Calculul celui mai mare divizor comun pentru două numere, calculul termenilor unui șir recurent

În acest material vom studia modul de determinare a celui mai mare divizor comun pentru două numere, lucru cu șiruri în care un termen este exprimat în funcție de alți termen și legărturile care există între aceste noțiuni.

Determinarea celui mai mare divizor comun pentru două numere.

Pentru soluțiile ce le vom prezenta presupunem că datele de intrare sunt două numere naturale nenule. Pentru cazul în care unul dintre numere este nul, rezultatul este chiar celălalt număr iar dacă ambele numere sunt nenule nu se definește rezultatul. Din start subliniem ca fiind mai puțin practică soluția bazată pe modul de calcul cunoscut la matematică (se descompun numerele în factori primi și se află cei comuni la puterea ce mai mică). Motivul este efortul de implementare dar și faptul că nu se obține cel mai bun timp de calcul posibil.

Iată mai departe abordări mai ușor de codat și cu diverse niveluri de complexitate în timp.

Soluția 1

Așa cum spune și numele, ne interesează un divizor comun și valoarea acestuia să fie cât mai mare. Vom încerca pe rând toți candidații la soluție. Aceștia sunt valori cuprinse între `1` și cel mai mic dintre cele două numere. În algoritmul de mai jos vom testa acești candidați în ordine descrescătoare, oprindu-ne la primul divizor comun întâlnit.

```
#include <iostream>
using namespace std;
int a, b, m, d;
int main () {
    cin>>a>>b;
    if (a < b)
        m = a;
    else
        m = b;
    for(d = m; d>=1; d--)
        if (a%d == 0 && b%d == 0)
            break;
    cout<<d;
    return 0;
}
```

Observăm că, datele de intrare fiind naturale nenule, vom avea mereu soluție, fie ea și `1`, în cazul valorilor de la intrare prime între ele.

Cazul cel mai defavorabil este când valorile date sunt mari și cu cmmdc mic, numărul de repetări fiind de ordinul valorii minime.

Soluția 2 (scăderi repetate)

Pornim de la următoarea observație: Dacă sunt egale cele două numere, rezultatul va fi valoarea lor comună. Altfel, fie d un divizor comun al numerelor a și b. Vor exista în aces caz două numere a' și b' așa încât:

a = d * a'

b = d * b'

Fără a restrânge generalitatea, vom presupune că a > b. În acest caz obținem

a - b = d * a' - d * b' = d*(a' - b')

Deducem de aici că d va divide valoarea a-b. Se conturează următoarea strategie: La fiecare pas, înlocuim numărul mai mare cu diferența celor două, continuând cu valoarea nou obținută și cu numărul mai mic. Pe de o parte d rămâne divizor al ambelor valori, pe de altă parte reducem valorile pe care le testăm (maximul celor două scade). Continuăm procedeul până ce ajung egale valorile de testat, caz în care s-a obținut rezultatul.

Exemplu

a	b
165	45
120	45
75	45
30	45
30	15
15	15

Algoritmul care implementează cele prezentate:

```
#include <iostream>
using namespace std;
int a, b, m, d;
int main () {
    cin>>a>>b;
    while (a!=b)
        if (a > b)
            a -= b;
        else
            b -= a;
    cout<<a; /// sau cout<<b;
    return 0;
}
```

Am observant că aceasta este varianta pe care majoritatea elevilor o reține cel mai ușor. Totuși, din punct de vedere al timpului de executare, nu este cea optimă. Un caz defavorabil nu este greu de găsit: a – o valoare foarte mare, b – o valoare foarte mică (chiar 1), caz în care numărul de scăderi este de ordinul valorii maxime.

