Dynamic Programming

Md. Tanvir Rahman

Lecturer, Dept. of ICT

Mawlana Bhashani Science and Technology University

Outline

- Dynamic Programming
- Fibonacci numbers
- Longest Common Subsequence (LCS)
- Matrix Chain Multiplication (MCM)

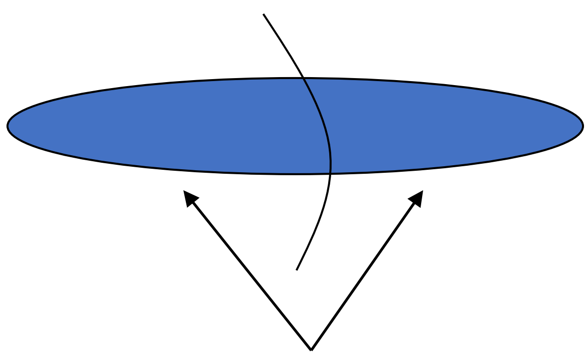
Dynamic Programming

- An algorithm design technique
- DP = Recursion + Reuse

Divide and Conquer	Dynamic Programming
1. Partitions a problem into independent smaller sub problems	1. Partitions a problem into overlapping and dependent sub problems
2. Doesn't store solutions of sub problems for future use	2. Stores solutions of sub problems for future use
3. Less amount of memory is required	3. Higher amount of memory is required

DP - Two key ingredients

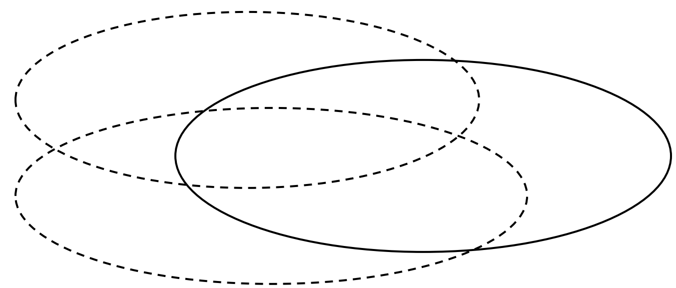
1. optimal substructures



Each substructure is optimal.

(Principle of optimality)

2. overlapping subproblems



Subproblems are dependent.

(otherwise, a divide-and-conquer approach is the choice.)

Three basic components

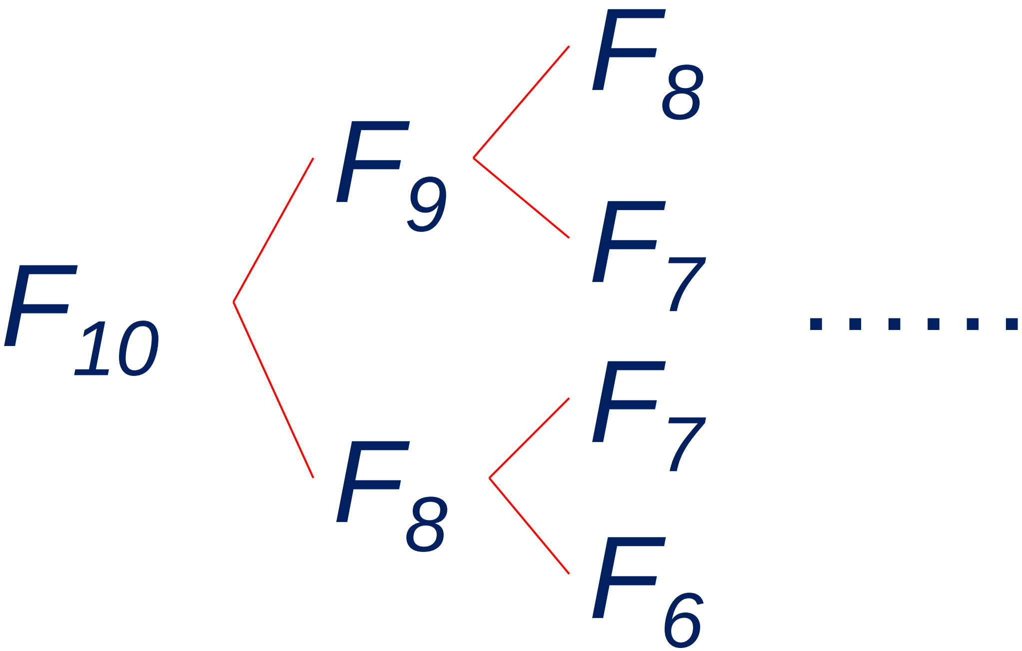
- The development of a dynamic-programming algorithm has three basic components:
    - The recurrence relation (for defining the value of an optimal solution);
    - The tabular computation (for computing the value of an optimal solution);
    - The traceback (for delivering an optimal solution).

