



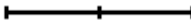
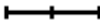




# Numere speciale

## Capitolul

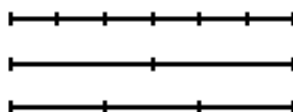
# 4

- ❖ Numere perfecte
- ❖ Numere prietene
- ❖ Numere triangulare
- ❖ Numere pătratice
- ❖ Numere piramidale
- ❖ Numere pitagoreice
- ❖ Numere prime
- ❖ Numere Fibonacci
- ❖ Implementări sugerate
- ❖ Probleme propuse
- ❖ Soluțiile problemelor

În antichitate numerele erau apreciate, în primul rând, pentru calitățile lor și, abia după aceea, pentru cantitatea pe care o reprezintă. Astfel, fiecare număr a fost privit ca o individualitate distinctă. *Rudolf Steiner* ne explică teoria conform căreia un număr se poate obține din *întreg* printr-un proces de împărțire (diviziune) asemănător proceselor naturale de diviziune celulară. Celălalt procedeu de obținere a unui număr ar fi prin *adunare*. Cele două procedee se pot prezenta comparativ astfel:

Operație de împărțire		Operație de adunare
	1	
	2	
	3	
	4	

*Unitatea* se considera mama tuturor numerelor, fiind superioară oricărui număr. Numerele se caracterizau prin dependența lor față de alte numere cu care formează o înlanțuire. De exemplu: numărul 6 s-ar putea reprezenta astfel:



Există multe motive pentru care cercul reprezintă o imagine mai adecvată unui număr care se divizează în părți. În acest caz, numărul 6 s-ar putea reprezenta astfel:



Unitatea din numărul 6 se regăsește, de asemenea, ca unitate în numerele doi și trei (*divizori* ai lui 6), pe când unitatea din 5 se regăsește doar pentru sine însuși, căci cinci nu este multiplul nici unuia dintre numerele care îl preced.



Dacă îl privim așa, *cinci* poate fi numit „primul număr”, **număr prim** cum îl numim noi. Aceeași denumire le este apoi atribuită și celorlalte numere care nu au alți divizori decât pe 1 și pe ei înșiși.

Analizând numărul 6, observăm că adunând divizorii săi (numerele 1, 2 și 3) ajungem din nou la numărul 6 ( $1 + 2 + 3 = 6$ ).

### 4.1. Numere perfecte

Vechii greci aveau un respect deosebit pentru aceste numere și le numeau „arithmos teleios” adică „numărul limitat la el însuși”, în traducerea actuală **număr perfect**.

Se spune că *Euclid* ar fi arătat că dacă  $2^n - 1$  este număr prim, atunci  $2^{n-1}(2^n - 1)$  este număr perfect. Până în prezent nu s-a găsit nici un număr perfect care să nu verifice condiția lui *Euclid*.

$n$	$2^n - 1$	Primalitate	$2^{n-1} \cdot (2^n - 1)$	Suma divizorilor
2	$2^2 - 1 = \underline{3}$	prim	$2^1 \cdot 3 = 2 \cdot 3 = \mathbf{6}$	$1 + 2 + 3 = \mathbf{6}$
3	$2^3 - 1 = \underline{7}$	prim	$2^2 \cdot 7 = 4 \cdot 7 = \mathbf{28}$	$1 + 2 + 4 + 7 + 14 = \mathbf{28}$
4	$2^4 - 1 = \mathbf{15}$	nu		
5	$2^5 - 1 = \underline{\mathbf{31}}$	prim	$2^4 \cdot 31 = 16 \cdot 31 = \mathbf{496}$	$1 + 2 + 4 + 8 + 16 + 31 + 62 + 124 + 248 = \mathbf{496}$
6	$2^6 - 1 = \mathbf{63}$	nu		
7	$2^7 - 1 = \mathbf{127}$	prim	$2^6 \cdot 127 = 64 \cdot 127 = \mathbf{8128}$	$1 + 2 + 4 + 8 + 16 + 32 + 64 + 127 + 254 + 508 + 1016 + 2032 + 4064 = \mathbf{8128}$

Alte proprietăți ale numerelor perfecte :

- 1) Toate numerele perfecte se termină cu cifra 6 sau cu cifra 8, după cum se poate observa din tabelul următor:

$n$	$2^{n-1}$	$2^n - 1$	$2^{n-1}(2^n - 1)$
2	2	3	<b>6</b>
3	4	7	<b>28</b>
4	8	15	120
5	16	31	<b>496</b>
6	32	63	2016
7	64	127	<b>8128</b>
8	128	255	32640
9	256	511	130816
10	512	1023	523776
11	1024	2047	2096128
12	2048	4095	8386560

Numerele de forma  $2^n - 1$  se numesc numere *Mersenne*.

