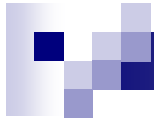


# Dinamikus programozás párhuzamosítási lehetőségekkel

A. Grama, A. Gupta, G. Karypis és  
V. Kumar: „Introduction to Parallel Computing”,  
Addison Wesley, 2003

könyv, valamint

Michael Goodrich (Univ. California) anyaga alapján



# Vázlat

- A dinamikus programozás (DP) áttekintése
- Soros, egyszerű-argumentumú (Monadic) DP
- Nem soros, egyszerű-argumentumú DP
- Soros, összetett-argumentumú (Polyadic) DP
- Nem soros, összetett-argumentumú DP



# A dinamikus programozás áttekintése

- A *dinamikus programozás* (DP) széles körben használt optimalizációs problémák megoldására: ütemezés, string-szerkesztés, csomagolási probléma, stb.
- A problémákat részproblémákra bontjuk és ezek megoldásait kombináljuk, hogy a nagyobb probléma megoldását megtaláljuk.
- Az „oszd meg és uralkodj” típusú megoldási módszerrel szemben, itt a részproblémák között kapcsolatok, átfedések állhatnak fent (*overlapping subproblems*).
- Az elnevezés matematikai optimalizálásra utal.



# A dinamikus programozás áttekintése

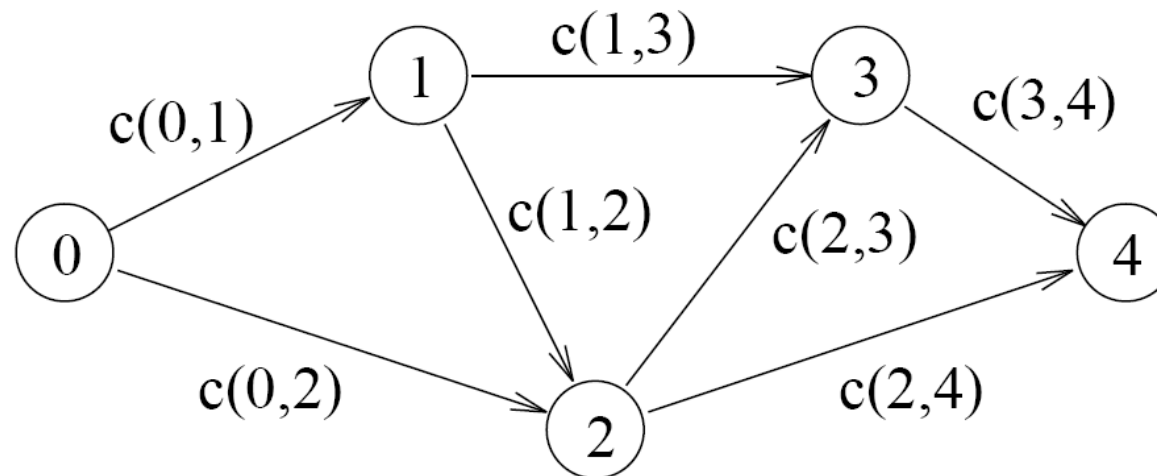
- A problémamegoldás menete:
  1. Részproblémákra osztás.
  2. A részproblémák optimális megoldása rekurzívan.
  3. Az optimális megoldások felhasználása az eredeti feladat optimális megoldásának megtalálásához.
- A dinamikus programozás (DP) során a részmegoldások eredményeit gyakran táblázatokban tároljuk, hogy azokat – szemben a rekurzív megközelítésekkel – ne kelljen újra és újra kiszámolni (*memoization*).

# Dinamikus programozás: példa

- Tekintsük a hurokmentes irányított gráfokban két csúcs (csomópont) között a legrövidebb út problémáját.
- Az  $i$  és  $j$  csomópontokat összekötő él költsége  $c(i, j)$ .
- A gráf  $n$  csomópontot tartalmaz:  $0, 1, \dots, n-1$ , és az  $i$  csomópontból a  $j$  csomópontba akkor vezethet él, ha  $i < j$ . A 0 csomópont a forrás és az  $n-1$  csomópont a cél.
- Jelölje  $f(x)$  a legrövidebb utat a 0 csomóponttól  $x$ -ig.

$$f(x) = \begin{cases} 0 & x = 0 \\ \min_{0 \leq j < x} \{f(j) + c(j, x)\} & 1 \leq x \leq n - 1 \end{cases}$$

# Dinamikus programozás: példa



- Irányított gráf, melyben a legrövidebb utat számolhatjuk a 0 és 4 csomópontok között.

$$f(4) = \min\{f(3) + c(3, 4), f(2) + c(2, 4)\}.$$



# Dinamikus programozás

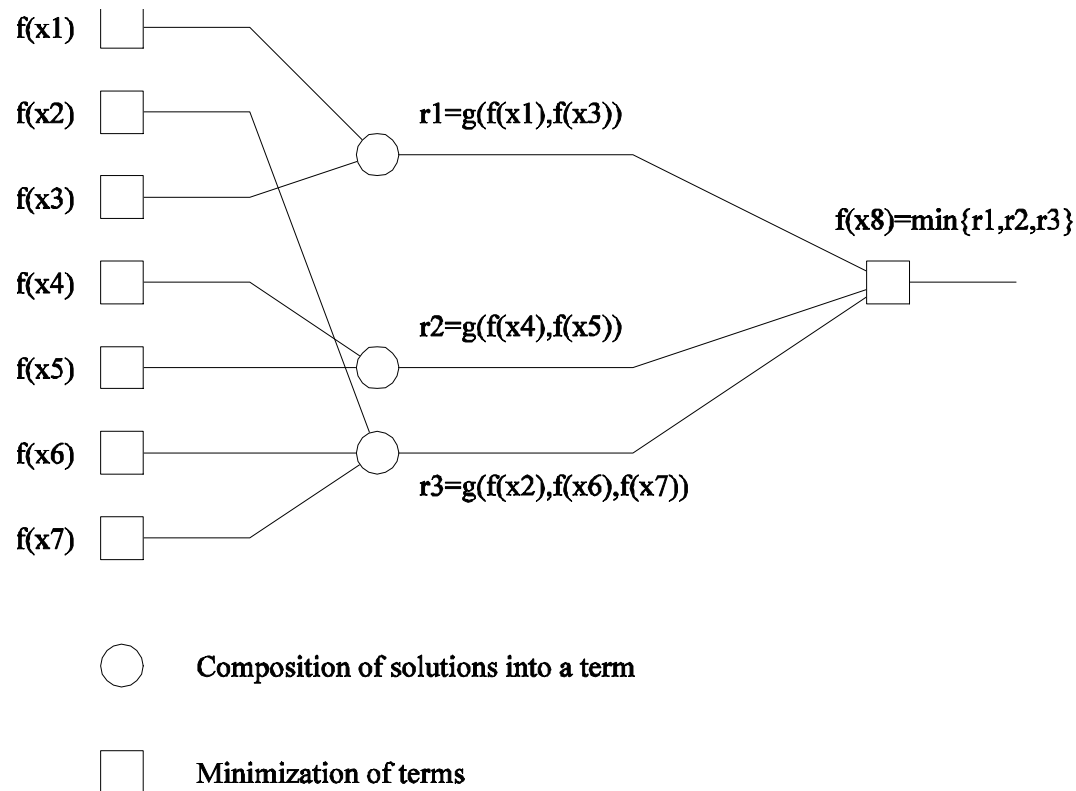
- A DP formátumú probléma megoldását tipikusan a lehetséges megoldások minimumaként vagy maximumaként fejezzük ki.
- Ha  $r$  az  $x_1, x_2, \dots, x_l$  részproblémák kompozíciójából meghatározott megoldás költségét jelenti, akkor  $r$  a következő alakban írható

$$r = g(f(x_1), f(x_2), \dots, f(x_l)).$$

Itt  $g$  az un. *kompozíciós függvény*.

- Ha minden probléma optimális megoldása a részfeladatok optimális megoldásának optimális kompozíciójaként kerül meghatározásra és a minimum (vagy maximum) érték kiválasztásra kerül, akkor DP formátumú megoldásról beszélünk. (Metamódszer, nem konkrét algoritmus.)

