# Metoda backtracking

Capitolul

# 19

- **Backtracking iterativ**
- **Backtracking recursiv**
- **Implementări sugerate**
- **Probleme propuse**
- **Soluțiile problemelor**

Metoda *backtracking* este o metodă de programare cu ajutorul căreia se rezolvă problemele în care:

- soluția se poate reprezenta sub forma unui tablou $X = (x_1, x_2, \ldots, x_n)$ cu $x_1 \in M_1$, $x_2 \in M_2$ etc. $M_1, M_2, ..., M_n$ fiind mulțimi finite având $s_1, s_2, ..., s_n$ elemente.
- între elementele tabloului $X$ există anumite legături impuse în enunț.

***Condiții interne***

În fiecare problemă sunt date anumite relații care trebuie să existe între componentele $x_1, x_2, ..., x_n$ ale vectorului $X$, numite *condiții interne*.

***Spațiul soluțiilor posibile***

Mulțimea $M_1 \times M_2 \times ... \times M_n$ se numeşte *spațiul soluțiilor posibile*.

***Condiții de continuare***

Dacă la pasul $k$, condițiile interne sunt satisfăcute, algoritmul se continuă în funcție de cerințe. Dacă mai trebuie căutate componente, deoarece numărul lor se cunoaşte şi încă nu le-am obținut pe toate, sau componentele încă nu constituie o soluție pe baza unor proprietăți, condițiile de continuare vor permite continuarea algoritmului.

***Soluții rezultat***

Acele soluții dintre soluțiile posibile care satisfac condițiile impuse de problemă (condițiile interne şi condițiile de continuare nu cer componente în plus) se numesc *soluții rezultat*.

O metodă de rezolvare a acestei categorii de probleme ar fi determinarea tuturor soluțiilor posibile şi apoi căutarea acelora care satisfac condițiile interne. Dezavantajul este uşor de observat deoarece timpul cerut de această căutare este foarte mare.

***Modul de lucru al metodei backtracking***

- Elementele din tabloul $X$ primesc *pe rând* valori, adică lui $x_k$ i se atribuie o valoare numai dacă au fost deja atribuite valori elementelor $x_1, x_2, ..., x_{k-1}$.
- În plus, lui $x_k$ i se atribuie o valoare numai dacă pentru valorile $x_1, x_2, ..., x_{k-1}$ şi valoarea propusă pentru $x_k$ sunt îndeplinite condiţiile interne şi cele de continuare impuse de problemă. (În cazul în care aceste condiţii nu sunt îndeplinite, oricum am alege următorii termeni $x_{k+1}, ..., x_n$, nu vom ajunge la o soluţie în care condiţiile interne să fie satisfăcute.)
- Dacă la pasul $k$ condiţiile de continuare nu sunt îndeplinite trebuie să se facă o altă alegere pentru $x_k$ din mulţimea $M_k$.
- Dacă mulţimea $M_k$ a fost epuizată (s-au testat toate valorile din mulţime) se *decrementează* $k$ şi se încearcă o altă alegere pentru $x_{k-1}$ din mulţimea $M_{k-1}$.

**Observaţii**

- Acestei decrementări a lui $k$ i se datorează numele metodei, exprimând faptul că atunci când nu putem avansa, se avansează *în sens invers* (înapoi) în secvenţa curentă a soluţiei.
- Între condiţiile interne şi cele de continuare există o strânsă legătură. O bună alegere a condiţiilor de continuare are ca efect reducerea numărului de calcule.

# 19.1. Backtracking iterativ

Condiţiile interne vor fi verificate în subalgoritmul `Posibil(k)`. Acesta returnează *adevărat* dacă adăugarea componentei $x_k$ a şirului soluţie este posibilă şi *fals* în caz contrar.

```
Subalgoritm Posibil(k):
  dacă condiţiile interne nu sunt îndeplinite atunci
    Posibil ← fals
    ieşire forţată
  sfârşit dacă
  Posibil ← adevărat
sfârşit subalgoritm
```

Generarea soluţiilor se realizează cu subalgoritmul `Back`.

```
Subalgoritm Back:
  k ← 1
  x[k] ← 0
  cât timp k > 0 execută:
    ok ← fals
```

```
    cât timp (x[k]< valoare maximă permisă) şi nu ok execută:
    { mai sunt valori netestate în mulţime, deci lui x[k] i se atribuie o altă valoare }
      x[k] ← x[k] + 1
      ok  ← Posibil(k)        { poate am găsit valoare posibilă pentru x[k] }
    sfârşit cât timp
    dacă nu ok atunci
      { dacă nu, se decrementează k pentru a alege o valoare nouă pentru aceasta }
      k ← k - 1
    altfel                                  { dacă am ajuns la sfârşitul generării }
      dacă k = numărul de elemente cerut atunci          { sau altă condiţie }
        scrie soluţia
      altfel
        k ← k + 1                  { trecem la următorul element din soluţie }
        x[k] ← 0                { iniţializăm noul element din tabloul soluţie }
      sfârşit dacă
    sfârşit dacă
  sfârşit cât timp
sfârşit subalgoritm
```

