# Capitolul 8

# Tehnica programării dinamice

Programarea dinamică (introdusă în 1950 de R. Bellman pentru rezolvarea problemelor de optimizare) este o metodă bazată pe construirea şi utilizarea unor tabele cu informaţii. La construirea tabelelor pentru completarea unui element se folosesc elemente completate anterior (construirea se realizează în manieră dinamică). Această tehnică se bazează pe descompunerea unei probleme în subprobleme şi este adecvată rezolvării problemelor (în particular celor de optimizare) care au următoarele proprietăţi:

- *Proprietatea de substructură optimă.* Orice soluţie optimă este constituită din soluţii optime ale subproblemelor. Această proprietatea este specifică şi problemelor ce pot fi rezolvate prin tehnica greedy. Pentru a verifica dacă o problemă posedă această proprietate se poate folosi metoda reducerii la absurd.
- *Proprietatea de suprapunere a problemelor.* Pentru ca programarea dinamică să fie eficientă este necesar ca numărul subproblemelor ce trebuie efectiv rezolvate în scopul obţinerii soluţiei problemei iniţiale să fie relativ mic (polinomial în raport cu dimensiunea problemei). Aceasta înseamnă că în procesul de descompunere a problemei se ajunge de mai multe ori la aceeaşi subproblemă care însă va fi rezolvată o singură dată iar soluţia ei va fi reţinută într-un tabel. Ideea descompunerii problemei în subprobleme este folosită şi în metoda divizării însă în acel caz era important ca subproblemele să fie independente.

## 8.1 Principiul tehnicii

La aplicarea tehnicii programării dinamice se parcurg următoarele etape:

(a) Analiza structurii unei soluţii optime şi a proprietăţilor acesteia prin punerea în evidenţă a relaţiei existente între soluţia problemei şi soluţiile subproblemelor (se verifică dacă problema posedă proprietatea de substructură optimă).

(b) Stabilirea unei relaţii de recurenţă referitoare la criteriul de optimizat sau la valoarea ce trebuie calculată. Aceasta descrie legătura dintre valoarea criteriului de optim corespunzător problemei şi cele corespunzătoare subproblemelor.

(c) Calculul valorii asociate soluţiei optime dezvoltând relaţia de recurenţă într-o manieră *ascendentă* şi reţinând valorile calculate asociate subproblemelor într-o structură tabelară. În funcţie de problemă, în această etapă se pot reţine şi alte informaţii care vor fi utilizate în momentul construirii soluţiei.

(d) Construirea unei soluţii optime folosind informaţiile determinate şi reţinute la etapa anterioară.

Deşi se bazează pe descompunerea unei probleme în subprobleme la fel ca tehnica divizării, specificul programării dinamice este că subproblemele nu sunt independente (ci se suprapun). În schimb tehnica divizării conduce la algoritmi eficienţi doar dacă subproblemele sunt independente.

**Dezvoltarea ascendentă şi descendentă a unei relaţii de recurenţă.** Suprapunerea problemelor face ca obţinerea valorii prin dezvoltarea relaţiei de recurenţă să nu fie eficientă dacă este abordată în manieră descendentă ("top-down") prin implementarea recursivă a relaţiei. Aceasta deoarece o aceeaşi valoare poate fi utilizată de mai multe ori şi, de fiecare dată când este utilizată, este recalculată.

Să considerăm problema determinării celui de-al $m$-lea element din şirul lui Fibonacci ($f_1 = f_2 = 1$, $f_n = f_{n-1} + f_{n-2}$, $n \geq 3$). O abordare descendentă conduce la algoritmul recursiv:

```
fib_rec(m)
if (m = 1) or (m = 2) then
  return 1
else
  return fib_rec(m − 1)+fib_rec(m − 2)
end if
```

Numărul de adunări efectuate pentru a determina pe $f_m$, $T(m)$ verifică relaţiile: $T(1) = T(2) = 0$, $T(m) = T(m-1) + T(m-2) + 1$. Prin urmare $T(m) \geq f_{m-1}$, pentru $m \geq 5$. Cum $f_m \in \Theta(\phi^m)$ cu $\phi = (1 + \sqrt{5})/2$ rezultă că `fib_rec` are complexitate exponenţială.