# Dinamikus programozás: példa



Az  $f(x_8)$  megoldás meghatározásának kompozíciója és számítása részfeladatok megoldásából.





# Dinamikus programozás

- A rekurzív DP egyenletet *funkcionális egyenletnek*, vagy *optimalizációs egyenletnek* is nevezzük.
- A legrövidebb út problémánál a kompozíciós függvény  $f(j) + c(j, x)$ . Ez egyszerű rekurzív tagként tartalmazza  $(f(j))$ . Az ilyen DP megfogalmazású feladatot monadic típusúnak (egyszerű-argumentumúnak) nevezzük.
- Ha a rekurzív kompozíciós függvény több tagú, akkor a DP feladat megszövegezése polyadic (összetett-argumentumú) típusú.



# Dinamikus programozás

- A részproblémák közötti kapcsolatokat gráf segítségével fejezhetjük ki.
- Ha a gráf szintekre bontható és a probléma megoldása egy bizonyos szinten csak az előző szinten megjelenő problémamegoldástól függ, ekkor a megfogalmazást *sorosnak* nevezzük, egyébként pedig *nem sorosnak*.
- A két kritérium alapján a DP feladat megfogalmazásokat osztályozhatjuk: serial-monadic, serial-polyadic, non-serial-monadic, non-serial-polyadic, vagy egyéb.
- Az osztályozás azért fontos, mert meghatározza konkurenciát és a függőségeket, amely a párhuzamos megoldásokhoz vezethet.



# Soros, egyszerű-argumentumú DP

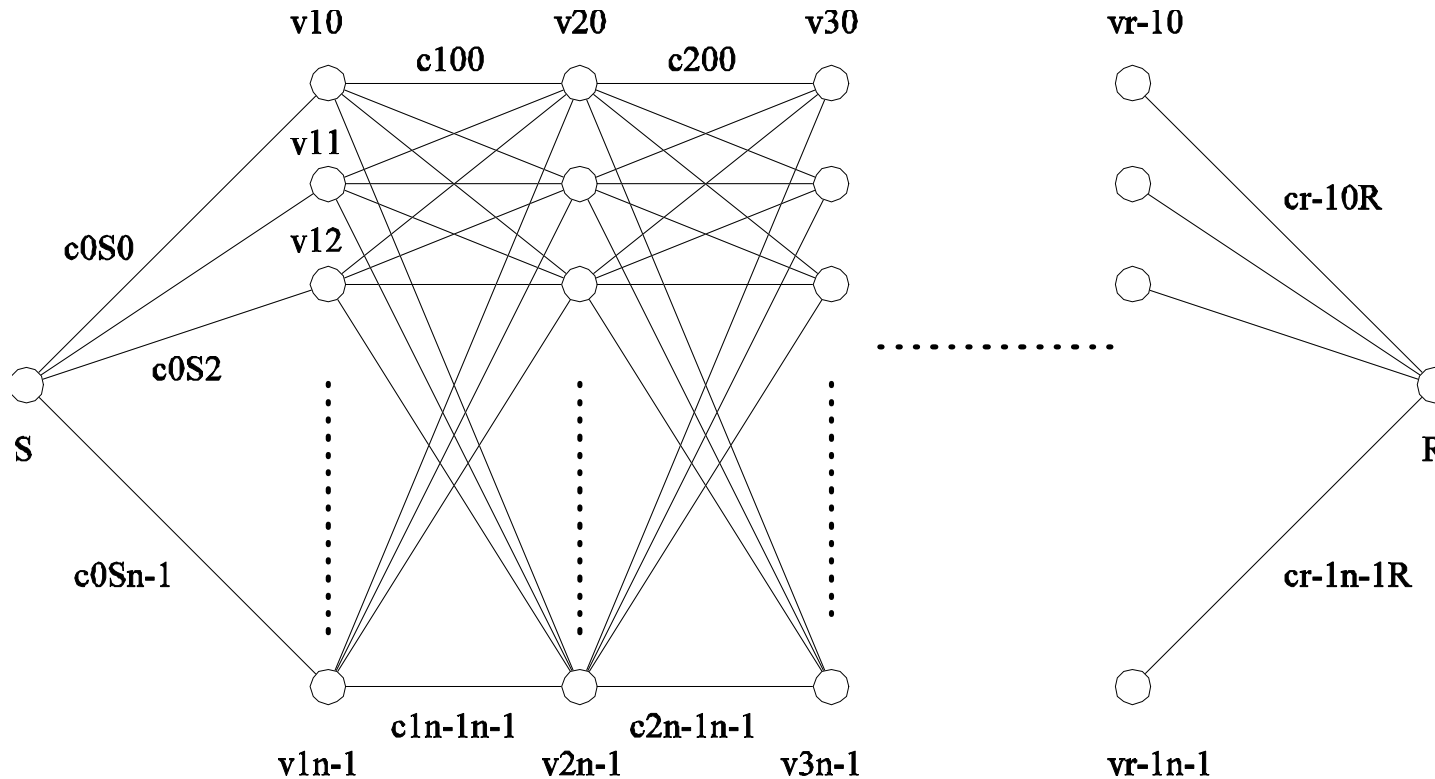
- Az egész osztályra vonatkozó általános megfogalmazást, mintát nehéz megadni.
- Két reprezentatív feladat kerül bemutatásra:
  - A legrövidebb út probléma többszintű gráfban (*shortest-path problem for a multistage graph*),
  - 0/1 típusú hátizsák probléma (*0/1 knapsack problem*).
- Célunk a feladatok párhuzamos megfogalmazása és ezek alapján az osztályon belüli közös elvek lefektetése.



# Legrövidebb út probléma

- A legrövidebb út problémának speciális osztálya, amikor a gráf súlyozott és többszintű. Jelen esetben a szintek száma  $r + 1$ .
- Minden szinten  $n$  csomópont van és minden csomópont az  $i$ . szintről össze van kötve az  $i + 1$  szint minden egyes csomópontjával.
- A 0. szint és az  $r$ . csak egy csomópontot tartalmaz, ezeket forrás ( $S$ ) és cél ( $R$ ) csomópontoknak nevezzük.
- Célunk legrövidebb utat találni  $S$ -től  $R$ -ig.

# Legrövidebb út probléma



Soros, egyszerű-argumentumú (monadic) DP megfogalmazása a legrövidebb út problémának olyan gráfban, ahol a csomópontokat szintekbe szervezhetjük.

# Legrövidebb út probléma

- A gráf  $l$  szintjének  $i$ . csomópontja:  $v_i^l$  és a  $v_i^l$  csomópontot  $v_j^{l+1}$  csomóponttal összekötött él súlya  $c_{i,j}^l$ .
- Bármely  $v_i^l$  csomópont és az cél csomópont  $R$  költsége:  $C_i^l$ .
- Ha  $n$  csomópont van egy szinten ( $l$ ), akkor a költségek (legrövidebb utak  $R$ -ig) egy vektorban foglalhatók össze  $[C_0^l, C_1^l, \dots, C_{n-1}^l]^T$ . Ez a vektor  $C^l$ . Megjegyzés:  $C^0 = [C_0^0]$ .
- Az  $l$  szint  $i$  eleme a céltól  $C_i^l = \min \{(c_{i,j}^l + C_j^{l+1}) \mid j \text{ az } l+1 \text{ szint egy csomópontjának indexe}\}$

# Legrövidebb út probléma

- Mivel a  $v_j^{r-1}$  csomópontokat csak egy-egy él köti össze a cél,  $R$  csomóponttal az  $r$  szinten, a  $C_j^{r-1}$  költsége:  $c_{j,R}^{r-1}$ .
- Ezért:

$$C^{r-1} = [c_{0,R}^{r-1}, c_{1,R}^{r-1}, \dots, c_{n-1,R}^{r-1}].$$

A feladat soros és egyszerű-argumentumú (monadic) típusú.