Soluția 3 (algoritmul lui Euclid)

Să analizăm puțin exemplul prezentat la soluția anterioară, a = 165 și b = 45. Prima valoare este mai mare ca a doua și rămâne așa la primii trei pași, după care ea ajunge mai mică (30). Noi am obținut asta cu trei operatii de scădere. Observăm că valoarea la care ajunge a când devine mai mic decât b este chiar restul împărțirii celor două valori inițiale (iar numărul de scăderi după care s-a obținut este câtul împărțirii celor

două valori inițiale). Profităm de faptul că limbajul dispune de operatorul de rest (`%`) și obținem astfel dintr-un singur pas valoarea în care trece `a` (evitând deci "cât" scăderi).

Această observație o vom pune în practică în felul următor: împărțim `a` la `b`, păstrând restul, iar la pasul următor reluăm pentru `b` și rest. Obținem, pentru exemplul prezentat, următorul șir.

`165, 45, 30, 15, 0.`

a	b	rest
165	45	30
45	30	15
30	15	0
15	0	Stop

Ultimul rest diferit de `0` este soluția. Implementarea o facem ca în cazul problemelor în care valoarea curentă se calculează în funcție de cele două anterioare, urmând ca apoi să actualizăm pe cele ce vor deveni "ultimele două" la pasul următor.

```
#include <iostream>
using namespace std;
int a, b, r;
int main () {
    cin>>a>>b;
    while (b!=0) {
        r = a%b;
        a = b;
        b = r;
    }
    cout<<a;
    return 0;
}
```

Iată câteva observații pe baza algoritmului de mai sus:

- Întrucât prima operație pe care o facem în blocul repetiției este o împărțire la `b`, reținem condiția de oprire ca fiind `b!=0`, altfel nu am putea împărți la `0`.
- După modul de actualizare a variabilelor, rezultatul rămâne în `a`.
- Dacă la început avem `a > b`, acest lucru se păstrează pe parcurs întrucât la pasul următor avem numere `b` și un rest la împărțirea la `b`, care este între `0` și `b-1`, deci mai mic decât `b`.
- Dacă inițial avem `a < b`, algoritmul de mai sus ne ajută prin faptul că după primul pas el interschimbă automat pe `a` și `b` (`a % b` este chiar `a` în acest caz).

Este o strânsă legătură între numărul de pași pe care îi face algoritmul lui Euclid și modul de creștere a termenilor din șirul lui Fibonacci. Practic, cazul cel mai defavorabil este atunci când numerela inițiale `a` și `b` sunt termeni Fibonacii consecutivi (algoritmul lui Euclid și soluția cu scăderi repetate fac în acest caz același număr de pași). Se poate arăta că, pe cazul cel mai defavorabil, numărul de pași pe care ii face algoritmul lui Euclid deste de ordinul $\log_x$`vaoarea_maxima`, unde `x` este $\frac{1+\sqrt{5}}{2}$, adică *raportul de aur* (`1,618033` aproximativ, pe noi interesându-ne faptul că această valoare este supraunitară). De exemplu, pentru oricare două numere de tip int se garantează obținerea rezultatului în `50` de pași.

Probleme rezolvate

1. Se dau două numere naturale `a` și `b`. Se cere calculul celui mai mic multiplu comun al lor.

Soluția se bazează cunoscuta relație: `a * b = cmmdc(a, b) * cmmmc(a, b)`

Așadar ne vom folosi de algoritmul lui Euclid. Se recomandă ca odată calculat cel mai mare divizor comun, obținerea rezultatului final să o scriem printr-o expresie de forma: `a / cmmdc * b`, în loc de `a * b / cmmdc` (pentru a evita riscul unei depășiri de valoare maximă a tipului de date în momentul înmulțirii).