Considerăm acum varianta generării lui $f_m$ in manieră ascendentă: se calculează si se *reţin* valorile lui $f_1, f_2, \ldots f_m$. În acest caz algoritmul poate fi descris prin:

```
fib_asc(m)
f[1] ← 1; f[2] ← 1
for i ← 3, m do
   f[i] = f[i − 1] + f[i − 2]
end for
return f[m]
```

Numărul de adunări efectuate este de această dată $T(m) = m - 2$ (complexitate liniară). Această variantă de implementare necesită însă utilizarea unui spaţiu auxiliar de memorie de dimensiune $m$. Algoritmul poate fi uşor transformat astfel încât să nu folosească decât trei variabile pentru reţinerea elementelor şirului dar să rămână cu complexitate liniară:

```
fib_asc2(m)
f1 ← 1; f2 ← 1
for i ← 3, m do
   f3 ← f1 + f2; f1 ← f2; f2 ← f3
end for
return f3
```

Numărul variabilelor de lucru poate fi redus chiar la două în modul următor:

```
fib_asc3(m)
f1 ← 1; f2 ← 1
for i ← 3, m do
   f2 ← f1 + f2; f1 ← f2 − f1;
end for
return f2
```

Un exemplu similar este cel al calculului combinărilor, $C_n^k$ $(n \geq k)$, pornind de la relaţia de recurenţă:

$$C_n^k = \begin{cases} 1 & \text{dacă } k = 0 \text{ sau } n = k \\ C_{n-1}^k + C_{n-1}^{k-1} & \text{altfel} \end{cases} \tag{8.1}$$

Algoritmul în varianta recursivă este descris în continuare.

```
comb_rec(n, k)
if (k = 0) or (n = k) then
   return 1
else
   return comb_rec(n − 1, k)+comb_rec(n − 1, k − 1)
end if
```

Algoritmul `comb_rec` are complexitate exponenţială întrucât utilizând arborele de apel se poate demonstra că $T(n,k) \geq 2^{\min\{n-k,k\}}$.

În abordarea ascendentă ideea este să se reţină toate valorile $C_i^j$ calculate pentru $0 \leq j \leq i \leq n$. Aceasta s-ar putea realiza prin completarea elementelor $a_{ij}$ din zona inferior triunghiulară a unei matrici $n \times n$. De fapt este suficient să se completeze până la coloana $k$. Dacă $n = k$ zona triunghiulară astfel

completată este cunoscută sub denumirea de *triunghiul lui Pascal*. Algoritmul este descris în 8.1.

**Algoritmul 8.1** Calculul combinărilor folosind triunghiul lui Pascal

```
comb_asc(n, k)
for i ← 0, n do
  for j ← 0, min{i, k} do
    if (i = j) or (j = 0) then
      a[i, j] ← 1
    else
      a[i, j] ← a[i − 1, j] + a[i − 1, j − 1]
    end if
  end for
end for
return a[n, k]
```

Numărul de adunări efectuate, $T(n,k)$ satisface

$$T(n,k) = \sum_{i=1}^{k}\sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^{n}\sum_{j=1}^{k} 1 = \sum_{i=1}^{k}(i-1) + \sum_{i=k+1}^{n} k = \frac{k(k-1)}{2} + k(n-k)$$

deci $T(n,k) \in \Theta(nk)$. Se observă că zona auxiliară utilizată poate fi redusă, întrucât pentru completarea liniei $i$ se folosesc doar valorile din linia $i-1$.

**Exemplul 8.1** (*Problema determinării celui mai lung subşir strict crescător*) Se consideră un şir de valori reale $a_1, a_2, \ldots, a_n$ şi se caută un subşir $a_{j_1}, a_{j_2}, \ldots, a_{j_k}$ cu $1 \le j_1 < j_2 < \ldots < j_k \le n$, $a_{j_1} < a_{j_2} < \ldots < a_{j_k}$ şi astfel încât $k$ să fie cât mai mare. Un astfel de subşir nu este neapărat unic. De exemplu pentru şirul $(2,5,1,3,6,8,2,10)$ există două subşiruri strict crescătoare de lungime 5: $(2,3,6,8,10)$ şi $(1,3,6,8,10)$. Pe de altă parte nu toate subşirurile de aceeaşi lungime se termină cu acelaşi element. De exemplu în şirul $(2,1,4,3)$ există patru subşiruri crescătoare de lungime maximă (2): $(2,4)$, $(2,3)$, $(1,4)$, $(1,3)$.

a) *Caracterizarea soluţiei optime.* Fie $a_{j_1} < a_{j_2} < \ldots < a_{j_{k-1}} < a_{j_k}$ un subşir de lungime maximă. Considerăm că $a_{j_k} = a_i$ iar $a_{j_{k-1}} = a_p$ (evident $p < i$). Prin reducere la absurd se poate arăta că $a_{j_1} < a_{j_2} < \ldots < a_{j_{k-1}}$ este cel mai lung dintre subşirurile crescătoare ale şirului parţial $(a_1, a_2, \ldots, a_p)$ care au proprietatea că ultimul element este chiar $a_p$. Prin urmare problema are proprietatea de substructură optimă şi lungimea unui subşir crescător care se termină în $a_i$ poate fi exprimată în funcţie de lungimea subşirurilor crescătoare care se termină cu elemente $a_p$ ce satisfac $a_p < a_i$.

