ALGORITMI ITERATIVI

Definiţie: Algoritmii care pot fi descrişi folosind pe lângă comenzi/instrucţiuni de citire (în pseudocod: **citeste**) şi scriere (în pseudocod: **scrie**) folosesc şi comanda iterativa (în pseudocod: **pentru**...**repeta**) se numesc *algoritmi iterativi*.

R1 (2*).

Enunţul problemei: Un lift coboară de la etajul *a* la etajul *b*. Afişati toate etajele pe care le parcurge. De exemplu, pentru *a*=8 şi *b*=3 se va scrie 8, 7, 6, 5, 4, 3.

Metoda de rezolvare: Fiind vorba de-o coborâre, se impune condiţia *a*>*b*. Se vor parcurge în ordine descrescătoare de la *a* la *b* şi se vor afişa valorile corespunzătoare.

Descrierea algoritmului în pseudocod:

```
citeşte a,b     *presupunem a>b
pentru i = a, b, -1  repeta
     scrie i
```

Descrierea algoritmului în C++(folosind CodeBlocks):

```
#include <iostream>
using namespace std;
int main()
{int a,b,i;
 cout<<"Dati nivelul etajului superior: ";
 cin>>a; //valoarea data se retine in variabila a
 cout<<"Dati nivelul etajului inferior: "; cin>>b;
 //controlam datele de intrare (noi vrem a>b)
 if (a<=b)
    cout<<"Liftul nu poate cobora de la "<<a<<" la "<<b;
 else //deci a>b
  for (i=a;i>=b;i--) cout<<i<<" ";
  //scriem valoarea lui i si lasam un spatiu dupa fiecare i
  //pentru a nu apare valorile "lipite" una de alta
 return 0;
}
```

Observație: Instrucțiunea

```
for (i=a;i>=b;i--) cout<<i<<" ";
```

este echivalentă cu

```
i=a;
while (i>=b)
{
  cout<<i<<" ";
  i--;
}
```

De aceea, for-ul inițial se poate citi: *pornind de la i=a, cât timp i este mare sau egal decât a, se afișează valoarea variabilei i și s epune un spațiu după aceea și apoi se trece la următorul i, prin decrementarea valorii varibilei i (i--;).*

R2 (2*).

Enunţul problemei: Un lift parcurge distanţa dintre două etaje *a* şi *b*. Afişaţi etajele parcurse în ordinea atingerii lor. De exemplu, pentru *a*=6 şi *b*=3 se va scrie 6, 5, 4, 3, pentru *a*=2 şi *b*=4 se va scrie 2, 3, 4, iar pentru *a*=2 şi *b*=2 se va scrie 2.

Metoda de rezolvare: Aici este esenţial să ştim dacă liftul coboară (*a*>*b*) sau urcă (*a*<*b*), egalitatea putându-se include într-unul din cazuri, pentru a vedea dacă parcurgem de la a la b în ordine descrescătoare (cu pasul –1 sau în ordine crescătoare cu pasul 1).

Descrierea algoritmului în pseudocod:

```
citeşte a,b
daca a>=b atunci
   ┌ pentru i = a, b, -1 repeta
   └   scrie i
altfel
   ┌ pentru i = a, b, 1 repeta
   └   scrie i
```

Descrierea algoritmului în C++(folosind CodeBlocks):

```
#include <iostream>
using namespace std;
int main()
{
 int a,b,i;
 cout<<"Dati valoarea etajului a: "; cin>>a;
 cout<<"Dati valoarea etajului b: "; cin>>b;
 if (a>=b) //liftul coboara
   for (i=a;i>=b;i--) cout<<i<<" ";
 else //liftul urca, a<b
   for (i=a;i<=b;i++) cout<<i<<" ";
 return 0;
}
```

R3 (2*).

Enunţul problemei: Să se afişeze tabla înmulţirii cu *n* dat.

De exemplu, tabla înmulţirii cu 5 arată astfel:

$0 \times 5 = 0$
$1 \times 5 = 5$
$2 \times 5 = 10$
$3 \times 5 = 15$
$4 \times 5 = 20$
$5 \times 5 = 25$
$6 \times 5 = 30$
$7 \times 5 = 35$
$8 \times 5 = 40$
$9 \times 5 = 45$
$10 \times 5 = 50$

Descrierea algoritmului în pseudocod:

```
citeşte n
┌pentru i = 0, 10, 1 repeta
└      scrie i,"*",n,"= ",i*n
```

Descrierea algoritmului în C++ (folosind CodeBlocks):

```
#include <iostream>
#include <iomanip> //pt setw
using namespace std;
int main()
{
 int n,i;
 cout << "n = ";  cin >> n;
 for (i=0;i<=10;i++)
   cout << setw(2) << i << " * " << n << " = " << i*n << endl;
 //se afis pe doua campuri, ocupandu-le de la dr. spre stanga
 //pt a alinia la dreapta i-ul care are o cifra,
 //respectiv doua cifre
 return 0;
}
```

R4 (3*).

Enunţul problemei: Descrieţi un algoritm pentru determinarea tripletelor pitagoreice mai mici sau egale ca *n* (adică determinaţi tripletele naturale (a, b, c) cu $1 \leq a < b < c \leq n$ şi $a^2 + b^2 = c^2$ – celelalte egalităţi nu mai sunt necesare datorită relaţiei de ordine dintre *a*, *b* şi *c* (*c* este ipotenuză, *b* și *c* sunt catete)).

Metoda de rezolvare: Putem folosi 3 for-uri pentru a testa diverse valori întregi pentru *a*, *b*, *c*, până la maxim *n*. O primă abordare ar fi:

```
int n,a,b,c;
cout<<"n="; cin>>n;
for (a=1;a<=n;a++)
    for (b=1;b<=n;b++)
        for (c=1;c<=n;c++)
            if (a<b && b<c && a*a+b*b==c*c)
                 cout<<a<<" "<<b<<" "<<c<<endl;
```

Dar, se poate optimiza, în primul rând, prin gestionarea for-urilor astfel încât inegalitățile $a<b<c$ să fie îndeplinite. Astfel, pentru un *a* fixat, *b* poate porni de la $a+1$ și astfel inegalitatea $a<b$ este îndeplinită; similar, pentru *a* și *b* fixate, *c* poate porni de la $b+1$ și astfel inegalitatea $b<c$ este îndeplinită. Apoi, cea mai mică valoare a lui *a* este evident 1, iar cea mai mare posibilă este n-2 pentru a mai "avea loc" și *b* și *c*. Apoi, cea mai mare valoarea a lui *b* este n-1, pentru a mai avea loc și *c* (adică c să mai poată avea măcar valoarea *n*). Iar în aceste cazuri, rămâne de testat inegalitatea $a^2 + b^2 = c^2$.

Descrierea algoritmului în C++ (folosind CodeBlocks):

```
#include <iostream>
using namespace std;
int main()
{     int n,a,b,c;
      cout<<"Dati valoarea lui n>=5: "; cin>>n;
      cout<<"Tripletele pitagoreice pana la n sunt:"<<endl;
      for (a=1;a<=n-2;a++)
         for (b=a+1;b<=n-1;b++)
            for (c=b+1;c<=n;c++)
               if (a*a+b*b==c*c)
                  cout<<"("<<a<<","<<b<<","<<c<<")"<<endl;
      return 0;
}
```

Rulare:

```
Dati valoarea lui n>5:  15 <Enter>
Tripletele pitagoreice pana la n sunt:
(3,4,5)
(5,12,13)
(6,8,10)
(9,12,15)
```

R5 (3*).

