Dynamic Programming

Examples

Matrix-Chain Multiplication Problem

- Given a chain $<A_1, A_2, \ldots, A_n>$ of n matrices where A_i is of size $P_{i-1} \times P_i$
- How can we multiply them so that min # of scalar multiplications is performed

Recursive solution

$$m[i,j] = \begin{cases} 0 & i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\}, & i < j \end{cases}$$

Pseudocode

Matrix-chain-order (p) ; $p = <p_o, p_1, \ldots, p_n>$
1) $n \leftarrow$ length$[p] - 1$
2) for $i \leftarrow 1$ **to** n
3) **do** $m[i, i] \leftarrow 0$
4) for $l \leftarrow 2$ **to** n ; l is length of chain product
5) **do for** $i \leftarrow 1$ **to** $n - l + 1$
6) **do** $j \leftarrow i + l - 1$
7) $m[i, j] \leftarrow inf$
8) **for** $k \leftarrow i$ **to** $j - 1$
9) **do** $q \leftarrow m[i, k] + m[k+1, j] + p_{i-1}p_kp_j$
10) **if** $q < m[i, j]$
11) **then** $m[i, j] \leftarrow q$
12) $s[i, j] \leftarrow k$
13)return m and s

Pseudocode

Matrix-chain-order (p) ; $p = <p_o, p_1, \ldots, p_n>$
1) $n \leftarrow$ length$[p] - 1$
2) for $i \leftarrow 1$ **to** n
3) **do** $m[i, i] \leftarrow 0$
4) for $l \leftarrow 2$ **to** n ; l is length of chain product
5) **do for** $i \leftarrow 1$ **to** $n - l + 1$
6) **do** $j \leftarrow i + l - 1$
7) $m[i, j] \leftarrow inf$
8) **for** $k \leftarrow i$ **to** $j - 1$
9) **do** $q \leftarrow m[i, k] + m[k+1, j] + p_{i-1}p_kp_j$
10) **if** $q < m[i, j]$
11) **then** $m[i, j] \leftarrow q$
12) $s[i, j] \leftarrow k$
13)return m and s

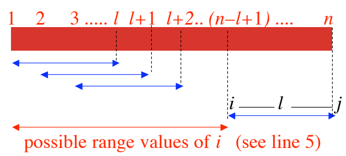
$$m[i,j] = \begin{cases} 0 & , i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\}, & i < j \end{cases}$$

Pseudocode

Matrix-chain-order (p) ; $p = <p_o, p_1, \ldots, p_n>$
1) $n \leftarrow$ length[p] − 1
2) **for** $i \leftarrow 1$ **to** n
3) **do** $m[i, i] \leftarrow 0$
4) **for** $l \leftarrow 2$ **to** n ; l is length of chain product
5) **do for** $i \leftarrow 1$ **to** $n - l + 1$
6) **do** $j \leftarrow i + l - 1$
7) $m[i, j] \leftarrow inf$

i 1 2 3 ...... $(n - l + 1)$
j l l+1 l+2 ... n

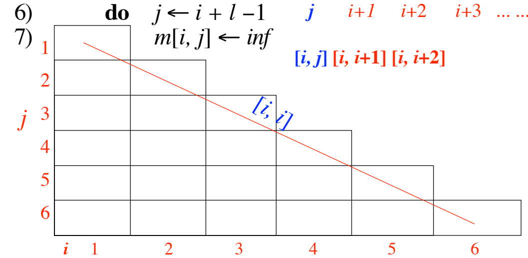
each $[i, j]$ has length l



Pseudocode

Matrix-chain-order (p) ; $p = <p_o, p_1, \ldots, p_n>$
1) $n \leftarrow$ length[p] − 1
2) **for** $i \leftarrow 1$ **to** n
3) **do** $m[i, i] \leftarrow 0$
4) **for** $l \leftarrow 2$ **to** n
5) **do for** $i \leftarrow 1$ **to** $n - l + 1$
6) **do** $j \leftarrow i + l - 1$
7) $m[i, j] \leftarrow inf$

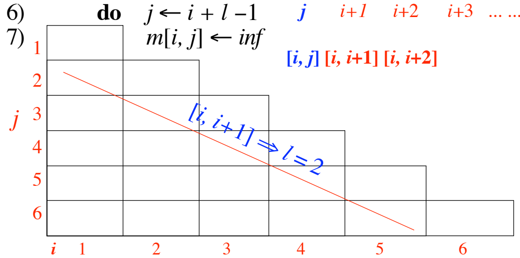
l 2 3 4 ...... n
j i+1 i+2 i+3 ... ... $i + n - 1$
[i, j] [i, i+1] [i, i+2] [$i, i+n-1$]