# Legrövidebb út probléma

- Az  $l$  szint valamely csomópontjától a célcsomópont,  $R$  elérésének költsége, ahol  $(0 < l < r - 1)$ :

$$C_0^l = \min\{(c_{0,0}^l + C_0^{l+1}), (c_{0,1}^l + C_1^{l+1}), \dots, (c_{0,n-1}^l + C_{n-1}^{l+1})\},$$

$$C_1^l = \min\{(c_{1,0}^l + C_0^{l+1}), (c_{1,1}^l + C_1^{l+1}), \dots, (c_{1,n-1}^l + C_{n-1}^{l+1})\},$$

⋮

$$C_{n-1}^l = \min\{(c_{n-1,0}^l + C_0^{l+1}), (c_{n-1,1}^l + C_1^{l+1}), \dots, (c_{n-1,n-1}^l + C_{n-1}^{l+1})\}.$$





# Legrövidebb út probléma

- A probléma megoldását egy módosított mátrix-vektor szorzat formájában fejezhetjük ki
- Helyettesítsük az összeadás operációt minimalizálással és a szorzás műveletet összeadással, az előző egyenletek a következő alakúak lesznek:

$$C^l = M_{l,l+1} \times C^{l+1},$$

ahol  $C^l$  és  $C^{l+1}$   $n \times 1$  méretű vektorok a cél csomópont elérésének költségeit reprezentálják az  $l$  és  $l + 1$  szint minden egyes csomópontjától mérve.

# Legrövidebb út probléma

- Az  $M_{l,l+1}$  mátrix  $n \times n$  méretű, és minden  $(i, j)$  eleme az  $l$  szint  $i$  csomópontjának az  $l + 1$  szint  $j$  csomópontjával történő összekötés költségét tartalmazza:

$$M_{l,l+1} = \begin{bmatrix} c_{0,0}^l & c_{0,1}^l & \cdots & c_{0,n-1}^l \\ c_{1,0}^l & c_{1,1}^l & \cdots & c_{1,n-1}^l \\ \vdots & \vdots & & \vdots \\ c_{n-1,0}^l & c_{n-1,1}^l & \cdots & c_{n-1,n-1}^l \end{bmatrix}.$$

- A legrövidebb út problémát  $r$  mátrix-vektor szorzatok sorozataként definiáltuk:  $C^{r-1}$ ,  $C^{r-k-1}$  ( $k = 1, 2, \dots, r-2$ ),  $C^0$ .



# Párhuzamos legrövidebb út probléma

- Az algoritmus úgy párhuzamosítható, mint bármelyik mátrix-vektor szorzást megvalósító algoritmus.
- $n$  processzor elem számolhat minden  $C'$  vektort.
- Sok gyakorlati esetben az  $M$  mátrix elemei ritkák lehetnek. Ilyenkor ritkamátrixokhoz kapcsolódó módszert alkalmazzunk.

# 0/1 hátizsák probléma

0/1 knapsack problem

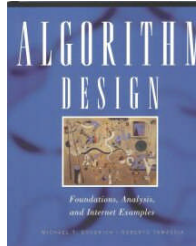
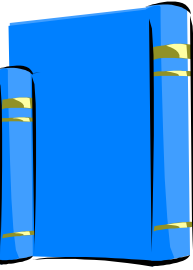
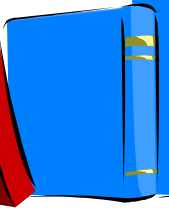
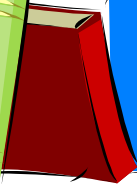
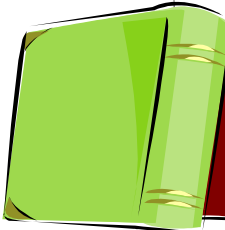


- „A thief is robbing the King's treasury, but he can only carry a load weighing at most  $W$ ...”
- Adott:  $S$  halmaz  $n$  elemmel és egy súlyhatár  $c$ , minden  $i$  elemnek
  - $p_i$  – pozitív egész értéke van (*profit*)
  - $w_i$  – és pozitív egész súly
- Cél: úgy válasszuk ki az elemeket, hogy az összérték maximális legyen, de az összsúly kisebb legyen, mint  $c$ .
  - $T$  jelölje azokat az elemeket, amit kiválasztunk,  $T \subseteq S$
  - **Cél:** összérték maximalizálása  $\sum_{i \in T} p_i$
  - **Feltétel:** a súlyhatáron belül kell maradni  $\sum_{i \in T} w_i \leq C$

# 0/1 hátizsák probléma: példa

- Adott:  $n$  elemű  $S$  halmaz, minden elem
  - $p_i$  – pozitív értékű és  $w_i$  – pozitív súlyú
- Cél: válasszunk ki elemeket, hogy az összérték maximális legyen, de az összsúly ne haladja meg  $c$ -t.
- Összesen  $2^n$  eset lehet!

Elemek:



1

2

3

4

5

Súly:

4 N

2 N

2 N

6 N

2 N

Érték:

\$20

\$3

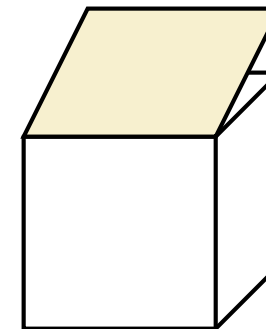
\$6

\$25

\$80

Dinamikus programozás

“knapsack”



9 N

**Megoldás:**

- 5 (2 N)
- 3 (2 N)
- 1 (4 N)

**Összérték:**

$$\begin{aligned} &\bullet 80+6+20 \\ &= \$106_{21} \end{aligned}$$

# 0/1 hátizsák probléma, első próbálkozás

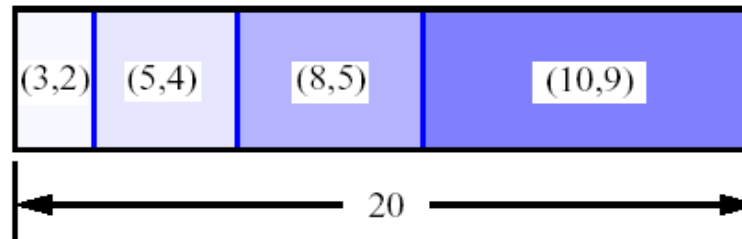


- $S_i$ : A halmaz elemeinek azonosítója 1-től  $i$ -ig ( $i < n$ ).
- Definiálja  $F[i]$  = a legjobb kiválasztást  $S_i$ -ből.
- Legyen pl.  $S = \{(3,2), (5,4), (8,5), (4,3), (10,9)\}$  érték-súly párok,  $c = 20$

A legjobb  $S_4$ ,  
ha csak négyet veszünk ki:  
Összsúly: 14, érték: 20



A legjobb  $S_5$ :  
az 5. benne van, de a 4. nem



- **Rossz hír:**  $S_4$  nem része az  $S_5$  optimális megoldásnak

# 0/1 hátizsák probléma, más megközelítés



- $S_i$ : Az elemek halmaza 1-től  $i$ -ig,  $i < n$ .
- Definiálja  $F[i, x]$  = az  $S_i$  halmazból a legjobb kiválasztást, ahol a súly legföljebb  $x$  – további paraméter
- Ez optimális részprobléma megoldáshoz vezet.
- Az  $S_i$  legjobb kiválasztás legföljebb  $x$  súllyal, két esetet jelenthet:
  - Ha  $w_i > x$ , akkor az  $i$ . elemet nem lehet hozzávenni, mert súlya nagyobb, mint az aktuális határ
  - Egyébként: ha a legjobb  $S_{i-1}$  részhalmaz súlya  $x - w_i$  és ehhez vagy jön  $i$ , vagy nem eredményez nagyobb értéket

$$F[i, x] = \begin{cases} F[i-1, x] & \text{if } w_i > x \\ \max\{F[i-1, x], F[i-1, x - w_i] + p_i\} & \text{else} \end{cases}$$

# 0/1 hátizsák algoritmus



- $F[i, x]$  rekurzív formula:

$$F[i, x] = \begin{cases} F[i-1, x] & \text{if } w_i > x \\ \max\{F[i-1, x], F[i-1, x - w_i] + p_i\} & \text{else} \end{cases}$$

- $F[i, x]$  = az 1-től  $i$ -ig tartó elemekből a legjobb kiválasztás, ahol az összsúly legfeljebb  $x$
- Alapeset:  $i = 0$ , nem került elem kiválasztásra. Összérték 0.
- A feladat megoldása: a legnagyobb érték az  $n$ . sorban, utolsó ( $c$ ) oszlopban
- Futási idő:  $O(nc)$ .  
Nem  $2^n$  ideig tart.