Enunţul problemei: Descrieţi un algoritm pentru afişarea tuturor triunghiurilor (măsurile laturilor) cu laturile numere naturale şi de perimetru P dat (adică determinaţi tripletele (a, b, c) astfel încât $1 \leq a \leq b \leq c \leq P$ astfel încât $a + b + c = P$). De exemplu, pentru $P = 12$ avem: (2, 5, 5), (3, 4, 5), (4, 4, 4).

Metoda de rezolvare: Algoritmul este similar cu cel anterior, adică se pot folosi tot 3 for-uri, un pic modificate, iar condițiile pe care trebuie să le îndeplinească a, b, c sunt ca suma lor să fie egală cu P și suma oricăror două să fie mai mare decât a treia (ca să poată forma un triunghi). Ca îmbunătățire, se poate evita for-ul după c, ținând cont că el este legat de a și b printr-o egalitate, dar se poate pierde relația de ordine între a, b, c.

```
#include <iostream>
using namespace std;
int main()
{   int P,a,b,c;
    cout<<"P="; cin>>P;
    cout<<"Tringhiurile de perimetru P au laturile:"<<endl;
    for (a=1;a<=P;a++)
      for (b=a;b<=P;b++)
        for (c=b;c<=P;c++)
           if ((a+b+c==P)&&(a+b>c)&&(b+c>a)&&(a+c>b))
                cout<<a<<" "<<b<<" "<<c<<endl;
    return 0;
}
```

Rulare:

```
P = 12 <Enter>
Triunghiurile de perimetru P au laturile:
(2,5,5)
(3,4,5)
(4,4,4)
```

R6 (4*).

Enunțul problemei: Determinați termenul de ordin n al șirului lui Fibonacci. De exemplu, F6 = 8.

Metoda de rezolvare: Șirul lui Fibonacci este determinat prin relația de recurență

$$F_n = F_{n-1} + F_{n-2},\ n \geq 2,$$

unde $F_0 = 0$, iar $F_1=1$.

Nu este necesară folosirea unui vector pentru a memora toate valorile șirului până la n. La un moment dat, se pot utiliza doar 3 variabile, de exemplu a, b, c, ce reprezintă 3 valori consecutive din șir, iar c se va calcula ca fiind suma dintre a și b.

a	b	c	
0	1	1	$= F_2$
1	1	2	$= F_3$
1	2	3	$= F_4$
2	3	5	$= F_5$
3	5	8	$= F_6$

Descrierea algoritmului în C++ (folosind CodeBlocks):

```
#include <iostream>
using namespace std;
int main()
{
    int n,a,b,c;
    cout<<"n = "; cin>>n; //ordinul nr.Fibonacci de determinat
    a = 0; //initial a = F0 = 0
    b = 1; //si b = F1 = 1
    for (int i=2;i<=n;i++)//det.F2, F3, ..., Fn
    {
        c = a+b;
        a = b; //pregatim urmatorul pas
        b = c;
    }
    cout<<"Fibonacci("<<n<<") = "<<c<<endl;
    return 0;
}
```

Rulare:

```
n = 6 <Enter>
Fibonacci(6) = 8
```

Observație: Dacă vrem toate valorile din șirul lui Fibonacci, afișarea se mută în for si se mai afișează separat valorile inițiale.

```
#include <iostream>
using namespace std;
int main()
{   int n,a,b,c;
    cout<<"n = "; cin>>n; //ordinul nr.Fibonacci de determinat
    a = 0; //initial a = F0 = 0
    b = 1; //si b = F1 = 1
    cout<<"Fibonacci(0) = 0"<<endl<<"Fibonacci(1) = 1"<<endl;
    for (int i=2;i<=n;i++)//det.F2, F3, ..., Fn
    {   c = a+b;
        cout<<"Fibonacci("<<i<<") = "<<c<<endl;
        a = b; //pregatim urmatorul pas
        b = c;
    }
    return 0;
}
```

Rulare:

```
n = 6 <Enter>
Fibonacci(0) = 0
Fibonacci(1) = 1
Fibonacci(2) = 1
Fibonacci(3) = 2
Fibonacci(4) = 3
Fibonacci(5) = 5
Fibonacci(6) = 8
```

R7 (2*).

Enunţul problemei: Să se afişeze toţi divizorii întregi ai unui număr natural nenul dat.

Metoda de rezolvare: De exemplu, divizorii lui 6 sunt {±1, ±2, ±3, ±6}, iar divizorii lui 5 sunt {±1, ±5}. Pentru un număr *n* în general ±1 şi ±n se numesc divizori improprii (orice număr nenul se divide cu 1 şi el însuşi), iar ceilalţi se numesc divizori proprii (2 fiind cel mai mic posibil, iar *n*/2 cel mai mare posibil).

Descrierea algoritmului în pseudocod:

```
citeşte n  *presupunem ca n este natural nenul
*valoarea 1 o scriem din oficiu caci orice nr se divide cu 1
scrie ±1
pentru i = 2, [n/2], 1 repeta       *posibilii divizori proprii
   daca n%i=0 atunci  *i este divizor al lui n
      scrie ±i
daca n≠1 atunci
   scrie ±n
*valoarea n o scriem din oficiu caci orice nr n se divide cu n
```

Descrierea algoritmului în C++ (folosind CodeBlocks):

```
#include <iostream>
using namespace std;
int main()
{
 int n,i;
 cout << "Dati valoarea lui n: "; cin >> n;
 //presupunem n>1
 cout << "Divizorii lui n: {-1, +1";
 for (i=2;i<=n/2;i++) //posibilii divizori proprii
   if (n%i==0) //i este chiar divizor al lui n
     cout << ", -" << i << ", +" << i;
 if (n!=1)
   cout << ", -" << n << ", +" << n << "}";
 else
   cout<<" }";
 cout<<endl;
 return 0;
}
```

Rulare:

```
      Dati valoarea lui n: 6
      Divizorii lui n: {-1, +1, -2, +2, -3, +3, -6, +6}
sau
      Dati valoarea lui n: 5
      Divizorii lui n: {-1, +1, -5, +5}
Sau
      Dati valoarea lui n: 1
      Divizorii lui n: {-1, +1 }
```

R8 (3*).

Enunţul problemei: Să se scrie un algoritm pentru a scrie un număr dat ca sumă de două numere impare (toate posibilităţile), dacă se poate. De exemplu, $n = 33$ nu se poate scrie ca sumă de două numere impare, iar $n = 24 = 1+23$ şi $n = 26 = 1 + 25$

$n = 24$	$= 1+23$	$n = 26$	$= 1 + 25$
	$= 3+21$		$= 3 + 23$
	$= 5+19$		$= 5 + 21$
	$= 7+17$		$= 7 + 19$
	$= 9+15$		$= 9 + 17$
	$= 11+13.$		$= 11+15$
			$= 13+13.$

Metoda de rezolvare: Se observă că pentru număr impar nu se poate scrie ca sumă de două numere impare şi că orice număr par se poate scrie ca cel puţin o sumă dintre două numere impare. În cazul unui număr par, putem parcurge cu un *i* toate numerele impare de la 1 la *n*/2 (pentru a nu mai scrie şi simetricele), scriind *n* ca sumă între *i* şi *n-i* (*n* fiind par şi *i* impar, rezultă că *n-i* este de asemenea impar şi nu mai necesită verificarea condiţiei de imparitate).