b) *Stabilirea unei relaţii de recurenţă între lungimile subşirurilor.* Fie $(B_i)_{i=\overline{1,n}}$ un tablou ce conţine pe poziţia $i$ numărul de elemente al celui mai lung subşir strict crescător al şirului $(a_1, \ldots a_n)$ care are pe $a_i$ ca ultim element (spunem

că subşirul se termină în $a_i$). Ţinând cont de proprietăţile soluţiei optime se poate stabili o relaţie de recurenţă pentru $B_i$:

$$B_i = \begin{cases} 1 & i = 1 \\ 1 + \max\{B_j | 1 \leq j < i \text{ şi } a_j < a_i\} & i > 1 \end{cases}$$

Dacă mulţimea $M_i = \{B_j | 1 \leq j < i \text{ şi } a_j < a_i\}$ este vidă atunci $\max M_i = 0$. De exemplu pentru şirul iniţial $a = (2, 5, 1, 3, 6, 8, 4)$ tabloul lungimilor subşirurilor optime care se termină în fiecare element al lui $a$ este: $B = (1, 2, 1, 2, 3, 4, 3)$. Cea mai mare valoare din $B$ indică numărul de elemente din cel mai lung subşir crescător.

c) *Dezvoltarea relaţiei de recurenţă în manieră ascendentă.* Se completează un tablou cu elementele lui $(B_i)_{i=\overline{1,n}}$ aşa cum este descris în Algoritmul 8.2.

**Algoritmul 8.2** Dezvoltarea relaţiei de recurenţă pentru determinarea celui mai lung subşir crescător

```
completare(a[1..n])
B[1] ← 1
for i ← 2, n do
   max ← 0
   for j ← 1, i − 1 do
      if (a[j] < a[i]) and (max < B[j]) then
         max ← B[j]
      end if
   end for
   B[i] ← max + 1
end for
return  B[1..n]
```

.

Luând în considerare doar comparaţia dintre două elemente ale şirului ($a[j] < a[i]$) costul algoritmului este $T(n) = \sum_{i=2}^{n} \sum_{j=1}^{i-1} 1 = \sum_{i=2}^{n} (i-1) = n(n-1)/2$ deci completarea tabelului $B$ este de complexitate $\Theta(n^2)$.

d) *Construirea unei soluţii optime.* Se determină cea mai mare valoare din $B$. Aceasta indică numărul de elemente ale celui mai lung subşir crescător iar poziţia pe care se află indică elementul, $a[m]$, din şirul iniţial cu care se termină subşirul. Pornind de la acest element subşirul se construieşte în ordinea inversă a elementelor sale căutând la fiecare etapă un element din şir mai mic decât ultimul completat în subşir şi căruia îi corespunde în $B$ o valoare cu 1 mai mică decât cea curentă. Algoritmul este descris în 8.3.

Dacă după completarea elementului $a[m]$ în subşir există două subşiruri din $a[1..m-1]$ de lungime maximă egală cu $B[m] - 1$: unul care se termină în $a[p]$ şi unul care se termină în $a[q]$ (astfel încât $a[p] < a[m]$ şi $a[q] < a[m]$),

**Algoritmul 8.3** Construirea unui cel mai lung subşir crescător

```
construire(a[1..n], B[1..n])
// determinarea valorii şi poziţiei maximului în B
imax ← 1
for i ← 2, n do
  if B[imax] < B[i] then
    imax ← i
  end if
end for
m ← imax
k ← B[m]
x[k] ← a[m]
while B[m] > 1 do
  i ← m − 1
  while (a[i] ≥ a[m]) or (B[i] ≠ B[m] − 1) do
    i ← i − 1
  end while
  m ← i
  k ← k − 1 // se adaugă un nou element în subşir
  x[k] ← a[m]
end while
return x[1..k]
```

.

algoritmul va selecta ca element pentru subşir pe cel cu indicele mai apropiat de $m$, adică mai mare. Astfel dacă $p < q$ va fi selectat $q$. Aplicând algoritmul de construire şirului $(2, 5, 1, 3, 6, 8, 4)$ se va porni cu $m = 6$ ($B[m] = 4$) ultimul element din subşir fiind astfel 8. Cum $B[5] = 3$ şi $6 < 8$ penultimul element va fi 6. Continuând procesul se selectează în continuare elementele 3 şi 1 obţinându-se subşirul crescător $(1, 3, 6, 8)$ de lungime 4.