Fibonacci numbers

The *Fibonacci numbers* are defined by the following recurrence:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2} \quad \text{for } i > 1 .$$

How to compute F_{10}?



Dynamic Programming

- Applicable when subproblems are not independent rather dependent
  - Subproblems share sub subproblems

E.g.: Fibonacci numbers:

- Recurrence: $F(n) = F(n-1) + F(n-2)$
- Boundary conditions: $F(1) = 0$, $F(2) = 1$
- Compute: $F(5) = 3$, $F(3) = 1$, $F(4) = 2$

- A divide and conquer approach would repeatedly solve the common subproblems
- Dynamic programming solves every subproblem just once and stores the answer in a table

Tabular computation

- The tabular computation can avoid re-compuation.

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
0	1	1	2	3	5	8	13	21	34	55

Result

Longest Common Subsequence (LCS)

- Application: comparison of two DNA strings
- Ex: X= {A B C B D A B }, Y= {B D C A B A}
- Longest Common Subsequence:
- X = A **B** **C** **B** D **A** B
- Y = **B** D **C** A **B** **A**
- Brute force algorithm would compare each subsequence of X with the symbols in Y

Longest Common Subsequence

- Given two sequences

$X = \langle x_1, x_2, \ldots, x_m \rangle$

$Y = \langle y_1, y_2, \ldots, y_n \rangle$

find a maximum length common subsequence (LCS) of X and Y

- *E.g.:*

$X = \langle A, B, C, B, D, A, B \rangle$

- Subsequences of X:
  - A subset of elements in the sequence taken in order

$\langle A, B, D \rangle$, $\langle B, C, D, B \rangle$, etc.

Example

$X = \langle A, B, C, B, D, A, B \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

$X = \langle A, B, C, B, D, A, B \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

- $\langle B, C, B, A \rangle$ and $\langle B, D, A, B \rangle$ are longest common subsequences of X and Y (length = 4)
- $\langle B, C, A \rangle$, however is not a LCS of X and Y

LCS Algorithm

- First, we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.
- Define X_i, Y_j to be the prefixes of X and Y of length i and j respectively
- Define $c[i, j]$ to be the length of LCS of X_i and Y_j
- Then the length of LCS of X and Y will be $c[m, n]$

$$c[i, j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

LCS recursive solution

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i]=y[j], \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

- We start with $i = j = 0$ (empty substrings of x and y)
- Since X_0 and Y_0 are empty strings, their LCS is always empty (i.e. $c[0,0] = 0$)
- LCS of empty string and any other string is empty, so for every i and j: $c[0, j] = c[i,0] = 0$

LCS recursive solution

$$c[i, j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- When we calculate *c[i,j]*, we consider two cases:
- **First case:** *x[i]=y[j]*:
  - one more symbol in strings X and Y matches, so the length of LCS X_i and Y_j equals to the length of LCS of smaller strings X_{i-1} and Y_{i-1}, plus 1

LCS recursive solution

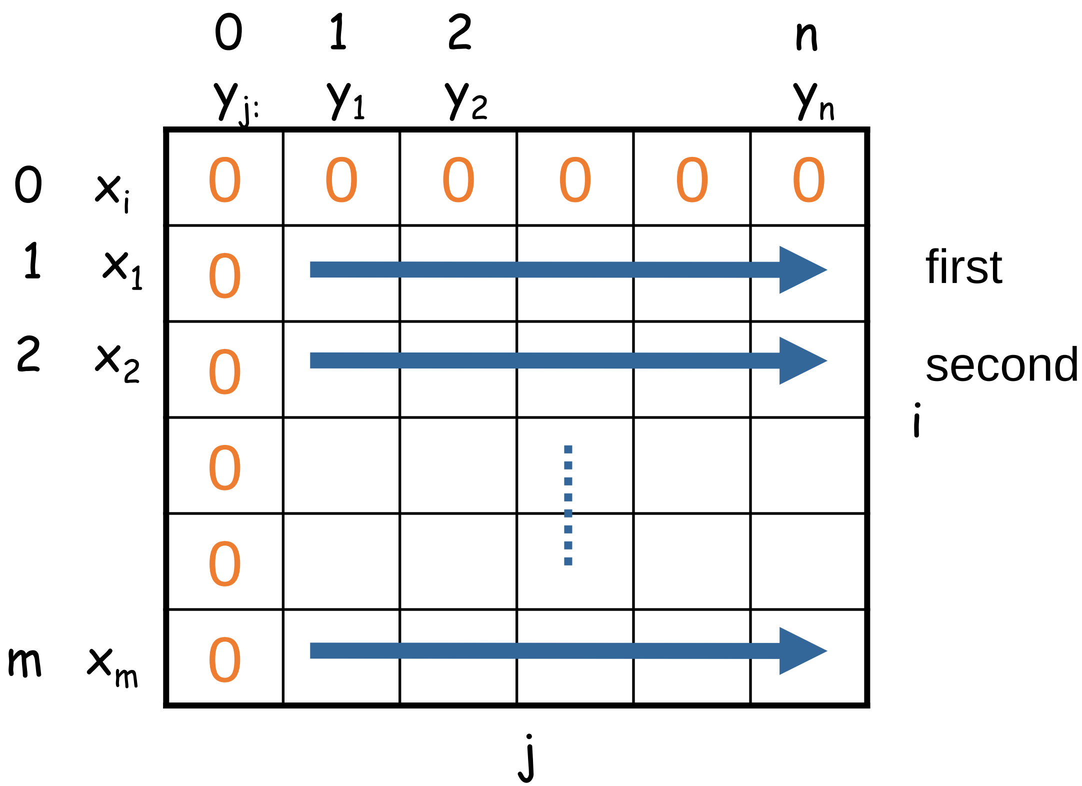
$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- **Second case:** *x[i] != y[j]*
  - As symbols don't match, our solution is not improved, and the length of LCS(X_i , Y_j) is the same as before (i.e. maximum of LCS(X_i, Y_{j-1}) and LCS(X_{i-1},Y_j)