- 2) Primele patru numere perfecte în reprezentare binară sunt:

<i>Numere perfecte</i>	<i>Reprezentare binară</i>
6	110
28	11100
496	11110000
8128	111111000000

Dacă facem o comparație între un număr și suma divizorilor săi, observăm că avem trei situații:

- Dacă suma divizorilor este egală cu numărul, spunem că numărul este *perfect*.
- Dacă suma divizorilor este mai mică decât numărul, spunem că numărul este *deficient*.
- Dacă suma divizorilor este mai mare decât numărul, spunem că numărul este *bogat*.

## 4.2. Numere prietene

Perechile de numere care se „acoperă” reciproc, se bucurau de un mare interes în vechea Grece și erau numite „philoi arithmoi” adică *numere prietene* (în unele surse le găsim cu numele de numere *amiabile*). Aici prin „acoperire” înțelegem proprietatea că fiecare dintre cele două numere are *suma divizorilor egală cu celălalt număr*.

Prima pereche de numere prietene este 220 cu 284. Suma divizorilor lui 220 este  $1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284$ , iar suma divizorilor lui 284 este  $1 + 2 + 4 + 71 + 142 = 220$ .

Matematicianul arab *Tabit ibn Korra* (secolul X) a găsit un algoritm de generare a unor perechi de numere prietene. Acest algoritm a fost publicat fără demonstrație de către *Descartes* în anul 1638.

Prezentăm în continuare algoritmul (*rețeta*) lui *Tabit* în forma în care se cunoștea în acele vremuri.

*Pasul 1:* Se pleacă de la numărul 6;

*Pasul 2:* Se construiește, prin dublări succesive șirul: 6, 12, 24, 48, 96, 192, ...

*Pasul 3:* În continuare se iau pe rând din acest șir perechi succesive. Exemplu: 6 și 12, 12 și 24, 24 și 48, ...

*Pasul 4:* Fiecărei perechi de numere  $i$  se adaugă un al treilea număr, care conține ambii membri ai dualității, adică produsul lor. Se obțin astfel:

(6, 12); (12, 24); (24, 48); (48, 96); ...  
72      288      1152      4608

*Pasul 5:* Se scade 1 din fiecare din cele trei numere. Se obține:

(5, 11); (11, 23); (23, 47); (47, 95); ...  
71      287      1151      4607

*Pasul 6:* Se construiește seria dublurilor pentru numărul 4 (4, 8, 16, 32, 64, ...) care se scrie sub numerele obținute la pasul anterior.

(5, 11); (11, 23); (23, 47); (47, 95); ...  
71      287      1151      4607  
4      8      16      32

*Pasul 7:* Începând cu acest pas ne interesează numai tripletele de numere prime. În cazul nostru se păstrează:

(5, 11); (23, 47); ...  
71      1151  
4      16

*Pasul 8:* La acest pas, se înmulțesc cele două numere ale perechii și se obține:

55, 1081, ...  
71, 1151

*Pasul 9:* În final, cele două numere scrise deasupra multiplilor lui 4 se înmulțesc cu multiplul corespunzător. Se obțin astfel perechi de numere prietene:

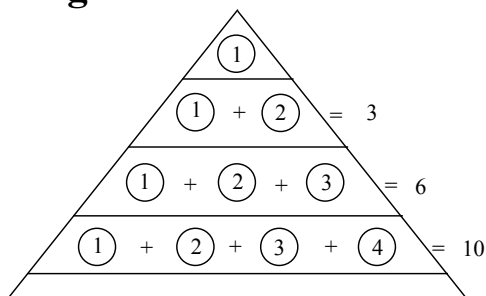
220, 17296, ...  
284, 18416, ...