Descrierea algoritmului în pseudocod:

```
citeste n
daca n%2 = 1 atunci *n este impar (restul imp. la 2 este 1)
     scrie "nu se poate"
altfel *n este par
     pentru i = 1, n/2 ,2 repeta
          scrie i, n-i
```

Descrierea algoritmului în C++ (folosind CodeBlocks):

```
#include <iostream>
using namespace std;
int main()
{
 int n,i;
 cout<<endl<<"n="; cin>>n;
 if (n%2==1) //sau (n%2)
  cout<<"Nu se poate descompune";
 else
  for (i=1; i<=n/2; i+=2)
   //pornind de la 1 din 2 in 2 (numere impare)
   //pana la cel mult jumatate (n/2)
   cout<<n<<" = "<<i<<" + "<<n-i<<endl;
 return 0;
}
```

Rulare:

```
n = 33 <Enter>
Nu se poate
```

sau

```
n = 24 <Enter>
24 = 1 + 23
24 = 3 + 21
24 = 5 + 19
24 = 7 + 17
24 = 9 + 15
24 = 11 + 13
```

R9 (2-3*).

Enunţul problemei: Să se determine un algoritm pentru a afişa toate numerele de 2 cifre care adunate cu răsturnatul lor dau 55. De exemplu, 23 caci 23 + 32 = 55.

Metoda de rezolvare: În primul rând, numerele de 2 cifre sunt între 10 şi 99. Dintre acestea vom afişa doar pe cele cu proprietatea cerută. Răsturnatul unui număr *n* de 2 cifre se obţine astfel: cifra unităţilor * 10 + cifra zecilor, adică : (restul împărţirii lui *n* la 10) * 10 + (câtul împărţirii lui *n* la 10), ceea ce în C/C++ şi am preluat şi în pseudocod este (*n*%10)*10 + *n*/10 = n%10*10 + n/10.

Descrierea algoritmului în pseudocod:

```
pentru n = 10, 99 repeta    *parcurgem toate nr. de 2 cifre
    daca n + (n%10)*10 + n/10 = 55 atunci
       * n + rasturnatul lui n = 55
         scrie n
```

Descrierea algoritmului în C++ (folosind CodeBlocks):

```
#include <iostream> //pentru cin, cout
using namespace std;
int main()
{
 for (int n=10; n<=99; n++) //parcurgem nr de doua cifre
  if (n + n%10*10 + n/10 == 55) //n + rasturnatul lui n = 55
     cout<<n<<" ";
 return 0;
}
```

Rulare:

```
14 23 32 41 50
```

R10 (2-3*).

Enunţul problemei: Să se scrie un algoritm pentru a afişa toate perechile de numere naturale nenule (a, b) astfel încât $a + b$=1000, 17 / a şi 19 / b.

Metoda de rezolvare: La prima vedere, ne putem gândi la două "for"-uri, unul pentru a şi unul pentru b, urmate de cele trei condiţii. Dar ţinând cont că a+ b = 100 (variabilele sunt legate printr-o relație de egalitate), se poate exprima b în funcţie de a şi anume b = 1000 – a şi astfel nu mai este necesar decât un singur "for", cel după a. Cum căutăm a multiplu de 17, putem merge cu a de la 17, din 17 în 17, până la 1000 (fără să calculăm care este cel mai mare mutiplu de 17 < 1000). Singura condiţie care se va mai impune în for va fi "19 / b", adică (1000-a) % 19 = 0.

Descrierea algoritmului în pseudocod:

```
pentru a = 17,1000,17 repeta *multiplii de 17 pana la 1000
    daca (1000-a)%19 = 0 atunci *daca b=100-a se divide cu 19
         scrie a, 1000-a
```

Descrierea algoritmului în C++ (CodeBlocks):

```
#include <iostream>
using namespace std;
int main()
{
 for (int a=17; a<=1000; a+=17)
 //pornind cu a de la 17, pana la cel mult 1000, din 17 in 17
  if ((1000-a)%19 == 0)
  //daca restul impartirii lui b = 1000-a l 19 este 0
    cout <<"("<<a<<","<<1000-a<<")"<<endl;
    //se afiseaza pe rand nou perechea (a,b) curenta
 return 0;
}
```

Rulare:

```
(221,779)
(544,456)
(867,133)
```

R11 (2-3*).

Enunţul problemei: Să se scrie un algoritm pentru a afişa toate tripletele naturale (a, b, c) astfel încât $1 \leq a < b < c \leq 10$, cu $a + b + c$ divizibil cu 10.

Metoda de rezolvare: De data aceasta sunt necesare trei "for"-uri, pentru că cele 3 variabile (a, b, c) nu mai sunt legate printr-o relaţie de egalitate şi astfel una dintre variabile nu mai poate fi scrisă în funcţie de celelalte. Valoare minimă a variabilei a este 1, valoarea minimă a lui b pentru un a dat este a+1 (a<b), iar valoarea minimă a lui c pentru un b dat este b+1 (b<c). Apoi, valoarea maximă a lui c este 10, valoarea maximă a lui b este 9 (b<c) şi valoarea maximă a lui a este 8 (a<b).

Descrierea algoritmului în pseudocod:

```
pentru a = 1,8 repeta
  pentru b = a+1,9 repeta
    pentru c = b+1,10 repeta
      daca (a+b+c)%10 = 0 atunci
        scrie a,b,c
```

Descrierea algoritmului în C++:

```
#include <iostream>
using namespace std;
int main()
{
 for (int a=1; a<=8; a++)
  for (int b=a+1; b<=9; b++)
   for (int c=b+1; c<=10; c++)
    if ((a+b+c)%10==0) //daca a+b+c se divide cu 10
        //adica restul impartirii lui a+b+c la 10 este 0
     cout<<"("<<a<<","<<b<<","<<c<<")   ";
 return 0;
}
```

Rulare:

```
      (1,2,7) (1,3,6) (1,4,5) (1,9,10) (2,3,5) (2,8,10) (3,7,10)
(3,8,9) (4,6,10) (4,7,9) (5,6,9) (5,7,8)
```

R12 (3*).

Enunţul problemei: Să se determine dacă un număr natural n este *perfect* (suma divizorilor proprii pozitivi + 1 este egală cu valoarea numărului). De exemplu, $n = 6$ este număr perfect deoarece $1 + 2 + 3 = 6$, $n = 28$ este număr perfect deoarece $1 + 2 + 4 + 7 + 14 = 28$.

Metoda de rezolvare: Avem de calculat o sumă, deci se poate considera variabila S care se iniţializează cu 0. Apoi, se parcurge mulţimea divizorilor posibili şi dacă se găseşte un divizor se adaugă la suma anterioară. La final, dacă $S + 1 = n$, atunci n este număr pefect, altfel nu este număr perfect.

Descrierea algoritmului în pseudocod:

```
citeşte n             *n≥1
S ← 0*suma adiv proprii
pentru i = 2,[n/2],1  repeta *posibilii divizori proprii
     daca n%i = 0 atunci    *i chiar este divizor al lui n
          S ← S + i         *il adun la S
daca S+1 = n atunci
     scrie "Este numar perfect"
altfel
     scrie "Nu este numar perfect"
```

Descrierea algoritmului în C++ (folosind CodeBlocks):
```
# include <iostream>
using namespace std;
int main()
{
 int i,n,s=1; //s este var. locala, declarata de tip int si
              //este initializata cu 1
 cout<<"n = "; cin>>n;
 for (i=2; i<=n/2; i++) //posibilii divizori proprii
   if (n%i == 0) //n se divide cu i => i este divizor al lui n
     s += i;    //se adauga i (divizorul) la s
 if (s == n) cout<<"Este numar perfect";
 else cout<<"Nu este numar perfect";
 return 0 ;
}
```

R13 (2*).

Enunţul problemei: Să se descrie un algoritm pentru determinarea tuturor pătratelor perfecte mai mici sau egale cu $n \in N^*$ dat.

Metoda de rezolvare: O metodă ar fi să parcurgem toate numerele de la 0 la n şi să testăm dacă acestea sunt pătrate perfecte sau nu ($\Leftrightarrow \sqrt{i} \overset{?}{\in} N \Leftrightarrow [\sqrt{i}] \overset{?}{=} \sqrt{i}$).