Why not just take the length of LCS (X_{i-1}, Y_{j-1}) ?

3. Computing the Length of the LCS

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



Additional Information

$$c[i, j] = \begin{cases} 0 & \text{if } i,j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } x_i \neq y_j \end{cases}$$

b & c:

		0	1	2	3		n
		y_j:	A	C	D		F
0	x_i	0	0	0	0	0	0
1	A	0					
2	B	0			c[i-1,j]		
3	C	0		↖ c[i,j-1]	↑		
		0					
m	D	0					

i

j

A matrix b[i, j]:

- For a subproblem [i, j] it tells us what choice was made to obtain the optimal value
- If $x_i = y_j$

  b[i, j] = "↖"
- Else, if

  c[i - 1, j] ≥ c[i, j-1]

  b[i, j] = " ↑ "

  else

  b[i, j] = " ← "

LCS-LENGTH(X, Y, m, n)

```
for i ← 1 to m
    do c[i, 0] ← 0
for j ← 0 to n
    do c[0, j] ← 0
for i ← 1 to m
    do for j ← 1 to n
        do if x_i = y_j
            then c[i, j] ← c[i - 1, j - 1] + 1
                b[i, j ] ← "↖"
            else if c[i - 1, j] ≥ c[i, j - 1]
                then c[i, j] ← c[i - 1, j]
                    b[i, j] ← "↑"
                else c[i, j] ← c[i, j - 1]
                    b[i, j] ← "←"
return c and b
```

The length of the LCS if one of the sequences is empty is zero

Case 1: $x_i = y_j$

Case 2: $x_i \neq y_j$

Running time: $\Theta(mn)$

Example

$X = \langle A, B, C, B, D, A \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

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

If $x_i = y_j$

$b[i, j] =$ " ↖ "

Else if

$c[i - 1, j] \geq c[i, j-1]$

$b[i, j] =$ " ↑ "

else

$b[i, j] =$ " ← "

		0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

4. Constructing a LCS

- Start at **b[m, n]** and follow the arrows
- When we encounter a " ↖ " in **b[i, j]** $\Rightarrow$ $x_i = y_j$ is an element of the LCS

		0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

PRINT-LCS(b, X, i, j)

```
if i = 0 or j = 0
  then return
if b[i, j] = "↖"
  then PRINT-LCS(b, X, i - 1, j - 1)
       print x_i
elseif b[i, j] = "↑"
       then PRINT-LCS(b, X, i - 1, j)
       else PRINT-LCS(b, X, i, j - 1)
```

Running time: $\Theta(m + n)$

Initial call: PRINT-LCS(b, X, length[X], length[Y])

Improving the Code

- If we only need the length of the LCS
  - LCS-LENGTH works only on two rows of c at a time
    - The row being computed and the previous row
  - We can reduce the asymptotic space requirements by storing only these two rows

LCS Algorithm Running Time

- LCS algorithm calculates the values of each entry of the array c[m,n]
- So what is the running time?