### Observație

Algoritmul lui *Tabit* **nu** generează toate perechile de numere prietene. El a găsit doar o regulă de a genera anumite numere perfecte.

În continuare prezentăm alte numere cărora anticii le acordau o deosebită atenție.

### 4.3. Numere triangulare



Un număr  $x$  se numește **triangular** dacă există un număr natural  $n$ , astfel încât suma primelor  $n$  numere naturale este egală cu numărul dat  $x$ :

$$1 + 2 + 3 + \dots + n = n \cdot (n + 1)/2 = x$$

#### 4.3.1. Relații între numerele perfecte și cele triangulare

Dacă se privește șirul numerelor triangulare : 1, 3, 6, 10, 15, 21, 28, 36, ... se observă că în șir apar numerele perfecte 6 și 28. Este ușor de demonstrat că orice număr perfect este totodată și număr *triangular*. Lăsăm pe seama cititorilor verificarea acestei observații, ținând cont de ipoteza că orice număr perfect are forma  $2^{n-1}(2^n - 1)$ .

### 4.4. Numere pătratice

Un număr  $x$  se numește **pătratic** dacă există un număr natural  $n$ , astfel încât numărul dat  $x$  să fie egal cu suma primelor  $n$  numere naturale impare:

$$1 + 3 + 5 + \dots + (2n - 1) = n^2$$

### 4.5. Numere piramidale

Un număr se numește **piramidal** dacă poate fi scris ca sumă de numere triangulare consecutive:

$$\begin{aligned} 1 \\ 1 + 3 = 4 \\ 1 + 3 + 6 = 10 \\ \dots \end{aligned}$$

### 4.6. Numere pitagoreice

Relația caracteristică laturilor unui triunghi dreptunghic (cunoscută empiric și de egipteni) este:

$$x^2 + y^2 = z^2$$

Numerele care respectă o astfel de relație se numesc **numere pitagoreice**.

## 4.7. Numere prime

Un număr prim are exact doi divizori. Rezultă imediat că 1 nu este prim și singurul număr prim par este 2. Verificarea primalității se poate realiza simplu, pornind de la definiție, dar se pot scrie și algoritmi mai performanți care se vor opri imediat după ce s-a găsit un divizor propriu. De asemenea, putem evita căutarea unor divizori care sunt numere pare în cazul în care numărul dat este impar (vezi rezolvarea problemei 4.10.7).

Generarea numerelor prime mai mici decât un număr dat i-a preocupat și pe matematicienii din Grecia antică. Strategia lui *Eratostene* și astăzi este un „algoritm” aplicat. El a scris pe un papirus, începând de la 2, un șir de numere naturale consecutive. Apoi, l-a marcat pe 2 ca fiind prim și a tăiat toate numerele divizibile cu 2. A procedat la fel cu 3. Pe 4 nu l-a mai „găsit” printre numere, acesta fiind tăiat, deci a continuat cu 5. În final, pe papirus au rămas netăiate doar numerele prime.

## 4.8. Numere *Fibonacci*

Șirul numerelor care poartă numele celebrului matematician *Fibonacci* are foarte multe proprietăți interesante:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Generarea termenilor mai mici sau egali cu un număr natural  $n > 1$  dat ai acestui șir este o problemă simplă și se realizează cu următorul algoritm.

**Algoritm Fibonacci:**

<b>citește</b> $n$	{ valoarea cea mai mare a unui termen }
$a \leftarrow 1$	{ primul termen }
$b \leftarrow 1$	{ al doilea termen }
$c \leftarrow a + b$	{ al treilea termen }
<b>scrie</b> $a, b$	{ deoarece $n > 1$ , primii doi se pot scrie }
<b>cât timp</b> $c \leq n$ <b>execută:</b>	{ cât timp suma celor doi termeni anteriori }
<b>scrie</b> $c$	{ satisface cerința, o scriem }
$a \leftarrow b$	{ noul $a$ va fi vechiul $b$ }
$b \leftarrow c$	{ noul $b$ va fi vechiul $c$ }
$c \leftarrow a + b$	{ calculăm noul termen }
<b>sfârșit cât timp</b>	
<b>sfârșit algoritm</b>	

O problemă interesantă care se poate formula este următoarea: dat fiind un număr natural, să se stabilească dacă acesta este sau nu număr *Fibonacci* și, în caz negativ, să se scrie numărul sub formă de sumă de număr *minim* de numere *Fibonacci*.