O altă metodă: i=? astfel încât $i^2 \leq n$ este echivalent cu i=? astfel încât $1 \leq i \leq [\sqrt{n}]$, astfel că este suficient să parcurgem toate numerele naturale i între 1 şi $[\sqrt{n}]$ şi să afişăm i^2.

Descrierea algoritmului în pseudocod:
```
citeste n  *presupunem n≥1
pentru i = 0, [√n], 1  repeta
     scrie i²
```

Descrierea algoritmului în C++ (CodeBlocks):
```
#include <iostream>
using namespace std;
int main()
{
 int n,i;
 cout<<"n="; cin>>n;
 cout<<"Patratele perfecte pana la n: ";
 for (i=0;i*i<=n;i++) //i*i<=n <=> i<=sqrt(n) si evitam cmath
   cout<<i*i<<" ";
 return 0;
}
```

Rulare:
```
n = 10 <Enter>
Patratele perfecte pana la n: 0 1 4 9
```
sau
```
n = 16 <Enter>
Patratele perfecte pana la n: 0 1 4 9 16
```

R14 (2*). (calculul sumelor)

Enunţul problemei: Pentru un întreg $n \geq 1$ dat, să se calculeze valoarea sumei $S = 1+2+3+...+n$, fără a folosi formula de calcul direct $S = n(n+1)/2$.

Metoda de rezolvare: Suma se poate scrie restrâns $S = \sum_{i=1}^{n} i$.

Algoritmul general pentru determinarea unei sume scris restrâns $S = \sum_{i=vi}^{vf} f(i)$ este următorul:

1. se iniţializează variabila S cu 0.
2. pentru $i = vi$, $vi+1$,.., vf se face $S \leftarrow S + f(i)$ (adică la vechea valoare a sumei se adaugă valoarea curentă din suma scrisă restrâns).

Descrierea algoritmului în pseudocod:

```
citeşte n
S ← 0                       *orice suma se initializeaza cu 0
pentru i=1,n  repeta
     S ← S + i              *la suma anterioara se aduna val. curenta
scrie S
```

Descrierea algoritmului în C++:

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

int n,i,S;  //orice var globala se initializeaza cu 0

int main()
{
 cout<<"Dati valoarea lui n (n>=1): "; cin>>n;
 /*S = 0;*/
 for (i=1;i<=n;i++)
      S += i; //sau S = S + i;
      //la suma anterioara se adauga i-ul curent
 cout<<"S = "<<S<<endl;
 return 0;
}
```

Dacă variabila S se declară global, iniţializarea variabilei S cu valoarea 0 nu mai este necesară pentru că orice variabilă globală declarată într-un program C/C++ se iniţializează automat cu 0. Acest lucru nu mai este valabil însă dacă variabila S este declarată local în funcţia "main" sau în alte funcţii.

R15 (2* suplimentar).

Enunţul problemei: Pentru un $n \geq 2$ întreg dat, să se calculeze sumei $S = 2 + 4 + 6 + ... + n$.

Metoda de rezolvare: Comparativ cu suma de la programul anterior, de data aceasta "for"-ul de la algoritmul de calcul al sumei are pasul 2 şi poate atinge marginea n sau nu, după cum n este par sau impar.

Descrierea algoritmului în pseudocod:

```
citeşte n
S ← 0
pentru i = 2,n,2  repeta
     //pornind de la 2, pana la maxim n din 2 in 2
   S ← S + i
scrie S
```

Descrierea algoritmului în pseudocod în C++(CodeBlocks):

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

int n,i,S; //orice variabila globala se initializeaza cu 0

int main()
{
 cout<<"Dati valoarea lui n (n>=2): ";  cin>>n;
 //S = 0;
 //n-ar mai fi necesara initializarea variabilei S cu 0
 for (i=2;i<=n;i+=2) //de la 2, pana la maxim n, din 2 in 2
      S += i; //adaug i-ul curent
 cout<<"S = "<<S<<endl;
 return 0;
}
```

R16 (2*). (determinarea valorii unui produs)

Enunţul problemei: Pentru un $n \geq 0$ întreg dat, să se calculeze $n!$.

Metoda de rezolvare: Cum $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$, putem scrie restrâns $n! = \prod_{i=1}^{n} i$.

Algoritmul general pentru determinarea unui produs scris restrâns $P = \prod_{i=vi}^{vf} f(i)$ este următorul:

1. se iniţializează variabila P cu 1
2. pentru $i = vi, vi+1, .., vf$ se face $P \leftarrow P \cdot f(i)$.

În cazul nostru, pentru că iniţializarea se face cu 1 şi această valoare reprezintă chiar 1! şi 0!, “for”-ul de la pasul 2 poate începe de la 2.

Descrierea algoritmului în pseudocod:

```
citeşte n        *presupunem n ≥ 0
P ← 1            *orice produs se initializeaza cu 1
┌pentru i = 2,n  repeta
└     P ← P*i    *produsul anterior se inmult. cu val. curenta i
scrie P
```

Descrierea algoritmului în C++ (folosind CodeBlocks):

Cum valoarea factorialului creşte foarte repede, trebuie acordată mare atenţie declarării variabilei care se foloseşte pentru memorarea factorialului:

- pentru $n = 8$ se depăşeşte valoarea 32767;
- pentru $n = 9$ se depăşeşte valoarea 65535;
- pentru $n = 13$ se depăşeşte valoarea 2147483647, respectiv 4294967295.
- pentru $n = 22$ se depăşeşte valoarea maximă a domeniului `unsigned long long`.

Astfel, se poate folosi tipul de date float, afişarea rezultatului făcându-se cu 0 zecimale.

```
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{ int n,i;
 float P;
 cout<<"Dati valoarea lui n (n>=0): "; cin>>n;

 P = 1; //orice produs se initializeaza cu 1
 for (i=2;i<=n;i++) P *= i; //sau P = P*i;
```

```
 cout<<"n! = "<<fixed<<setprecision(0)<<P; //cu fix zero zecimale
 return 0;
}
```

R17 (3*- suplimentar)

Enunţul problemei: Să se afişeze 1!, 2!, …, *n*!, pentru *n*≥1 întreg citit de la tastatură.

Metoda de rezolvare: O primă abordare ar fi aceea că pentru *k* de la 1 la *n* se poate calcula *k*! pornind de la valoarea 1.

Descrierea algoritmului în pseudocod:

```
citeşte n              *n ≥ 1
pentru k = 1,n,1  repeta *pentru k=1,2,...,n calculam k!
      k_fact ← 1       *orice factorial se initializeaza cu 1
      pentru i = 2,k,1  repeta
            k_fact ← k_fact*i
      scrie k_fact
```

Descrierea algoritmului în C++:

```
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{int n,i,k;
 float k_fact;
 cout<<"Dati valoarea lui n (n>=1): "; cin>>n;
 for (k=1;k<=n;k++) //pentru fiecare numar de la 1 la n
  {               //calculam si afisam k!
   k_fact = 1;    //orice produs se initializeaza cu 1
   for (i=2;i<=k;i++)
      k_fact *= i; //inmultim cu 2,3,…,k
   cout<<k<<"! = "<<fixed<<setprecision(0)<<k_fact<<endl;
  }
 return 0;
}
```

sau folosind funcția factorial

```
#include<iostream>
#include<iomanip>
using namespace std;
float Factorial(int n)
{   float k_fact = 1; //orice produs se initializeaza cu 1
    for (int i=2;i<=n;i++)
        k_fact *= i; //inmultim cu 2,3,…,n
    return k_fact;
}
int main()
{int n,i,k;
 cout<<"Dati valoarea lui n (n>=1): "; cin>>n;
 for (k=1;k<=n;k++) //pentru fiecare numar de la 1 la n
   cout<<k<<"! = "<<fixed<<setprecision(0)<<Factorial(k)<<endl;
 return 0;
}
```

Dar ținând cont că la un moment dat am calculat (*k*-1)!, următoarea valoare *k*! se poate determina folosind valoarea anterioară (*k*! = (*k*-1)! · *k*). Așasar, de fiecare dată când calculăm un factorial (care inițial este 1), înmulțim cu valoarea *k* curentă.

