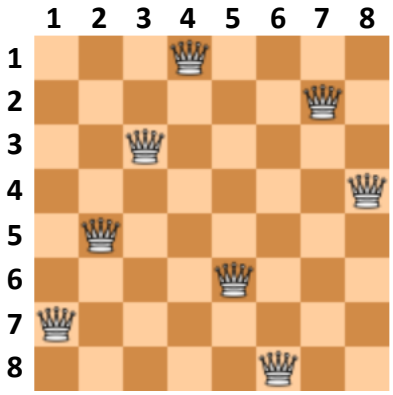
TEHNICA DE PROGRAMARE "BACKTRACKING" (continuare)

1. Problema celor n regine

Fiind dată o tablă de şah de dimensiune $n \times n$, problema cere să se determine toate modurile în care pot fi plasate n regine pe tablă astfel încât oricare două să nu se atace între ele. Două regine se atacă pe tabla de şah dacă se află pe aceeaşi linie, coloană sau diagonală. De exemplu, pentru $n = 8$, o posibilă soluţie dintre cele 92 existente, este următoarea (sursa: https://en.wikipedia.org/wiki/Eight_queens_puzzle):



Problema a fost formulată de către creatorul de probleme şahistice Max Bezzel în 1848 pentru $n = 8$. În 1850 Franz Nauck a publicat primele soluţii ale problemei şi a generalizat-o pentru orice număr natural $n \geq 4$ (pentru $n \leq 3$ problema nu are soluţii), ulterior ea fiind analizată de mai mulţi matematicieni (e.g., C. F. Gauss) şi informaticieni (e.g., E.W. Dijkstra) celebri.

Problema poate fi rezolvată prin mai multe metode, astfel:

- considerând reginele numerotate de la 1 la n^2, generăm, pe rând, toate cele $C_{n^2}^n$ submulţimi formate n regine şi apoi le testăm (de exemplu, pentru $n = 8$ vom genera şi testa $C_{64}^8 = 4.426.165.368$ submulţimi). Evident, această metodă de tip forţă-brută este foarte ineficientă, deoarece vom genera şi testa inutil foarte multe submulţimi care sigur nu pot fi soluţii (de exemplu, toate submulţimile care cuprind cel puţin două regine pe aceeaşi linie, coloană sau diagonală);
- observând faptul că pe o linie se poate poziţiona exact o regină, vom genera, pe rând, toate cele n^n tupluri conţinând coloanele pe care se află reginele de pe fiecare linie şi le vom testa (de exemplu, pentru $n = 8$ vom genera şi testa $8^8 = 16.777.216$ tupluri). Deşi această metodă, tot de tip forţă-brută, este de aproximativ 260 de ori mai rapidă decât precedenta, tot va genera şi testa inutil multe tupluri care nu pot fi soluţii (de exemplu, toate tuplurile care conţin cel puţin două valori egale, deoarece acest lucru înseamnă faptul că mai mult de două regine se află pe aceeaşi coloană, deci se atacă între ele);
- observând faptul că pe o linie şi o coloană se poate poziţiona exact o regină, vom genera, pe rând, utilizând metoda Backtracking, toate cele $n!$ permutări cu n elemente, testând la fiecare pas şi condiţia ca reginele să nu se atace pe diagonală (de exemplu, pentru $n = 8$ vom genera şi testa $8! = 40.320$ permutări). Evident,

această metodă este mult mai eficientă decât primele două, fiind de aproximativ 420 de ori mai rapidă decât a doua metodă și de peste 110000 de ori decât prima!