Vom genera, cu algoritmul de mai sus, numere Fibonacci și ne vom opri în momentul în care ultimul termen este mai mare sau egal cu numărul dat. Dacă ultima valoare a variabilei  $c$  este număr *Fibonacci*, algoritmul se termină. În caz contrar, afișăm cel mai mare număr *Fibonacci* mai mic sau egal cu numărul dat (disponibil în variabila  $b$ ) și scădem din numărul dat valoarea acestuia. Dacă  $b$  a fost egal cu numărul dat, algoritmul se termină, în caz contrar reluăm algoritmul de verificare. Cât timp  $n$  (din care mereu se scade numărul *Fibonacci*  $b$  care se afișează) este pozitiv, determinăm câte un număr Fibonacci  $b$  mai mic sau egal cu valoarea actuală a lui  $n$ , pe care o scădem din  $n$  și o afișăm.

**Algoritm** Sumă\_de\_Numere\_Fibonacci:

```

citește n { valoarea dată }
a ← 1
b ← 1
c ← a + b
cât timp c < n execută:
    a ← b
    b ← c
    c ← a + b
sfârșit cât timp { acum b este cel mai mare număr Fibonacci ≤ n }
dacă c = n atunci
    scrie 'Numar Fibonacci'
altfel
    scrie n, '=', b
    n ← n - b
    cât timp n > 0 execută: { cât timp mai avem rest }
        cât timp b > n execută:
            c ← b
            b ← a
            a ← c - b
        sfârșit cât timp { acum b este cel mai mare număr Fibonacci ≤ n }
        scrie '+', b
        n ← n - b
    sfârșit cât timp
sfârșit dacă
sfârșit algoritm

```

## 4.9. Implementări sugerate

Pentru a vă familiariza cu modul în care se rezolvă problemele în care verificarea divizibilității are rol important, vă sugerăm să încercați să implementați algoritmi pentru:

1. numărarea (generarea) divizorilor unui număr;
2. descompunerea unui număr în factori primi;
3. determinarea factorului prim al unui număr dat, care este la puterea cea mai mare;
4. cel mai mare divizor comun a două numere întregi;
5. cel mai mic multiplu comun a două numere întregi;
6. verificarea primalității unui număr;
7. determinarea celui mai mic număr prim mai mare decât un număr dat;
8. determinarea celui mai mare număr prim mai mic decât un număr dat;
9. determinarea prefixelor prime ale unui număr;
10. determinarea sufixelor prime ale unui număr;
11. determinarea primelor  $n$  numere prime;
12. determinarea tuturor numerelor prime dintr-un interval;
13. determinarea primelor  $n$  numere perfecte;
14. determinarea primelor  $n$  numere perfecte folosind ideea lui *Euclid*;
15. determinarea primelor  $n$  perechi de numere prietene;
16. generarea perechilor de numere prietene, folosind algoritmul lui *Tabit ibn Korra*;
17. afișarea numerelor *triangulare* dintr-un șir de numere naturale;
18. afișarea numerelor *triangulare* care sunt pătrate perfecte și sunt mai mici sau egale cu un număr natural  $n$  dat;
19. afișarea numerelor *pătratice* dintr-un șir dat de numere naturale;
20. afișarea numerelor *pătratice* care sunt *pătrate perfecte* și sunt mai mici sau egale cu un număr natural  $n$  dat;
21. generarea primelor  $n$  perechi de numere *pitagoreice*;
22. generarea termenilor șirului *Fibonacci* mai mici decât un număr natural dat;
23. generarea primilor  $n$  termeni ai șirului *Fibonacci*;
24. scrierea unui număr natural  $n$  sub formă de sumă de numere *Fibonacci*.

## 4.10. Probleme propuse

### 4.10.1. Divizori

Să se scrie un program care determină cel mai mic număr care are exact  $k$  divizori, unde  $k$  este un număr natural dat.

#### Date de intrare

Numărul natural  $k$  se citește din fișierul de intrare **NUMAR.IN**.