Pseudocode

Matrix-chain-order (p) ; $p = <p_o, p_1, \ldots, p_n>$
1) $n \leftarrow$ length[p] − 1
2) **for** $i \leftarrow 1$ **to** n
3) **do** $m[i, i] \leftarrow 0$
4) **for** $l \leftarrow 2$ **to** n
5) **do for** $i \leftarrow 1$ **to** $n - l + 1$
6) **do** $j \leftarrow i + l - 1$
7) $m[i, j] \leftarrow inf$

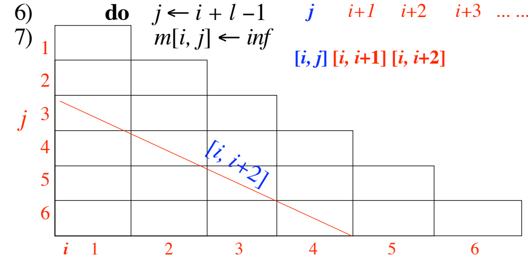
l 2 3 4 ...... n
j i+1 i+2 i+3 ... ... $i + n - 1$
[i, j] [i, i+1] [i, i+2] [$i, i+n-1$]



Pseudocode

Matrix-chain-order (p) ; $p = <p_o, p_1, \ldots, p_n>$
1) $n \leftarrow$ length[p] − 1
2) **for** $i \leftarrow 1$ **to** n
3) **do** $m[i, i] \leftarrow 0$
4) **for** $l \leftarrow 2$ **to** n
5) **do for** $i \leftarrow 1$ **to** $n - l + 1$
6) **do** $j \leftarrow i + l - 1$
7) $m[i, j] \leftarrow inf$

l 2 3 4 ...... n
j i+1 i+2 i+3 ... ... $i + n - 1$
[i, j] [i, i+1] [i, i+2] [$i, i+n-1$]

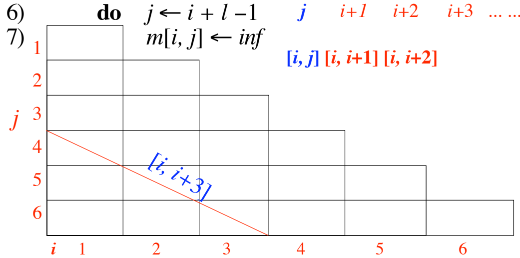


Pseudocode

Matrix-chain-order (p) ; $p = <p_o, p_1, \ldots, p_n>$

```
n ← length[p] – 1
for i ← 1 to n
   do m[i, i] ← 0
for l ← 2 to n
   do for i ← 1 to n – l + 1
         do   j ← i + l –1
              m[i, j] ← inf
```

l	2	3	4	......	n
j	$i+1$	$i+2$	$i+3$	... ...	$i+n-1$
[i, j]	**[i, i+1]**	**[i, i+2]**			**[i, i+n – 1]**

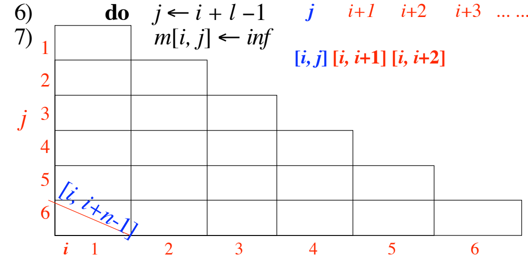


Pseudocode

Matrix-chain-order (p) ; $p = <p_o, p_1, \ldots, p_n>$

```
n ← length[p] – 1
for i ← 1 to n
   do m[i, i] ← 0
for l ← 2 to n
   do for i ← 1 to n – l + 1
         do   j ← i + l –1
              m[i, j] ← inf
```

l	2	3	4	......	n
j	$i+1$	$i+2$	$i+3$	... ...	$i+n-1$
[i, j]	**[i, i+1]**	**[i, i+2]**			**[i, i+n – 1]**



