**ALGORITMICĂ. Seminar 7:** Aplicaţii ale programării dinamice şi căutării cu revenire.

---

**Problema 1** (*Problema celui mai lung subşir comun a două şiruri.*) (S+L) Fie $A = (a_1, a_2, \ldots, a_m)$ şi $B = (b_1, b_2, \ldots, b_n)$ două şiruri. Să se determine cel mai lung subşir comun al celor două şiruri, adică $(c_1, c_2, \ldots, c_l)$ cu proprietatea că există $i_1 < i_2 < \ldots < i_l$ şi $j_1 < j_2 < \ldots < j_l$ astfel încât $c_k = a_{i_k} = b_{j_k}$ pentru $k = \overline{1, l}$.

*Rezolvare.* De exemplu pentru şirurile $a = (2, 1, 4, 3, 2)$ şi $b = (1, 3, 4, 2)$ există două subşiruri comune de lungim maximă (3) şi anume: $(1, 4, 2)$ şi $(1, 3, 2)$.

(a) *Analiza structurii unei soluţii optime.* Fie $c = (c_1, c_2, \ldots, c_{l-1}, c_l)$ o soluţie optimă. Presupunem că $c_l = a_i = b_j$. $c$ poate fi considerată soluţie optimă a problemei $P(i, j)$ ce constă în determinarea celui mai lung subşir comun al şirurilor $a_{1..i} = (a_1, \ldots, a_i)$ şi $b_{1..j} = (b_1, \ldots, b_j)$. Se poate demonstra prin reducere la absurd că $(c_1, c_2, \ldots, c_{l-1}$ este soluţie optimă a subproblemei $P(i-1, j-1)$.

(b) *Deducerea unei relaţii de recurenţă.* Se notează cu $L(i, j)$ numărul de elemente al celui mai lung subşir comun al lui $(a_1, a_2, \ldots, a_i)$ şi $(b_1, b_2, \ldots, b_j)$. Relaţia de recurenţă pentru calculul lui $L(i, j)$ este:

$$L(i,j) = \begin{cases} 0 & \text{dacă } i = 0 \text{ sau } j = 0 \\ L(i-1, j-1) + 1 & \text{dacă } a_i = b_j \\ \max\{L(i-1, j), L(i, j-1)\} & \text{în celelalte cazuri} \end{cases}$$

În cazul exemplului de mai sus matricea $L$ are următoarea structură:

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 2 & 2 \\ 0 & 1 & 2 & 2 & 2 \\ 0 & 1 & 2 & 2 & 3 \end{matrix}$$

(c) *Dezvoltarea relaţiei de recurenţă.* Este descrisă în Algoritmul 1.

(d) *Construirea soluţiei.* Dacă $a_i = b_j$ atunci soluţia problemei $P(i, j)$ se obţine din soluţia subproblemei $P(i-1, j-1)$ prin adăugarea valorii comune celor două şiruri $a_i = b_j$. Dacă $a_i \neq b_j$ atunci soluţia lui $P(i, j)$ coincide cu soluţia lui $P(i-1, j)$ (dacă $L[i-1, j] \geq L[i, j-1]$) respectiv cu soluţia lui $P(i, j-1)$ (dacă $L[i-1, j] < L[i, j-1]$). Presupunând că tabloul în care se colectează soluţia ($x$) şi variabila ce conţine numărul de elemente ale subşirului comun ($k$) sunt variabile globale (iar $k$ este iniţializată cu 0), algoritmul pentru construirea soluţiei este descris recursiv în 2.

**Problema 2** (*Problema monedelor.*)(S) Se consideră monede de valori $d_n > d_{n-1} > \ldots > d_1 = 1$. Să se găsească o acoperire minimală (ce foloseşte cât mai puţine monede) a unei sume date $S$.

*Rezolvare.* Întrucât există monedă de valoare 1 orice sumă poate fi acoperită exact. In cazul general, tehnica greedy (bazată pe ideea de a acoperi cât mai mult posibil din suma cu moneda de valoarea cea mai mare) nu conduce întotdeauna la soluţia optimă. De exemplu dacă $S = 12$ şi se dispune de monede cu valorile 10, 6, 1 atunci aplicând tehnica greedy s-ar folosi o monedă de valoare 10 şi două monede de valoare 1. Soluţia optimă este însă cea care foloseşte două monede de valoare 6.