**Date de ieșire**

Primul număr natural având exact  $k$  divizori se va scrie în fișierul **NUMAR.OUT**.

**Restricții și precizări**

- $2 \leq k \leq 100$

**Exemplu**

NUMAR.IN	NUMAR.OUT
4	6

**Timp maxim de execuție: 2 secunde**

**4.10.2. Numere perfecte**

Să se genereze toate numerele perfecte posibile de reprezentat cu un număr de tip întreg pe două cuvinte. Implementați algoritmul sugerat de *Euclid* pentru obținerea numerelor perfecte.

**Date de ieșire**

Numerele perfecte mai mici decât cel mai mare număr întreg posibil de reprezentat se vor scrie în fișierul **PERFECT.OUT**.

**4.10.3. Numere prietene**

Să se genereze perechi de numere prietene, posibile de reprezentat cu numere de tip întreg pe două cuvinte. Implementați algoritmul lui *Tabit*.

**Date de ieșire**

Perechile de numere prietene se vor scrie pe câte o linie a fișierului **PRIETENE.OUT**.

**4.10.4. Numere triangulare**

Se consideră mai multe numere naturale scrise într-un fișier. Să se copieze într-un alt fișier toate numerele triangulare și într-un al treilea acelea care nu sunt triangulare.

**Date de intrare**

Numerele naturale se vor citi de pe prima linie a fișierului **TRI.IN**.

**Date de ieșire**

Numerele citite din fișierul de intrare care sunt triangulare se vor scrie pe prima linie a fișierului **TRI.OUT** (despărțite prin câte un spațiu), iar cele care nu sunt triangulare se vor scrie pe prima linie a fișierului **REST.OUT**.

**Restricții și precizări**

- $1 \leq \text{orice număr dat} \leq 1000000000$ ;
- după ultimul număr existent în fișier urmează imediat marcajul de sfârșit de fișier.

**Exemplu**

TRI . IN	TRI . OUT	REST . OUT
22 3 55 6 10 11	3 55 6 10	22 11

**4.10.5. Aplicație la numerele triangulare**

Se consideră șirul de numere: 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, ...

Să se calculeze valoarea celui de-al  $k$ -lea element din șir ( $k$  număr natural dat) fără să se genereze șirul de elemente.

**Date de intrare**

Numărul natural  $k$  se citește din fișierul de intrare **SIR . IN**.

**Date de ieșire**

Valoarea corespunzătoare celui de-al  $k$ -lea element din șir se va scrie în fișierul de ieșire **SIR . OUT**.

**Restricții și precizări**

- $1 \leq k \leq 1000000000$ .

**Exemplu**

SIR . IN	SIR . OUT
14	5

**4.10.6. Alte numere speciale**

Notăm cu  $n_k^c$  ( $k = 1, 2, \dots, 9, c \in \{2, 3, \dots, 9\}$ ) numerele naturale formate din  $c$  cifre identice și egale cu  $k$ . Notăm suma cifrelor:  $s_k = c \cdot k$ . Se știe că există mai multe numere  $nr$ , astfel încât se poate găsi o valoare  $c$  pentru care:

$$s_k \cdot nr = n_k^c, \forall k = 1, 2, \dots, 9.$$

Să se scrie un program care determină toate perechile  $(nr, c)$  având proprietatea descrisă.

**Date de ieșire**

Pe fiecare linie a fișierului de ieșire **NUMAR . OUT** se vor scrie două numere naturale, separate printr-un spațiu, reprezentând numărul  $nr$  și numărul  $c$  de cifre identice pentru care proprietatea cerută este adevărată.

### 4.10.7. Prim

Stabiliți dacă un număr dat este prim sau nu!

#### Date de intrare

În fișierul de intrare **PRIM.IN** este scris un singur număr natural.

#### Date de ieșire

Dacă numărul citit din fișierul de intrare este prim, în fișierul de ieșire **PRIM.OUT** se va scrie DA, altfel se va scrie NU.

#### Exemplu

<b>PRIM.IN</b>	<b>PRIM.OUT</b>
2	DA

## 4.11. Soluțiile problemelor propuse

### 4.11.1. Divizori

Pentru a analiza modul de rezolvare a acestei probleme, vom proiecta algoritmul pentru un exemplu. Fie  $k = 4$ . Vom determina divizorii numerelor naturale, începând cu 2, și terminând în momentul în care am găsit primul număr care are exact  $k$  divizori.