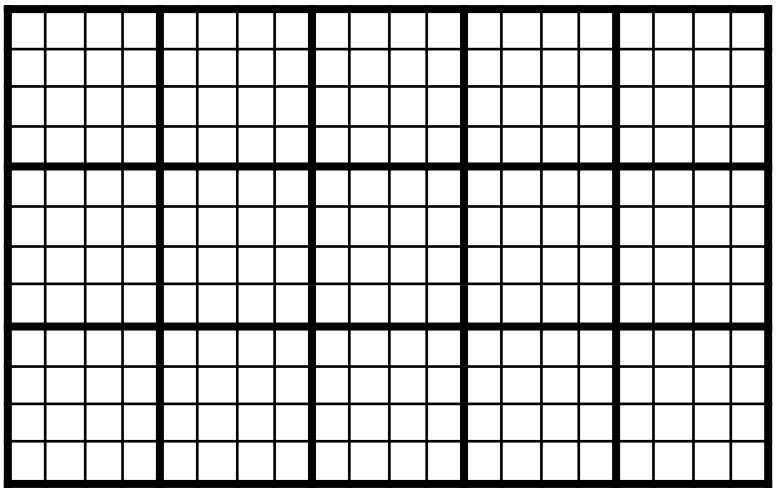
```
#include <iostream>
using namespace std;
int a, b, r, A, B;
int main () {
    cin>>A>>B;
    a = A;
    b = B;
    while (b!=0) {
        r = a%b;
        a = b;
        b = r;
    }
    cout<<A/a*B;
    return 0;
}
```

2. Se citește `n` și apoi `n` numere naturale. Să se afișeze cel mai mare divizor comun al tuturor celor `n` numere. De exemplu, pentru `n=5` și numerele: `18 48 30 24 60`, rezultatul obținut trebuie să fie `6`.

```
#include <iostream>
using namespace std;
int a, b, r, n, i;
int main () {
    cin>>n;
    cin>>a;
    for (i=2;i<=n;i++ ){
        cin>>b;
        while (b!=0) {
            r = a%b;
            a = b;
            b = r;
        }
    }
    cout<<a;
}
```

Aici am presupus primul număr ca fiind cel mai mare divizor comun și am aplicat algoritmul lui Euclid între el și al doilea număr citit. La implementarea de mai sus ne ajută mult faptul că rezultatul rămâne tot în prima variabilă, așadar la pașii următori putem doar să reluăm raționamentul pentru noul număr citit și valoare rămasă în `a` de la pasul anterior.

3. Se dau două numere `a` și `b` ce reprezintă lungimile laturilor unei zone dreptunghiulare. Se dorește pavarea completă a zonei cu dale pătrate, toate de aceeași latură și în număr cât mai mic. Noi trebuie să determinăm latura unei astfel de dale și câte sunt necesare. De exemplu, pentru datele de intrare `20` și `12`, rezultatul trebuie să fie `4` (latura) și `15` (numărul de dale necesare)



Observăm că pentru a se încadra perfect pe linii și pe coloane dalele pătrate, latura unei astfel de dale trebuie să fie un divizor comun pentru lungimile laturilor dreptunghiului dat. Orice astfel de divizor asigură o acoperire completă (cu dale de formă pătrată).

Fie `d` un divizor comum. Pe o linie încap `a/d` date de latură `d`, iar pe o coloană încap `b/d` astfel de dale. Numărul de dale necesar acoperirii este `(a/d) * (b/d)`. Numărul minim de dale se obține pentru `d` = cel mai mare divizor comun dintre `a` și `b`.

```
#include <iostream>
using namespace std;
long long a, b, x, y, r, latura, numar;
int main()
{
    cin>>a>>b;
    x = a;
    y = b;
    while (b != 0) {
        r = a % b;
        a = b;
        b = r;
    }
    latura = a;
    numar = (x / latura) * ( y / latura);
    cout<<numar<<" "<<latura;
}
```

Șiruri recurente

Sunt șiruri în care un termen este definit în funcție de valoarea altor termeni, anteriori. De exemplu, dacă notăm T_n ca fiind `n!` (numit "`n` factorial" și reprezentând produsul primelor n numere naturale nenule), avem:

$$T_n = \{1\ dacă\ n = 0\ respectiv\ n * T_{n-1}\ dacă\ n > 0$$

Șirul factorialelor este deci: `1, 1, 2, 6, 24, 120, 720 ...` (am început numerotarea de la `0`).