Descrierea algoritmului în C++:

```
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{int n,i,k;
 float k_fact=1; //0!=1, apoi folosin k! = (k-1)!*k
 cout<<"Dati valoarea lui n (n>=1): "; cin>>n;
 for (k=1;k<=n;k++) //pentru fiecare numar de la 1 la n
  {              //calculam si afisam k!
    k_fact *= k; //pt. k! curent, inmultim val. anterioara cu k
    cout<<k<<"! = "<<fixed<<setprecision(0)<<k_fact<<endl;
  }
 return 0;
}
```

Rulare:

```
Dati valoarea lui n (n>=1): 8
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
```

R18 (4*-suplimentar).

Enunţul problemei: Să se calculeze P_n, A_n^k şi C_n^k, pentru n, $k \geq 0$, $n \geq k$ întregi, citite de la tastatură.

Metoda de rezolvare: Reamintim că:

$$\boxed{P_n} = 1 \cdot 2 \cdot \ldots \cdot n = n! = \boxed{\prod_{i=1}^{n} i}$$

$$\boxed{A_n^k} = \frac{n!}{(n-k)!} = (n-k+1) \cdot (n-k+2) \cdot \ldots \cdot n = \boxed{\prod_{i=n-k+1}^{n} i}$$

$$\boxed{C_n^k} = \frac{\cancel{n!}}{k!\cancel{(n-k)!}} = \frac{(n-k+1)(n-k+2)\cdot \ldots \cdot n}{1 \cdot 2 \cdot \ldots \cdot k} = \boxed{\frac{A_n^k}{k!}}$$

Descrierea algoritmului în pseudocod:

```
citeşte n,k           *n,k≥0, n≥k
perm ← 1
pentru i = 2,n,1 repeta    *nu mai inmultim perm si cu 1
     perm ← perm*i
aranj ← 1
pentru i = n-k+1,n,1 repeta
     aranj ← aranj*i
k_fact ← 1                 *k! ne trebuie la combinari
pentru i = 2,k,1 repeta
     k_fact ← k_fact*i
comb ← aranj/k_fact
scrie perm,aranj,comb
```

Descrierea algoritmului în C+:

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

```
int main()
{
 int n,i,k;
 float perm,aranj,comb,k_fact;
 cout<<"Dati valoarea lui n (n>=0): ";
 cin>>n;
 cout<<"Dati valoarea lui k (k>=0 si k<=n): ";
 cin>>k;

 perm = aranj = k_fact = 1;   //atribuire multipla - se exec de la
 //dreapta la stanga <=> k_fact = 1; aranj = k_fact; perm = aranj;

 for (i=2;i<=n;i++) perm *= i;
 cout<<"Permutari de "<<n<<":  "<<perm<<endl;

 for (i=n-k+1;i<=n;i++) aranj *= i;
 cout<<"Aranjamente de "<<n<<" luate cate "<<k<<": "<<aranj<<endl;

 for (i=2;i<=k;i++) k_fact *= i;
 comb = aranj / k_fact;

 cout<<"Combinari de "<<n<<" luate cate "<<k<<": "<<comb<<endl;
 return 0;
}
```

R19 (3*).

Enunţul problemei: Pentru un număr întreg $n \geq 0$ citit de la tastatură, să se stabilească dacă este număr prim.

Metoda de rezolvare: Prin definiţie, un întreg natural este număr prim dacă are numai doi divizori care aparţin lui **N** (1 nu se este număr prim).
Aşadar, o metodă de a determina dacă un număr $n \in \mathbf{N}$ este prim este de a parcurge mulţimea divizorilor proprii *posibili*, adică $\{2, 3, \ldots, n/2\}$ şi dacă se găseşte unul ca fiind divizor al lui n se decide că n nu este prim şi se opreşte parcurgerea; altfel acesta este prim.

```
citeşte n   *n>=0
┌daca n=0 sau n=1 atunci
│     scrie "Nu este numar prim"
└     iesire
prim ← adevarat       * presupunem ca n este prim
┌pentru i = 2,[n/2],1 repeta*posibilii div proprii
│    ┌daca n%i = 0 atunci    *n se divide cu i
└    └     prim ← fals      *si iesire din "for" cu break
┌daca prim = adevarat atunci
│     scrie "Numarul este prim"
│altfel
└     scrie "Numarul nu este prim"
```

Există o variantă optimizată a metodei anterioare în ceea ce locul priveşte considerarea tuturor divizorilor proprii: pentru a stabili dacă un număr natural n este prim este suficient să parcurgem mulţimea $\{2, 3, \ldots, [\sqrt{n}]\}$ şi să verificăm dacă vreunul este divizor. De exemplu, pentru n=12 nu este necesar să considerăm toti divizorii proprii, ci doar 2 (si 3), iar pentru n=13 se poate decide doar dacă se consideră doar multimea $\{2, 3\}$.

```
citeşte n             *n≥0
┌daca n=0 sau n=1 atunci
│     scrie "Nu este numar prim"
└     iesire
┌daca n=3 atunci
│     scrie "Este numar prim"
└     iesire
```

```
daca n%2=0 atunci     *singurul numar prim par e 2
     daca n=2 atunci
          scrie "Este numar prim"
     altfel
          scrie "Nu este numar prim"
          iesire
altfel
     prim ← adevarat      * presupunem ca n este prim
     pentru i = 3,√n,2 repeta
          daca n%i = 0 atunci
               prim ← fals     *si iesire din "for"
     daca prim = adevarat atunci
          scrie "Numarul este prim"
     altfel
          scrie "Numarul nu este prim"
```

În general, pentru problemele care verifică o *proprietate* de genul:

proprietate = adevărată dacă pentru orice *i* din mulţimea *I* se verifică relaţia *p*.

⬄

proprietate = falsă dacă există *i* din mulţimea *I* pentru care nu se verifică relaţia *p*.

În cazul anterior,

n este număr prim dacă $(\forall)i \in \{2,3,...,\sqrt{n}\} : n\%i \neq 0$

⬄ n nu este număr prim dacă $(\exists)i \in \{2,3,...,\sqrt{n}\} : n\%i = 0$.

De aceea, se poate proceda astfel:

```
proprietate ← adevarat
pentru i din M
     daca relatia p nu este verificata pentru i atunci
          proprietate ← fals
```

Descrierea algoritmului în C++:

```
# include <iostream>
//# include <cmath> //pentru sqrt
using namespace std;
int main()
{
 int n,i,prim;
 cout<<"n="; cin>>n;
 if (n==0 || n==1) prim = 0;
 else
  {
   prim = 1; //presupunem ca n este prim
   for (i=2; i*i<=n; i++)//multimea posibililor div. proprii
   //i<=sqrt(n) <=> i*i<=n si nu mai este necesar fisierul cmath
    if (n%i == 0)  //am gasit un divizor propriu
     {
      prim = 0; //este clar, n nu este prim
      break; //se opreste trecerea la alt i
       //(break = iesire din ciclul curent (aici, for))
     }
  }
 if (prim==1) cout<<"Numarul dat este prim"<<endl;
 else cout<<"Numarul dat nu este prim"<<endl;
 return 0;
}
```

sau, folosind funcţii

```
# include <iostream>
# include <cmath>
using namespace std;
int Prim(int n)
{//se testeaza daca parametrul global n este prim sau nu
 if (n<=1) return 0; //orice n<=1 nu este prim
           //se iese din functie cu valoarea returnata 0
 for (int i=2; i*i<=n; (i==2)?i=3:(i+=2))
 //cand trec la urmatoarea valoare:
 //daca i este 2 atunci trec la 3, altfel trec la i+2
    if (n%i == 0)  return 0;
 return 1; //daca nu s-a iesit pana acum cu valoarea 0, atunci
 //n este prim si valoarea returnata este 1
}
int main()
{int n;
 cout<<"n="; cin>>n;
 if (Prim(n))   //sau (Prim(n)==1)
    cout<<"Numarul dat este prim"<<endl;
 else cout<<"Numarul dat nu este prim"<<endl;
 return 0;
}
```