n = 4

	1	2	3	4
1		R	Q	
2	Q			R
3	R			Q
4		Q	R	

`s[i] = coloana pe care se află regina de pe linia i`

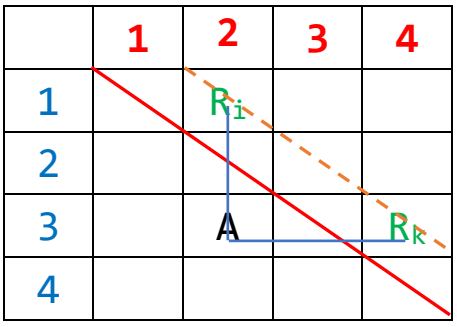
`s = (1, 1, 1, 1), ..., s = (1, 2, 1, 3),..., s = (4, 4, 4, 4)`

s_Q = (3, 1, 4, 2) este soluția simetrică pentru s_R = (2, 4, 1, 3)

În continuare, vom detalia puțin cea de-a treia variantă de rezolvare prezentată mai sus, bazată pe metoda Backtracking.

Revenind la observațiile generale de la metoda Backtracking, acestea se vor particulariza, astfel:

- `s[k]` reprezintă coloană pe care este poziționată regina de pe linia `k`. De exemplu, soluției prezentate în figura de mai sus îi corespunde tuplul $s = (4,7,3,8,2,5,1,6)$;
- deoarece pe o linie `k` regina poate fi poziționată pe orice coloană `s[k]` cuprinsă între `1` și `n`, obținem $\min_k$=1 și $\max_k$=n;
- `succ(v)=v+1`, deoarece valorile posibile pentru `s[k]` sunt numere naturale consecutive cuprinse între `1` și `n`;
- `s[1],…,s[k-1],s[k]` este soluție parțială dacă regina curentă R_k(k,s[k]), adică regina aflată pe linia `k` și coloana `s[k]`, nu se atacă pe coloană sau diagonală cu nicio regină anterior poziționată pe o linie `i` și o coloană `s[i]`, pentru orice `i`∈{1,…,k-1}. Condiția referitoare la coloană se deduce imediat, respectiv trebuie ca `s[k]`≠`s[i]` pentru orice `i`∈{1,2,…,k-1}. Condiția referitoare la diagonală se poate deduce, de exemplu, astfel: regina $R_k(k, s[k])$ se atacă pe diagonală cu o altă regină $R_i(i, s[i])$ dacă și numai dacă dreapta R_kR_i este paralelă cu una dintre cele două diagonale ale tablei de șah.



$R_i(i, s[i]) \Rightarrow AR_i = |i-k|$
$R_k(k, s[k]) \Rightarrow AR_k = |s[i]-s[k]|$

- pentru a testa că `s[1],…,s[k]` este soluție vom ține cont de faptul că `s[1],…,s[k]` este soluție parțială, deci nu vom retesta condițiile de continuare, ci doar condiția suplimentară `k=n`.

Implementare în limbajul C++:

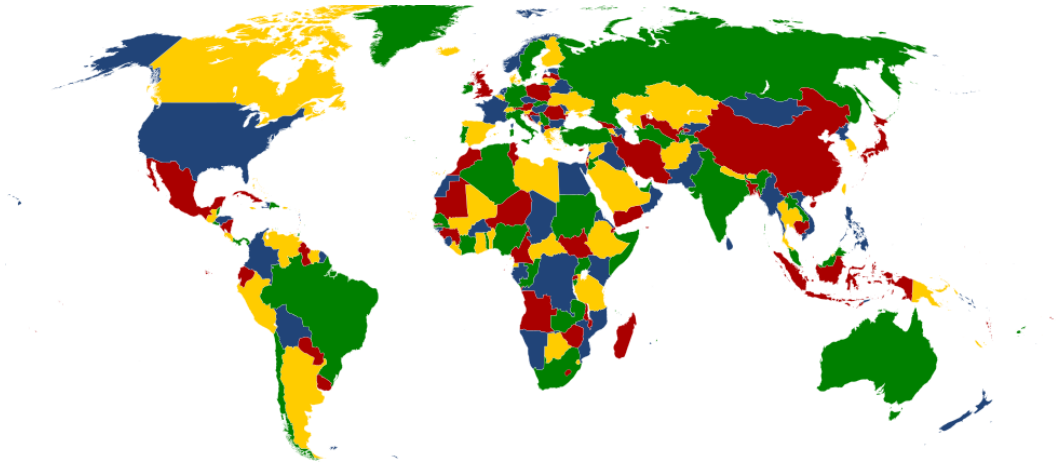
```
#include <iostream>
#include <cmath>
using namespace std;

//nrsol = numarul solutiilor gasite (initializat automat cu 0)
int s[101], n, nrsol;

//verifică dacă s[1],…,s[k] este o soluție parțială sau nu,
//respectiv dacă regina aflată pe linia k și coloana s[k]
//este atacata de o regină poziționată anterior
bool solp(int k)
{
    for(int i = 1; i < k; i++)
        //daca s[k] == s[i] avem două regine pe aceeași coloană
        //daca abs(s[k]-s[i]) == abs(k-i) avem două regine
        //pe aceeași diagonală
        if((s[i] == s[k]) || (abs(s[k]-s[i]) == abs(k-i)))
            return false;
    return true;
}

//functie care afiseaza o solutie sub forma unei table de sah
void afisare()
{
    cout << "Solutia " << ++nrsol << ":" << endl;

    for(int i = 1; i <= n; i++)
    {
        for(int j = 1; j <= n; j++)
            //regina de pe linia i se gaseste pe coloana j
            if(j == s[i])
                cout << " R ";
            else
                cout << " * ";
        cout << endl;
    }
    cout << endl << endl;
```

```
}

//k reprezintă indicele componentei curente s[k] dintr-un
//tablou unidimensional s indexat de la 1
void bkt(int k)
{
    int v;
    //parcurgem toate valorile posibile v pentru s[k]
    //mink = 1, maxk = n
    //succ(v) = v + 1
    for(v = 1; v <= n; v++)
    {
        //atribuim componentei curente s[k] valoarea v
        s[k] = v;
        //dacă s[1],…,s[k] este soluție parțială
        if(solp(k) == true)
        //verific dacă s[1],…,s[k] este o soluție
        //stiind că s[1],…,s[k] este o soluție partiala,
        //deci verific doar conditiile suplimentare!!!
        //in acest caz, trebuie sa avem k == n
            if(k == n)
            {
                //avem o solutie s[1],…,s[k] pe care o prelucram,
                //adica o afisam (in acest caz)
                afisare();
            }
            else
                //s[1],…,s[k] este soluție parțială, dar nu este
                //soluție finala, deci adăugăm o nouă
                //componentă s[k+1]
                bkt(k+1);
    }
}

int main()
{
    cout << "n = ";
    cin >> n;

    //"pornesc" algoritmul de backtracking
    //incepand cu prima componenta
    bkt(1);
    return 0;
}
```

Complexitate: $\mathcal{O}(n!)$

2. Problema colorării hărţilor

Fiind dată o hartă cu n ţări, problema cere să se determine toate modurile în care aceasta poate fi colorată folosind cel mult 4 culori, astfel încât oricare două ţări vecine să fie colorate diferit. De exemplu, o colorare de acest tip a mapamondului este dată în figura de mai jos (sursa: https://commons.wikimedia.org/wiki/File:Four_color_world_map.svg):



Pentru a putea fi colorată folosind cel mult 4 culori, o hartă trebuie să îndeplinească următoarele două condiţii (https://en.wikipedia.org/wiki/Four_color_theorem):

- graniţa dintre două ţări nu se reduce la un singur punct;
- dacă teritoriul unei ţări nu este continuu (de exemplu, statul Alaska aparţine Statelor Unite ale Americii, dar este complet separat de restul teritoriului său), atunci părţile sale nu trebuie să fie colorate folosind aceeaşi culoare.