Pseudocode

Matrix-chain-order (p) ; $p = <p_o, p_1, \ldots, p_n>$

```
n ← length[p] – 1
for i ← 1 to n
   do m[i, i] ← 0
for l ← 2 to n                    ; l is length of chain product
   do for i ← 1 to n – l + 1
         do   j ← i + l –1
              m[i, j] ← inf
              for k ← i to j – 1
                 do   q ← m[i, k] + m[k+1, j] + p_{i-1} p_k p_j
                      if q < m[i, j]
                        then m[i, j] ← q
                             s[i, j] ← k
return m and s
```

$$m[i,j] = \begin{cases} 0 & , i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & , i < j \end{cases}$$

Example: Table *m*

$$m[i,j] = \begin{cases} 0 & , i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & , i < j \end{cases}$$

j \ i	1	2	3	4	5	6
1	0					
2	400	0				
3		800	0			
4			160	0		
5				120	0	
6					108	0

$p = <p_o, p_1, \ldots, p_6>$

$p = <5, 10, 8, 10, 2, 6, 9>$

$m[i, i+1]$

$m[1, 2] = m[1, 1] + m[2, 2] + p_0p_1p_2 = 5 \times 10 \times 8 = 400$

$m[2, 3] = m[2, 2] + m[3, 3] + p_1p_2p_3 = 10 \times 8 \times 10 = 800$

$s[i, j] \leftarrow k$

Example: Table S

j \ i	1	2	3	4	5
2	1				
3		2			
4					
5					
6					

Example: Table m

$$m[i, j] = \begin{cases} 0 & , i = j \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} & , i < j \end{cases}$$

$p = \langle p_0, p_1, \ldots, p_6 \rangle$

$p = \langle 5, 10, 8, 10, 2, 6, 9 \rangle$

j \ i	1	2	3	4	5	6
1	0					
2	400	0				
3	800	800	0			
4		320	160	0		
5			286	120	0	
6				288	108	0

$m[i, i+2]$ e.g., $m[1, 3]$

min of $k = 1$ ... $m[1, 1] + m[2, 3] + p_0 p_1 p_3 = 0 + 800 + 5 \times 10 \times 10 = 1300$

$k = 2$ ... $m[1, 2] + m[3, 3] + p_0 p_2 p_3 = 400 + 0 + 5 \times 8 \times 10 = 800$

Example: Table S

j \ i	1	2	3	4	5
2	1				
3	2	2			
4			3		
5				4	
6					5

Example: Table m

$$m[i, j] = \begin{cases} 0 & , i = j \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} & , i < j \end{cases}$$

$p = \langle p_0, p_1, \ldots, p_6 \rangle$

$p = \langle 5, 10, 8, 10, 2, 6, 9 \rangle$

j \ i	1	2	3	4	5	6
1	0					
2	400	0				
3	800	800	0			
4		320	160	0		
5		440	286	120	0	
6				288	108	0

$m[2, 5]$ find *min* when

$k = 2$ ... $m[2, 2] + m[3, 5] + p_1 p_2 p_5 = 0 + 286 + 10 \times 8 \times 6 = 766$

$k = 3$ ... $m[2, 3] + m[4, 5] + p_1 p_3 p_5 = 800 + 120 + 10 \times 10 \times 6 = 1520$

$k = 4$ ... $m[2, 4] + m[5, 5] + p_1 p_4 p_5 = 320 + 0 + 10 \times 2 \times 6 = 440$

Example: Table m

$$m[i,j] = \begin{cases} 0 & , i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & , i < j \end{cases}$$

$p = \langle p_o, p_1, \ldots, p_6 \rangle$

$p = \langle 5, 10, 8, 10, 2, 6, 9 \rangle$

j \ i	1	2	3	4	5	6
1	0					
2	400	0				
3	800	800	0			
4		320	160	0		
5		440	286	120	0	
6				288	108	0

How many times $m[2, 4]$ is used?

Three times: for $m[2, 5]$, $m[2, 6]$, $m[1,4]$

m

j \ i	1	2	3	4	5	6
1	0					
2	400	0				
3	800	800	0			
4	420	320	160	0		
5	480	440	286	120	0	
6	618	608	412	288	108	0

S

j \ i	1	2	3	4	5
2	1				
3	2	2			
4	1	2	3		
5	4	4	4	4	
6	4	4	4	4	5

Constructing an optimal solution

Matrix-chain-multiply (A, S, i, j)