Funcția *Prim* poate fi îmbunătățită, scotând din regula/calculul general și numerele pare (doar 2 este prim, restul nu sunt prime) etc.

```
int Prim(int n)
{
 if (n<=1) return 0; //orice n<=1 nu este prim
           //se iese din functie cu valoarea returnata 0
 if (n==2) return 1; //2 este singurul nr. par prim
 if (n%2==0) return 0; //orice alt numar par nu este prim
 //daca s-a ajuns aici inseamna ca n>=3
 for (int i=3; i*i<=n; i+=2) //posibilii divizori: de la 3 din 2 in 2
    if (n%i == 0)  return 0; //cum am gasit un divizor propriu,
   //este clar, nu este prim si se iese din functie cu valoarea 0
 return 1; //daca nu s-a iesit pana acum cu valoarea 0, atunci
 //n este prim si valoarea returnata este 1
}
```

R20 (3*).

Enunţul problemei: Să se determine toate numerele prime mai mici sau egale decât un număr natural $n \geq 2$ dat.

Metoda de rezolvare:De exemplu, pentru n=10 numerele prime mai mici sau egale decât n sunt 2, 3, 5, 7. Vom parcurge toate numerele de la 2 (cel mai mic număr prim) până la n şi testăm primalitatea numărului curent. În C++, primalitatea se poate testa prin apelarea unei funcţii.

O descriere a algoritmului în C++:

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

```
int Prim(int n)
{
 //if (n<=1) return 0; //orice n<=1 nu este prim
           //se iese din functie cu valoarea returnata 0
 if (n==2) return 1; //2 este singurul nr. par prim
 if (n%2==0) return 0; //orice alt numar par nu este prim
 //daca s-a ajuns aici inseamna ca n>=3
 for (int i=3; i*i<=n; i+=2) //posibilii divizori: de la 3 din 2 in 2
    if (n%i == 0)  return 0; //cum am gasit un divizor propriu,
    //este clar, nu este prim si se iese din functie cu valoarea 0
 return 1; //daca nu s-a iesit pana acum cu valoarea 0, atunci
 //n este prim si valoarea returnata este 1
}
int main()
{
 int n;
 cout<<"n="; cin>>n;
 cout<<"Numerele prime pana la n sunt: ";
 for (int i=2; i<=n; i++) //pornind de la 2, din 1 in 1
   if (Prim(i)) //sau if (Prim(i)==1)
     cout<<i<<" ";
 return 0;
}
```

R21 (4*- suplimentar**).**

Enunţul problemei: Descrieţi un algoritm pentru determinarea primului număr prim mai mare sau egal decât un $n \in N^*$ citit de la tastatură.

Metoda de rezolvare: De exemplu, pentru n = 10 primul număr prim mai mare decât n este 11, pentru n = 13 primul număr prim mai mare decât n este 13, iar pentru n = 14 primul număr prim mai mare decât n este 17. Ideea ar fi să pornim de la n şi ne oprim dacă am dat de un număr prim, altfel mergem mai departe.

Descrierea algoritmului în C++:

```
#include <iostream>
using namespace std;
int Prim(int n)
{
 if (n<=1) return 0; //orice n<=1 nu este prim
           //se iese din functie cu valoarea returnata 0
 if (n==2) return 1; //2 este singurul nr. par prim
 if (n%2==0) return 0; //orice alt numar par nu este prim
 //daca s-a ajuns aici inseamna ca n>=3
 for (int i=3; i*i<=n; i+=2) //posibilii divizori: de la 3 din 2 in 2
    if (n%i == 0)  return 0; //cum am gasit un divizor propriu,
    //este clar, nu este prim si se iese din functie cu valoarea 0
 return 1; //daca nu s-a iesit pana acum cu valoarea 0, atunci
 //n este prim si valoarea returnata este 1
}

int main()
{
 int n,i;
 cout<<"n="; cin>>n;
 for (i=n ; Prim(i)==0 ; i++) ;
 //pornind de la i=n, cat timp i nu este prim,
```

```
 //nu fac nimic ( ; = instructiune vida) si avanzez din 1 in 1
 cout<<"Primul nr. prim >= "<<n<<" este: "<<i<<endl;
 return 0;
}
```

Observaţie: Efectul instrucţiunilor

```
for (i=n ; Prim(i)==0 ; i++) ;
cout<<"Primul nr. prim >= "<<n<<" este: "<<i<<endl;
```

este echivalent cu

```
int i=n;
while (Prim(i)==0)    //cat timp i-ul curent nu este prim
   i++;               //trecem mai departe
cout<<"Primul nr. prim >= "<<n<<" este: "<<i<<endl;
```

sau

```
for (i=n ; ; i++) //pornind de la n, din 1 in 1
  if (Prim(i)) //sau (Prim(i)==1) <=> n-am gasit element prim
   {
    cout<<"Primul nr. prim >= "<<n<<" este: "<<i<<endl;
    break;
   }
```

Rulare:

```
n=14 <Enter>
Primul numar prim mai mare ca 14 este: 17
```

R22 (3-4*- suplimentar).

Enunţul problemei: Descrieţi un algoritm pentru determinarea primelor $n \in N^*$ numere prime. De exemplu, pentru *n*=5 (primele 5 numere prime) se va afişa 2, 3, 5, 7, 11.

Metoda de rezolvare: Ideea ar fi să pornim de la 2 care este cel mai mic număr prim şi să numărăm doar numele prime.

Descrierea algoritmului în C++:

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

int Prim(int n)
{
 if (n<=1) return 0;
 if (n==2) return 1;
 if (n%2==0) return 0;
 for (int i=3; i*i<=n; i+=2)
    if (n%i == 0)  return 0;
 return 1;
}
int main()
{
 int n,i,contor=0; //initializam contorul cu 0
 cout<<"n="; cin>>n;
 cout<<"Primele "<<n<<" numere prime sunt: ";
 for (i=2 ; contor < n ; i++)
 //pornind de la 2, cat timp n-am gasit cele n numere prime
    if (Prim(i)) //numarul curent este prim
    {
        contor++;        //il numaram
        cout<<i<<" ";   //il afisam
    }
 cout<<endl;
 return 0;
```

```
}
```

Observaţie: Instrucţiunile

```
for (i=2 ; contor < n ; i++)
 //pornind de la 2, cat timp n-am gasit cele n numere prime
    if (Prim(i)) //numarul curent este prim
    {
        contor++;        //il numaram
        cout<<i<<" ";   //il afisam
```

sunt echivalente cu