# 19.2. Backtracking recursiv

Algoritmul poate fi implementat şi recursiv (funcţia `Posibil(k)` are forma prezentată):

```
Subalgoritm Back(k):
  pentru i = 1,valoarea maximă impusă de problemă execută:
    x[k] ← i
    dacă Posibil(k) atunci
                                          { dacă am ajuns la sfârşitul generării }
      dacă (*) k = numărul de elemente cerut atunci
                                              { sau altă condiţie de continuare }
        scrie soluţia(k)
      altfel
        Back(k+1)
      sfârşit dacă
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

Dacă în condiţia de continuare `(*)` expresia este formată din subexpresii logice (relaţionale) şi condiţia nu este îndeplinită, pe ramura **altfel** va trebui să decidem motivul pentru care condiţia nu este îndeplinită. Dacă de exemplu, numărul de ele-

mente cerut s-a atins, dar din alte motive soluția obținută nu este corectă şi deci nu se scrie, atunci evident, nu apelăm subalgoritmul `Back` pentru valoarea $k + 1$, ci vom ieşi din subprogram, asigurând totodată revenirea la pasul precedent şi căutarea unei valori noi pentru $k - 1$.

**Observație**

Algoritmii de rezolvare a problemelor cu metoda backtracking, în principiu, vor respecta forma generală de mai sus, dar de fiecare dată vom căuta posibilitățile de optimizare şi implementări cât mai sugestive. Un caz special îl reprezintă acele probleme în care soluția se compune din două (sau mai multe) şiruri care descriu, de exemplu, coordonate în plan.

## 19.3. Implementări sugerate

Pentru a vă familiariza cu analiza problemelor care se rezolvă cu metoda backtracking, precum cu implementarea programelor de rezolvare, vă recomandăm să încercați să rezolvați următoarele probleme:

1. generarea tuturor permutărilor şirului de numere 1, 2, ..., $n$;
2. aşezarea a 8 regine pe tabla de şah, fără ca acestea să se atace;
3. generarea produsului scalar a două mulțimi;
4. generarea tuturor aranjamentelor de $n$ luate câte $m$;
5. generarea tuturor perechilor de paranteze care se închid corect;
6. generarea tuturor funcțiilor injective cunoscând mulțimea de argumente şi cea a valorilor funcțiilor;
7. generarea tuturor funcțiilor surjective cunoscând mulțimea de argumente şi cea a valorilor funcțiilor;
8. generarea tuturor partițiilor unui număr dat;
9. generarea partițiilor mulțimii $\{1, 2, \ldots, n\}$;
10. generarea tuturor turnurilor de cuburi în funcție de anumite proprietăți date;
11. ieşirea din labirint;
12. ieşirea din labirint (pereții celulelor păstrați pe biți);
13. aranjarea unor piese de *tetris*, astfel încât să completeze integral un dreptunghi de dimensiuni date;
14. determinarea turnului de cuburi de înălțime minimă;
15. determinarea drumului pe care broasca ajunge la lăcustă şi se întoarce la locul de pândă pe suprafața unui lac înghețat, unde gheața se rupe în urma efectuării săriturilor efectuate conform săriturii calului pe tabla de şah;
16. mutarea elementelor unei matrice de dimensiuni $m \times n$ într-un şir $a$, astfel încât două elemente aflate pe poziții consecutive în şirul $a$ să fie vecine în matrice şi suma $1 \cdot a_1 + 2 \cdot a_2 + \ldots + m \cdot n \cdot a_{m \cdot n}$ să fie minimă.

# 19.4. Probleme propuse

## 19.4.1. Numere

Se dă un tablou unidimensional de *n* numere întregi. Să se genereze toate tablourile alternante (şiruri în care după fiecare element strict pozitiv urmează unul strict negativ şi după fiecare element strict negativ urmează unul strict pozitiv) care să conţină elemente din şirul dat în ordinea iniţială a lor.

**Date de intrare**
Elementele tabloului se află în fişierul de intrare **NUMERE.IN**, un număr pe o linie.

**Date de ieşire**
Soluţiile se vor scrie în fişierul de ieşire **NUMERE.OUT**. O soluţie (un şir alternant) se va scrie pe o linie. În cadrul acestor şiruri două numere vor fi separate printr-un spaţiu.

**Restricţii şi precizări**
- $1 \le n \le 100$ (se determină pe baza numărului numerelor din fişierul de intrare)
- Dacă din tabloul dat nu se poate forma nici un subşir care îndeplineşte cerinţele problemei, în fişierul de ieşire se va scrie mesajul `'imposibil'`.

**Exemple**

**NUMERE.IN**
```
-1 0 -3 -4
```
**NUMERE.OUT**
```
imposibil
```
**NUMERE.IN**
```
1 2 -3
```
**NUMERE.OUT**
```
1 -3
2 -3
```

## 19.4.2. Rechizite

Pentru prima zi de şcoală un elev doreşte să-şi cumpere rechizite şcolare ale căror costuri le cunoaşte. Acesta are la dispoziţie o sumă de bani *S* şi doreşte să aleagă din mai multe obiecte cum ar fi:
- creioane
- radiere
- caiete
- liniare
- rezerve stilou
- creioane colorate
- acuarele

Ştiind că elevul a stabilit un necesar maxim pentru fiecare obiect, afişaţi câte bucăţi din fiecare obiect reuşeşte să cumpere.

**Date de intrare**
Datele de intrare se citesc din fişierul **RECHIZIT.IN** care are următoarea structură:
- Pe prima linie se găseşte suma de bani *S* de care dispune elevul.
- Pe a doua linie se găsesc costurile celor 7 obiecte, separate prin câte un spaţiu.
- Pe a treia linie se găsesc cantităţile maxime necesare din fiecare obiect, separate prin câte un spaţiu.

**Date de ieşire**
Datele de ieşire se vor scrie pe linii distincte în fişierul de ieşire **RECHIZIT.OUT** sub forma următoare:
*buc*cost–nume produs*
unde *buc* reprezintă numărul bucăţilor din rechizitele având denumirea *nume produs* şi preţul unitar *cost*.

Dacă elevul nu reuşeşte să efectueze cumpăraturile în condiţiile specificate în enunţ, în fişierul de ieşire se va scrie mesajul `'imposibil'`.

**Restricţii şi precizări**
- $1 \le S \le 1000000$;
- costurile maxime sunt numere pozitive mai mici decât 500000;
- cantităţile maxime sunt numere pozitive mai mici decât 100;
- nu este necesar ca elevul să cumpere din toate obiectele; lăsaţi-l pe el să aleagă ce anume şi în ce cantităţi cumpără, cu condiţia să-şi cheltuie toţi banii.

**Exemple**

**RECHIZIT.IN**
```
100000
2000 5000 3000 12000 6000 80000 70000
1 2 1 2 1 0 0
```

**RECHIZIT.OUT**
```
imposibil
```

**RECHIZIT.IN**
```
100000
1000 2000 10000 3000 4000 20000 80000
1 2 1 1 1 1 1
```

**RECHIZIT.OUT**
```
1*20000-set creioane colorate
1*80000-set acuarele

1*1000-creioane
1*2000-radiere
1*10000-caiete
1*3000-liniare
1*4000-rezerve stilou
1*80000-set acuarele
```

## 19.4.3. Expresie

Se dă un şir de *n* numere întregi. Să se plaseze între aceste numere *n* – 1 operatori aritmetici (+, –, *, /), astfel încât valoarea expresiei obținute să fie egală cu un număr întreg dat *m*.

**Date de intrare**

Pe prima linie a fişierului de intrare **EXPRES.IN** se află numărul întreg *n.* Pe a doua linie se află cele *n* numere, separate prin câte un spațiu, iar pe a treia linie se află numărul *m*.

**Date de ieşire**

În fişierul de ieşire **EXPRES.OUT** fiecare soluție este scrisă pe linie nouă. O soluție are forma:

*m*=*expresie*, unde *expresie* este o succesiune de operanzi şi operatori, primul şi ultimul fiind operanzi.

Dacă pentru numerele date nu se poate găsi o combinație de operatori astfel încât rezultatul expresiei obținute să fie numărul *m* dat, în fişier se va scrie mesajul `'imposibil'`.

**Restricții şi precizări**

- $2 \le n < 10$;
- $-100 \le nr_i \le 100$, $i = 1, 2, \ldots, n$;
- valoarea expresiei trebuie să fie un număr întreg la fiecare pas al evaluării;
- nu se respectă prioritatea operatorilor cunoscută din aritmetică (operațiile se vor efectua de la stânga la dreapta în ordinea în care apar în expresie).

**Exemplu**

**EXPRES.IN**
```
4
12 12 1 1
24
```

**EXPRES.OUT**
```
24=12+12+1-1
24=12+12-1+1
24=12+12*1*1
24=12+12*1/1
24=12+12/1*1
24=12+12/1/1
```

## 19.4.4. Maşinuțe

Un copil doreşte să aşeze *n* maşinuțe numerotate de la 1 la *n* în *n* cutii identificate prin numere de la 1 la *n*. Să se afişeze toate modalitățile în care copilul poate să aşeze maşinuțele în cutii, ştiind că într-o cutie încap cel mult *m* maşinuțe.

**Date de intrare**
Fişierul de intrare **MASINI.IN** conţine o singură linie pe care sunt trecute valorile *n* şi *m*, separate printr-un spaţiu.

**Date de ieşire**
Fişierul de ieşire **MASINI.OUT** va conţine pentru fiecare soluţie obţinută câte *n* linii care vor descrie conţinutul cutiilor. Pe linia *i* (*i* = 1, 2, …, *n*) se scriu *m* numere de ordine ale maşinuţelor care sunt puse în cutia având numărul de ordine *i*. Aceste numere se vor despărţi printr-un spaţiu, iar în faţa numerelor de ordine a maşinuţelor se va scrie cuvântul `cutia`, apoi un spaţiu şi numărul de ordine *i*, care la rândul lui este urmat de `': '`. Dacă într-o cutie nu se pune nici o maşinuţă, ea nu va conţine după `': '` nimic.

**Restricţii şi precizări**
- $1 \le n, m \le 7$.

**Exemplu**

**MASINI.IN**
```
2 3
```

**MASINI.OUT**
```
cutia 1: 1 2
cutia 2:

cutia 1: 1
cutia 2: 2

cutia 1: 2
cutia 2: 1

cutia 1:
cutia 2: 1 2
```

## 19.4.5. Depozit

Într-un depozit compartimentat conform unui caroiaj a izbucnit focul în *m* locuri diferite. Magazionerul aflat în depozit ar vrea să iasă din clădire. Ştiind că ieşirea se află în colţul din dreapta-jos a depozitului, să se determine toate drumurile posibile spre ieşire.

În anumite compartimente, corespunzătoare unor pătrăţele din caroiaj, se află obiecte peste care magazionerul nu va putea trece sau sări. De asemenea, pe drumul său spre ieşire, magazionerul nu poate trece prin compartimentele vecine orizontal, vertical sau pe diagonală celor cuprinse de incendiu. Magazionerul va putea să traverseze depozitul trecând printr-o succesiune de compartimente vecine orizontal sau vertical.

**Date de intrare**
În fişierul de intrare **DEPOZIT.IN** se vor afla datele referitoare la depozitul având forma pătratică, poziţia magazionerului şi poziţiile cuprinse de flăcări.

- Pe prima linie a fişierului se află dimensiunile depozitului (*n*);
- Pe următoarele *n* linii se află câte *n* valori, reprezentând datele referitoare la depozit:
  – 0 reprezintă spaţiu liber;
  – 1 reprezintă un obstacol;
  – 2 reprezintă un foc;
  – 3 reprezintă poziţia magazionerului.

**Date de ieşire**
Soluţia se va afişa în fişierul de ieşire **DEPOZIT.OUT** sub forma unui tablou bidimensional având *n* linii şi *n* coloane. Acest tablou va conţine traseul magazionerului. Pentru fiecare valoare *k* (cuprinsă între 1 şi lungimea traseului) în tablou va exista un singur element având valoarea *k*, (corespunzător pasului *k* al magazionerului), în rest tabloul va conţine elemente egale cu 0.

**Restricţii şi precizări**
- $2 \le n < 8$;
- două soluţii în fişierul de ieşire vor fi separate printr-o linie goală;
- dacă nu se poate ajunge la ieşire din cauză că este imposibil să se iasă din depozit datorită focurilor şi obstacolelor, în fişierul de ieşire se va scrie `'imposibil'`.

**Exemplu**

**DEPOZIT.IN**
```
5
3 0 0 2 0
0 1 0 0 0
0 0 0 0 0
0 0 0 1 0
2 0 0 0 0
```

**DEPOZIT.OUT**
```
1 0 0 0 0
2 0 0 0 0
3 4 5 6 7
0 0 0 0 8
0 0 0 0 9

1 0 0 0 0
2 0 0 0 0
3 4 5 0 0
0 0 6 0 0
0 0 7 8 9
```

## 19.4.6. Cal şi nebun pe tabla de şah

Pe o tablă de şah de dimensiune $n \times n$ se află un nebun în poziţia $(x, y)$ şi un cal în poziţia iniţială $(x_0, y_0)$. Calul trebuie să ajungă din această poziţie iniţială într-o poziţie finală $(x_1, y_1)$ fără să treacă de două ori prin aceeaşi poziţie şi fără să se oprească pe un pătrat aflat în raza de acţiune a nebunului.

Afişaţi toate secvenţele de paşi de lungime minimă prin care calul poate să ajungă din poziţia iniţială în poziţia finală fără a fi capturat de nebun.

**Date de intrare**
Fişierul de intrare **CALNEBUN.IN** are următoarea structură:

- pe prima linie se află un număr natural diferit de 0, reprezentând dimensiunea tablei;
- pe cea de-a doua linie sunt scrise două numere naturale, despărţite printr-un spaţiu, corespunzător poziţiei nebunului;
- pe următoarele două linii sunt înregistrate, în formatul liniei doi, poziţia iniţială a calului, precum şi poziţia pe care trebuie să ajungă.

**Date de ieşire**
Fişierul de ieşire **CALNEBUN.OUT** va conţine câte o linie, corespunzătoare unei soluţii. Fiecare soluţie va fi scrisă sub forma unor perechi de numere naturale indicând traseul calului. Aceste perechi reprezintă coordonatele pătratelor tablei de şah atinse de cal pe traseul lui începând cu poziţia iniţială şi terminând cu poziţia finală. Perechile se vor scrie încadrate între paranteze rotunde şi în interiorul parantezei despărţite de o virgulă.

Dacă din start calul se află în bătaia nebunului sau coordonatele citite din fişier nu se află în interiorul tablei, în fişierul de ieşire se va scrie mesajul `'Imposibil'`.

**Restricţii şi precizări**

- $2 \le n \le 8$;
- calul nu are voie să iasă de pe tablă;
- poziţia iniţială nu poate să coincidă cu poziţia finală;
- în cazul în care coordonatele iniţiale şi finale ale calului se află în interiorul tablei şi poziţia iniţială nu este atacată de nebun, atunci va exista întotdeauna cel puţin o soluţie.

**Exemplu**

**CALNEBUN.IN**
```
6
2 3
2 2
4 4
```

**CALNEBUN.OUT**
```
(2,2)(4,3)(2,4)(3,6)(4,4)
(2,2)(4,3)(5,5)(3,6)(4,4)
(2,2)(4,3)(5,5)(6,3)(4,4)
(2,2)(4,3)(6,4)(5,2)(4,4)
(2,2)(4,3)(5,1)(6,3)(4,4)
(2,2)(4,3)(3,1)(5,2)(4,4)
```

# 19.5. Soluțiile problemelor propuse

## 19.5.1. Numere

Problema dată se rezolvă prin metoda *backtracking*, deoarece se cere generarea *tuturor* subşirurilor care au proprietățile cerute în enunț.

Fie *a* şirul dat având *n* elemente numere întregi. Vom lucra la nivelul indicilor, adică vom genera mulțimi de indici ai elementelor din *a*, formând soluția în şirul de indici *x*.

**Exemplu**

Să presupunem că şirul *a* are 7 elemente numere întregi: `1 -2 2 3 -1 -4 7`.
O soluție a problemei este şirul de numere: `1 -2 2 -1 7`.
Şirul de indici *x* care se generează în acest caz, este: `1 2 3 5 7`.

Se observă că în enunț nu se specifică lungimea *soluției*, deci este necesară generarea tuturor soluțiilor de lungimi variabile (cel puțin două elemente şi cel mult *n*). Vom asigura generarea lor, specificând lungimile posibile în programul principal într-o structură de tip **pentru**. Pentru fiecare lungime posibilă de la 2 la *n* vom genera toate subşirurile de lungimea respectivă care respectă cerințele din enunț.

Vom construi un subalgoritm de validare care verifică dacă există cel puțin un element pozitiv în şirul dat şi cel puțin un element negativ, astfel încât să fie asigurată generarea a cel puțin a unei soluții de lungime 2. Dacă în şirul dat nu se găseşte cel puțin un element pozitiv şi cel puțin un element negativ, în fişierul de ieşire se va scrie mesajul `'imposibil'`.