(a) *Analiza structurii unei soluţii optime.* Considerăm problema generică $P(i, j)$ care constă în acoperirea sumei $j$ folosind monede de valori $d_1 < \ldots < d_i$. Soluţia optimă corespunzătoare acestei probleme va fi un şir, $(s_1, s_2, \ldots, s_k)$ de valori corespunzătoare monedelor care acoperă suma $j$. Dacă $s_k = d_i$ atunci $(s_1, s_2, \ldots, s_{k-1})$ trebuie să fie soluţie optimă pentru subproblema $P(i, j-d_i)$ ($i$ rămâne nemodificat întrucât pot fi folosite mai multe monede de aceeaşi valoare). Dacă $s_k \neq d_i$ atunci $(s_1, s_2, \ldots, s_{k-1})$ trebuie să fie soluţie optimă a subproblemei $P(i-1, j)$.

**Algorithm 1** Dezvoltarea relaţiei de recurenţă pentru determinarea celui mai lung subşir comun

```
construire (integer a[1..m], b[1..n])
for i ← 0, m do
   L[i, 0] ← 0
end for
for j ← 0, n do
   L[0, j] ← 0
end for
for i ← 1, m do
   for j ← 1, n do
      if a[i] = b[j] then
         L[i, j] ← L[i − 1, j − 1] + 1
      else
         if L[i − 1, j] > L[i, j − 1] then
            L[i, j] ← L[i − 1, j]
         else
            L[i, j] ← L[i, j − 1]
         end if
      end if
   end for
end for
return L[0..m, 0..n]
```

**Algorithm 2** Determinarea celui mai lung subşir comun a două şiruri

```
solutie(i, j)
if L[i, j] ≠ 0 then
   if a[i] = b[j] then
      solutie(i-1,j-1)
      k ← k + 1
      x[k] ← a[i]
   else
      if L[i − 1, j] ≥ L[i, j − 1] then
         solutie(i − 1, j)
      else
         solutie(i, j − 1)
      end if
   end if
end if
```

**Algorithm 3** Dezvoltarea relaţiei de recurenţă în cazul problemei monedelor şi construirea soluţiei

```
completare(d[1..n],S)
for i ← 1,n do
  R[i,0] ← 0
end for
for j ← 1,s do
  R[1,j] ← j
end for
for i ← 1,n do
  for j ← 1,s do
    if j < d[i] then
      R[i,j] ← R[i−1,j]; Q[i,j] ← 0
    else
      if R[i−1,j] < 1 + R[i,j−d[i]] then
        R[i,j] ← R[i−1,j]; Q[i,j] ← 0
      else
        R[i,j] ← R[i,j−d[i]]+1;Q[i,j] ← 1
      end if
    end if
  end for
end for
```

```
solutie(i,j)
if j ≠ 0 then
  if Q[i,j] = 1 then
    k ← k+1
    a[k] ← d[i]
    solutie(i,j−d[i])
  else
    solutie(i−1,j)
  end if
end if
```

(b) *Deducerea relaţiei de recurenţă.* Fie $R(i,j)$ numărul minim de monede de valori $d_1,\ldots,d_i$ care acoperă suma $j$. $R(i,j)$ satisface:

$$R(i,j)=\begin{cases} 0 & j=0\\ j & i=1\\ R(i-1,j) & j<d_i\\ \min\{R(i-1,j),1+R(i,j-d_i)\} & j\geq d_i\end{cases}$$

Pentru exemplul de mai sus se obţine matricea:

| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 6 | 0 | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 | 6 | 2 |
| 10 | 0 | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 1 | 2 | 2 |

Pentru a uşura construirea soluţiei se poate construi încă o matrice $Q[1..n,0..S]$ caracterizată prin:

$$Q(i,j)=\begin{cases} 1 & d_i \text{ se foloseşte pentru acoperirea sumei } j\\ 0 & d_i \text{ nu se foloseşte pentru acoperirea sumei } j\end{cases}$$

In cazul exemplului analizat matricea $Q$ va avea următorul conţinut:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

(c)*Dezvoltarea relaţiei de recurenţă.* Matricile $R$ şi $Q$ pot fi completate după cum este descris în Algoritmul 3 ($R[n,s]$ reprezintă numărul minim de monede necesare pentru a acoperi suma $S$).

(d)*Construirea solutiei.* Considerând că tabloul $a$ în care se colectează soluţia şi variabila $k$ ce conţine numărul de elemente ale lui $a$ sunt variabile globale, soluţia poate fi construită în manieră recursivă aşa cum e descris în algoritmul 3.

