**Expresii relaționale**

Sunt ceea ce popular numim "condiție" ↔ o expresie relațională verifică valoarea de adevăr a relației dintre două elemente (mai mic, mai mare, ...)

Operatorii în C++ sunt:

< mai mic

<= mai mic sau egal

> mai mare

>= mai mare sau egal

== egal

!= diferit

(Ex: cout<<(1<=3); //afișează 1

cout<<(1==3); //afișează 0)

**Expresii logice**

Sunt cele cunoscute de la logica matematică, adică, în ordinea priorității efectuării lor:

! → negație

&& → conjuncție

|| → disjuncție

**Regulile negației:**

!(E1&&E2) = !E1 || !E2

!(E1||E2) = !E1 && !E2

**Exemple**

Condiția ca valoarea variabilei de tip int x să fie cifră:

x>=0 && x<=9

Condiția ca valoarea variabilei de tip int x să NU fie cifră:

!(x>=0 && x<=9) ↔ !(x>=0) || !(x<=9) ↔ x<0 || x>9

**Variabile de tip flag (switch, semafor)**

Sunt variabile de tip indicator, pe care le folosim DOAR cu două stări, și anume 0 (în general 0 = fals) și 1 (adevărat).

Cu ajutorul lor se pot verifica următoarele două clase de probleme:

1) Testarea dacă TOATE elementele unei mulțimi satisfac o proprietate

2) Testarea dacă există cel puțin un element al unei mulțimi care satisface o proprietate

Principiul constă în inițializarea flag-ului cu 1 sau 0, în funcție de presupunerea făcută și, ulterior, verificarea elementelor mulțimii unul câte unul.

Dacă la un moment dat un anumit element NU convine, schimbăm valoarea flag-ului.

Este esențial ca în timpul verificării element cu element să NU permitem revenirea flag-ului la valoarea inițială.

**Numere prime** (prime numbers)

Def: Se numește număr prim un număr care are doar doi divizori **distincți** și anume 1 și numărul însuși

Exemple de numere prime:

2 3 5 7 11 13 17 19 97 101 127 1237 12347 123457

Exemple de numere NEprime:

0 1 4 6 8 10 91(=7\*13) 119(=7\*17)

**Teoremă:** Dacă un număr natural **mai mare ca 1** NU are niciun divizor propriu cuprins între 2 și radicalul său, atunci numărul este prim.

Teorema ne dă și unul dintre algoritmii eficienți de verificare dacă un număr este prim sau nu: dat fiind un număr n, testăm toți potențialii divizori săi **cuprinși între 2 și** . Dacă îl prindem pe n că se divide la vreunul dintre ei → NU este prim.

Verificarea se face cu ajutorul unei variabile de tip flag.

Iată două variante de program (una clasică și alta ușor eficientizată) care verifică dacă un număr natural "n" este prim:

**V1) Varianta clasică**

is\_prime=1;//asta e variabila de tip flag

for(d=2;d\*d<=n;d++)//in pseudocod trebuie scris pentru d <- 2, execută

if(n%d==0) //sau pe while

{is\_prime=0;break;}

if(n<=1)is\_prime=0;//aceste două cazuri trebuie tratate separat

if(is\_prime) -> este prim

else -> NU este prim

**V2) Varianta îmbunătățită**

Pleacă de la observația că putem tăia din start numerele pare, care oricum NU sunt prime, și care sunt destul de numeroase.

Dacă le-am tăiat pe cele pare, putem să considerăm divizori începând DOAR de la 3, și mergând din 2 în 2.

is\_prime=1;//asta e variabila de tip flag

if(n<2 || n>2 && n%2==0) is\_prime=0;

else

for(d=3;d\*d<=n;d=d+2)

if(n%d==0)

is\_prime=0; //obs: în practică aici un break e binevenit

if(is\_prime) -> este prim

else -> NU este prim

**CMMDC (Cel mai mare divizor comun) GCF (Greatest Common Factor)**

**(GGT Grösster Gemeinsamer Teiler)**

**Cazuri excepție:**

cmmdc(a,0) = cmmdc(0,a) = a (daca a nenul)

cmmdc(0,0) = ∞ (nedeterminare)

**Obs:** Două numere al căror cmmdc este egal cu 1 se numesc "prime între ele".

Ex: 14 și 15 sunt prime între ele, pt. că cmmdc(14,15)=1

Unul dintre algoritmii pe care e bine să-i recunoașteți, dar să nu-i folosiți **NICIODATĂ** (din cauză că este neoptim) este cel **prin scăderi repetate**.