```
Subalgoritm Validare(a,n):
  i ← 1                    { verificăm dacă există cel puțin un element pozitiv }
  cât timp (i ≤ n) şi (a[i] ≤ 0) execută:
    i ← i + 1                  { cât timp numărul nu este pozitiv, avansăm }
  sfârşit cât timp
  okpozitiv ← i ≤ n                  { am găsit cel puțin un număr pozitiv? }
  i ← 1                  { verificăm dacă există cel puțin un element negativ }
  cât timp (i ≤ n) şi (a[i] ≥ 0) execută:
    i ← i + 1                  { cât timp numărul nu este negativ avansăm }
   sfârşit cât timp
  oknegativ ← i ≤ n                  { am găsit cel puțin un număr negativ? }
              { dacă avem cel puțin un element pozitiv şi cel puțin unul negativ }
                                                { vom avea cel puțin o soluție }
  Validare ← okpozitiv şi oknegativ
sfârşit subalgoritm
```

Condițiile de continuare se verifică într-un alt subalgoritm, unde ne asigurăm că:
- elementul care se adaugă pe poziția *k* nu a mai fost introdus în șirul soluție;
- dacă se adaugă un element pozitiv pe poziția *k*, atunci elementul de pe poziția *k* – 1 nu va fi tot pozitiv;
- dacă se adaugă un element negativ pe poziția *k*, atunci elementul de pe poziția *k* – 1 nu va fi tot negativ;
- indicii trebuie să apară în ordine crescătoare, astfel încât în soluții să se păstreze ordinea numerelor din șirul inițial, așa cum cere problema.

Dacă în urma execuției subalgoritmului de mai jos valoarea aleasă pe poziția *k* este acceptată, subalgoritmul `Posibil(k)` returnează *adevărat*, în caz contrar *fals*. Cazul în care *k* = 1 (generăm prima componentă din soluție) se va trata cu atenție, deoarece în această situație nu există `x[k-1]`.

```
Subalgoritm Posibil(k):
  i ← 1
  cât timp (i ≤ k-1) şi (x[i] ≠ x[k]) execută:
    i ← i + 1    { dacă x[k] a fost inclus deja în soluție nu îl mai putem pune }
  Posibil ← i > k-1
  sfârşit cât timp
  dacă (k > 1) şi (a[x[k-1]] > 0) şi (a[x[k]] > 0) atunci
   { dacă în fața lui a[x[k]] (număr pozitiv) avem tot un număr pozitiv, nu îl punem }
    Posibil ← fals
    ieşire forțată din subprogram
  sfârşit dacă
  dacă (k > 1) şi (a[x[k-1]] < 0) şi (a[x[k]] < 0) atunci
{ dacă în fața lui a[x[k]] (număr negativ) avem tot un număr negativ, nu îl punem }
    Posibil ← fals
    ieşire forțată din subprogram
  sfârşit dacă
  dacă (k > 1) şi (x[k-1] > x[k]) atunci
    Posibil ← fals       { dacă indicele x[k] este mai mic decât cel din fața lui }
    ieşire forțată din subprogram
  sfârşit dacă
                         { dacă am ajuns până aici, alegerea lui x[k] este posibilă }
  Posibil ← adevărat
sfârşit subalgoritm
```

Afișarea unei soluții se realizează scriind elementele tabloului *a* corespunzătoare indicilor dați în șirul *x*. Deoarece șirurile soluție au lungime variabilă, trebuie transmis ca parametru și lungimea *k* a lor.

```
Subalgoritm ScrieSoluție(k)
  pentru i=1,k execută
    scrie a[x[i]]
  sfârşit pentru
sfârşit subalgoritm
```

În subalgoritmul care realizează paşii metodei backtracking:

- valoarea lui $x_k$ este mai mică sau egală cu $n$, $x_k$ reprezentând un indice din şirul $a$;
- afişarea se face atunci când $k = q$, unde variabila $q$ este iniţializată în programul principal, ea luând valori între 2 şi $n$ şi o folosim pentru a putea genera subşiruri de lungimi variabile.

```
Subalgoritm Back(q):
  k ← 1
  x[k] ← 0
  cât timp k > 0 execută:
    ok ← fals                    { încă nu am găsit valoarea curentă a soluției }
    cât timp (x[k] < n) şi nu ok execută:
      x[k] ← x[k]+1
      ok ← Posibil(k)          { verificăm dacă x[k] poate fi pus în soluție }
    sfârşit cât timp
    dacă nu ok atunci
      k ← k - 1              { căutăm altă valoare pentru elementul precedent }
    altfel
      dacă k = q atunci        { dacă avem q elemente în soluție, o scriem }
        ScrieSoluție(k)
      altfel
        k ← k + 1                  { dacă nu, pregătim următorul element }
        x[k] ← 0
      sfârşit dacă
    sfârşit dacă
  sfârşit cât timp
sfârşit subalgoritm
```

Programul principal conţine secvenţa de mai jos care asigură verificarea existenţei cel puţin a unei soluţii şi lansarea în execuţie a subprogramelor descrise mai sus.