**Problema 3** (*Problema submulţimii de sumă dată.*)(L) Fie $A = \{a_1, \ldots, a_n\}$ o mulţime de valori naturale şi $c$ un număr natural. Să se determine o submulţime $S \subset A$ pentru care suma elementelor nu depăşeşte valoarea $c$ dar este maximală ($\sum_{s \in S} s \leq c$ şi $\sum_{s \in S} s$ este maximă).

*Rezolvare.* In cazul general problema $P(i, j)$ se referă la determinarea unei submulţimi a mulţimii $\{a_1, \ldots, a_i\}$ pentru care suma elementelor este cât mai apropiată de $j$. Notând cu $S(i, j)$ suma elementelor unei soluţii optimale a problemei $P(i, j)$ relaţia de recurenţă corespunzătoare este:

$$S(i,j) = \begin{cases} 0 & i = 0 \text{ sau } j = 0 \\ S(i-1,j) & a_i > j \\ \max\{S(i-1,j), S(i-1,j-a_i) + a_i\} & a_i \leq j \end{cases}$$

O dată completată matricea $S[0..n, 0..c]$, soluţia $R[1..k]$ poate fi construită ca în Algoritmul 4.

**Algorithm 4** Determinarea unei submulţimi de sumă dată

```
solutie(i, j)
if i ≠ 0 and j ≠ 0 then
  if a[i] > j or (a[i] ≤ j and S[i − 1, j] > S[i − 1, j − a_i] + a_i) then
    solutie(i − 1, j)
  else
    solutie(i − 1, j − a[i]); k ← k + 1; R[k] ← a[i]
  end if
end if
```

**Problema 4** (*Calculul distanţei de editare.*)(S+L) Fie $x[1..m]$ şi $y[1..n]$ două şiruri. Distanţa de editare (distanţa Levenshtein) dintre cele două şiruri se defineşte ca fiind numărul minim de operaţii de înlocuire/ inserţie/ ştergere element necesare pentru a transforma unul dintre şiruri în celălalt. De exemplu cuvântul "carte" este la distanţa 1 faţă de cuvintele "caste" (un caracter înlocuit), "care" (un caracter eliminat) sau "cartel" (un caracter inserat), la distanţa 2 faţă de cuvântul "castel" ("carte"→"caste"→"castel") şi la distanţa 3 faţă de cuvântul "caiet" ("carte"→"cairte" → "caiete" → "caiet"). Aceasta distanţă este utilizată în bioinformatică, la compararea secvenţelor ADN şi în "spell checking".

*Rezolvare.* Problema generică $P(i, j)$ poate fi formulată ca fiind cea a determinării distanţei de editare dintre şirurile parţiale $x[1..i]$ şi $y[1..j]$. La fel ca în cazul problemei determinării celui mai lung subşir comun $P(i, j)$ se reduce la una dintre subproblemele $P(i-1, j-1)$, $P(i, j-1)$ respectiv $P(i-1, j)$. Notând cu $d(i, j)$ distanţa de editare dintre $x[1..i]$ şi $y[1..j]$, relaţia de recurenţă asociată poate fi descrisă prin:

$$d(i,j) = \begin{cases} i & j = 0 \\ j & i = 0 \\ d(i-1,j-1) & x[i] = y[j] \\ \min\{d(i-1,j-1)+1, d(i,j-1)+1, d(i-1,j)+1\} & x[i] \neq y[j] \end{cases}$$

În cazul în care valoarea minimă se obţine pentru $d(i-1, j-1) + 1$ înseamnă că e necesară înlocuirea elementului $x[i]$ cu elementul $y[j]$. Dacă valoarea minimă se obţine pentru $d(i, j-1) + 1$ înseamnă că e necesară înserţia elementului $y[j]$, iar dacă valoarea minimă se obţine pentru $d(i-1, j) + 1$ atunci e necesară eliminarea elementului $x[i]$.