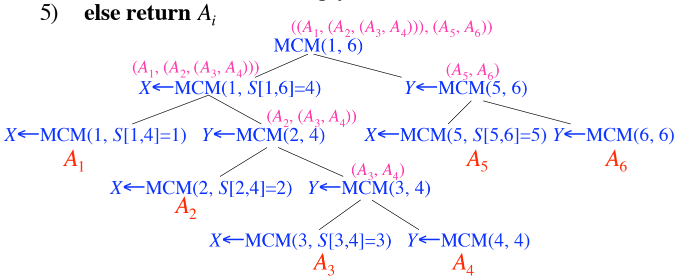
1) if $j > i$

2) **then** $X \leftarrow$ matrix-chain-multiply $(A, S, i, S[i, j])$

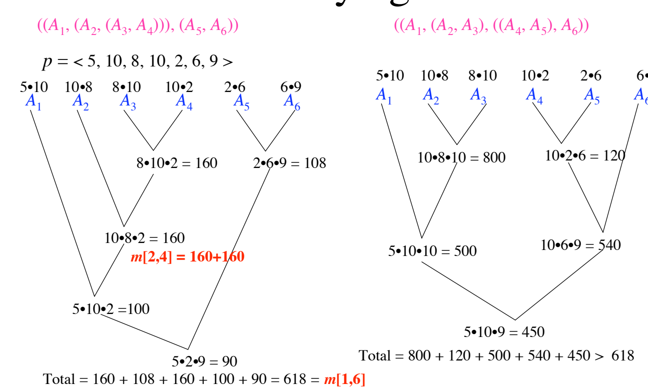
3) $Y \leftarrow$ matrix-chain-multiply $(A, S, S[i, j] + 1, j)$

4) **return** matrix-multiply (X, Y)

5) **else return** A_i



Verifying



Analysis of

Matrix-chain-order

Elements of dynamic programming

- **Optimal substructure**
  - How do we determine reasonable subprogram space?

    E.g., matrix-chain problem: input $<A_1, .. A_n>$

    subspace1: $<A_i, .. A_j>$ subspace1 is preferred

    subspace2: $<A_{i_1}, .. A_{i_n}>$
- **Overlapping subproblems**
  - Want total # of distinct sub-problems to be polynomial

    E.g., matrix-chain-order has $\theta(n^2)$ distinct sub-problems

    (there are n (case $i=j$) + Combination(n, 2) (case $i<j$) sub-problems)
  - Take advantage by sharing(reusing) solutions of sub-problems
- **Find optimal solution from recurrences**
  - Bottom up strategy, e.g., **matrix-chain-order**
  - Topdown strategy, e.g., **recursive-matrix-chain**

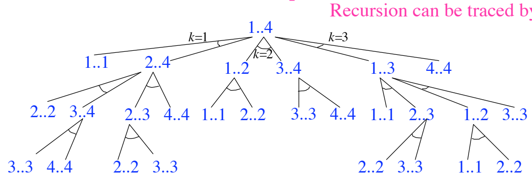
Recursive topdown

Recursive-matrix-chain (p, i, j) ; $p = <p_o, p_1, \ldots, p_n>$

1) **if** $i=j$
2) **then return** 0
3) $m[i, j] \leftarrow$ inf
4) **for** $k \leftarrow i$ **to** $j - 1$
5) **do** $q \leftarrow$ **recursive-matrix-chain** (p, i, k) + **recursive-matrix-chain** $(p, k+1, j) + p_{i-1}p_kp_j$
6) **if** $q < m[i, j]$
7) **then** $m[i, j] \leftarrow q$
8) **return** $m[i, j]$

$$m[i, j] = \begin{cases} 0 & , i = j \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} & , i < j \end{cases}$$

Example: Call recursive-matrix-chain(p, 1, 4)
Recursion can be traced by DFS



Recursive topdown: Analysis

Recursive-matrix-chain (p, i, j) ; $p = <p_o, p_1, \ldots, p_n>$

1) **if** $i=j$
2) **then return** 0
3) $m[i, j] \leftarrow$ inf
4) **for** $k \leftarrow i$ **to** $j - 1$
5) **do** $q \leftarrow$ **recursive-matrix-chain** (p, i, k) + **recursive-matrix-chain** $(p, k+1, j) + p_{i-1}p_kp_j$
6) **if** $q < m[i, j]$
7) **then** $m[i, j] \leftarrow q$
8) **return** $m[i, j]$

$$m[i, j] = \begin{cases} 0 & , i = j \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} & , i < j \end{cases}$$