Întrucât în ciclul de construire căutarea continuă de la poziţia noului element selectat, chiar dacă sunt două cicluri **while** suprapuse ordinul de complexitate este $\mathcal{O}(n)$. Cum şi determinarea maximului lui $B$ are acelaşi ordin de complexitate rezultă că algoritmul de construire are complexitate liniară.

## 8.2 Aplicaţii

### 8.2.1 Înmulţirea optimală a unui şir de matrici

Fie $A_1$, $A_2$, ..., $A_n$ un şir de matrici având dimensiuni compatibile pentru a putea calcula produsul: $A_1 \cdot A_2 \cdots A_n$. Să presupunem că aceste dimensiuni sunt: $(p_0, p_1, \ldots, p_n)$, adică matricea $A_i$ are $p_{i-1}$ linii şi $p_i$ coloane. Pentru

a calcula produsul $A = A_1 \cdot A_2 \cdots A_n$ trebuie stabilit un mod de grupare a factorilor astfel încât să se pună în evidenţă ordinea în care vor fi efectuate înmulţirile, la fiecare etapă înmulţindu-se două matrici. Considerăm că produsul a două matrici se efectuează după metoda clasică astfel că la produsul $A_i \cdot A_{i+1}$ se efectuează $p_{i-1}p_ip_{i+1}$ înmulţiri scalare.

Modul de grupare a factorilor (plasarea parantezelor pentru a indica ordinea de efectuare a înmulţirilor) nu influenţează rezultatul final (înmulţirea este asociativă) însă poate influenţa numărul de operaţii efectuate.

Să considerăm cazul a trei matrici $A_1$, $A_2$ şi $A_3$ având dimensiunile $(2, 20)$, $(20, 5)$ şi $(5, 10)$. În acest caz există două modalităţi de plasare a parantezelor:

1. $A_1 \cdot A_2 \cdot A_3 = (A_1 \cdot A_2) \cdot A_3$. În acest caz se efectuează $2 \cdot 20 \cdot 5 + 2 \cdot 5 \cdot 10 = 300$ înmulţiri scalare.

2. $A_1 \cdot A_2 \cdot A_3 = A_1 \cdot (A_2 \cdot A_3)$. În acest caz se efectuează $20 \cdot 5 \cdot 10 + 2 \cdot 20 \cdot 10 = 1400$ înmulţiri scalare.

Pentru $n$ mare, numărul de *parantezări* (variante de plasare a parantezelor) poate fi foarte mare astfel că este exclusă generarea tuturor parantezărilor posibile şi alegerea celei mai bune. Pentru calculul produsului $A_1 \cdot A_2 \cdots A_n$ notând cu $K(n)$ numărul de parantezări posibile obţinem:

$$K(n) = \begin{cases} 1 & n = 1 \\ \sum_{i=1}^{n-1}(K(i)K(n-i)) & n > 1 \end{cases} \tag{8.2}$$

întrucât ultimul nivel de paranteze poate fi aplicat în oricare dintre poziţiile $i \in \{1, 2, \ldots, n-1\}$ şi fiecare dintre cele $K(i)$ parantezări ale produsului $A_1 \cdots A_i$ poate fi combinată cu fiecare dintre cele $K(n-i)$ parantezări ale produsului $A_{i+1} \cdots A_n$. Se poate demonstra prin inducţie matematică că $K(n) = \dfrac{C_{2(n-1)}^{n-1}}{n}$ şi $K(n) \in \Omega(4^{n-1}/(n-1)^{3/2})$.

Problema înmulţirii optimale a unui şir de matrici (un caz particular al problemelor de planificare optimală a prelucrărilor) constă în a determina parantezarea (ordinea de efectuare a înmulţirilor) care conduce la un număr minim de înmulţiri scalare.

Aplicăm pentru rezolvarea problemei etapele specifice programării dinamice.

a) *Caracterizarea soluţiei optime.* Pentru a specifica produsele parţiale introducem notaţia $A_{i..j} = A_i \cdot A_{i+1} \cdots A_j$. Fie o soluţie optimă caracterizată prin faptul că parantezele cele mai exterioare sunt plasate pe poziţiile 1, $k$ şi $n$, adică ultima înmulţire efectuată este: $A_{1..k} \cdot A_{k+1..n}$. Atunci parantezările asociate lui $A_{1..k}$ şi $A_{k+1..n}$ trebuie să fie şi ele optime (în caz contrar ar există o parantezare mai bună pentru $A_{1..n}$).

b) *Stabilirea unei relaţii de recurenţă pentru numărul de înmulţiri scalare.* Fie $c_{ij}$ numărul de înmulţiri scalare efectuate pentru calculul lui $A_{i..j}$. Valorile