# 0/1 hátizsák algoritmus



**Algorithm 0-1Knapsack( $S, c$ ):**

**Input:**  $S$  halmaz elemei  $p_i$  értékkel,  $w_i$  súllyal; valamint a max. súly  $c$

**Output:** a legjobb részhalmaz értéke ( $F[n, c]$ ), hogy teljesül: összsúly  $\leq c$

**for**  $x \leftarrow 0$  **to**  $c$  **do**

$F[0, x] \leftarrow 0$

**for**  $i \leftarrow 0$  **to**  $n$  **do**

$F[i, 0] \leftarrow 0$

**for**  $i \leftarrow 0$  **to**  $n$  **do**

**for**  $x \leftarrow 0$  **to**  $c$  **do**

**if**  $w_i \leq x$

$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$

**else**

$F[i, x] \leftarrow F[i - 1, x]$

- Mivel  $F[i, x]$  csak  $F[i - 1, *]$  értékeitől függ, elég lehet vektorokat használni



## 0/1 hátizsák példa

- $n = 4$  az adatok száma
  - $c = 5$  kapacitás (maximális összsúly)
  - Elemek:
- | $i$ | súly ( $w_i$ ), | profit ( $p_i$ ) |
|-----|-----------------|------------------|
| 1   | 2               | 3                |
| 2   | 3               | 4                |
| 3   | 4               | 5                |
| 4   | 5               | 6                |



# Példa

$i \backslash x$	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

for  $x \leftarrow 0$  to  $c$  do  
     $F[0, x] \leftarrow 0$



# Példa

$i \backslash x$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

for  $i \leftarrow 0$  to  $n$  do

$F[i, 0] \leftarrow 0$



# Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

$i \setminus x$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

if  $w_i \leq x$

$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$

else

$F[i, x] \leftarrow F[i - 1, x]$

$i = 1$
$p_i = 3$
$w_i = 2$
$x = 1$
$x - w_i = -1$



# Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

$i \setminus x$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0					
3	0					
4	0					

if  $w_i \leq x$

$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$

else

$F[i, x] \leftarrow F[i - 1, x]$

$i = 1$   
 $p_i = 3$   
 $w_i = 2$   
 $x = 2$   
 $x - w_i = 0$



# Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

$i \setminus x$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3		
2	0					
3	0					
4	0					

if  $w_i \leq x$

$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$

else

$F[i, x] \leftarrow F[i - 1, x]$

$i = 1$   
 $p_i = 3$   
 $w_i = 2$   
 $x = 3$   
 $x - w_i = 1$



# Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

$i \setminus x$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	
2	0					
3	0					
4	0					

if  $w_i \leq x$

$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$

else

$F[i, x] \leftarrow F[i - 1, x]$

$i = 1$   
 $p_i = 3$   
 $w_i = 2$   
 $x = 4$   
 $x - w_i = 2$



# Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

$i \setminus x$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					

if  $w_i \leq x$

$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$

else

$F[i, x] \leftarrow F[i - 1, x]$

$i = 1$   
 $p_i = 3$   
 $w_i = 2$   
 $x = 5$   
 $x - w_i = 3$



# Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

$i \setminus x$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0				
3	0					
4	0					

if  $w_i \leq x$

$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$

else

$F[i, x] \leftarrow F[i - 1, x]$

$i = 2$
$p_i = 4$
$w_i = 3$
$x = 1$
$x - w_i = -2$



# Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

$i \setminus x$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3			
3	0					
4	0					

if  $w_i \leq x$

$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$

else

$F[i, x] \leftarrow F[i - 1, x]$

$i = 2$
$p_i = 4$
$w_i = 3$
$x = 2$
$x - w_i = -1$



# Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

$i \setminus x$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4		
3	0					
4	0					

if  $w_i \leq x$

$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$

else

$F[i, x] \leftarrow F[i - 1, x]$

$i = 2$   
 $p_i = 4$   
 $w_i = 3$   
 $x = 3$   
 $x - w_i = 0$



# Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

$i \setminus x$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	
3	0					
4	0					

if  $w_i \leq x$

$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$

else

$F[i, x] \leftarrow F[i - 1, x]$

$i = 2$
$p_i = 4$
$w_i = 3$
$x = 4$
$x - w_i = 1$



# Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

$i \setminus x$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

if  $w_i \leq x$

$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$

else

$F[i, x] \leftarrow F[i - 1, x]$

$i = 2$   
 $p_i = 4$   
 $w_i = 3$   
 $x = 5$   
 $x - w_i = 2$

# Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

$i \setminus x$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4		
4	0					

if  $w_i \leq x$

$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$

else

$F[i, x] \leftarrow F[i - 1, x]$

$i = 3$   
 $p_i = 5$   
 $w_i = 4$   
 $x = 1..3$   
 $x - w_i = -3..-1$

# Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

$i \setminus x$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	
4	0					

if  $w_i \leq x$

$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$

else

$F[i, x] \leftarrow F[i - 1, x]$

$i = 3$   
 $p_i = 5$   
 $w_i = 4$   
 $x = 4$   
 $x - w_i = 0$



# Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

$i \setminus x$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0					

if  $w_i \leq x$

$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$

else

$F[i, x] \leftarrow F[i - 1, x]$

$i = 3$   
 $p_i = 5$   
 $w_i = 4$   
 $x = 5$   
 $x - w_i = 1$

# Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

$i \setminus x$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	

if  $w_i \leq x$

$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$

else

$F[i, x] \leftarrow F[i - 1, x]$

$i = 4$   
 $p_i = 6$   
 $w_i = 5$   
 $x = 1..4$   
 $x - w_i = -4..-1$

# Példa

$w_i; p_i$
1: (2; 3)
2: (3; 4)
3: (4; 5)
4: (5; 6)

$i \setminus x$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

if  $w_i \leq x$

$F[i, x] \leftarrow \max(F[i - 1, x], F[i - 1, x - w_i] + p_i)$

else

$F[i, x] \leftarrow F[i - 1, x]$

$i = 4$   
 $p_i = 6$   
 $w_i = 5$   
 $x = 5$   
 $x - w_i = 0$

# Példa

- Az algoritmus a maximális összsúlyt vette figyelembe úgy, hogy a zsákba tehető  $F[n, c]$  érték a lehető legnagyobb legyen.
- Az elemek kiolvasásához egy visszafele haladó algoritmus szükséges, amely a táblázatot használja:

```
 $i \leftarrow n, x \leftarrow c$   
while ( $i, x > 0$ )  
    if  $F[i, x] \neq F[i - 1, x]$   
        Jelöljük meg az  $i$ . elemet, hogy a zsákban van  
         $i = i - 1, x = x - w_i$   
    else  
         $i = i - 1$ 
```

# 0/1 hátizsák probléma (könyv szerint)

- Adott a  $c$  kapacitású hátizsák és  $n$  objektumból álló halmaz  $1, 2, \dots, n$ . Minden objektumot  $w_i$  súly és  $p_i$  profit jellemez.
- Legyen  $v = [v_1, v_2, \dots, v_n]$  egy megoldás vektor, amelyben  $v_i = 0$  ha az  $i$ . objektum nincs a hátizsákban  $v_i = 1$  ha a hátizsákban van.
- A cél az objektumok olyan részhalmazát megtalálni, amit a hátizsákba helyezhetünk, azaz

$$\sum_{i=1}^n w_i v_i \leq c$$

(az összsúly nem nagyobb, mint a kapacitás) és

$$\sum_{i=1}^n p_i v_i$$

a profit maximalizált.