Matricea de distanţe pentru cazul cuvintelor "carte" şi "caiet" este:

**Algorithm 5** Utilizarea tehnicii memoizării în calculul distanţei de editare

```
memo(integer i, j)
if i >= 0 and j >= 0 then
  if d[i, j] = −1 then
    if i = 0 then
      d[i, j] ← j
    else
      if j = 0 then
        d[i, j] ← i
      else
        if x[i] = y[j] then
          d[i, j] ← memo(i − 1, j − 1)
        else
          d[i, j] ← min(memo(i − 1, j − 1) + 1,memo(i, j − 1) + 1,
                  memo(i − 1, j) + 1)
        end if
      end if
    end if
  end if
  return d[i, j]
end if
```

| | | c | a | i | e | t |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 0 | 1 | 2 | 3 | 4 | 5 |
| c 1 | 1 | 0 | 1 | 2 | 3 | 4 |
| a 2 | 2 | 1 | 0 | 1 | 2 | 3 |
| r 3 | 3 | 2 | 1 | 1 | 2 | 3 |
| t 4 | 4 | 3 | 2 | 2 | 2 | 2 |
| e 5 | 5 | 4 | 3 | 3 | 2 | 3 |

Matricea de distanţe poate fi completată în mod clasic sau se poate folosi tehnica memoizării care permite completarea doar a elementelor necesare pentru calculul lui $d(m, n)$. Tehnica se caracterizează prin faptul că matricea $d$ este iniţializată cu o valoare virtuală (de exemplu, $-1$) iar completarea elementelor se face in manieră top-down aşa cum este ilustrat în Algoritmul 5.

Pentru exemplul de mai sus matricea de distanţe completată folosind tehnică memoizării are următorul conţinut:

| | | c | a | i | e | t |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 0 | 1 | 2 | 3 | 4 | -1 |
| c 1 | 1 | 0 | 1 | 2 | 3 | -1 |
| a 2 | 2 | 1 | 0 | 1 | 2 | -1 |
| r 3 | 3 | 2 | 1 | 1 | 2 | -1 |
| t 4 | 4 | 3 | 2 | 2 | 2 | 2 |
| e 5 | -1 | -1 | -1 | -1 | 2 | 3 |

Vizualizarea operaţiilor de editare prin care şirul $x[1..m]$ poate fi transformat în şirul $y[1..n]$ poate fi realizată prin Algoritmul 6.

**Problema 5** (S) Fie $a[1..n, 1..n]$ o matrice de valori reale. Se defineşte un traseu în matrice ca o succesiune de elemente $a[i_1, 1], a[i_2, 2], \ldots, a[i_n, n]$ cu proprietatea că $i_{k+1} \in \{i_k, i_k - 1, i_k + 1\}$ (evident dacă $i_k = 1$ atunci $i_{k+1} \in \{i_k, i_k + 1\}$ iar dacă $i_k = n$ atunci $i_{k+1} \in \{i_k, i_k - 1\}$). Un astfel de traseu vizitează câte

**Algorithm 6** Determinarea operaţiilor corespunzătoare distanţei de editare

```
transformari(integer x[1..m], y[1..n], d[1..m, 1..n])
i ← m; j ← n
while i > 0 and j > 0 and d[i, j] > 0 do
  if x[i] = y[j] then
    i ← i − 1; j ← j − 1;
  else
    if d[i, j] = d[i − 1, j − 1] + 1 then
      "inlocuire x[i] cu y[j]"; i ← i − 1; j ← j − 1;
    else
      if d[i, j] = d[i, j − 1] + 1 then
        "inserare y[j]"; j ← j − 1;
      else
        "stergere x[i]"; i ← i − 1;
      end if
    end if
  end if
end while
if i = 0 then
  while j > 0 and d[i, j] > 0 do
    "inserare y[j]"; j ← j − 1
  end while
end if
if j = 0 then
  while i > 0 and d[i, j] > 0 do
    "stergere x[i]"; i ← i − 1
  end while
end if
```

un element de pe fiecare coloană trecând de la un element curent la unul "vecin" de pe coloană următoare (modulul diferenţei dintre indicii de linie este cel mult 1). Să se determine un traseu având suma elementelor maximă.

*Rezolvare.* Fie $P(i,j)$ problema determinării unui traseu de sumă maximă care se termină în $a[i,j]$. Soluţia optimă a acestei probleme conţine soluţia optimă a uneia dintre subproblemele $P(i-1,j-1)$, $P(i,j-1)$ respectiv $P(i+1,j-1)$. Dacă notăm cu $V(i,j)$ suma asociată traseului optim care se termină în $a[i,j]$ atunci relaţia de recurenţă asociată este:

$$V(i,j) = \begin{cases} a(i,j) & j = 1 \\ \max\{V(i-1,j-1) + a(i,j), V(i,j-1) + a(i,j), & j > 1 \\ \qquad V(i+1,j-1) + a(i,j)\} & \end{cases}$$