```
...
  dacă Validare(a,n) atunci
    pentru q=2,n execută:
      Back(q)
    sfârşit pentru
  altfel scrie 'imposibil'
  sfârşit dacă
...
```

## 19.5.2. Rechizite

Deoarece în problemă se cer *toate* modalităţile de cheltuire a sumei *S*, o vom rezolva prin metoda *backtracking*. Problema revine la a scrie un număr întreg sub forma unei sume de termeni numere întregi, adică suma *S* dată trebuie descompusă în funcţie de restricţiile impuse în enunţul problemei.

Tabloul soluţie *x* va conţine pe poziţia *i* numărul de obiecte de papetărie de tipul *i*, pe care elevul le cumpără; cu alte cuvinte *i* reprezintă indicele din tabloul de articole de papetărie şi $x_i$ reprezintă cantitatea din articolul având indicele *i*.

În rezolvare folosim un subalgoritm care calculează costul total al produselor cumpărate până la inclusiv pasul curent *k*.

```
Subalgoritm Suma(k):
  sum ← 0
  pentru i=1,k execută:
                              { x[i] bucăţi din produsul care costă bani[i] }
    sum ← sum + x[i] * bani[i]
  sfârşit_pentru
  Suma ← sum
sfârşit subalgoritm
```

În algoritm se verifică la fiecare pas dacă cheltuielile realizate până la pasul curent nu depăşesc suma de bani pe care o are elevul la dispoziţie. Acest test se realizează în cadrul unui subalgoritm de validare.

```
Subalgoritm Posibil:
  Posibil ← Suma(k) ≤ s
sfârşit subalgoritm
```

Subalgoritmul care implementează metoda backtracking are câteva particularităţi:

- Fiecare element din şir (reprezentând cantităţi de produse) poate avea ca valoare maximă valoarea corespunzătoare din şirul `necesar`.
- Din moment ce nu este obligatoriu să se cumpere din fiecare obiect este posibilă şi cantitatea 0.
- Dacă la un anumit pas costul obiectelor deja cumpărate este egal cu suma deţinută de elev, atunci aceasta se consideră a fi o soluţie şi se scrie în fişierul de ieşire, altfel se încearcă cumpărarea a încă unui obiect.

```
Subalgoritm Back(k):
  pentru j=0,necesar[k] execută    { cantităţi posibile între nimic şi maxim }
    x[k] ← j
    dacă Posibil(k) atunci
```

```
      dacă suma(k) = s atunci Scrie_sol(k)
      altfel
                { există în total 7 obiecte din care elevul îşi face cumpărăturile }
        daca k < 7 atunci
          Back(k+1);
        sfârşit dacă
      sfârşit dacă
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

Dacă nu se poate scrie nici o soluție, în fişier se va scrie mesajul `'imposibil'`.

## 19.5.3. Expresii

Problema se rezolvă prin metoda *backtracking* deoarece dorim să obținem *toate* soluțiile care respectă cerințele problemei. Pentru aceasta am reținut valorile operanzilor în tabloul *a*. Vom genera toate tablourile *x* care conțin operatori codificați astfel:

1 : adunare
2 : scădere
3 : înmulțire
4 : împărțire

Vom genera un şir de operatori şi la fiecare adăugare a unui operator în şirul *x* vom calcula rezultatul curent al expresiei. Dacă operatorul pe care l-am adăugat pe poziția *k* este 4, adică urmează să efectuăm o împărțire, verificăm dacă rezultatul obținut este un număr întreg sau nu.

Pentru rezolvare avem nevoie de un subalgoritm care calculează valoarea expresiei inclusiv la pasul curent *k*:

```
Subalgoritm Calcul(k):
  s ← a[1]
  pentru i=1,k execută:
    dacă x[i]=1 atunci s ← s + a[i+1]
    altfel
      dacă x[i]=2 atunci s ← s - a[i+1]
      altfel
        dacă x[i]=3 atunci s ← s * a[i+1]
        altfel s ← s / a[i+1]
        sfârşit dacă
      sfârşit dacă
    sfârşit dacă
  sfârşit pentru
  Calcul ← s
sfârşit subalgoritm
```

În programul Pascal variabila locală *s* şi funcţia `Calcul(k)` vor fi de tip `Real` din cauza că vom folosi operatorul /.

Condiţiile interne în această problemă au fost definite în enunţ: toate rezultatele parţiale trebuie să fie numere întregi. Subalgoritmul `Posibil(k)` va returna *adevărat* în caz afirmativ şi *false* în caz contrar.

```
Subalgoritm Posibil(k):
  Posibil ← nu ((x[k] = 4) şi
                ((Calcul(k-1)/a[k+1]) ≠ [(Calcul(k-1)/a[k+1]))])
sfârşit subalgoritm
```

Algoritmul de backtracking are câteva particularităţi:

- Elementele din soluţia *x* vor avea valori de la 1 la 4, deoarece cei patru operatori s-au codificat cu cifre.
- Generarea se încheie atunci când s-a ajuns la pasul $n - 1$ şi valoarea expresiei este egală cu *m*.
- Valoarea expresiei se calculează apelând subalgoritmul `Calcul(k)` descris mai sus.
- Dacă nu s-a ajuns la pasul $n - 1$, se continuă generarea.

```
Subalgoritm Back(k):
  pentru i=1,4 execută:                          { avem 4 operatori posibili }
    x[k] ← i                                   { punem un operator în şirul soluţie }
    dacă Posibil(k) atunci                        { verificăm condiţiile interne }
                                                  { dacă am obţinut o soluţie }
        dacă (k = n-1) şi (Calcul(k) = m) atunci
        Afişare(x,este)                                          { o afişăm }
      altfel
        dacă k < n-1 atunci                      { dacă mai avem operanzi }
          Back(k+1)                           { încercăm să le plasăm în expresie }
        sfârşit dacă
      sfârşit dacă
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

În enunţul acestei probleme se cere ca în cazul în care pentru expresia generată nu se poate obţine rezultatul *m* printr-o combinaţie de operatori, să se afişeze mesajul `'imposibil'`. Atunci când afişăm o soluţie, schimbăm valoarea variabilei booleene globale `este` în *adevărat*, urmând ca în programul principal să verificăm valoarea acestei variabile şi să scriem în fişierul de ieşire mesajul respectiv.

## 19.5.4. Maşinuțe

Această problemă se rezolvă prin metoda *backtracking* deoarece trebuie generate *toate* soluțiile. În această idee vom realiza următoarea codificare a datelor:

- $x$ va fi şirul soluție şi vom păstra în $x_i$ numărul de ordine al cutiei în care punem maşinuța $i$;
- în şirul $nr_i$ păstrăm numărul (cantitatea) maşinuțelor puse în cutia $i$;
- în şirul $pus_i$ vom avea valoarea *adevărat* dacă maşinuța $i$ a fost pusă deja în soluția curentă, iar în caz contrar $pus_i$ va avea valoarea *fals*.

**Exemplu**

Fie $n = 2$ şi $m = 3$.
$x = (1, 1)$ cu semnificația că atât maşinuța 1, cât şi maşinuța 2 se introduc în cutia 1;
$x = (1, 2)$ maşinuța 1 se introduce în cutia 1 şi maşinuța 2 în cutia 2;
$x = (2, 1)$ maşinuța 1 se introduce în cutia 2 şi maşinuța 2 în cutia 1;
$x = (2, 2)$ ambele maşinuțe se introduc în cutia 2.

Pentru rezolvarea problemei avem nevoie de un algoritm care verifică condițiile de continuare. Am putea număra aparițiile fiecărei cutii până la pasul curent $k$ şi am putea verifica apoi dacă numerele obținute nu sunt cumva mai mari decât $m$. Renunțăm la această idee şi vom verifica la fiecare pas $k$, dacă maşinuța $k$ nu este pusă deja în vreo cutie şi dacă cutia în care vrem s-o punem nu este deja plină, folosindu-ne de şirurile *pus* şi *nr*.