Principiul său constă în scăderea în prostie a valorii mai mici din cea mai mare, până când cele două devin egale.

Valoarea la care devin egale reprezintă cmmdc-ul lor.

Ex: a=90 b=12

|  |  |
| --- | --- |
| a | b |
| 90 | 12 |
| 78 | 12 |
| 66 | 12 |
| 54 | 12 |
| 42 | 12 |
| 30 | 12 |
| 18 | 12 |
| 6 | 12 |
| 6 | 6 |

În momentul egalității, valoarea respectivă reprezintă cmmdc.

Iată exprimarea acestui algoritm:

while(a!=b)

if(a>b)

a=a-b;

else

b=b-a;

Obs:

- este foarte neoptim. Imaginați-vă că a=2 și b=3.000.000

- dacă sau a=0 sau b=0 se blochează în buclă infinită

- valorile inițiale ale lui a și b se distrug.

**Algoritmul optim este algoritmul lui Euclid**

Este dedus de fapt din algoritmul precedent, plecând de la observația că, dacă scădem un număr b din alt număr a până când valoarea lui a devine mai mică decât b, de fapt în a va rămâne RESTUL împărțirii sale la b.

Așadar, algoritmul lui Euclid împarte inițial cele două numere date. Se fac apoi împărțiri repetate, împărțind **împărțitorul** la **rest** (considerându-le pe cele de la ultima împărțire efectuată). Când în acest mod dăm peste o împărțire care are împărțitorul 0, ne oprim. Deîmpărțitul acesteia este de fapt cmmdc-ul dorit.

Ex:

|  |  |
| --- | --- |
| 12 | **90** |
| 0  ==  **12** | 0 |

|  |  |
| --- | --- |
| **90** | **12** |
| 84  ==  6 | 7 |

|  |  |
| --- | --- |
| 12 | 6 |
| 12  ==  0 | 2 |

|  |  |
| --- | --- |
| **6** | 0 |
|  | STOP |

Iată codul:

while(b)

{

r=a%b;

a=b;

b=r;

}

cmmdc este dat de a.

Obs:

- este foarte optim. Pentru orice numere a și b cu maxim 9 cifre, sunt suficienți cel mult 45 de pași pt. calculul său.

- dacă a=0 sau b=0 rezultatul este corect

- valorile inițiale ale lui a și b se distrug.

**Eficiența algoritmului lui Euclid**

Algoritmul lui Euclid este ușor dependent de valorile cărora le calculăm cmmdc.

E limpede, spre exemplu, că:

⋅ pentru a=3000000 și b=2 algoritmul va face 1 pas (cazul favorabil)

⋅ Pentru numere ca a=192 și b=90 va face 3 pași (cazul mediu)

⋅ Pentru numere ca a=21 și b=13 va face 6 pași (cazul nefavorabil)

Dacă facem o analiză ceva mai atentă a alg. lui Euclid, vom constata că numărul maxim de pași (cazul defavorabil) este atins în momentul în care cele două numere reprezintă termeni vecini ai șirului lui Fibonacci.

De fapt, chiar și acest caz, așa nașpa cum pare, face un număr foarte mic de pași.

Spre exemplu dacă luăm cei mai mari 2 termeni din șirul lui Fibonacci reprezentabili pe int, aceștia fiind 1836311903 și 1134903170 se fac 46 de pași, ceea ce oricum este puțin.

De fapt, pentru că orice termen din Fibonacci se poate scrie ca

F_{n} = \cfrac{1}{\sqrt{5}}\cdot\left(\cfrac{1+\sqrt{5}}{2}\right)^n-\cfrac{1}{\sqrt{5}}\cdot\left(\cfrac{1-\sqrt{5}}{2}\right)^n~.

din punctul nostru de vedere, înseamnă că ordinul de complexitate este logaritmic, adică,

față de valorile lui a și b, numărul de pași este de genul logxa sau logxb.

**CMMMC**

Se poate calcula foarte rapid dacă știm cmmdc pe baza formulei:

cmmdc\*cmmmc = a\*b

=>pt. a calcula cmmmc trebuie mai întâi calculat cmmdc, iar atribuirea preferabil s-o faceți sub forma:

**cmmmc = a/cmmdc\*b;** (am preferat să facem mai întâi împărțirea pt. că SIGUR e

posibilă și să evităm eventuale depășiri)