;

int main()

{

//Declarăm și citim cele două numere

int a, b;

cin >> a >> b;

while(b != 0) {

int r = a % b; //Restul împărțirii lui a la b

a = b;

b = r;

}

cout << "cmmdc-ul este " << a;

return 0;

}

**Parcurgerea divizorilor unui număr**

Ne propunem, înainte de toate, să înțelegem cum facem rost de fiecare divizor în parte înainte de a aplica diferite operații cu acestea.

Vom lua pe rând fiecare metodă în parte, de la algoritmul naiv la cel eficient.

**Algoritmul naiv**

Ştim clar că pentru un număr natural n, orice divizor d aparține [1; n]. Astfel, algoritmul naiv se bazează pe parcurgerea tuturor numerelor de la 1 la n şi verificarea n % d == 0:

//Parcurgere//pentru mai multe detalii: https://100din100.netlify.app

for(int d = 1; d <= n; d++) {

if(n % d == 0) { //am găsit un divizor//pentru mai multe detalii: https://100din100.netlify.app

cout<<d<<" ";

}

}

Este evident că pentru valori mari ale lui n, de exemplu 1.000.000.000, programul iese din timp. Aşadar, ce alte îmbunătățiri putem face?

**Algoritmul eficient**

Observăm că pentru fiecare divizor d ≤ sqrt(n), există o pereche a sa, n / d ≥ sqrt(n). Aşadar în loc să parcurgem toate numerele din [1; n], parcurgem doar de la 1 la sqrt(n) şi efectuăm operația dorită atât pe d, cât şi pe n / d.

//Parcurgere//pentru mai multe detalii: https://100din100.netlify.app

int d;

for(d = 1; d \* d < n; d++) {

if(n % d == 0) { //am găsit doi divizori//pentru mai multe detalii:

cout<<d<<" "<<n / d<<" ";

}

}

if(d \* d == n) { //Numărul este pătrat perfect, prelucrăm şi radicalul.//pentru mai multe detalii:

cout<<d<<" ";

}