$c_{ij}$ au sens doar pentru $i \leq j$ iar relaţia de recurenţă dedusă din proprietatea de substructură optimă este:

$$c_{ij} = \begin{cases} 0 & i = j \\ \min_{i \leq k < j}(c_{ik} + c_{k+1,j} + p_{i-1}p_k p_j) & i < j \end{cases}$$

Relaţia se bazează pe faptul că dintre toate parantezările posibile ale lui $A_{i..j}$ se alege cea mai bună (de cost minim).

c) *Dezvoltarea relaţiei de recurenţă.* Valorile $c_{i,j}$ pot fi reţinute în porţiunea superior triunghiulară a unei matrici. Construirea în manieră recursivă a matricii este ineficientă. Construirea în manieră ascendentă se bazează pe ideea completării matricii astfel încât în momentul calculului lui $c_{ij}$ toate elementele $c_{ik}$ şi $c_{k+1,j}$ pentru $k \in \{i, i+1, \ldots, j-1\}$ să fie deja completate. Din acest motiv elementele se completează în ordinea crescătoare a diferenţei $j-i$: prima dată se completează elementele diagonalei principale ($i = j$), apoi se completează elementele aflate imediat deasupra diagonalei principale ($j = i+1$) ş.a.m.d. (Algoritmul 8.4).

**Algoritmul 8.4** Dezvoltarea relaţiei de recurenţă în cazul înmulţirii optimale a unui şir de matrici

```
completare(p[0..n])
for i ← 1, n do
  c[i, i] = 0
end for
for l ← 1, n − 1 do
  for i ← 1, n − l do
    j ← i + l
    c[i, j] ← c[i, i] + c[i + 1, j] + p_{i−1} p_i p_j
    s[i, j] ← i
    for k ← i + 1, j − 1 do
      cost = c[i, k] + c[k + 1, j] + p_{i−1} p_k p_j
      if cost < c[i, j] then
        c[i, j] ← cost;
      end if
      s[i, j] ← k
    end for
  end for
end for
return c[1..n, 1..n], s[1..n, 1..n]
```

O dată cu completarea matricii de costuri se reţine şi poziţia în care se descompune un produs în doi factori: $s[i, j] = k$ indică faptul că produsul $A_{i..j}$ se descompune în $A_{i..k} \cdot A_{k+1..j}$. Aceste poziţii sunt utile la construirea soluţiei. Cum pentru determinarea valorii fiecăruia dintre cele $n(n-1)/2$ e-

lemente este necesară determinarea unui minim dintr-un tablou cu cel mult $n$ elemente rezultă că ordinul de complexitate al algoritmului de completare este $\mathcal{O}(n^3)$.

Pentru exemplul celor trei matrici cu dimensiunile $(2, 20)$, $(20, 5)$ şi $(5, 10)$ se obţin matricile:

$$c = \begin{bmatrix} 0 & 200 & 300 \\ - & 0 & 1000 \\ - & - & 0 \end{bmatrix} \qquad s = \begin{bmatrix} 0 & 1 & 2 \\ - & 0 & 2 \\ - & - & 0 \end{bmatrix}$$

d) În funcţie de cerinţele problemei se poate determina: (i) numărul minim de operaţii; (ii) rezultatul înmulţirii matricilor; (iii) modul de descompunere a produsului (parantezare).

(i) Numărul minim de înmulţiri este chiar $c_{1n}$.

(ii) Calculul produsului poate fi efectuat în manieră recursivă în Algoritmul 8.5. În acest algoritm se presupus că pot fi accesate matricile din şir (matricea $A_i$ este elementul $i$, $A[i]$, al tabloului tridimensional ce contine toate matricile) precum şi elementele matricii $s$. Algoritmul `produs` calculează produsul a două matrici.

**Algoritmul 8.5** Înmulţirea optimală a unui şir de matrici

```
produs_optimal(i, j)
if i = j then
  return A[i]
else
  X ← produs_optimal(i, s[i, j])      // calcul A_{i..s[i,j]}
  Y ← produs_optimal(s[i, j] + 1, j)  // calcul A_{(s[i,j]+1)..j}
  R ← produs(X, Y)      // produsul a două matrici
end if
return R
```

(iii) Ordinea în care se efectuează înmulţirile poate fi determinată prin Algoritmul 8.6.