```
i=2;                    //pornind de la 2
while (contor < n)      //cat timp n-am gasit cele n numere prime
{
    if (Prim(i))        //numarul curent este prim
    {
        contor++;       //il numaram
        cout<<i<<" ";   //il afisam
    }
    i++; //oricum trec la urmatorul numar
}
```

R23 (3-4*-suplimentar).

Enunţul problemei: Determinaţi un algoritm pentru determinarea tuturor variantelor de scriere a unui număr întreg dat ca sumă de două numere prime; în cazul în care nu există nicio descompunere se va afişa un mesaj.

Metoda de rezolvare: De exemplu, n = 12 = 5+7, iar n = 10 = 3+7 = 5+5, iar pentru n = 23 se va afişa mesajul „nu se descompune”.

Vom căuta să descompunem n în suma dintre două numere prime: i şi n-i. Cea mai mică valoare a lui i este 2 pentru că acesta este cel mai mic divizor propriu, iar valoarea cea mai mare este aceea pentru care cele două componente sunt ordonate: $i \le n-i \Leftrightarrow i \le n / 2$.

Pentru cazurile când nu există nicio descompunere, putem lua o variabilă în care să memorăm valoarea 1 în care am găsit cel puţin o decompunere şi valoarea 0 în cazul în care n există nicio descompunere. Putem iniţializa această variabilă cu 0, iar în cazul găsirii unei descompuneri să schimbăm valoarea variabilei la valoarea 1. La final, dacă variabila a rămas cu valoarea iniţializată 0, atunci se afişează un mesaj.

O descriere a algoritmului în C++:

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

int Prim (int n)
{//nu testez pentru n<=1 pt. ca nu apelam pentru astfel de val.
 for (int i=2;i*i<=n;(i==2)?i=3:(i+=2))
   if (n%i==0) return 0;
 return 1;
}
int main() {
 int n;
 bool OK = false; //initializam variabila OK cu false
 //ca si cum am presupune ca nu exista nicio descompunere
 cout<<"n="; cin>>n;
```

```
     for (int i=2;i<=n/2; i++)
      if (Prim(i)==1 && Prim(n-i)==1)  {
        cout<<n<<" = "<<setw(2)<<i<<" + "<<n-i<<endl;
        OK = true; //am gasit o descompunere
       }
     if (OK == false) //daca nu am gasit nicio descompunere => mesaj
      cout<<"Nu se poate descompune"<<endl;
     return 0;
    }
```

Rulare:

```
n=11 <Enter>
Nu se poate descompune
```

sau

```
n=24 <Enter>
24 = 5 + 19
24 = 7 + 17
24 = 11 + 13
```

R24 (3*-suplimentar).

Enunţul problemei: Descrieţi un algoritm pentru calculul sumei $S = \sum_{k=1}^{n} k!$, pentru $n \geq 1$. De exemplu, pentru $n = 4$: $S = 1! + 2! + 3! + 4! = 1 + 2 + 6 + 24 = 33$.

Metoda de rezolvare: Avem de calculat o sumă de factoriale, deci o sumă de produse. Folosind algoritmii clasici de determinarea unei sume şi a unui produs, un prim algoritm constă în: iniţializarea sumei cu 0, apoi pentru i de la 1 la n se adună la suma anterioară $i!$, care se poate calcula clasic ca un produs: orice produs se iniţializează cu 1, apoi pentru $k = 2$ la n înmulţim produsul anterior cu k (k începe de la 2 şi nu de la 1 pentru a nu mai înmulţi cu 1).

Descrierea algoritmului 1 în pseudocod:

```
citeste n                        *presupunem n≥1
S ← 0
┌pentru k = 1, n  repeta
│     kfact ← 1
│    ┌pentru i = 2, k  repeta     *i=1 a fost omis
│    └    kfact ← kfact * i       *pentru a nu mai inmulti cu 1
└     S ← S + kfact
scrie S
```

Algoritmul anterior calculează fiecare factorial de la capăt ($1 \cdot 2 \cdot \ldots \cdot k$). Dar ţinând cont că la un moment dat, adăugăm un factorial, la următorul nu ar trebui decât să înmulţim cu k curent ($k! = (k-1)! \cdot k$). Acest lucru se aplică la orice algoritm în care termenul curent al sumei se poate scrie (în întregime sau parţial) în funcţie de termenul anterior (în întregime sau parţial):

$$S = 1! + 2! + 3! + ... + n! = \sum_{k=1}^{n} k!,\ \text{iar}\ k! = (k-1)! \cdot k,\ \text{pentru}\ k \geq 1\ \text{(reamintim } 0! = 1\text{)}.$$

Descrierea algoritmului 2 în pseudocod:

```
citeste n                        *presupunem n≥1
S ← 0                            *initializarea sumei cu 0
kfact ← 1  *initializarea termenului curent al sumei  cu 1
┌pentru k = 1, n  repeta
│     kfact ← kfact * k
└     S ← S + kfact
```

```
scrie S
```

Se mai poate optimiza algoritmul anterior, prin iniţializarea sumei cu primul termen, apoi k începe de la valoarea 2.

Descrierea algoritmului în C++ (folosind CodeBlocks):

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{int n,k;
 float S=1, kfact=1;  //declarare si initializare
 //am inclus in suma deja 1!
 //factorilele este bine sa fie memorate in tip de date float
 cout<<"n="; cin>>n; //presupunem ca
 for (k=2;k<=n;k++)  //pornim cu k de la 2
 {  kfact *= k;  //factorialul anterior se inmulteste cu k
    S += kfact; //se adauga la suma
 }
 cout<<"S = 1!+2!+...+n! = "<<fixed<<setprecision(0)<<S<<endl;
 return 0;
}
```

R25 (suplimentar).

Enunţul problemei: Descrieţi un algoritm pentru calculul sumei $S = \sum_{k=1}^{n} k \cdot a^k$, pentru $n \in \boldsymbol{N}^*$ şi $a \in \boldsymbol{R}$ date. De exemplu, pentru $n = 3$ şi $a = 2$ avem $S = 1\cdot 2^1 + 2\cdot 2^2 + 3\cdot 2^3 = 34$.

Metoda de rezolvare: Avem de calculat o sumă de puteri (a^k) înmulţite cu k. Având în vedere observaţiile din algoritmul anterior, aici putem calcula parte din termenul general în funcţie de parte din termenul general anterior: $a^k = a^{k-1}\cdot a$, pentru $k\geq 1$).

Descrierea algoritmului în pseudocod:

```
citeste n,a                    *presupunem n≥1
S ← 0                          *initializarea sumei cu 0
ak ← 1                         *initializarea puterii cu 1
pentru k = 1, n  repeta
     ak ← ak * a
     S ← S + k*ak
scrie S
```

Descrierea algoritmului în C++ (CodeBlocks):

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

int main()
{
 int n,k;
 float S=0,ak=1,a;
 //functie putere care creste repede => memorare in "float"
 //factorilele este bine sa fie memorate in tip de date float
 cout<<"n="; cin>>n; //pp. n>=1
 cout<<"a="; cin>>a;
 for (k=1;k<=n;k++)
 {
      ak *= a;   //puterea anteioara se inmulteste cu a
      S += k*ak;
 }
```

```
    cout<<"S = "<<fixed<<setprecision(0)<<S<<endl;
    return 0;
   }
```

Rulare:

$n = 3$
$a = 2$
S = 34

R26 (4*-suplimentar).

Enunţul problemei: Să se scrie un algoritm pentru calculul mediei aritmetice, geometrice şi armonice a tuturor divizorilor naturali unui număr natural nenul dat.

Metoda de rezolvare: De exemplu,

- pentru n = 6 divizorii naturali sunt {1, 2, 3, 6} şi mediile sunt: $m_a = \frac{1+2+3+6}{4}$=3, $m_g = \sqrt[4]{1\cdot 2\cdot 3\cdot 6}$ = 2.45 şi $m_{arm} = \frac{4}{\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\frac{1}{6}}$ = 2.
- pentru n = 5 divizorii naturali sunt {1, 5} şi mediile sunt: $m_a = \frac{1+5}{2}$=3, $m_g = \sqrt{1\cdot 5}$ ~ 2,24 şi $m_{arm} = \frac{2}{\frac{1}{1}+\frac{1}{5}}$ = 1,67.
- pentru n = 1 singurul divizor natural este 1 şi mediile sunt: $m_a = m_g = m_{arm}$ = 1 (la media geometrică radicalul este calculat ca ridicarea la puterea 1).