# 0/1 hátizsák probléma

- Legyen a maximális profit  $F[i, x]$  az  $x$  kapacitású hátizsákhoz kapcsolódóan ha a következő objektum-részhalmazt használjuk  $\{1, 2, \dots, i\}$ . A DP megfogalmazása:

$$F[i, x] = \begin{cases} 0 & x \geq 0, i = 0 \\ -\infty & x < 0, i = 0 \\ \max\{F[i-1, x], (F[i-1, x - w_i] + p_i)\} & 1 < i < n \end{cases}$$



# 0/1 hátizsák probléma

- Konstruáljuk meg az  $n \times c$  méretű  $F$  táblázatot soronként haladva.
- Egy elem kitöltése két elem ismeretét igényli az előző sorból: egy ugyanabból az oszlopból és egy olyan oszloptávolságnyra, mint amilyen súllyal az objektum a sorhoz tartozik.
- Minden elem kiszámolása konstans idejű; a soros futási komplexitás  $\Theta(nc)$ .
- A megfogalmazás soros, egyszerű-argumentumú.

# 0/1 hátizsák probléma

Table F

n								
i					$F_{ij}$			
2								
1								
Weights →	1		$j-w_i$		$j$		$c-1$	$c$
Processors →	$P_0$		$P_{i-w_i-1}$		$P_{i-1}$		$P_{c-2}$	$P_{c-1}$


A 0/1 hátizsák probléma  $F$  táblájának számítása. Az  $F[i,j]$  meghatározásához szükséges kommunikáció processzorelemekkel, amelyek tartalmazzák  $F[i-1,j]$  és  $F[i-1,j-w_i]$  elemeket.





## 0/1 hátizsák probléma

- Ha  $c$  processzorelemet használunk (CREW PRAM ideális, közös memóriás modellen), párhuzamos algoritmust készíthetünk az oszlopok processzorokhoz particionálásával és  $O(n)$  futási idővel.
- Ha osztott memóriás modellen minden processzorelem lokálisan tárolja az objektumok súlyait és profitjait a  $j$ . iteráció során,  $F[j, r]$  számításához a  $P_{r-1}$  processzoron lokálisan rendelkezésre áll  $F[j-1, r]$ , de  $F[j-1, r-w_j]$ -t másik processzorelemről kell megkapni.  $n$  iteráció ekkor  $O(n \log c)$  ideig tart



## Nem soros, egyszerű-argumentumú DP megfogalmazás: Longest-Common-Subsequence (LCS)

- Adott egy  $A = \langle a_1, a_2, \dots, a_n \rangle$  sorozat; az  $A$  részsorozatát kapjuk, ha valahány elemet törölünk belőle.
- Cél: Adott két részsorozat:  $A = \langle a_1, a_2, \dots, a_n \rangle$  és  $B = \langle b_1, b_2, \dots, b_m \rangle$ , keressük meg a leghosszabb sorozatot, ami részsorozata egyaránt  $A$ -nak és  $B$ -nek.
- Példa: ha  $A = \langle c, a, d, b, r, z \rangle$  és  $B = \langle a, s, b, z \rangle$ , a leghosszabb közös részsorozata  $A$ -nak és  $B$ -nek  $\langle a, b, z \rangle$ .

# Leghosszabb közös részsorozat

- Jelölje  $F[i, j]$  az  $A$  első  $i$  elemének és  $B$  első  $j$  elemének leghosszabb közös részsorozat hosszát. Az LCS célja, hogy megtaláljuk  $F[n, m]$  értékét (és a sorozat elemeit).
- Ekkor igaz, hogy:

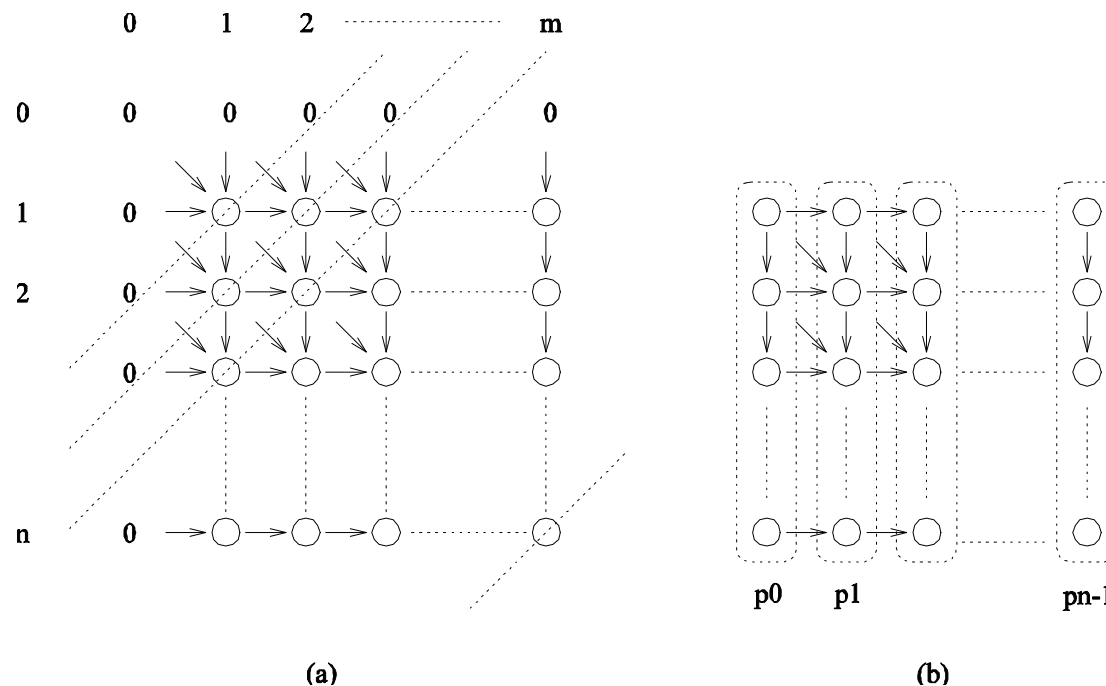
$$F[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ F[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max \{F[i, j - 1], F[i - 1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$



# Leghosszabb közös részsorozat

- Az algoritmus kiszámolja a két-dimenziós  $F$  táblát sor-oszlop sorrendben.
- Az átlós csomópontok mindegyike két részproblémához kapcsolódik, az előző szinthez és az azt megelőző szinthez. Ezért ez a DP megfogalmazás nem soros, egyszerű-argumentumú típusú.

# Leghosszabb közös részsorozat



(a) Az LCS táblázat számítási elemei. A számítás a jelzett átlós irányban halad. (b) A táblázat elemeinek leképezése a processzor elemekre.

# Leghosszabb közös részsorozat

- Consider the LCS of two amino-acid sequences H E A G A W G H E E and P A W H E A E. For the interested reader, the names of the corresponding amino-acids are A: Alanine, E: Glutamic acid, G: Glycine, H: Histidine, P: Proline, and W: Tryptophan.

		H	E	A	G	A	W	G	H	E	E
		0	0	0	0	0	0	0	0	0	0
P		0	0	0	0	0	0	0	0	0	0
A		0	0	0	1	1	1	1	1	1	1
W		0	0	0	1	1	1	2	2	2	2
H		0	1	1	1	1	1	2	2	3	3
E		0	1	2	2	2	2	2	3	4	4
A		0	1	2	3	3	3	3	3	4	4
E		0	1	2	3	3	3	3	3	4	5

- Az LCS: A W H E E.



## Parallel LCS

- A táblázat elemeinek kiszámolása átlós irányban történik a bal felső saroktól a jobb alsó irányban.
- Ha  $n$  processzort használunk (CREW PRAM), minden elem kiszámolása átlósan konstans ideig tart.
- A két  $n$  hosszú szekvenciával az algoritmus  $2n-1$  átlós lépést tesz összesen.

# Parallel LCS

- Az algoritmus processzorok lineáris tömbjére is adaptálható: a  $P_i$  processzor elem az  $F$  tábla  $(i+1)$ . oszlopának meghatározásáért felelős.
- Az  $F[i,j]$  kiszámolásához a  $P_{j-1}$  processzorelemnek szüksége van  $F[i-1,j-1]$  vagy  $F[i,j-1]$  értékre a tőle balra lévő processzor elemtől, amely kommunikációt jelent.
- A számítás konstans idejű ( $t_c$ ) minden elemre.
- A teljes parallel idő:

$$T_P = (2n - 1)(t_s + t_w + t_c).$$

- A hatékonyság felső korlátja 0.5!