Să considerăm următorul exemplu:

$$A = \begin{bmatrix} 10 & 4 & 1 & 6 \\ 3 & -2 & 8 & 2 \\ 2 & 15 & -6 & 4 \\ 7 & -3 & 4 & 9 \end{bmatrix}$$

Printr-o abordare de tip greedy (în care se porneşte de la cel mai mare element de pe prima coloană şi la fiecare etapă se alege elementul cel mai mare din cele vecine aflate pe coloana următoare) s-ar obţine traseul

(10,4,8,6) având suma 28. Aplicând relaţia de recurenţă de mai sus se obţine matricea:

$$V = \begin{bmatrix} 10 & 14 & 15 & 36 \\ 3 & 8 & 30 & 32 \\ 2 & 22 & 16 & 34 \\ 7 & 4 & 26 & 35 \end{bmatrix}$$

ceea ce conduce la traseul: (7,15,8,6) având suma 36.

O dată completată matricea $V$, soluţia ($s = (i_1, i_2, \ldots, i_n)$) poate fi construită începând de la ultimul element aşa cum este descris în Algoritmul 7.

**Algorithm 7** Determinarea unui traseu de sumă maximă

```
solutie(integer a[1..n, 1..n], s[1..n], V[1..n, 1..n])
s[n] ← imax(V[1..n, n]) // indicele celui mai mare element din ultima
                        //coloană a matricii V
for k ← n − 1, 1, −1 do
  if V[s[k + 1], k + 1] = V[s[k + 1], k] + a[s[k + 1], k + 1] then
    s[k] ← s[k + 1]
  else
    if s[k + 1] > 1 and V[s[k + 1], k + 1] = V[s[k + 1] − 1, k] + a[s[k + 1], k + 1] then
      s[k] ← s[k + 1] − 1
    else
      if s[k + 1] < n and V[s[k + 1], k + 1] = V[s[k + 1] + 1, k] + a[s[k + 1], k + 1] then
        s[k] ← s[k + 1] + 1
      end if
    end if
  end if
end for
```

**Problema 6** (*Problema turistului în Manhattan.*) (S) Se consideră o grilă pătratică $n \times n$ (care poate fi interpretată ca harta străzilor din Manhattan). Fiecare muchie în cadrul grilei are asociată o valoare (ce poate fi interpretată ca fiind câştigul turistului care parcurge porţiunea respectivă de stradă). Se caută un traseu care porneşte din nodul (1,1), se termină în nodul $(n, n)$, şi care are proprietatea că la fiecare etapă se poate trece din nodul $(i, j)$ fie în nodul $(i, j + 1)$ (deplasare la dreapta) fie în nodul $(i + 1, j)$ (deplasare în jos). În plus valoarea asociată traseului trebuie să fie maximă.

*Rezolvare.* Presupunem că valorile asociate muchiilor grilei sunt stocate în două matrici: $C_D[1..n, 1..n-1]$ ($C_D[i, j]$ conţine valoarea asociată trecerii din nodul $(i, j)$ în nodul $(i, j + 1)$) iar $C_J[1..n-1, 1..n]$ conţine valoarea asociată trecerii din nodul $(i, j)$ în nodul $(i + 1, j)$.

Dacă notăm cu $P(i, j)$ problema determinării unui traseu optim care se termină în $(i, j)$ atunci soluţia optimă a acestei probleme conţine soluţia optimă a uneia dintre subproblemele $P(i, j-1)$ respectiv $P(i-1, j)$. Notând cu $C(i, j)$ valoarea asociată soluţiei optime a problemei $P(i, j)$, relaţia de recurenţă asociată este:

$$C(i,j) = \begin{cases} 0 & i = 1, j = 1 \\ C(i, j-1) + C_D(i, j-1) & i = 1, j > 1 \\ C(i-1, j) + C_J(i-1, j) & i > 1, j = 1 \\ \max\{C(i, j-1) + C_D(i, j-1), & i > 1, j > 1 \\ \quad C(i-1, j) + C_J(i-1, j)\} & \end{cases}$$