**Algoritmul 8.6** Determinarea poziţiilor în care trebuie plasate parantezele în cazul înmulţirii optimale a unui şir de matrici

```
parantezare(i, j)
if i < j then
  parantezare(i, s[i, j])
  write s[i, j]
  parantezare(s[i, j] + 1, j)
end if
```

### 8.2.2 Varianta discretă a problemei rucsacului

Considerăm varianta discretă a problemei rucsacului. Presupunem că obiectele se caracterizează prin dimensiunile $(d_1, d_2, \ldots, d_n)$ şi profiturile (valorile): $(p_1, p_2, \ldots, p_n)$ iar capacitatea maximă a rucsacului este $C$. Se pune problema selectării unui subset de obiecte, $((p_{i_1}, d_{i_1}), \ldots, (p_{i_k}, d_{i_k}))$ care să nu depăşească capacitatea rucsacului ($\sum_{j=1}^{k} d_{i_j} \leq C$) şi să asigure un profit maxim ($\sum_{j=1}^{k} p_{i_j}$ este maximă). Se consideră că dimensiunea fiecărui obiect este mai mică decât cea maximă ($d_i \leq C$).

Pentru a uşura aplicarea tehnicii programării dinamice vom considera că $(d_i)_{i=\overline{1,n}}$ şi $C$ sunt valori naturale. În varianta 0-1 a problemei un obiect poate fi selectat doar în întregime. În acest caz tehnica greedy nu garantează obţinerea soluţiei optimale.

a) *Analiza structurii unei soluţii optime.* Problema selectării dintre cele $n$ obiecte poate fi redusă la problema rucsacului corespunzătoare primelor $n-1$ obiecte şi a capacităţii maxime $1 \leq C_{n-1} \leq C$. O soluţie optimă a problemei iniţiale va fi constituită dintr-o soluţie optimă a subproblemei corespunzătoare primelor $n-1$ obiecte la care se adaugă eventual al $n$-lea obiect (dacă $C_{n-1} + d_n \leq C$). Dacă soluţia subproblemei nu ar fi optimă atunci ar putea fi înlocuită cu una mai bună ceea ce ar conduce la o soluţie mai bună a problemei iniţiale.

b) *Stabilirea unei relaţii de recurenţă.* Fie $V(i, j)$ valoarea obiectelor selectate în soluţia optimă a problemei corespunzătoare primelor $i$ obiecte şi capacităţii $j$. Cum soluţia optimă a problemei corespunzătoare lui $i$ poate fi exprimată prin soluţia optimă a problemei corespunzătoare lui $i-1$ rezultă următoarea relaţie de recurenţă pentru $V(i, j)$:

$$V(i,j) = \begin{cases} V(i-1,j) & \text{dacă } j - d_i < 0 \\ \max\{p_i + V(i-1, j-d_i), V(i-1,j)\} & \text{dacă } j - d_i \geq 0 \end{cases}$$

pentru $i = \overline{1,n}$, $j = \overline{1,C}$ iar $V(0,j) = 0$ pentru $j = \overline{0,C}$ şi $V(i,0) = 0$ pentru $i = \overline{1,n}$.

Cele două cazuri din relaţia de recurenţă provin din faptul că o soluţie optimă a problemei asociate primelor $i$ obiecte şi capacităţii $j$ poate să nu conţină sau să conţină obiectul $i$. În primul caz dimensiunea obiectului $i$ este prea mare pentru ca acesta să poată fi selectat. Astfel valoarea este identică cu cea corespunzătoare subproblemei primelor $i-1$ obiecte, $V(i-1, j)$.

În al doilea caz obiectul $i$ ar putea fi selectat însă el va fi selectat doar dacă conduce la o valoare totală mai mare decât dacă nu ar fi fost selectat. Din acest motiv se analizează ambele valori şi se alege cea maximă. Valoarea corespunzătoare cazului în care obiectul este selectat este obţinută adăugând valoarea celui de-al $i$-lea obiect ($p_i$) la valoarea soluţiei problemei corespunzătoare celor $i-1$ obiecte şi capacităţii $j-d_i$ (capacitatea rămasă disponibilă după selectarea obiectului $i$). $V(n, C)$ reprezintă valoarea maximă a unui set de obiecte a căror

dimensiuni însumate nu depăşeşte pe $C$ adică valoarea asociată soluţiei optime a problemei.

c) *Dezvoltarea relaţiei de recurenţă.* Dacă $C$ şi $(d_i)_{i=\overline{1,n}}$ sunt valori întregi atunci valorile lui $V(i,j)$ pot fi reţinute într-o matrice cu $n+1$ linii şi $C+1$ coloane. Algoritmul de construire a tabelului de valori este descris în 8.7.