#### Exemplu

- 4 are 3 divizori (1, 2 și 4);
- 5 are 2 divizori (1, 5);
- 6 are 4 divizori (1, 2, 3 și 6), deci ne oprim. Numărul căutat este 6.

Analizând exemplul dat, așa pare că divizorii sunt numere prime, dar trebuie să fim atenți. De exemplu, numărul 840 are 32 de divizori: 1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 14, 15, 20, 21, 24, 30, 35, ..., deci printre divizori avem numere prime și puteri ale lor, respectiv produse de două sau mai multe numere prime.

În descrierea următorului algoritm notăm cu  $nr$  numărul natural curent, cu *divizor* valoarea cu care efectuăm împărțirea și cu  $nr\_d$  numărul divizorilor lui  $nr$ . Acest algoritm generează numere naturale și, începând cu numărul 2, determină toți divizorii acestora, numărându-le. Generarea numerelor se oprește în momentul în care s-a găsit primul număr având  $k$  divizori. Dacă însă nu suntem atenți și nu căutăm posibilități de optimizare, programul nostru va rula pentru  $k = 100$  mult mai mult timp decât permite restricția privind timpul de execuție.

În concluzie, vom căuta divizori până la rădăcina pătrată a numărului, deoarece, chiar dacă există divizori mai mari decât această valoare, aceștia sunt egali cu câtul împărțirii întregi ai unui divizor mai mic decât rădăcina pătrată. În concluzie, pentru fiecare divizor găsit mărim contorul divizorilor cu 2 (odată pentru divizorul `divizor` și încă odată pentru divizorul `[nr/divizor]`). Să observăm că dacă numărul este pătrat perfect, divizorul `divizor` este egal cu divizorul `[nr/divizor]`, deci în cazul numerelor care sunt pătrate perfecte, în final vom micșora cu 1 numărul divizorilor contorizați.

**Algoritm** Divizori:

```

citește k
nr ← 1
repetă
    nr ← nr + 1      { dacă k = 2, atunci cel mai mic număr cu 2 divizori este 2 }
    nr_d ← 2          { orice număr are doi divizori proprii }
    pentru divizor=2, parte întreagă din rădăcina pătrată a lui n execută:
        dacă rest[nr/divizor] = 0 atunci
            nr_d ← nr_d + 2          { numărăm divizorii lui nr }
        sfârșit dacă
    sfârșit pentru
    dacă nr este pătrat perfect atunci
        nr_d ← nr_d - 1
    până când nr_d = k          { până când obținem k divizori }
    scrie nr
sfârșit algoritm

```

#### 4.11.2. Numere perfecte

În enunțul problemei ni se cere să generăm toate numerele perfecte mai mici decât un număr dat  $n$ . După cum s-a sugerat în enunț, vom folosi condiția necesară enunțată de către *Euclid*, prezentată anterior.

Condiția spune că dacă numărul  $2^m - 1$  este prim, atunci numărul  $2^{m-1}(2^m - 1)$  este perfect.

Astfel, cel mai simplu procedeu este de a da pe rând valori lui  $m$  și, în cazul în care numărul  $2^m - 1$  este prim, de a afișa numărul perfect  $2^{m-1}(2^m - 1)$ .

**Algoritm** Generare\_Numere\_Perfecte:

```

citește n
actual ← 2          { prima putere a lui 2 }
repetă
    actual ← actual*2          { 2 la puterea m }
    prim ← adevărat          { verificăm dacă actual - 1 este prim }

```

```

pentru divizor=2, parte întreagă din rădăcina pătrată a lui actual-1
                                                    execută:
    dacă rest[(actual-1)/divizor] = 0 atunci
        prim ← fals
        ieșire forțată din pentru
    sfârșit dacă
sfârșit pentru
dacă prim atunci
    depășire ← fals { ne ferim de o înmulțire care ar provoca depășire }
    dacă calcularea valorii [actual/2]*(actual-1) ar produce depășire
                                                    atunci
        depășire ← adevărat
    altfel
        scrie [actual/2]*(actual-1)
    sfârșit dacă
până când depășire
sfârșit algoritm

```

### 4.11.3. Numere prietene

Ni se cere implementarea unui algoritm descris și exemplificat. Acest algoritm generează unele perechi de numere prietene conform unui model descris de *Tabit*.