```
Subalgoritm Posibil(k):
  Posibil ← (nr[i] < m) şi (nu pus[k])
sfârşit subalgoritm
```

Afişarea cerută în enunț are un format special, adică trebuie să scriem cuvântul „cutia”, apoi numărul de ordine al cutiei şi după `': '` numerele maşinuțelor care sunt în acea cutie separate prin câte un spațiu. Subalgoritmul prin care obținem afişarea cerută este:

```
Subalgoritm Afişare(x):
  pentru i=1,n execută:
    scrie 'cutia ',i,': '
    pentru j=1,n execută:
      dacă i = x[j] atunci scrie j,' '
      sfârşit dacă
    sfârşit pentru
  sfârşit pentru
sfârşit subalgoritm
```

Algoritmul de backtracking în formă recursivă este:

```
Subalgoritm Back(k):
  pentru i=1,n execută:                          { putem alege dintre n cutii }
    x[k] ← i                          { încercăm să punem maşinuţa k în cutia i }
    dacă Posibil(k,i) atunci                                  { dacă este posibil }
      pus[k] ← adevărat                  { o punem, deci maşinuţa k este pusă }
      nr[i] ← nr[i] + 1                { creşte numărul maşinuţelor în cutia i }
      dacă k = n atunci                          { dacă am pus şi a n-a maşinuţă }
        Afişare(x)                                  { avem o soluţie, o afişăm }
      altfel
        Back(k+1)    { în caz contrar încercăm să plasăm următoarea maşinuţă }
      sfârşit dacă
      pus[k] ← fals                  { la revenire scoatem maşinuţa k din cutia i, }
      nr[i] ← nr[i] - 1          { deci descreşte numărul maşinuţelor în cutia i }
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

În această implementare a metodei backtracking am înlocuit traversarea şirului soluţiei cu scopul de a verifica dacă o anumită componentă nu este deja prezentă în soluţie, prin utilizarea şirului de valori logice `pus`. În plus, pentru o mai uşoară verificare a spaţiului ocupat în cutii, ne-am folosit de şirul `nr`. Ceea ce apare ca element nou din această cauză se poate formula în felul următor: în momentul alegerii unei componente din spaţiul soluţiilor posibile pentru soluţia curentă, toate entităţile care sunt „atinse" de această alegere se modifică în mod corespunzător (`pus[k]` devine *adevărat*, `nr[i]` creşte), iar în momentul revenirii dintr-un apel recursiv aceste valori se restaurează la starea anterioară (`pus[k]` devine *fals*, `nr[i]` descreşte), deoarece renunţăm la valoarea componentei *k* şi astfel toate modificările făcute din cauza alegerii acesteia la pasul precedent se anulează. Această abordare accelerează algoritmul, deoarece în loc să traversăm un şir, adresăm direct un element (valoare logică). Astfel evităm generarea inutilă a multor soluţii care ulterior s-ar dovedi a fi neinteresante (incorecte).

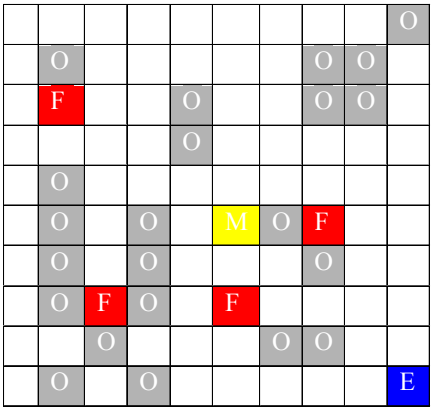
## 19.5.5. Depozit

În problema dată depozitul se memorează printr-un tablou bidimensional.

**Exemplu**

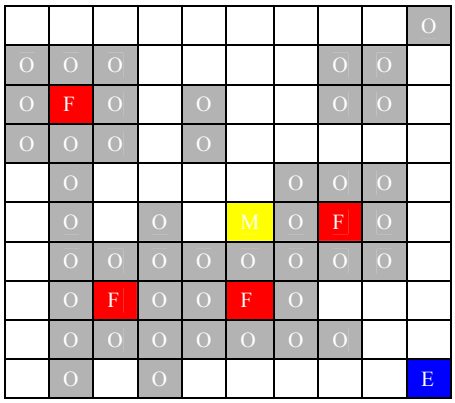
Fie $n = 10$. În figura următoare depozitul codificat cu tabloul de numere se poate vedea în stânga, iar vizualizat mai sugestiv în dreapta:

```
0 0 0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 1 1 0
0 2 0 0 1 0 0 1 1 0
0 0 0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0
0 1 0 1 0 3 1 2 0 0
0 1 0 1 0 0 0 1 0 0
0 1 2 1 0 2 0 0 0 0
0 0 1 0 0 0 1 1 0 0
0 1 0 1 0 0 0 0 0 0
```




În figură sunt marcate compartimentele incendiate, compartimentele în care se află obstacole, locul unde se află magazionerul şi ieşirea.

Enunţul problemei precizează că nu se poate trece prin zonele vecine incendiului. Zone vecine cu focul (pe baza enunţului) sunt toate căsuţele învecinate pe orizontală, verticală şi diagonală. Prin urmare, vom marca aceste zone ca fiind ocupate de „obstacole”. Depozitul arată acum în felul următor:




În secvenţa de algoritm care realizează această prelucrare a depozitului folosim şirul deplasamentelor indicilor de linie şi de coloană ca în problema *Biliard* (capitolul 6).

```
x=(-1,-1,0,1,1,1,0,-1)
y=(0,1,1,1,0,-1,-1,-1)

Subalgoritm Citire(a,n,i0,j0):
  citeşte n                                  { dimensiunea depozitului }
  pentru i=1,n execută:
    pentru j=1,n execută:
      citeşte a[i,j]                         { „conţinutul” depozitului }
  pentru i=1,n execută:                      { căutăm poziţiile incendiate }
    pentru j=1,n execută:
      dacă a[i,j] = 2 atunci
        pentru k=1,8 execută:                { cele 8 poziţii vecine }
          ii ← i + x[k]
          jj ← j + y[k]
```

```
          dacă (ii ∈ {1,2,...,n}) şi (jj ∈ {1,2,...,n}) atunci
                                { dacă poziţia vecină este în interiorul depozitului }
            a[ii,jj] ← 1          { o considerăm de asemenea ocupată }
          altfel
            dacă a[i,j] = 3 atunci
              i0 ← i              { poziţia în care se află magazionerul }
              j0 ← j
            sfârşit dacă
          sfârşit dacă
        sfârşit pentru
      sfârşit dacă
    sfârşit pentru
  sfârşit pentru
sfârşit subalgoritm
```

Coordonatele iniţiale ale magazionerului (`i0` şi `j0`) le determinăm din datele depozitului.