După completarea matricii $C$ (folosind fie varianta clasică fie cea bazată pe tehnica memoizării) elementul $C(n, n)$ indică valoarea asociată soluţiei optime. Traseul propriu-zis poate fi determinat simplu dacă o dată cu construirea matricii $C$ se construieşte şi o matrice $P$ ale cărei elemente indică direcţia asociată ultimului pas făcut pentru a ajunge în nodul respectiv: "dreapta" sau "jos":

$$P(i,j) = \begin{cases} "dreapta" & i=1, j>1 \\ "jos" & i>1, j=1 \\ "dreapta" & i>1, j>1, \\ & C(i,j-1)+C_D(i,j-1) > C(i-1,j)+C_J(i-1,j) \\ "jos" & i>1, j>1, \\ & C(i,j-1)+C_D(i,j-1) < C(i-1,j)+C_J(i-1,j) \end{cases}$$

In varianta recursivă algoritmul poate fi descris prin:

```
traseu(integer i, j)
if i > 1 or j > 1 then
  if P[i,j] = "jos" then
    traseu(i, j-1); write "jos"
  else
    traseu(i-1, j); write "dreapta"
  end if
end if
```

Pentru a obţine traseul se apelează algoritmul `traseu`$(n,n)$.

**Problema 7** (*Problema submulţimii de sumă dată.*) (L) Fie $A = \{a_1, a_2, \ldots, a_n\}$ o mulţime de valori întregi. Să se determine toate submulţimile lui $A$ pentru care suma elementelor este egală cu o valoare dată $V$.

*Rezolvare.*(a)*Reprezentarea soluţiei.* O soluţie a problemei este o submulţime $S$ a mulţimii $A$. Aceasta poate fi reprezentată prin tabloul elementelor sale $S = (s_1, \ldots, s_m)$ cu $m \leq n$ şi $s_i \in A$. Deci mulţimile de valori posibile pentru componentele soluţiei sunt toate egale cu $A$.

(b) *Restricţii şi condiţii de continuare.* O soluţie trebuie să satisfacă următoarele condiţii: $s_i \neq s_j$ pentru orice $i \neq j$ şi $\sum_{i=1}^{m} s_i = V$. Condiţia de continuare dedusă din aceste restricţii este $s_k \neq s_i$ pentru $i = \overline{1, k-1}$. În cazul în care elementele lui $A$ şi valoarea $V$ ar fi valori naturale s-ar putea adăuga ca şi condiţie de continuare $\sum_{i=1}^{k} s_i \leq V$.

(c)*Criteriul de decizie pentru a verifica că s-a obţinut o soluţie finală.* O soluţie parţială $(s_1, \ldots, s_k)$ poate fi considerată soluţie finală dacă $\sum_{i=1}^{k} s_i = V$.

Cu aceste ipoteze algoritmul de generare a tuturor submulţimilor cu suma $V$ este descris în 8.

**Algorithm 8** Generarea submulţimilor de sumă dată

```
subm(integer k)
if suma(s[1..k-1]) = V then
  write s[1..k-1]
else
  if k ≤ n then
    for i ← 1, n do
      s[k] ← a[i]
      if validare(s[1..k]) =true then
        subm(k+1)
      end if
    end for
  end if
end if
```

Algoritmul `suma` returnează suma elementelor şirului pentru care este apelat iar algoritmul `validare` verifică dacă $s[k]$ este diferit de elementele şirului $s[1..k-1]$ (acelaşi criteriu de validare ca la problema generării permutărilor). Algoritmul recursiv `subm` va fi apelat pentru $k = 1$. Tablourile $a[1..n]$ şi $s[1..n]$ sunt considerate variabile globale.

**Problema 8** (*Problema comis voiajorului (TSP-Traveling Salesman Problem)).* (S) Se consideră un set de $n$ oraşe şi un comis voiajor care trebuie să viziteze toate cele $n$ oraşe pornind din primul oraş, trecând o singură dată prin fiecare oraş şi întorcându-se în primul oraş. Se consideră cunoscută matricea $D[1..n, 1..n]$ în care elementul $D[i, j]$ este 0 dacă nu există drum direct între oraşul $i$ şi oraşul $j$ respectiv lungimea drumului direct dintre cele două oraşe. (i) Să se genereze toate circuitele pe care le poate parcurge comis voiajorul pornind din oraşul 1; (ii) Să se determine cel mai scurt circuit.