call 1.. n size n — 1.. k size k; k+1 .. n size n-k

$$T(n) \ge \underbrace{\theta(1)}_{1)-2),\ 6)-8)} + \underbrace{\sum_{k=1}^{n-1}}_{4)} [\underbrace{T(k) + T(n-k) + \theta(1)}_{5)}]$$

$$T(n) \geq \theta(1) + \sum_{k=1}^{n-1} [T(k) + T(n-k) + \theta(1)]$$

$$= \theta(1) + 2\sum_{i=1}^{n-1} T(i) + \theta(n-1)$$

Assume $T(k) \geq 2^{k-1}$ for all $k < n$

$$T(n) \geq 2\sum_{i=1}^{n-1} 2^{i-1} + \theta(n) = \sum_{i=0}^{n-1} 2^{i} - 1 + \theta(n) \geq 2^{n} - 2 + cn \geq 2^{n} > 2^{n-1}$$

for some c > 0

Therefore, $T(n) = \Omega(2^n)$

Topdown recursive

- not always efficient
  - repeatedly resolves each sub-problem each time it reappears in the recursive tree
  - for independent sub-problems

    --> Divide & conquer
  - when sub-problems overlap with small total number of different sub-problems

    -->Dynamic Programming (maybe)

Memoization

Idea: use topdown recursive alg to maintain table with sub-problem solutions

Memoization-matrix-chain (p)

1) $n \leftarrow$ length[p] – 1
2) **for** $i \leftarrow 1$ **to** n
3) **do for** $j \leftarrow i$ **to** n
4) **do** $m[i, j] \leftarrow inf$
5) **return** lookup-chain(p, 1, n)

Lookup-chain (p, i, j)

1) **if** $m[i, j] < inf$
2) **then return** $m[i, j]$
3) **if** $i = j$
4) **then** $m[i, j] \leftarrow 0$
5) **else for** $k \leftarrow i$ **to** j-1

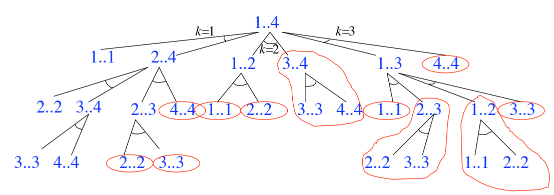
Recursive-matrix-chain (p, i, j)

1) **if** $i=j$
2) **then return** 0
3) $m[i, j] \leftarrow$ inf
4) **for** $k \leftarrow i$ **to** j – 1
5) **do** $q \leftarrow$ **recursive-matrix-chain** (p,i, k) + **recursive-matrix-chain** (p, k+1, j) + $p_{i-1}p_k p_j$
6) **if** $q < m[i, j]$
7) **then** $m[i, j] \leftarrow q$
8) **return** $m[i, j]$

Memoization

Example:

Using memoization will save computation as shown in enclosed areas



Memoization: Analysis

Idea: there are $\theta(n^2)$ calls to lookup-chain. Each call takes $O(n)$ time

$T(n) = O(n^3)$

Memoization-matrix-chain (p) — $\theta(n^2)$

```
1)   n ← length[p] − 1
2)   for i ← 1 to n
3)   do for j ← i to n
4)        do m[i, j] ← inf
5)   return lookup-chain(p, 1, n)
```

Recursive-matrix-chain (p, i, j)

```
1)   if i=j
2)   then return 0
3)   m[i, j] ← inf
4)   for k ← i to j − 1
5)        do q ← recursive-matrix-chain (p,i, k) +
                 recursive-matrix-chain (p, k+1, j) + p_{i-1}p_k p_j
6)    if q < m[i, j]
7)       then m[i, j] ← q
8)   return m[i, j]
```

Lookup-chain (p, i, j)

```
1)   if m[i, j] < inf
2)       then return m[i, j]
3)   if i = j
4)       then m[i, j] ← 0
5)       else for k ← i to j-1
```

Lines 1–4: $O(1)$; line 5: $O(n)$

Summary

Given an optimation problem

- check optimal subtructure
- find recurive definition of value of an opt. soln.
- compute opt. value and construct opt. solution
  - exhaustive enumeration - $\Omega(4^n/n^{3/2})$
  - repeatedly solve subprob - topdown recursive $\Omega(2^n)$, $O(2^{n-1}n)$
  - reuse opt sol of subprob
    - memoization - $O(n^3)$
    - bottom up DP - $O(n^3)$ - better than memoization by a const factor