Magazionerul poate traversa depozitul, avansând dintr-o poziţie în alta care este vecină pe orizontală sau verticală cu cea în care se află. Vom pregăti şirul deplasamentelor indicilor de linie şi de coloană ca în problema *Biliard*, de data aceasta din patru componente:

`x=(-1,0,1,0)` şi `y=(0,1,0,-1)`.

Dacă magazionerul se află în poziţia marcată cu M, şirul *x* conţine valori care trebuie adunate valorii indicelui de linie a poziţiei magazionerului pentru a ajunge pe liniile vecine, iar *y* conţine valori care trebuie adunate valorii indicelui de coloană a poziţiei magazionerului pentru a ajunge pe coloanele vecine (marcate cu 1, 2, 3 şi 4 în figura de mai jos).

| | | |
|---|---|---|
| | **1** | |
| **4** | M | **2** |
| | **3** | |

Condiţiile interne sunt precizate în enunţ sub forma restricţiilor în deplasare. Acestora se adaugă şi condiţia de a nu ieşi din depozit în altă parte decât este ieşirea propriu-zisă:

- coordonatele încăperii în care se trece trebuie să fie între 1 şi *n*;
- în încăperea în care se face deplasarea nu poate exista obstacol sau foc;
- prin încăperea în care se trece la pasul curent acest drum nu a trecut încă.

Aceste condiţii sunt verificate în subalgoritmul `Posibil` (`a` este tabloul în care se memorează depozitul, iar `b` este tabloul care reţine drumul parcurs de magazioner):

```
Subalgoritm Posibil(i,j):
          { presupunem că se poate efectua pas în încăperea de coordonate (i,j) }
  Posibil ← adevărat
  dacă nu (i ∈ {1,2,...,n}) sau nu (j ∈ {1,2,...,n}) atunci
                { dacă încăperea se află în afara depozitului, pasul nu este posibil }
    Posibil ← fals
  sfârşit dacă
  dacă a[i,j] ≠ 0 atunci
             { dacă poziția este ocupată de obstacol sau este cuprinsă de incendiu }
    Posibil ← fals
  sfârşit dacă
  dacă b[i,j] ≠ 0 atunci                          { dacă s-a mai trecut pe aici }
    Posibil ← fals
  sfârşit dacă
sfârşit subalgoritm
```

Subalgoritmul `Posibil(i,j)` este apelat în algoritmul `Back(i,j,pas)`, atunci când, la pasul `pas`, din poziția `(i,j)` se încearcă trecerea într-o nouă poziție `(ii,jj)`.

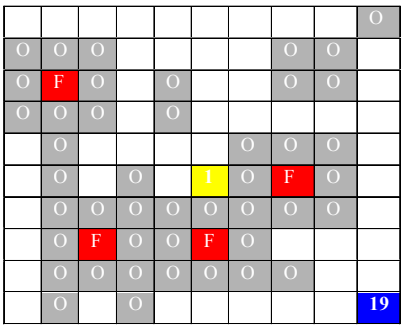
```
Subalgoritm Back(i,j,pas):
  pentru k=1,4 execută:      { se încearcă deplasările în cele 4 poziții vecine }
    ii ← i + x[k]            { se determină indicele de linie a poziției vecine }
    jj ← j + y[k]          { se determină indicele de coloană a poziției vecine }
    dacă Posibil(ii,jj) atunci { dacă pasul în această poziție este posibil }
      b[ii,jj] ← pas                                   { se efectuează pasul }
      dacă ((ii=n) şi (jj=n)) atunci { dacă poziția curentă este ieşirea }
        afişare(b)                                  { se afişează o soluție }
      altfel                                              { în caz contrar }
        Back(ii,jj,pas+1)
               { se încearcă un pas nou care se efectuează din poziția (ii,jj) }
      sfârşit dacă
         { la revenire se şterge ultimul pas efectuat din poziția (i,j) în (ii,jj) }
      b[ii,jj] ← 0{ deoarece din (i,j) vom încerca să trecem în altă poziție }
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

Datele cu care lucrează algoritmul `Back` sunt cunoscute. Mai rămâne să interpretăm modul în care acesta lucrează.

- Se determină coordonatele poziției în care urmează să treacă magazionerul: `(ii,jj)`.

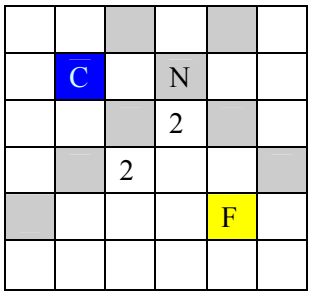
- Se verifică condițiile de continuare prin apelarea algoritmului `Posibil(ii,jj)`.
- Dacă sunt îndeplinite condițiile de continuare, deci `Posibil(ii,jj)` are valoarea *adevărat*, atunci pasul curent se trece în tabloul `b` care reține drumul parcurs spre ieşire.
- Dacă s-a ajuns în colțul din dreapta-jos, atunci se face afişarea tabloului `b`, altfel se continuă deplasarea.

O soluție pentru exemplul considerat este vizualizată în figura următoare. Observăm că traseul trece inutil prin pozițiile 4–9, dar acest lucru nu constituie greşeală din moment ce nu s-au cerut trasee de lungime minimă.




## 19.5.6. Cal şi nebun pe tabla de şah

În figura de mai jos sunt haşurate pozițiile pe care bate nebunul când se află pe poziția N având coordonatele (2, 4). Calul se află inițial în poziția de start marcată cu C (2, 2) şi trebuie să ajungă în poziția finală F (5, 5). El are două trasee de aceeaşi lungime minimă pentru a ajunge în poziția finală, si anume:
(2, 2)(3, 4)(5, 5)
(2, 2)(4, 3)(5, 5)




Prima observație pe care o facem se referă la faptul că în enunț se cer *toate* soluțiile problemei care corespund cerințelor. În acest moment ştim că metoda de rezolvare va fi *backtracking*. Analizând cerințele, constatăm că trebuie să determinăm un minim. În această problemă minimul nu este doar un număr (care de altfel nici nu trebuie afişat la ieşire), ci un şir de coordonate de pe tabla de şah care corespund *traseului de lungime minimă*. În plus, se cer toate aceste şiruri.

Prima idee ar fi să stabilim (conform algoritmului clasic) un astfel de minim, apoi să afişăm toate soluţiile care corespund acestuia. Astfel ar trebui să efectuăm algoritmul realizat cu metoda backtracking o dată pentru stabilirea tuturor soluţiilor, reţinând valoarea minimă a traseului, apoi să executăm *încă odată* acelaşi algoritm pentru găsirea tuturor soluţiilor (traseelor) având lungimea minimă.

Este evident că o astfel de soluţie nu este acceptabilă, deoarece ar conduce la un timp de execuţie mult prea mare. Analizând în continuare problema, ajungem la concluzia că trebuie să găsim o modalitate care nu execută integral procedura care implementează metoda backtracking de două ori.

Mai întâi ne propunem să găsim o rezolvare în care s-o executăm o singură dată. Dacă am putea stabili minimul şirului, traversându-l o singură dată, am găsit simplificarea pe care ne-am propus-o.

În următorul algoritm presupunem că se caută (mai simplu) toate poziţiile din şirul *x* pe care se află valoarea minimă a şirului *x*. Rezultatul se va furniza programului apelant în şirul *rez* de lungime *j*:

```
Subalgoritm Minim(n,x,j,rez):
  min ← x[1]
  j ← 1
  rez[j] ← 1
  pentru i = 2, n execută
    dacă min > x[i] atunci
      j ← 1
      rez[j] ← i
      min ← x[i]
    altfel
      dacă min = x[i] atunci
        j ← j + 1
        rez[j] ← i
      sfârşit dacă
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

Dar în cazul nostru trebuie reţinute soluţii care sunt şiruri de coordonate şi nu doar indici. Alegem altă cale de rezolvare. Vom scrie rezultatul curent în fişier şi atunci când găsim unul care corespunde aceluiaşi minim, deschidem fişierul cu `Append` şi îl scriem, iar când găsim o soluţie corespunzătoare unui minim mai mic decât cel curent, deschidem fişierul cu `ReWrite`, astfel începând scrierea grupului nou de soluţii.

Se poate observa că pe lângă această modalitate mai există şi altele care rezolvă problema. De exemplu, putem „presupune” că traseul minim are lungimea minimă 1. Apelăm procedura `Back(i,j,pas)` şi căutăm trasee având această lungime. Dacă gă-

sim un astfel de traseu, îl scriem în fişier, dacă nu, atunci mărim valoarea minimului la 2 şi căutăm trasee având această lungime. Algoritmul se opreşte, evident, atunci când am găsit cel puţin o soluţie care corespunde minimului presupus la pasul respectiv. Suntem siguri că astfel găsim soluţiile corespunzătoare minimului, deoarece am luat în considerare valorile posibile în ordine crescătoare. Acest algoritm este următorul:

```
Subalgoritm Minim:
  min ← 2
  ok ← fals
  repetă
    Back(x0,y0,2,min,ok)
    min ← min +1
  până când ok
sfârşit algoritm
```

Soluţia o vom codifica într-un tablou `t`, corespunzător tablei de şah, iniţializându-i elementele cu 0. Atunci când în căutarea soluţiei efectuăm un pas permis într-o anumită poziţie, valoarea acelui element va fi egală cu numărul de ordine al pasului. Evident, la revenire vom avea grijă să restabilim valoarea iniţială a elementului respectiv pentru ca în timpul generării unei soluţii noi să putem efectua un pas în această poziţie.

În procedura `Back(i,j,pas)` vom genera poziţiile `(ii,jj)` corespunzătoare paşilor noi pe care un cal le poate efectua dintr-o poziţie curentă `(i,j)` cu ajutorul a două şiruri de constante (`di` şi `dy`) care reprezintă deplasamentele liniilor, respectiv coloanelor celor 8 poziţii în care poate sări un cal.

```
sir=array[1..8] of -2..2;
const di:sir=(-2,-1,1,2,2,1,-1,-2);
      dj:sir=(1,2,2,1,-1,-2,-2,-1);
```

Pentru fiecare astfel de poziţie nouă, vom verifica dacă ea se află sau nu pe tabla de şah (**if** `(ii` **in** `[1..n])` **and** `(jj` **in** `[1..n])`).

Dacă ne aflăm pe tablă, vom verifica dacă această poziţie este în afara bătăii nebunului (aflat în poziţia `(x,y)` cu instrucţiunea **if** `(Abs(x-ii) <> Abs(y-jj))`. Totodată verificăm şi elementul corespunzător din tabloul `t`. Dacă acest element are valoarea 0, înseamnă că încă nu a fost inclus în traseu, deci putem efectua un pas aici. Acum vom verifica dacă această poziţie nu este cumva cea finală, caz în care urmează să ne convingem că lungimea traseului este egală cu lungimea minimă propusă.

Dacă am atins poziţia finală şi numărul paşilor este egal cu `min`, am obţinut o soluţie şi o putem afişa. În caz contrar vom continua traseul început, dar *numai dacă numărul de ordine al pasului curent este mai mic decât lungimea minimă propusă*, deoa-

rece nu are sens să determinăm soluții care conduc la o lungime mai mare decât cea propusă din moment ce acestea nu se valorifică.

În concluzie, algoritmul recursiv care implementează metoda backtracking este următorul:

```
Subalgoritm Back(i,j,pas):
  pentru k=1,8 execută:
    ii ← i + di[k]                          { indicele de linie a poziției noi }
    jj ← j + dj[k]                         { indicele de coloană a poziției noi }
                                           { dacă poziția aleasă este pe tablă }
    dacă (ii ∈ {1,2,...,n}) şi (jj ∈ {1,2,...,n}) atunci
      dacă (|x-ii|) ≠ |y-jj|) şi (t[ii,jj] = 0) atunci
                          { dacă poziția nu se află pe traiectoriile nebunului şi }
                            { în această poziție nu am trecut la un pas anterior }
        t[ii,jj] ← pas                  { efectuăm pasul (marcăm poziția) }
                                          { dacă am ajuns în poziția finală }
        dacă (ii = x1) şi (jj = y1) atunci
                        { dacă numărul paşilor efectuați este egal cu minimul }
          dacă pas = min atunci
            Afişare(t)
            ok ← adevărat
          sfârşit dacă
        altfel
          dacă pas < min atunci
             { dacă numărul paşilor efectuați este mai mic decât minimul curent }
            Back(ii,jj,pas+1)                    { continuăm drumul }
          sfârşit dacă
        sfârşit dacă
        t[ii,jj] ← 0
       { ştergem urma pasului efectuat, deoarece am revenit la o stare anterioară }
      sfârşit dacă
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

Afişarea rezultatului se realizează pe baza valorilor diferite de zero din tabloul `t`. Deoarece ştim că numărul acestor elemente este `min` (accesibilă, deoarece algoritmul s-a terminat când am găsit un traseu având cea mai mică valoare), în tabloul `t` vom căuta valori cuprinse între 1 şi `min`, în această ordine afişând indicii de linie şi de coloană a acestora.

```
Subalgoritm Afişare(t):
  k ← 1
  cât timp k ≤ min execută
    pentru i=1,n execută
      pentru j=1,n execută
        dacă t[i,j] = k atunci
          scrie i, j
          k ← k + 1
        sfârşit dacă
      sfârşit pentru
    sfârşit pentru
  sfârşit cât timp
sfârşit subalgoritm
```