În algoritm am notat cele două numere din perechea curentă cu  $a$  și cu  $b$  (multiplii lui 6) și cu  $prod$  produsul lor. Variabilele logice *prima*, *primb* și *primprod* păstrează valoarea de adevăr privind primalitatea numerelor  $a - 1$ ,  $b - 1$  și  $prod - 1$ . Cu *coef* s-a notat valoarea curentă a multiplilor lui 4 prin înmulțire cu 2. Descrierea algoritmului în pseudocod este următorul:

```

Algoritm Generare_Perechi_De_Numere_Prietene;
    coef ← 2 { valoare inițială pentru dublii lui 4; îi vom obține prin înmulțire cu 2 }
    a ← 6 { primul număr din prima pereche }
    prima ← adevărat { valoare inițială necesară la primul pas }
    depășire ← fals
    repetă
    { primul număr în perechea curentă este al doilea număr din perechea precedentă }
    b ← a
    a ← b*2 { al doilea număr în perechea curentă este dublul precedentului }
    { primalitatea celui de-al doilea număr din perechea precedentă }
    primb ← prima
    dacă a*b ar produce depășire atunci { ne ferim de o posibilă depășire }
        depășire ← adevărat

```

```

altfel
    prod ← a*b           { efectuăm produsul, deoarece nu vom avea depășire }
sfârșit dacă
    prima ← adevărat      { ne pregătim să stabilim primalitatea lui a }
    primprod ← adevărat   { și a lui prod }
    divizor ← 2
cât timp prima și (divizor ≤ rădăcina pătrată a lui prod-1) execută:
    dacă rest[(a-1)/divizor] = 0 atunci
        prima ← fals
    sfârșit dacă
    dacă rest[(prod-1)/divizor] = 0 atunci
        primprod ← fals
    sfârșit dacă
    divizor ← divizor + 1
sfârșit cât timp
    coef ← 2*coef         { pregătim următorul multiplu al lui 4 prin dublare }
                        { dacă a-1, b-1 (perechea) și prod-1 sunt numere prime }
    dacă prima și primb și primprod atunci
        scrie (a-1)*(b-1)*coef, ' ', (prod-1)*coef
    până când depășire
sfârșit algoritm

```

#### 4.11.4. Numere triangulare

Cerința la această problemă este să se elimine numerele triangulare dintr-un șir de numere date. Reamintim că un număr se numește *triangular* dacă se poate obține prin însumarea unui număr de numere naturale consecutive, începând cu numărul 1.

Algoritmul de rezolvare prezentat în continuare verifică pe rând dacă numerele din fișier sunt triangulare, scriind în fișierul rezultat acele numere care nu verifică această proprietate.

**Algoritm** Triangulare:

```

cât timp nu urmează marca de sfârșit de fișier execută:
    citește x
    suma ← 0           { verificăm dacă x este triangular }
    n ← 1              { primul număr natural în sumă }
    repetă
        suma ← suma + n
        n ← n + 1      { crește valoarea termenului }
    până când x ≤ suma
    dacă x ≠ suma atunci
        scrie x        { dacă x nu este triangular, îl copiem în REST.OUT }

```

```

altfel
    scrie x                                { dacă x este triangular, îl copiem în TRI.OUT }
    sfârșit dacă
    sfârșit cât timp
sfârșit algoritm

```

#### 4.11.5. Aplicație la numerele triangulare

Pentru a înțelege mai bine această problemă, se urmărim în tabelul de mai jos modul de construire a șirului dat.

$k$	1	2	<b>3</b>	4	5	<b>6</b>	7	8	9	<b>10</b>	11	12	13	14	<b>15</b>	...
Elementele șirului	1	2	2	3	3	3	4	4	4	4	5	5	5	5	5	...
Număr triangular = 3	$1 + \underline{2} = \mathbf{3}$															
Număr triangular = 6	$1 + 2 + \underline{3} = \mathbf{6}$															
Număr triangular = 10	$1 + 2 + 3 + \underline{4} = \mathbf{10}$															
	...															
			<b>3</b>			<b>6</b>				<b>10</b>					<b>15</b>	...