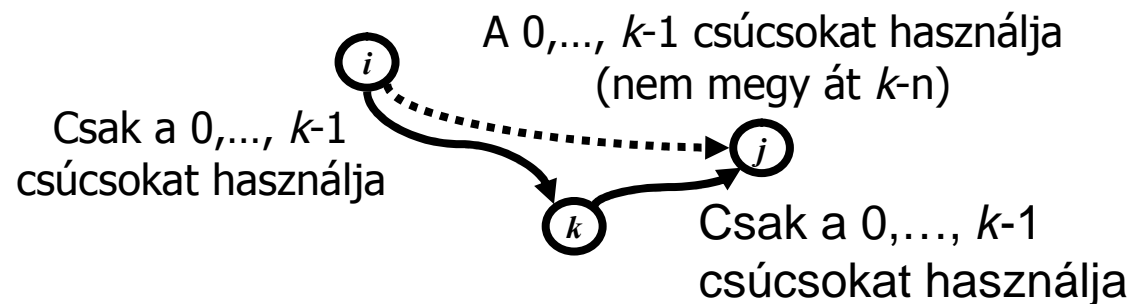


# Soros, összetett-argumentumú DP: Floyd legrövidebb út algoritmus, minden párra

- Adott egy súlyozott gráf  $G(V, E)$ , Floyd algoritmus a csúcspárok közötti  $d_{i,j}$  költségeket határozza meg, hogy a legrövidebb utat célozza meg.
- Legyen  $d_{i,j}^k$  a minimum költségű út az  $i$  csomóponttól a  $j$  csomópontig, és csak a következő csúcsokat használja  $V_0, V_1, \dots, V_{k-1}$ .

■ Így:

$$d_{i,j}^k = \begin{cases} c_{i,j} & k = 0 \\ \min \{d_{i,j}^{k-1}, (d_{i,k}^{k-1} + d_{k,j}^{k-1})\} & 0 \leq k \leq n-1 \end{cases}$$



# Nem soros összetett-argumentumú DP megfogalmazás: Optimális mátrix zárójelezési probléma

- Amikor mátrixok sorozatát kell összeszorozni, akkor a szorzás sorrendje alapvetően meghatározza a műveletek (szorzások) számát.
- Legyen  $C[i,j]$  (*egy résznél  $N[i,j]$  a jelölés!*) az  $A_i, \dots, A_j$  mátrixok optimális szorzásának költsége.
- A láncszorzást két kisebb lánc szorzására lehet bontani,  $A_i, A_{i+1}, \dots, A_k$  és  $A_{k+1}, \dots, A_j$ .
- Az  $A_i, A_{i+1}, \dots, A_k$  eredménye  $d_i \times d_{k+1}$  méretű mátrix, az  $A_{k+1}, \dots, A_j$  lánc eredménye  $d_{k+1} \times d_{j+1}$  méretű mátrix.
- A két mátrix összeszorzásának költsége (a szorzások száma)  $d_i d_{k+1} d_{j+1}$ .

# Mátrixok láncszorzása

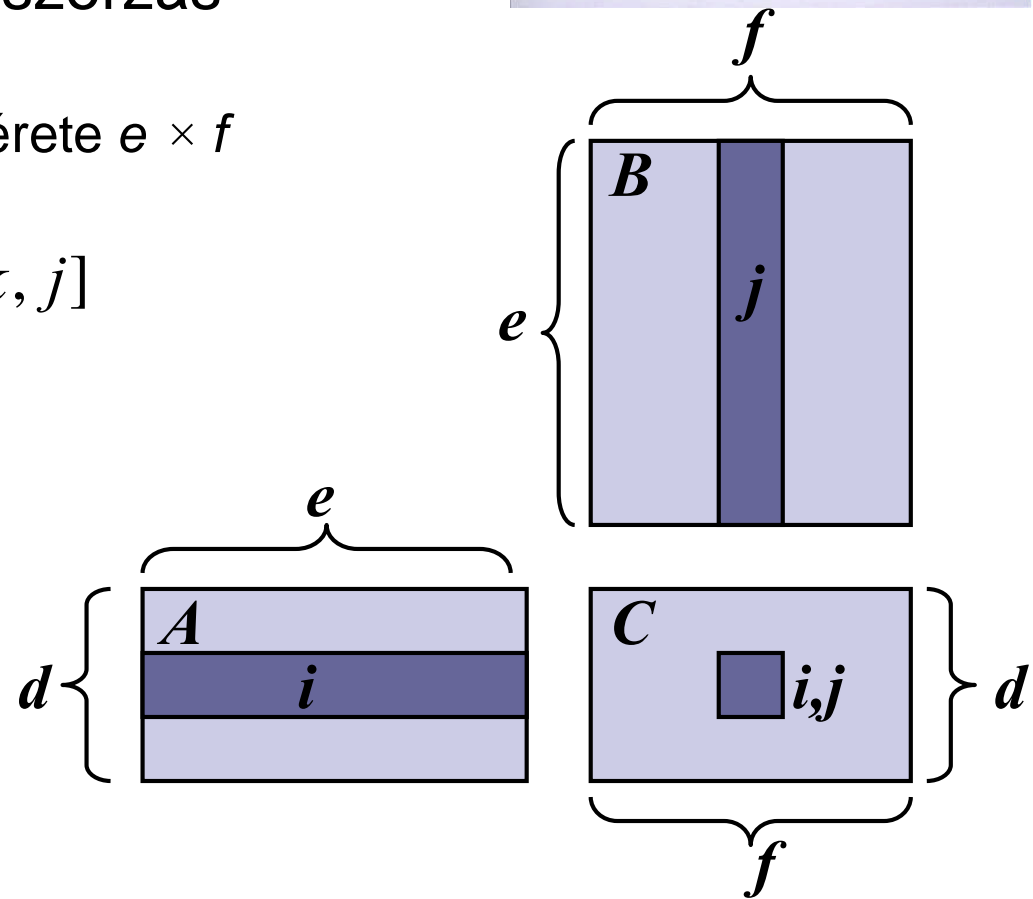
## ■ Emlékeztető: Mátrix-szorzás

□  $C = AB$

□  $A$  mérete  $d \times e$ ,  $B$  mérete  $e \times f$

$$C[i, j] = \sum_{k=0}^{e-1} A[i, k] \times B[k, j]$$

□  $O(def)$  idő



# Mátrixok láncszorzása



## ■ Mátrixok láncszorzása:

- Számítandó  $A = A_0 A_1 \dots A_{n-1}$
- $A_i$  mérete  $d_i \times d_{i+1}$
- Feladat: hogyan zárójelezzük?

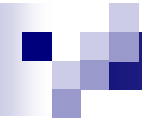
## ■ Példa

- B:  $3 \times 100$
- C:  $100 \times 7$
- D:  $7 \times 5$
- (BC)D      $3 \times 100 \times 7 + 3 \times 7 \times 5 = 2305$  szorzás
- B(CD)      $3 \times 100 \times 5 + 100 \times 7 \times 5 = 5000$  szorzás

# Felsorolásos megközelítés



- Mátrixok láncszorzása algoritmus:
  - Próbáljuk minden lehetséges módon zárójelezni  $A=A_0A_1\dots A_{n-1}$
  - Számoljuk ki minden esetre a szorzások számát
  - A legjobbat vegyük
- Futási idő:
  - A zárójelezések száma = az  $n$  csomópontú bináris fák számával azonos
    - Exponenciálisan nő
  - Gyakorlatban nem megvalósítható



# Egy mohó algoritmus

Ötlet: a legkevesebb műveletet használó szorzatokat válasszuk ki elsőként.