**Algoritmul 8.7** Dezvoltarea relaţiei de recurenţă în cazul problemei rucsacului

```
tabel_valori(p[1..n],d[1..n],C)
for i ← 0, n do
  V[i, 0] ← 0
end for
for j ← 1, C do
  V[0, j] ← 0
end for
for i ← 1, n do
  for j ← 1, C do
    if j < d[i] then
      V[i, j] = V[i − 1, j]
    else
      V[i, j] = max(V[i − 1, j], p[i] + V[i − 1, j − d[i]])
    end if
  end for
end for
return V[0..n, 0..C]
```

Să considerăm cazul în care $n = 5$, $C = 5$, dimensiunile obiectelor sunt $(2, 4, 2, 1, 3)$ iar valorile asociate $(10, 20, 13, 15, 18)$. În acest caz matricea de valori este cu 6 linii şi 6 coloane (indiciate de la 0 la 5) şi conţine elementele:

$$V = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 10 & 10 & 10 & 10 \\ 0 & 0 & 10 & 10 & 20 & 20 \\ 0 & 0 & 13 & 13 & 23 & 23 \\ 0 & 15 & 15 & 28 & 28 & 38 \\ 0 & 15 & 15 & 28 & 33 & 38 \end{bmatrix}$$

Pornind de la elementul $V[5,5]$ se poate construi soluţia astfel. Se observă că $V[5,5] = V[4,5]$. Aceasta înseamnă că obiectul $o_5$ nu este selectat. În schimb $V[4,5] \neq V[3,5]$ adică obiectul $o_4$ este selectat. Aceasta reduce problema la selecţia dintre obiectele $\{o_1, o_2, o_3\}$ pentru un rucsac de capacitate $5 - d_4 = 4$. Valoarea unei soluţii optime a acestei subprobleme este $V[3,4] = 23$ (se observă că diferenţa $V[5,5] - V[3,4] = 38 - 23 = 15$ reprezintă chiar valoarea obiectului $o_4$). În continuare, se observă că $V[3,4] \neq V[2,4]$ adică obiectul $o_3$ este selectat.

Astfel problema se reduce la subproblema corespunzătoare setului $\{o_1, o_2\}$ şi capacităţii $4 - d_3 = 2$. Valoarea unei soluţii optime a acestei subprobleme este $V[2,2] = 10$. Cum $V[2,2] = V[1,2]$ şi $V[1,2] \neq 0$ rezultă că $o_2$ nu este selectat în schimb este selectat $o_1$. Astfel soluţia problemei iniţiale este $\{o_1, o_3, o_4\}$ cu dimensiunea totală 5 şi valoarea totală 38.

d) *Construirea soluţiei.* Pentru construirea soluţiei se analizează tabelul de valori construit la etapa anterioară iniţiind parcurgerea din $V[n, C]$ după strategia descrisă în exemplul de mai sus. Algoritmul este descris în 8.8.

**Algoritmul 8.8** Rezolvarea variantei discrete a problemei rucsacului

```
construire_soluţie(V[0..n, 0..C],d[1..n],C)
i ← n; j ← C
k ← 0
while j > 0 do
  while V[i, j] = V[i − 1, j] and i ≥ 1 do
    i ← i − 1
  end while
  k ← k + 1; s[k] = i
  j = j − d[i]
  i ← i − 1
end while
return s[1..k]
```

### 8.2.3 Tehnica funcţiilor de memorie

Principalul dezavantaj al abordării ascendente este faptul că se bazează pe completarea unui întreg tabel de valori. În aplicaţii e posibil ca anumite valori din tabel să nu fie necesare nici în calculul celorlalte valori şi nici în construcţia soluţiei (de exemplu elementele $V[5,1]$, $V[5,2]$, $V[5,3]$ şi $V[5,4]$ din tabelul construit pentru problema rucsacului).

Dacă se foloseşte abordarea top-down atunci se vor calcula doar valorile asociate subproblemelor care sunt necesare pentru a obţine valoarea corespunzătoare problemei iniţiale. Dar dacă o subproblemă este întâlnită de mai multe ori atunci ea este rezolvată de fiecare dată.

O soluţie de compromis care îmbină avantajele abordării ascendente cu a celei descendente este de a dezvolta relaţia de recurenţă într-o manieră recursivă reţinând însă fiecare dintre valorile calculate. În felul acesta dacă o valoare deja calculată este necesară din nou ea nu va fi recalculată ci se va folosi valoarea reţinută anterior. Această variantă hibridă este cunoscută ca *tehnica funcţiilor de memorie* sau *tehnica memoizării* (termenul în engleză este "memoization"). Aplicarea ei în practică constă în:

- iniţializarea tabelului cu o valoare *virtuală* ce va fi diferită de valorile ce

se vor obţine din calcule; în felul acesta se va putea verifica dacă o poziţie din tabel a fost calculată deja sau nu;

- calculul valorii corespunzătoare soluţiei în manieră recursivă (cu reţinerea valorilor calculate); aceasta presupune că tabelul de valori este o structură globală ce poate fi accesată la fiecare apel recursiv.