*Rezolvare.* În ambele variante soluţia poate fi reprezentată printr-un şir $(s_1, \ldots, s_n)$ unde $s_i \in \{1, \ldots, n\}$ reprezintă indicele oraşului vizitat la etapa $i$. Pentru a obţine circuitul trebuie doar adăugat oraşul de pornire (de exemplu, 1). Restricţiile ce trebuie satisfăcute sunt: $s_i \neq s_j$ pentru orice $i \neq j$ (nu se vizitează de două ori acelaşi oraş) şi $D(s_{i-1}, s_i) > 0$ pentru $i = \overline{2, n}$ (există drum direct între oraşele vizitate la etape succesive).

Condiţiile de continuare deduse din restricţiile de mai sus sunt: $s_k \neq s_i$ pentru $i = \overline{1, k-1}$ şi $D(s_{k-1}, s_k) > 0$. În cazul căutării celui mai scurt traseu numărul traseelor generate poate fi redus dacă se completează condiţiile de continuare cu $\sum_{i=2}^{k} D(s_{i-1}, s_i) < Lmin$ unde $Lmin$ este lungimea celui mai scurt traseu generat până la momentul curent. Variabila $Lmin$ se iniţializează înainte de apelul algoritmului de generare cu o valoare suficient de mare ($Lmin = \infty$). In ambele variante ale problemei se consideră că s-a obţinut o soluţie când au fost completate toate cele $n$ componente. Pentru a obţine lungimea circuitului se adaugă la suma $\sum_{i=2}^{n} D(s_{i-1}, s_i)$ distanţa dintre oraşul $s_n$ şi primul oraş (stocată în $D(s_n, 1)$).

În prima variantă, algoritmul este descris în 9.

**Algorithm 9** Problema comis voiajorului. Generarea tuturor circuitelor (stânga). Generarea unui circuit de lungime minimă (dreapta)

```
TSP(integer k)
if k − 1 = n then
   write s[1..n]
else
   for i ← 2, n do
      s[k] ← i
      if validare(s[1..k]) =true
      then
         TSP(k + 1)
      end if
   end for
end if
```

```
1:  optTSP(integer k)
2:  if k − 1 = n then
3:     if L + D[s[n], s[1]] < Lmin then
4:        Lmin ← L + D[s[n], s[1]]
5:        smin[1..n] ← s[1..n]
6:     end if
7:  else
8:     for i ← 2, n do
9:        s[k] ← i
10:       if validare(s[1..k]) =true then
11:          L ← L + D[s[k − 1], s[k]]
12:          optTSP(k + 1)
13:          L ← L − D[s[k − 1], s[k]]
14:       end if
15:    end for
16: end if
```

Înainte de apelul lui `TSP`, $s[1]$ se setează pe 1 (se porneşte întotdeauna din primul oraş) iar algoritmul se apelează pentru $k = 2$. În algoritmul `validare` se verifică faptul că $s[k] \neq s[i]$ pentru $i = \overline{1, k-1}$ şi faptul că $D[s[k-1], s[k]] > 0$. În cazul în care una dintre aceste condiţii este încălcată se returnează **false**.

În a doua variantă se folosesc variabilele globale $smin[1..n]$ şi $Lmin$ pentru a stoca cel mai scurt traseu generat până la momentul curent, respectiv lungimea lui. $Lmin$ se iniţializează cu o valoare suficient de mare. În plus se foloseşte variabila globală $L$ care conţine la fiecare etapă lungimea traseului dintre primul şi ultimul oraş vizitat. Înainte de apel $L$ se iniţializează cu 0. Algoritmul este descris în 9. Algoritmul `optTSP` poate fi transformat pe baza ideii de la tehnica "ramifică şi mărgineşte" calculând pentru fiecare soluţie parţială o margine inferioară a lungimii traseului şi efectuând apelul recursiv de la linia 12 doar dacă această limită este mai mică decât $Lmin$ (iniţializată înaintea apelului `optTSP`(1) cu o valoare suficient de mare - de exemplu $n \max_{i,j} D_{ij}$). O variantă de calcul a limitei inferioare pentru lungimea traseului în cazul soluţiei

parţiale $(s_1, \ldots, s_k)$ este:

$$L_B = \sum_{i=1}^{k-1} D_{s_i s_{i+1}} + (n-k) \sum_{i=k+1}^{n} d_i + \min_{i=\overline{k+1,n}} D_{i1}$$

unde $d_i$ este media aritmetică a distanţelor dintre oraşul $i$ şi cele mai apropiate alte două oraşe care nu au fost vizitate încă.