Példa:

- ☐ A:  $101 \times 11$
- ☐ B:  $11 \times 9$
- ☐ C:  $9 \times 100$
- ☐ D:  $100 \times 99$

- ☐ Ötlet szerint:  $A((BC)D)$       $109989+9900+108900=228789$  szorzás
- ☐ A legjobb  $(AB)(CD)$       $9999+89991+89100=189090$  szorzás



# Másik mohó algoritmus

Ötlet: a legtöbb műveletet használó szorzatokat válasszuk ki elsőként.

Példa:

- ☐ A:  $10 \times 5$
- ☐ B:  $5 \times 10$
- ☐ C:  $10 \times 5$
- ☐ D:  $5 \times 10$

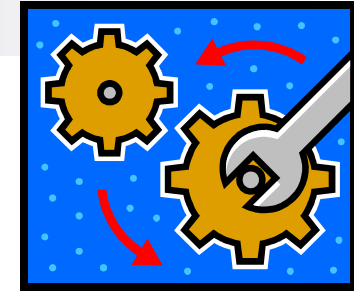
- ☐ Ötlet szerint  $(AB)(CD)$        $500+1000+500 = 2000$  szorzás
- ☐ Legjobb  $A((BC)D)$        $500+250+250 = 1000$  szorzás

# Egy rekurzív megközelítés



- Definiáljunk részproblémákat:
  - Keressük meg az  $A_i A_{i+1} \dots A_j$  legjobb zárójelezését.
  - Legyen  $N_{i,j}$  = a műveletek száma ennél a részfeladatnál.
  - Az egész feladatra optimális megoldás  $N_{0,n-1}$ .
- A részfeladat optimalizálás: Az optimális megoldás optimális részproblémákként lehet definiálni
  - Kell, hogy legyen néhány legvégül futó szorzás (a kifejezésfa gyökere) az optimális megoldásnál.
  - Mondjuk a végső szorzás az  $i$  indexnél van:  
 $(A_0 \dots A_i)(A_{i+1} \dots A_{n-1})$ .
  - Ekkor az optimális megoldás  $N_{0,n-1}$  két optimális részfeladat megoldása:  $N_{0,i}$  és  $N_{i+1,n-1}$ , valamint az utolsó szorzás költsége.
  - Ha a globális optimumnak nem ezek az optimális részproblémái, akkor tudunk definiálni még jobb „optimális” megoldást.





# Karakterisztikus egyenlet

- A globális optimális megoldást optimális részproblémákként kell definiálni, attól függően, hogy hol van az utolsó szorzás.
- Tételezzük fel az összes lehetséges helyet a végső szorzatra:
  - Az  $A_i$  mérete  $d_i \times d_{i+1}$ .
  - Így a karakterisztikus egyenlet  $N_{i,j}$ -re a következő:

$$N_{i,j} = \min_{i \leq k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$

- Ezek a részfeladatok nem függetlenek, hanem átfedésben vannak egymással.

# Dinamikus programozási algoritmus



- Mivel a részfeladatok átfedik egymást, nem használunk rekurziót.
- Helyette “bottom-up” módon építkezünk.
- $N_{i,j}$ -k meghatározása egyszerű, ezzel kezdünk.
- Ezután 2, 3, ... részfeladat.
- Futási idő:  $O(n^3)$

**Algorithm *matrixChain*( $S$ ):**

**Input:** a szorzandó mátrixok  $S$  sorozata

**Output:** az  $S$  optimális zárójelezésének a száma

**for**  $i \leftarrow 1$  **to**  $n-1$  **do**

$N_{i,i} \leftarrow 0$

**for**  $b \leftarrow 1$  **to**  $n-1$  **do**

**for**  $i \leftarrow 0$  **to**  $n-b-1$  **do**

$j \leftarrow i+b$

$N_{i,j} \leftarrow +\infty$

**for**  $k \leftarrow i$  **to**  $j-1$  **do**

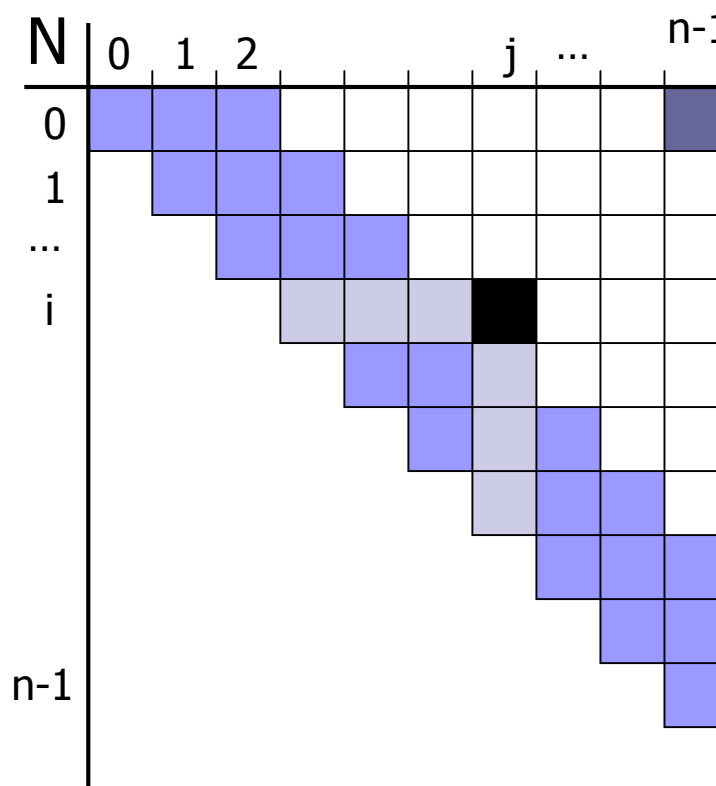
$N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$

# Dinamikus programozás



- A bottom-up konstrukció átlósan tölti ki az  $N$  tömböt.
- $N_{i,j}$  értékeket kap az előző  $i$ . sor és  $j$ . oszlop elemeiből.
- A tábla elemeinek kitöltése  $O(n)$  idejű.
- Teljes futási idő:  $O(n^3)$
- A zárójelezést úgy kaphatjuk meg, ha eltároljuk " $k$ "-t  $N$  minden elemére.

$$N_{i,j} = \min_{i \leq k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$



# Láncszorzás algoritmus



**Algorithm** *matrixChain*( $S$ ):

**Input:** a szorzandó mátrixok  $S$  sorozata

**Output:** az  $S$  optimális zárójelezésének száma

for  $i \leftarrow 0$  to  $n-1$  do

$N_{i,i} \leftarrow 0$

for  $b \leftarrow 1$  to  $n-1$  do //b a műveletek száma

for  $i \leftarrow 0$  to  $n-b-1$  do

$j \leftarrow i+b$

$N_{i,j} \leftarrow +\text{infinity}$

for  $k \leftarrow i$  to  $j-1$  do

$\text{sum} = N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}$

if ( $\text{sum} < N_{i,j}$ ) then

$N_{i,j} \leftarrow \text{sum}$

$O_{i,j} \leftarrow k$

return  $N_{0,n-1}$

■ Példa: ABCD

□ A:  $10 \times 5$

□ B:  $5 \times 10$

□ C:  $10 \times 5$

□ D:  $5 \times 10$

$N$	0	1	2	3
0	0 A	500 AB	500 A(BC)	1000 (A(BC))D
1		0 B	250 BC	500 (BC)D
2			0 C	500 CD
3				0 D

# A műveletek visszanyerése



## ■ Példa: ABCD

- A:  $10 \times 5$
- B:  $5 \times 10$
- C:  $10 \times 5$
- D:  $5 \times 10$

$N$	0	1	2	3
0	0 A	500 <sub>0</sub> AB	500 <sub>0</sub> A(BC)	1000 <sub>2</sub> (A(BC))D
1		0 B	250 <sub>0</sub> BC	500 <sub>1</sub> (BC)D
2			0 C	500 <sub>0</sub> CD
3				0 D

// return expression for multiplying  
 // matrix chain  $A_i$  through  $A_j$

**exp( $i,j$ )**

if ( $i=j$ ) then // base case, 1 matrix  
 return ' $A_i$ '

else

$k = O[i,j]$  // see red values on left

$S1 = \text{exp}(i,k)$  // 2 recursive calls

$S2 = \text{exp}(k+1,j)$

return '(' S1 S2 ')'