Aplicată la construirea tabelului de valori pentru problema rucsacului această tehnică constă în utilizarea algoritmilor `initializare` şi `calcul_valori` descrişi în continuare.

```
iniţializare(n,C)
for i ← 1, n do
   for j ← 1, C do
      V[i, j] ← −1 // iniţializare cu o valoare virtuală
   end for
end for
for j ← 0, C do
   V[0, j] ← 0
end for
for i ← 0, n do
   V[i, 0] ← 0
end for
return V[0..n, 0..C]
```

şi

```
calcul_valori(i,j)
if i = 0 or j = 0 then
   return 0
end if
if V[i, j] < 0 then
   if j < d[i] then
      val ← calcul_valori(i − 1, j)
   else
      val ← max(calcul_valori(i − 1, j), p[i]+calcul_valori(i − 1, j − d[i]))
   end if
   V[i, j] ← val // reţinerea valorii calculate
end if
return V[i, j]// returnarea valorii preluate din tabel sau calculate
```

Algoritmul `calcul_valori` are acces la talourile $d[1..n]$ şi $V[0..n, 0..C]$. Apelul `calcul_valori`$(n, C)$ se plasează după iniţializarea tabloului.

### 8.2.4 Problema închiderii tranzitive

Nu doar problemele de optimizare pot fi rezolvate cu tehnica programării dinamice ci şi alte probleme a căror soluţie poate fi exprimată în funcţie de

soluţiile unor subprobleme.

Considerăm o relaţie binară $R \subset \{1,\ldots,n\} \times \{1,\ldots,n\}$. Închiderea sa tranzitivă este o relaţie binară $R^* \subset \{1,\ldots,n\} \times \{1,\ldots,n\}$ având proprietatea: dacă pentru $i,j \in \{1,\ldots,n\}$ există $i_1,\ldots,i_m \in \{1,\ldots,n\}$ cu proprietăţile: $(i_1,i_2) \in R$, $(i_2,i_3) \in R$, ... ,$(i_{m-1},i_m) \in R$ iar $i = i_1$ şi $j = i_m$ atunci $(i,j) \in R^*$.

Pentru a construi $R^*$ se consideră următorul set de relaţii binare $R^0 = R$, $R^1$, ..., $R^n = R^*$, definite astfel: dacă pentru $i,j \in \{1,\ldots,n\}$ există $i_1,\ldots,i_m \in \{1,\ldots,k\}$ cu proprietăţile: $(i_1,i_2) \in R$, $(i_2,i_3) \in R$, ..., $(i_{m-1},i_m) \in R$ iar $i = i_1$ şi $j = i_k$ atunci $(i,j) \in R^k$. Relaţiile pot fi descrise prin recurenţe astfel: $(i,j) \in R^k$ dacă $(i,j) \in R^{k-1}$ sau $(i,k) \in R^{k-1}$ şi $(k,j) \in R^{k-1}$.

Pentru a elabora algoritmul de construire a lui $R^*$ considerăm relaţiile binare pe $\{1,2,\ldots,n\}$ reprezentate prin matrici ale căror elemente sunt definite astfel:

$$r_{ij} = \begin{cases} 1 & \text{dacă } (i,j) \in R \\ 0 & \text{dacă } (i,j) \notin R \end{cases}$$

Relaţiile de recurenţă pentru construirea matricilor asociate relaţiilor binare $R^0, R^1, \ldots, R^n$ sunt:

$$r_{ij}^k = \begin{cases} 1 & \text{dacă } r_{ij}^{k-1} = 1 \vee (r_{ik}^{k-1} = 1 \wedge r_{kj}^{k-1} = 1) \\ 0 & \text{altfel} \end{cases} \qquad k = \overline{1,n}$$

cu $r_{ij}^0 = r_{ij}$. Folosind aceste relaţii de recurenţă se poate construi matricea asociată relaţiei binare $R^n$, utilizând doar două matrici auxiliare în modul descris în Algoritmul 8.9.

Algoritmul pentru determinarea închiderii tranzitive este cunoscut şi sub numele de algoritmul lui Warshall.

**Algoritmul 8.9** Determinarea închiderii tranzitive a unei relaţii binare

```
închidere_tranzitivă(R[1..n, 1..n])
R2[1..n, 1..n] ← R[1..n, 1..n]
for k ← 1, n do
   R1[1..n, 1..n] ← R2[1..n, 1..n]
   for i ← 1, n do
      for j ← 1, n do
         if (R1[i, j] = 1) or (R1[i, k] = 1 and R1[k, j] = 1) then
            R2[i, j] ← 1
         else
            R2[i, j] ← 0
         end if
      end for
   end for
end for
return  R2[1..n, 1..n]
```