Studiind tabelul de mai sus putem trage câteva concluzii:

- Observăm că pentru un număr de ordine  $k$  valoarea elementului poate fi determinată prin calcularea numărului triangular *cel mai apropiat de  $k$* .
- Valoarea căutată va fi egală cu *ultimul* număr natural pe care l-am adăugat numărului triangular.

Aplicând observațiile de mai sus algoritmul este următorul (vă recomandăm să realizați o rezolvare a acestei probleme folosind suma primelor  $n$  numere naturale).

**Algoritm** Problema5:

```

citește k
i ← 1
TRI ← 1
cât timp TRI < k execută
    i ← i + 1
    TRI ← TRI + i
sfârșit cât timp
scrie i
sfârșit algoritm

```

### 4.11.6. Alte numere speciale

Un exemplu de număr care îndeplinește această proprietate este 37:

$$\begin{aligned}(1 + 1 + 1) \cdot 37 &= 111 \\(2 + 2 + 2) \cdot 37 &= 222 \\(3 + 3 + 3) \cdot 37 &= 333 \\(4 + 4 + 4) \cdot 37 &= 444 \\(5 + 5 + 5) \cdot 37 &= 555 \\(6 + 6 + 6) \cdot 37 &= 666 \\(7 + 7 + 7) \cdot 37 &= 777 \\(8 + 8 + 8) \cdot 37 &= 888 \\(9 + 9 + 9) \cdot 37 &= 999\end{aligned}$$

Algoritmul sugerat va căuta pentru fiecare lungime  $c$  de la 2 la 9 numărul care îndeplinește proprietatea cerută pentru orice număr având cifre identice pe lungimea  $c$ . Secretul constă în a observa că numărul căutat este de fapt un *cât* comun în cazul celor 9 numere împărțite la rândul lor la suma cifrelor corespunzătoare.

Am notat variabilele având semnificațiile din enunț, *vechi* reprezintă valoarea  $nr$ , obținută la pasul precedent.

**Algoritm** Speciale:

```

pentru  $c=2,9$  execută:                { numere de lungimi cuprinse între 2 și 9 }
     $ok \leftarrow adevărat$ 
     $k \leftarrow 1$ 
    cât timp ( $k \leq 9$ ) și  $ok$  execută:    { construim numărul din  $c$  cifre }
         $număr \leftarrow 0$ 
        pentru  $i=1,c$  execută:                { construim numărul din  $c$  cifre }
             $număr \leftarrow număr \cdot 10 + k$ 
        sfârșit pentru
         $s \leftarrow c \cdot k$ 
        dacă  $rest[număr/s]=0$  atunci
            { numărul căutat trebuie să fie câtul întreg dintre numărul format }
             $nr \leftarrow [număr/s]$                 { din  $c$  cifre identice și suma celor  $c$  cifre }
            dacă  $k > 1$  atunci                    { dacă  $k = 1$ , încă nu avem vechi }
                dacă  $nr = vechi$  atunci
                     $ok \leftarrow adevărat$     { dacă am obținut același număr, vom continua }
                altfel
                     $ok \leftarrow fals$ 
                sfârșit dacă
            sfârșit dacă
             $vechi \leftarrow nr$ 
             $k \leftarrow k + 1$ 

```





```
altfel
  dacă nr este număr par atunci { numere pare }
    prim ← nr=2 { singurul număr par prim este 2 }
  altfel { numere impare }
    diviz ← 3 { primul divizor posibil al unui număr impar }
    rad ← parte întreagă din rădăcina pătrată a lui nr
        { calculăm radicalul o singură dată în afara structurii repetitive }
    prim ← adevărat { presupunem că numărul este prim }
    cât timp (diviz ≤ rad) și prim execută:
      dacă rest[nr/diviz]=0 atunci
        prim ← fals { am găsit un divizor, deci nr nu este prim }
      altfel
        diviz ← diviz + 2
        { un număr impar se poate divide doar cu numere impare }
    sfârșit dacă
  sfârșit cât timp
sfârșit dacă
afișare
sfârșit algoritm
```