O numim pe aceasta recurență de ordin `1` (un termen depinde de un singur termen anterior).

Alt exemplu este șirul lui Fibonacci:

$$T_n = \{1\ dacă\ n = 1\ sau\ n = 2\ respectiv\ T_{n-1} + T_{n-2}\ dacă\ \ n > 2$$

Șirul termenilor Fibonacci este: `1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 ...` (am început numerotarea de la `1`).

În acest caz este vorba despre o recurență de ordin `2` (un termen al șirului este exprimat printr-o relație în care apar doi termeni anteriori).

Observăm deci că pentru o recurență de ordin `1` trebuie cunoscut un termen la început iar pentru una de ordin `2` trebuie cunoscuți inițial doi termeni.

Se pun următoarele probleme:

- Se dă un număr `n` și se cere determinarea celui de-al `n`-lea termen al șirului dat.
- Se dă un termen `x` și se cere să verificăm dacă se află în șir.

Să analizăm cele două probleme în contextul unei recurențe de ordin `1`, apoi pentru recurența de ordin `2`.

1. Să se determine al `n`-lea termen din șirul factorialelor (`n!`).

```
#include <iostream>
using namespace std;
long long n, t, i;
int main(){
      cin >> n;
      t = 1;
      for (i = 1; i <= n; i++){
            t = t*i;
      }
      cout << t;
      return 0;
}
```

Vom calcula așadar termenii unul din altul și observăm că pentru asta ne este suficientă o singură variabilă `t`, cea în care actualizăm valoarea termenului curent din cel anterior.

2. Se citește un număr `x`, se cere să verificăm dacă face parte din șirul factorialelor.

La astfel de probleme analizăm separat două cazuri: dacă numărul de testat este primul termen al șirului respectiv cazul când este un alt termen, fiind aici nevoie de calcule. Ne vom folosi de faptul că șirul factorialelor este *crescător* și, aplicând formula recurenței, vom genera termeni până întâlnim sau depășim valoarea dată.

```
#include <iostream>
using namespace std;
int t, x, i;
int main () {
    cin>>x;
    if (x == 1) {
        cout<<"DA\n";
        return 0;
    }
    t = 1;
    i = 2;
    while (t < x) {
        t *= i;
        i++;
    }
    if (t == x)
        cout<<"DA\n";
    else
        cout<<"NU\n";
    return 0;
}
```

Trecem acum la analizarea celor două cerințe pentru o recurență de ordin `2`.

3. Se citește un număr `n`, determinați al `n`-lea termen Fibonacci.

```
#include <iostream>
using namespace std;
int n, a, b, c, i;
int main () {
    cin>>n;
    if (n == 1 || n == 2) {
        cout<<1;
        return 0;
    }
    a = 1;
    b = 1;
    for (i=3;i<=n;i++) {
        c = a+b;
        a = b;
        b = c;
    }
    cout<<c;
    return 0;
}
```

Observăm că în acest caz nu ne mai este suficientă o singură variabilă, întrucât din urmă trebuie păstrati doi termeni. Modalitatea comodă de implementare este cu trei variabile, așa cum se vede mai sus, chiar dacă este posibilă și o implementare doar folosind două variabile.

4. Se citește un număr `x`. Se cere să verificăm dacă este termen al șirului lui Fibonacci.

```
#include <iostream>
using namespace std;
```

```
int x, a, b, c;
int main () {
    cin>>x;
    if (x == 1) {
        cout<<"DA";
        return 0;
    }
    a = 1;
    b = 1;
    while (a + b <= x) {
        c = a+b;
        a = b;
        b = c;
    }
    if (c == x)
        cout<<"DA";
    else
        cout<<"NU";
    return 0;
}
```

Și în acest caz ne-am folosit de faptul că șirul dat este monoton (crescător) și am generat termeni până ajungem în jurul valorii pe care o testăm (analizând separat cazul particular).