În general, pentru un n natural nenul, notând numărul divizorilor naturali ai lui n cu *nr_div*, sumele se pot calcula astfel:

$$\boxed{m_a = \frac{\sum\limits_{\substack{i/n\\ i\in N}} i}{nr\_div}}$$

(suma divizorilor naturali ai lui n împărţită la numărul de divizori naturali)

$$\boxed{m_g = \sqrt[nr\_div]{\prod\limits_{\substack{i/n\\ i\in N}} i}}$$

(radical de ordinul numărul divizorilor din produsul divizorilor naturali ai lui n)

$$\boxed{m_{arm} = \frac{nr\_div}{\sum\limits_{\substack{i/n\\ i\in N}} \frac{1}{i}}}$$

(raportul dintre numărul divizorilor naturali ai lui n şi suma inverselor divizorilor)

Pentru calculul mediilor, la început putem folosi variabila m_a pentru calculul sumei divizorilor, variabila m_g pentru calculul produsului divizorilor şi variabila m_{arm} pentru calculul sumei inverselor divizorilor. Apoi, se fac rectificările împărţind suma calculată în variabila m_a la numărul de divizori, calculând radical de ordin *nr_div* din produsul calculat în m_g, respectiv împăţind *nr_div* la suma calculată în m_{arm}.

O descriere a algoritmului în pseudocod:

```
citeşte n
ma ← 0 *suma divizorilor se initializeaza cu 0
mg ← 1 *produsul divizorilor se initializeaza cu 0
marm ← 0 *suma inverselor divizorilor se initializeaza cu 0
nr_div ← 0 *contorul divizorilor se initializeaza cu 0
pentru i = 1,n repeta *se parcurg posibilii divizori
 daca n se divide cu i atunci
    nr_div ← nr_div + 1
    ma ← ma + i *se adauga divizorul curent la suma
    mg ← mg * i *se inmulteste divizorul curent la produs
    marm ← marm + 1/i *se adauga inversul lui i la suma
ma ← ma / nr_div
mg ← nr_div√mg
marm ← nr_div/ marm

scrie ma, mg, marm
```

O descriere a algoritmului în C++ (CodeBlocks):

```
#include <iostream>
#include <iomanip>    //pentru setprecision
#include <math.h>     //pentru pow
using namespace std;
int main()
{
 int n,i,nr_div;
 float ma,mg,marm;
 cout<<"n="; cin>>n;
 ma = marm = nr_div = 0; //operatorul de atribuire multipla
 mg = 1;
 for (i=1;i<=n;i++)  //posibilii divizori (toti)
  if (n%i==0) //restul impartirii lui n la i este zero
              //adica i este divizor al lui n
  {
   nr_div ++; //numaram divizorul curent i
   ma += i; //adaugam divizorul curent la suma din ma
   mg *= i; //adaugam divizorul curent la produsul din mg
   marm += 1.0/i; //adaugam inversul lui i la suma
  }
 //urmeaza "rectificarile":
 ma /= nr_div; //impartim suma curenta la numarul divizorilor
 mg = pow(mg,1.0/nr_div);
 //radicalul se poate calcula ca fiind puterea 1/nr_div
 marm = nr_div / marm;
 cout<<"Media aritmetica  = "<<setprecision(2)<<ma;
 cout<<endl<<"Media geometrica  = "<<mg;
 cout<<endl<<"Media armonica  = "<<marm<<endl;
 return 0;
}
```

Temă suplimentară

1. Descrieţi un algoritm pentru a afişa toate numerele de 2 cifre (între 10 şi 99) care au cel puţin o cifră de 5 (unităţile sau zecile) şi sunt divizibile cu 17. Soluţie: 51 şi 85.
2. Descrieţi un algoritm pentru pentru determinarea tuturor numerelor $\overline{a23a}$ care se împart exact la 6. Soluţie: 2232, 8238. Sugestie: se pot parcurge toate variantele cifrei *a* şi se construieşte numărul astfel a*1000 + 230 + a.
3. Să se găsească un algoritm pentru afişarea tuturor numerelor dintre 100 şi 599 ce au cifrele în ordine crescătoare şi suma cifrelor egală cu 18. Soluţie: 189, …, 567. Sugestie: în C/C++, pentru un număr de 3 cifre: cifra unităţilor = n%10, cifra zecilor = (n%100)/10, cifra sutelor = n / 100.
4. Descrieţi un algoritm ce stabileşte dacă un număr întreg $n \geq 1$ dat este deficient (suma divizorilor proprii+1 < n) sau abundent (suma divizorilor proprii + 1 > n). De exemplu, n=12 este abundent deoarece 1+2+3+4+6=16>12, iar n=14 este deficient deoarece 1+2+7<14.
5. Descrieţi un algoritm ce stabileşte dacă două numere întregi sunt prietene (1 + suma divizorilor proprii ai unuia = celălalt). De exemplu 220 şi 284 sunt prietene, deoarece sd(220) = 1 + (2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110) = 284 şi sd(284) = 1 + (2 + 4 + 71 + 142) = 220.
6. Descrieţi un algoritm pentru calculul sumei $S = 1\cdot 3 + 2\cdot 4 + \ldots + n(n+2)$, pentru $n \geq 1$.
7. Descrieţi un algoritm pentru a calcula $S = 1 + 1/2 + 1/3 + \ldots + 1/n$, pentru $n \in \mathbf{N}^*$ dat.
8. Descrieţi un algoritm pentru a calcula m^n prin înmulţiri repetate. $m^n = \underbrace{m \cdot \ldots \cdot m}_{de\, n\, ori} = \prod_{i=1}^{n} m$
9. Să se descrie un algoritm pentru a afişa divizorii primi ai unui număr natural dat.
10. Să se descrie un algoritm pentru afişarea tuturor numerelor prime de 3 cifre, care citite invers sunt tot numere prime.
11. Să se descrie un algoritm pentru a stabili dacă două numere sunt gemene (p şi q sunt 2 numere gemene dacă ambele sunt numere prime şi $q - p = 2$). De exemplu, 3 şi 5, 5 şi 7, 11 şi 13, 17 şi 19, 29 şi 31 sunt numere gemene.
12. Să se descrie un algoritm pentru a determina câte numere prime sunt mai mici sau egale decât n. De exemplu, pentru n=6 sunt 3 numere prime (2, 3 şi 5).
13. Să se descrie un algoritm pentru a determina primele 10 numere prime de forma $4k$-1, k număr natural.
14. Să se calculeze valoarea finală a unui depozit după n ani, știind suma S inițială din depozit și rata dobânzii. Se presupune că nu există niciun comision.
15. Să se înlocuiască literele cu cifre în scăderea următoare:

```
a b c b e –
  e d  a b
----------
  e  b c e
```

Pentru avansați:

1) Generarea tuturor pătratelor perfecte mai mici sau egale decât un prag n dat (x este număr perfect dacă este egal cu 1 + suma divizorilor săi proprii).
2) Descompunerea în factori primi a unui număr natural nenul dat.
3) Determinarea divizorilor primi ai unui număr natural dat.
4) Pentru un n dat, să se determine între ce valori Fibonacci consecutive se găsește.
5) Aplicarea Ciurului lui Eratostene pentru determinarea numerelor prime până la un prag n dat, cu afișarea, la fiecare pas, a elementelor rămase în șir. (Se scriu toate elementele de la 2 la n, apoi pornind de după 2 din 2 în 2 se elimină elemente, se continuă pornind de după 3 din 3 în 3 se elimină elemente …).