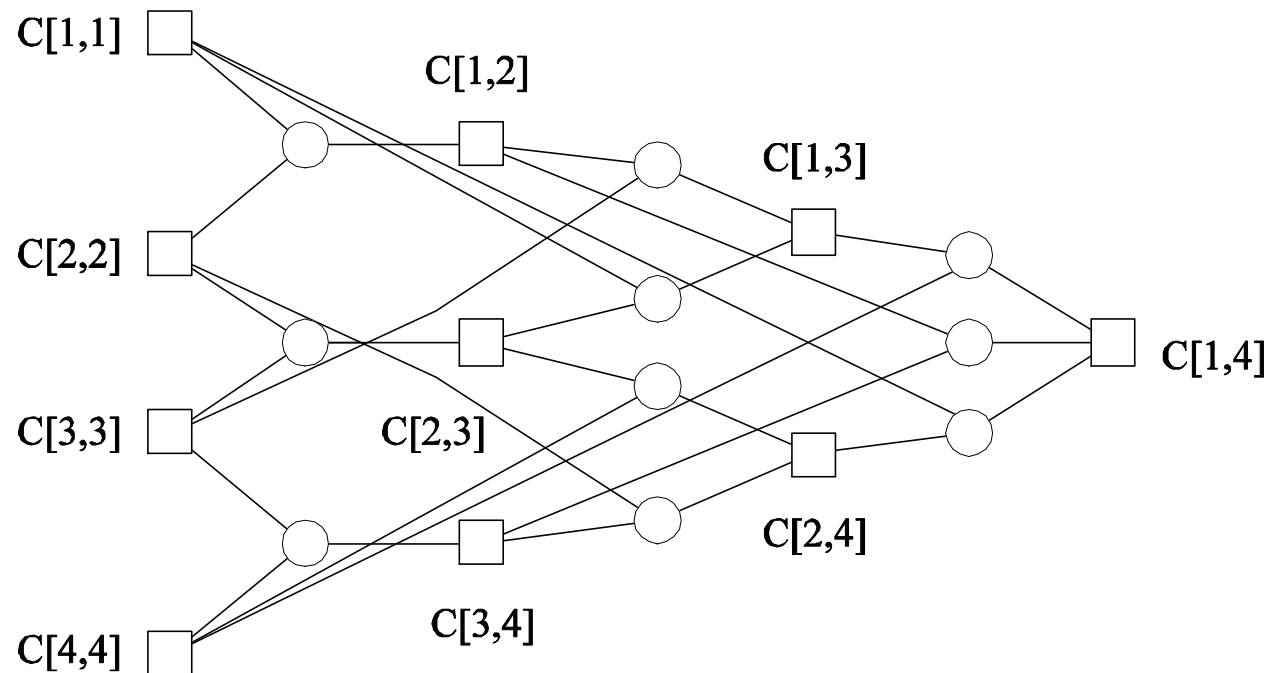


# Optimal Matrix-Parenthesization Problem

- We have:

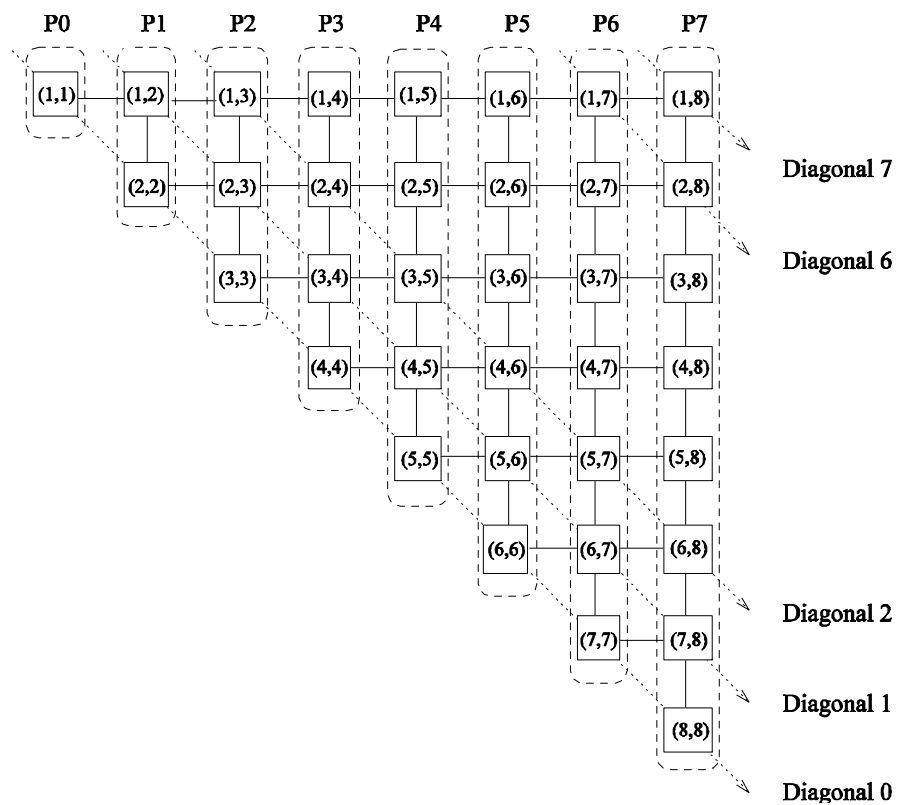
$$C[i, j] = \begin{cases} \min_{i \leq k < j} \{C[i, k] + C[k + 1, j] + r_{i-1}r_kr_j\} & 1 \leq i < j \leq n \\ 0 & j = i, 0 < i \leq n \end{cases}$$

# Optimális mátrix láncszorzás



Nem soros, összetett-argumentumú DP megfogalmazás mátrixok láncszorzásának optimális meghatározására. A négyzetek a mátrixok láncszorzatának optimális költségét jelentik. A körök lehetséges zárójelezést.  $C[i, j]$  jelölés azonos a korábbi  $N_{i,j}$ -vel

# Optimális mátrix láncszorzás



Átlós számítási sorrend

[http://docs.linux.cz/programming/algorithms/Algorithms-Morris/mat\\_chain.html](http://docs.linux.cz/programming/algorithms/Algorithms-Morris/mat_chain.html)

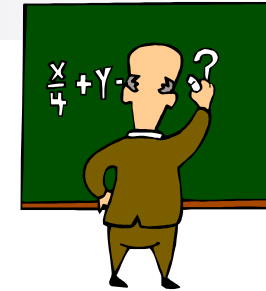




## Párhuzamosítási lehetőség

- Tételezzünk fel egy gyűrűs logikai processzor topológiát. Az  $l$  lépésben minden processzor elem egyetlen elemet számol ki, amely az  $l$ . átlóhoz tartozik.
- A  $C$  tábla kiszámítása során, minden processzor elküldi broadcasting módon az általa kiszámolt elemeket az összes többi processzornak.
- A következő érték lokálisan számítható.

# Az általános dinamikus programozási technika



- Általában olyan feladatoknál alkalmazzuk, amelyek első ránézésre rengeteg időt vesznek igénybe.

Részei:

- **Egyszerű részproblémák:** a részproblémákat néhány változó függvényeként kell definiálni.
- **Részprobléma optimalitás:** a globális optimum a részproblémák optimumaként definiálható.
- **Átfedő, kapcsolódó részproblémák:** a részfeladatok nem függetlenek, hanem átfedőek (ezért bottom-up konstrukcióban kell dolgozni).



# Parallel dinamikus programozási algoritmusok

- A számítási menetet gráfként reprezentálva, háromféle párhuzamosítási lehetőséget azonosíthatunk: csomópontokon belüli párhuzamosság; egy szinten a csomópontok közötti párhuzamosítás; és futószalagosított csomópontok (pipelining) eltérő szintű csomópontok között. AZ első kettő soros megfogalmazású, a harmadik nem soros DP.
- Az adatok helyzete (hová rendeljük) a teljesítmény szempontjából kritikus.