Problema are o istorie foarte interesantă, fiind formulată în 1852 de către matematicianul şi botanistul sud-african Francis Guthrie sub forma unei conjecturi (i.e., "*Orice hartă poate fi colorată folosind cel mult 4 culori astfel încât oricare două ţări vecine să fie colorate diferit.*") pe care fratele său, Frederick Guthrie, i-a prezentat-o marelui matematician britanic Augustus De Morgan. Ulterior, a fost reformulată folosind noţiuni din teoria grafurilor, fiind studiată de mai mulţi matematicieni de prestigiu (e.g., A. De Morgan, P.G. Tait, A. Kempe etc.) , dar fără a reuşi să o rezolve (i.e., fie arătând că este falsă printr-un contraexemplu, fie arătând că este adevărată printr-o demonstraţie). Prima demonstraţie a corectitudinii sale a fost realizată abia în anul 1976 de către matematicienii americani K. Appel şi W. Haken de la Universitatea din Illinois, combinând noţiuni de algebră cu rezultate obţinute cu ajutorul unui computer. Astfel, pentru prima dată în istoria matematicii, o demonstraţie era efectuată, chiar şi numai parţial, cu ajutorul unui computer, astfel demonstraţia fiind foarte greu acceptată de comunitatea matematică! Ulterior, au fost găsite unele erori în demonstraţia sa, dar acestea au fost corectate de către K. Appel şi W. Haken, iar în 1989 aceştia au publicat varianta finală a demonstraţiei lor. În 1996, matematicienii americani N. Robertson, D.P. Sanders, P. Seymour şi R. Thomas au simplificat demonstraţia lui K. Appel şi W. Haken, realizând şi o optimizare a algoritmului utilizat. În 2005, B. Werner şi G. Gonthier au reuşit să demonstreze teorema utilizând Coq - un instrument software pentru demonstrarea interactivă a teoremelor (https://en.wikipedia.org/wiki/Coq). Astfel, deşi au trecut mai mult de 150 de ani de la formularea problemei colorării hărţilor, încă nu a fost realizată nicio demonstraţie a sa care să nu necesite utilizarea unui computer!

În cadrul demonstrației lor, K. Appel și W. Haken au utilizat un algoritm doar pentru a testa dacă o anumită hartă poate fi colorată conform restricțiilor, deci nu pentru a genera toate modurile în care poate fi colorată harta respectivă. Cu toate acestea, computerul utilizat a avut nevoie de peste 1000 de ore pentru a testa în jur de 1800 de hărți!

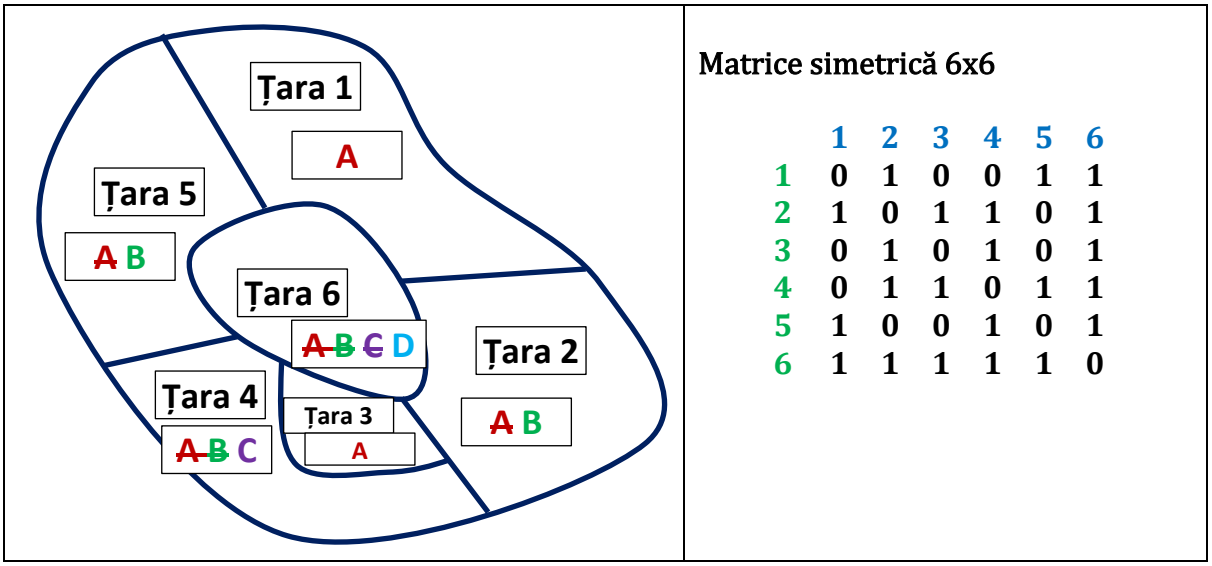
Pentru a rezolva problema generării tuturor modurilor în care poate fi colorată o hartă dată respectând restricțiile indicate vom utiliza metoda Bactracking particularizată, astfel:

- considerând cele n țări ca fiind numerotate cu numerele naturale de la 1 la n, relațiile de vecinătate dintre țări vor fi memorate utilizând o matrice de adiacență A de dimensiune $n \times n$, definită astfel:

$$a[i][j] = \begin{cases} 1, \text{dacă țara } i \text{ este vecină cu țara } j \\ 0, \text{dacă țara } i \text{ nu este vecină cu țara } j \end{cases}$$

  Se observă ușor faptul că matricea A are toate elementele de pe diagonala principală egale cu 0 (deoarece o țară nu poate fi vecină cu ea însăși) și este simetrică față de diagonala principală (i.e., $a[i][j] = a[j][i]$, deoarece dacă țara i este vecină cu țara j, atunci și țara j este vecină cu țara i).
- `s[k]` reprezintă culoarea atribuită țării `k`, deci obținem $\texttt{min}_k=1$ și $\texttt{max}_k=4$;
- `succ(v)=v+1`, deoarece valorile posibile pentru `s[k]` sunt numere naturale consecutive cuprinse între `1` și `n`;
- `s[1],…,s[k-1],s[k]` este soluție parțială dacă nu există nicio țară `i` colorată anterior, deci `1 ≤ i < k`, care să fie vecină cu țara `k` și colorată la fel, adică folosind tot culoarea `s[k]`;
- pentru a testa că `s[1],…,s[k]` este soluție vom ține cont de faptul că `s[1],…,s[k]` este soluție parțială, deci nu vom retesta condițiile de continuare, ci doar condiția suplimentară `k=n`.

Complexitatea acestui algoritm de tip Backtracking poate fi aproximată prin $\mathcal{O}(4^n)$, deoarece fiecare dintre cele n țări poate fi colorată folosind cel mult 4 culori, deci se vor testa cel mult $\underbrace{4 \cdot 4 \cdot \ldots \cdot 4}_{n \text{ ori}} = 4^n$ tupluri reprezentând posibile colorări.



Matrice simetrică 6x6

	1	2	3	4	5	6
1	0	1	0	0	1	1
2	1	0	1	1	0	1
3	0	1	0	1	0	1
4	0	1	1	0	1	1
5	1	0	0	1	0	1
6	1	1	1	1	1	0