Problemele enunțate mai sus și rezolvate pe cele două șiruri particulare alese (factorial pentru recurență de ordin `1` și Fibonacci pentru cea de ordin `2`), pot fi rezolvate pe orice șir cu aceleași proprietăți, chiar dacă diferă formula de calcul al termenului curent.

5. În cazul șirului lui Fibonacci există multe rezultate interesante. În acest material prezentăm unul pentru care vom face demonstrația și vom scrie și un algoritm constructiv: orice număr natural se poate scrie ca sumă de termeni distincți și neconsecutivi din șirul lui Fibonacci.

Pentru rezolvare se utilizează următoarea strategie: calculăm pentru numărul dat (notat mai jos cu `x`) cel mai mare termen Fibonacci mai mic sau egal decât el (notat cu `T`). Afișăm termenul găsit și continuăm procedeul pentru diferență, mergând până se ajunge la `0`.

Exemplu:

x	T
45	34
11	8
3	3
0	Stop

Algoritm

```
#include <iostream>
using namespace std;
int x, a, b, c;
int main () {
    cin>>x;
    while (x!=0) {
        a = 1;
```

```
        b = 1;
        c = 1;
        while (a + b <= x) {
            c = a+b;
            a = b;
            b = c;
        }
        cout<<c<<" ";
        x -= c;
    }
    return 0;
}
```

Observații

- Faptul că nu pot fi termeni consecutivi se probează prin reducere la absurd: existența a doi termeni consecutivi ar însemna că la alegerea celui mai mare dintre cei doi termeni se putea alege și suma lor (care era tot termen Fibonacci și cu o valoare mai mare), deci nu s-a făcut corect alegerea celui mai mare termen posibil la acel pas.
- Faptul că nu pot fi termeni egali se probează prin faptul că un temen Fibonacci se poate scrie ca sumă a altor termeni Fibonacci mai mici ca el.
- Algoritmul de mai sus oferă termenii sumei în ordine descrescătoare.
- Se observă că nu am mai tratat special termenul `1`. Este în continuare corect din faptul că am inițializat înainte `c` cu valoarea `1`, ceea ce face ca în acest caz, neintrându-se în repetiție, să se considere totuși la soluție valoarea care corectă.

Să analizăm în continuare următoarea problemă: se citește `n` și apoi `n` numere naturale și trebuie să determinăm de câte ori s-au introdus la rând trei valori în ordine crescătoare.

Exemplu: pentru `n = 11` și numerele `4 1 2 7 8 1 4 2 3 5 3`, se va afișa `3`, în contul tripletelor(`1 2 7`) (`2 7 8`) și (`2 3 5`).

```
#include <iostream>
using namespace std;
int n, a, b, c, i, sol;
int main () {
    cin>>n;
    sol = 0;
    cin>>a>>b;
    for (i=3; i<=n; i++) {
        cin>>c;
        if (a < b && b < c)
            sol++;
        a = b;
        b = c;
    }
    cout<<sol;
    return 0;
}
```

Observăm că și aici este necesară păstrarea a două valori din urmă (ultimele două) pentru a putea realiza testul. Prezentarea exemplului este pentru a sublinia abordarea comună pentru această problemă, pentru Euclid și pentru Fibonacci. Toate trei pot fi rezolvate cu următoarea abordare:

```
Repetă {
      Procesează numărul curent folosind-ul pe penultimul și pe ultimul
      Actualizează penultim (penultim = ultim)
      Actualizează ultim (ultim = curent)
}
```

Euclid	Fibonacci	Triplete
repetiție (…) { `    c = a % b;` **`    a = b;`** **`    b = c;`** **}**	**repetiție (…) {** `    c = a + b;` **`    a = b;`** **`    b = c;`** **}**	**repetiție (…) {** `    citește c, daca(a<b și b<c) sol++;` **`    a = b;`** **`    b = c;`** **}**