**Problema 9** (*Problema labirintului.*)(S) Se consideră o grilă pătratică $n \times n$ în care anumite celule sunt libere iar altele sunt ocupate. Celulele libere ale grilei pot fi vizitate prin deplasare din poziţia curentă în oricare dintre poziţiile libere vecine (stânga, dreapta, sus, jos). Presupunând că celulele $(1,1)$ şi $(n,n)$ sunt libere să se genereze toate traseele care permit trecerea doar prin celule libere de la $(1,1)$ la $(n,n)$. Un exemplu de labirint si de traseu care uneşte cele două celule este ilustrat în Figura 1.

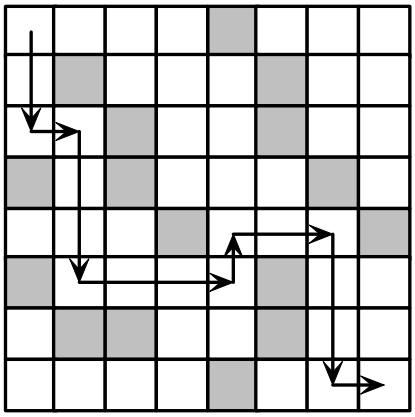

Figure 1: Exemplu de labirint şi de traseu care uneşte colţul din stânga sus cu colţul din dreapta jos

*Rezolvare.* Considerăm că informaţia privind starea unei celule este stocată în matricea $M[1..n, 1..n]$ în care $M[i,j]$ este 0 dacă celula e liberă şi 1 în caz contrar. O soluţie a problemei este un şir de perechi de coordonate identificând celulele libere prin care se trece: $s = (s_1, \ldots, s_m)$ cu $s_l = (i_l, j_l)$. Restricţiile ce trebuie satisfăcute de către soluţie sunt: elementele lui $s$ sunt perechi distincte şi se trece doar prin celule libere ($M[s[l].i, s[l].j] = 0$). Când se ajunge în celula $(n,n)$ se consideră că s-a obţinut o soluţie. Dintr-o celulă de coordonate $(i,j)$ se poate trece într-una dintre cele patru celule vecine având coordonatele: $(i-1,j)$, $(i+1,j)$, $(i,j-1)$ şi $(i,j+1)$ (ţinându-se cont de limitările existente pentru celulele de pe frontieră). Algoritmul de validare returnează **false** dacă cel puţin una dintre următoarele condiţii este încălcată:

1. $1 \leq s[k].i \leq n$, $1 \leq s[k].j \leq n$

2. $M[s[k].i, s[k].j] = 0$

3. $s[k] \neq s[l]$, $l = \overline{1, k-1}$

Algoritmul care generează traseele este descris în 10.

**Problema 10** (*Turul calului.*)(S) Se consideră un cal plasat pe o poziţie $(i_0, j_0)$ pe o tablă de şah. Se pune problema determinării unei secvenţe de poziţii prin care trebuie să treacă calul pentru a vizita o singură dată fiecare poziţie.

*Indicaţie.* Soluţia poate fi reprezentată printr-un tablou conţinând 64 de perechi distincte de indici $(i,j)$ $(i,j \in \{1, \ldots, 8\})$. Două perechile consecutive ale soluţiei, $(i,j)$ şi $(i',j')$ trebuie să reprezinte poziţii între

**Algorithm 10** Determinarea unui traseu prin labirint

```
labirint(integer k)
if s[k − 1] = (n, n) then
  write s[1..k − 1]
else
  s[k] ← (s[k − 1].i − 1, s[k − 1].j)
  if validare(s[1..k]) then
    labirint (k + 1)
  end if
  s[k] ← (s[k − 1].i + 1, s[k − 1].j)
  if validare(s[1..k]) then
    labirint (k + 1)
  end if
  s[k] ← (s[k − 1].i, s[k − 1].j − 1)
  if validare(s[1..k]) then
    labirint (k + 1)
  end if
  s[k] ← (s[k − 1].i, s[k − 1].j + 1)
  if validare(s[1..k]) then
    labirint (k + 1)
  end if
end if
```

care calul se poate deplasa printr-o singură mutare, adică: $|i - i'| = 1$ şi $|j - j'| = 2$ sau $|i - i'| = 2$ şi $|j - j'| = 1$.

**Problema 11** (*Partiţionarea unui număr.*)(L) Pentru un număr natural $C$ să se determine toate mulţimile de numere naturale care au proprietatea că suma elementelor lor este egală cu $C$.

*Indicaţie.* Problema este similară cu cea a determinării submulţimilor de sumă dată.