harta.txt	colorari.txt
6 1 2 1 5 1 6 2 3 2 4 2 6 3 4 3 6 4 5 4 6 5 6	Solutia 1: Tara 1: culoarea A Tara 2: culoarea B Tara 3: culoarea A Tara 4: culoarea C Tara 5: culoarea B Tara 6: culoarea D ..... Solutia 72: Tara 1: culoarea D Tara 2: culoarea C Tara 3: culoarea D Tara 4: culoarea B Tara 5: culoarea C Tara 6: culoarea A

```
#include <fstream>

using namespace std;

//nrsol = numarul solutiilor gasite (initializat automat cu 0)
//a = matricea de adiacenta a tarilor (initializata automat cu 0)
int s[101], n, nrsol, a[101][101];

//fisierele de intrare si iesire
ifstream fin;
ofstream fout;

//verifica daca s[1],…,s[k] este o solutie partiala sau nu,
//respectiv daca nu am folosit deja culoarea s[k] a tarii k
//pentru o alta tara vecina cu ea, colorata anterior
bool solp(int k)
{
    for(int i = 1; i < k; i++)
        //daca a[k][i] == 1 (sau a[i][k] == 1),
        // atunci tarile i si k sunt vecine
        //daca s[k] == s[i], atunci tarile i si k sunt vecine
        if((a[k][i] == 1) && (s[k] == s[i]))
            return false;
    return true;
}
```

```
//functie care afiseaza o solutie a problemei
void afisare()
{
    fout << "Solutia " << ++nrsol << ":" << endl;

    for(int i = 1; i <= n; i++)
        fout << "Tara " << i << ": culoarea " << (char)('A'+s[i]-1)
<< endl;
    fout << endl << endl;
}

//k reprezinta indicele componentei curente s[k] dintr-un
//tablou unidimensional s indexat de la 1
void bkt(int k)
{
    int v;
    //parcurgem toate valorile posibile v pentru s[k]
    //mink = 1, maxk = 4, deoarece
    //numarul maxim de culori este 4
    //succ(v) = v + 1
    for(v = 1; v <= 4; v++)
    {
        //atribuim componentei curente s[k] valoarea v
        s[k] = v;
        //daca s[1],…,s[k] este solutie partiala
        if(solp(k) == true)
        //verific daca s[1],…,s[k] este o solutie
        //stiind ca s[1],…,s[k] este o solutie partiala,
        //deci verific doar conditiile suplimentare!!!
        //in acest caz, trebuie sa avem k == n
            if(k == n)
            {
                //avem o solutie s[1],…,s[k] pe care o prelucram,
                //adica o afisam (in acest caz)
                afisare();
            }
            else
                //s[1],…,s[k] este solutie partiala, dar nu este
                //solutie finala, deci adaugam o noua
                //componenta s[k+1]
                bkt(k+1);
    }
}
```

```
int main()
{
    //citim datele de intrare din fisierul text "harta.txt"
    fin.open("harta.txt");

    //de pe prima linie citim numarul n de tari
    fin >> n;

    //citesc perechi de tari vecine si completez corepsunzator
    //matricea de adiacenta
    int x, y;
    while(fin >> x >> y)
        a[x][y] = a[y][x] = 1;

    //inchidem fisierul de intrare
    fin.close();

    //deschidem fisierul de iesire "colorari.txt"
    fout.open("colorari.txt");

    //"pornesc" algoritmul de backtracking
    //incepand cu prima componenta
    bkt(1);

    //inchidem fisierul de iesire
    fout.close();

    return 0;
}
```

3. Determinarea tuturor numerelor având cifre distincte şi suma cifrelor dată

Problema a fost dată la primul examen de Bacalaureat în care elevii de la profilul Informatică au susţinut obligatoriu o probă la disciplina Informatică. Problema cerea ca pentru un număr natural *c* citit de la tastatură să se afişeze toate numerele formate din cifre distincte şi suma cifrelor egală cu *c*. De exemplu, pentru $c = 3$, trebuie să fie afişate numerele: 102, 12, 120, 201, 21, 210, 3 şi 30 (nu neapărat în această ordine).

Observaţii:

- Orice soluţie are cel mult 10 cifre (e.g., orice permutare a cifrelor numărului 9876543210 care nu începe cu cifra 0).

- Problema are soluţie doar pentru valori ale lui *c* cuprinse între 0 şi 45 = 9+8+...+1.

- `s[k]` reprezintă cifra aflată pe poziţia `k` într-un număr (considerăm cifrele unui număr ca fiind numerotate de la stânga spre dreapta), deci obţinem $min_k=1$ pentru $k=1$ (prima cifră a unui număr nu poate fi 0) sau $min_k=0$ pentru $k \geq 2$, respectiv $max_k=9$;

- `s[1],…,s[k-1],s[k]` este soluţie parţială dacă cifra curentă `s[k]` nu a mai fost utilizată anterior, adică `s[k]≠s[i]` pentru orice `i∈{1,2,…,k-1}`, şi `s[1]+…+s[k]≤c`;

- pentru a testa dacă `s[1],…,s[k]` este soluţie vom ţine cont de faptul că cifrele `s[1],…,s[k]` sunt distincte (din condiţiile de continuare), deci vom verifica doar condiţia suplimentară `s[1]+…+s[k]==c`.

Implementând algoritmul Backtracking corespunzător observaţiilor de mai sus, nu vom obţine toate soluţiile, ci doar o parte dintre ele! De exemplu, pentru $c = 3$, **nu** vom obţine numerele scrise îngroşat: 102, 12, **120**, 201, 21, **210**, 3 şi **30**. Explicaţia acestui fapt necesită o înţelegere aprofundată a metodei Backtracking: numerele scrise îngroşat sunt soluţii care se obţin din soluţiile care nu conţin cifra 0 (de exemplu, numărul **120** se obţine din numărul 12, care nu conţine cifra 0, prin adăugarea unui 0 la sfârşitul său)! În forma sa generală, algoritmul Backtracking **nu** va furniza niciodată numerele scrise îngroşat, deoarece după găsirea unei soluţii a problemei, algoritmul **nu** va încerca niciodată să adauge încă un element (o cifră, în acest caz) la ea! Din acest motiv, o soluţie completă care nu modifică foarte mult algoritmul general Backtracking se poate obţine astfel: în momentul afişării unei soluţii, verificăm dacă ea conţine deja o cifră egală cu 0, iar în caz negativ o afişăm încă o dată şi-i adăugăm un 0 la sfârşit.

```
#include<stdio.h>
int s[11], c;

//funcţie care verifică dacă cifra s[k] a fost utilizată anterior
int distincte(int k)
{
    int i;

    for(i = 1; i < k; i++)
        if(s[k] == s[i]) return 0;
    return 1;
}

//funcţie care calculează suma s[1]+...+s[k]
int suma(int k)
{
    int i, scrt;

    scrt = 0;
    for(i = 1; i <= k; i++)
        scrt = scrt + s[i];
    return scrt;
}
void bkt(int k)
{
    int i, v, scrt, z;

    //valoarea minimă a unei cifre depinde de poziţia sa k
    for(v = (k == 1 ? 1 : 0); v <= 9; v++)
    {
        s[k] = v;
```

```
        scrt = suma(k);

        //verificăm condiţiile de continuare
        if(scrt <= c && distincte(k) == 1)
            //verificăm condiţia de soluţie
            if(scrt == c)
            {
                //verificăm dacă soluţia curentă conţine cifra 0
                z = 0;
                for(i = 1; i <= k; i++)
                {
                    printf("%d", s[i]);
                    if(s[i] == 0) z = 1;
                }
                printf("\n");

                //dacă soluţia curentă nu conţine cifra 0,
                //o reafişăm şi adăugăm cifra 0 la sfârşitul său
                if(z == 0)
                {
                    for(i = 1; i <= k; i++) printf("%d", s[i]);
                    printf("0\n");
                }
            }
            else
                bkt(k+1);
    }
}

int main()
{
    printf("c = ");
    scanf("%d", &c);

    //verificăm dacă problema are soluţie sau nu
    if(c < 0 || c > 45)
        printf("Problema nu are solutie!");
    else
        bkt(1);

    return 0;
}
```