O (m * n)

since each c[i,j] is calculated in constant time, and there are m*n elements in the array

Matrix-chain Multiplication

- Suppose we have a sequence or chain $A_1, A_2, \ldots, A_n$ of n matrices to be multiplied
  - That is, we want to compute the product $A_1A_2\ldots A_n$

- There are many possible ways (parenthesizations) to compute the product

Matrix-chain Multiplication ...contd

- Example: consider the chain A_1, A_2, A_3, A_4 of 4 matrices
  - Let us compute the product $A_1A_2A_3A_4$
- There are 5 possible ways:

1. $(A_1(A_2(A_3A_4)))$
2. $(A_1((A_2A_3)A_4))$
3. $((A_1A_2)(A_3A_4))$
4. $((A_1(A_2A_3))A_4)$
5. $(((A_1A_2)A_3)A_4)$

Matrix-chain Multiplication ...contd

- To compute the number of scalar multiplications necessary, we must know:
  - Algorithm to multiply two matrices
  - Matrix dimensions
- Can you write the algorithm to multiply two matrices?

Algorithm to Multiply 2 Matrices

Input: Matrices $A_{p\times q}$ and $B_{q\times r}$ (with dimensions $p\times q$ and $q\times r$)
Result: Matrix $C_{p\times r}$ resulting from the product $A\cdot B$

MATRIX-MULTIPLY$(A_{p\times q}, B_{q\times r})$

1. **for** $i \leftarrow 1$ **to** p
2. **for** $j \leftarrow 1$ **to** r
3. $C[i, j] \leftarrow 0$
4. **for** $k \leftarrow 1$ **to** q
5. $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
6. **return** C

Scalar multiplication in line 5 dominates time to compute C
Number of scalar multiplications = pqr

Matrix-chain Multiplication ...contd

- Example: Consider three matrices $A_{10\times100}$, $B_{100\times5}$, and $C_{5\times50}$
- There are 2 ways to parenthesize
  - $((AB)C) = D_{10\times5} \cdot C_{5\times50}$
    - $AB \Rightarrow 10\cdot100\cdot5=5{,}000$ scalar multiplications
    - $DC \Rightarrow 10\cdot5\cdot50 =2{,}500$ scalar multiplications
  - $(A(BC)) = A_{10\times100} \cdot E_{100\times50}$
    - $BC \Rightarrow 100\cdot5\cdot50=25{,}000$ scalar multiplications
    - $AE \Rightarrow 10\cdot100\cdot50 =50{,}000$ scalar multiplications

Total: 7,500

Total: 75,000

Matrix-Chain Multiplication

- Matrix-chain multiplication problem
  - Given a chain $A_1, A_2, \ldots, A_n$ of n matrices, where for i=1, 2, ..., n, matrix A_i has dimension $p_{i-1} \times p_i$.
  - Parenthesize the product $A_1A_2 \ldots A_n$ such that the total number of scalar multiplications is minimized.
- Note that in the matrix-chain multiplication problem, we are not actually multiplying matrices.
- Our aim is only to determine the order for multiplying matrices that has the lowest cost.

Dynamic Programming Approach

- Step 1: The structure of an optimal solution
  - Let us use the notation $A_{i..j}$ for the matrix that results from the product $A_i A_{i+1} \ldots A_j$
  - An optimal parenthesization of the product $A_1 A_2 \ldots A_n$ splits the product between A_k and A_{k+1} for some integer k where $1 \le k < n$
  - First compute matrices $A_{1..k}$ and $A_{k+1..n}$; then multiply them to get the final matrix $A_{1..n}$

Dynamic Programming Approach

- Step 2: A Recursive solution
  - Let $m[i, j]$ be the minimum number of scalar multiplications necessary to compute $A_{i..j}$
  - Minimum cost to compute $A_{1..n}$ is $m[1, n]$
  - Suppose the optimal parenthesization of $A_{i..j}$ splits the product between A_k and A_{k+1} for some integer k where $i \leq k < j$
  - $A_{i..j} = (A_i A_{i+1} \ldots A_k) \cdot (A_{k+1} A_{k+2} \ldots A_j) = A_{i..k} \cdot A_{k+1..j}$
  - Cost of computing $A_{i..j}$ = cost of computing $A_{i..k}$ + cost of computing $A_{k+1..j}$ + cost of multiplying $A_{i..k}$ and $A_{k+1..j}$
  - Cost of multiplying $A_{i..k}$ and $A_{k+1..j}$ is $p_{i-1} p_k p_j$
  - $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j \qquad for\ i \leq k < j$
  - $m[i, i] = 0$ for i=1,2,…,n
  - But… optimal parenthesization occurs at one value of k among all possible $i \leq k < j$
  - Check all these and select the best one

Dynamic Programming Approach

- Step 2: A Recursive solution (contd..)
  - Thus, our recursive definition for the minimum cost of parenthesizing the product $A_i A_{i+1} \dots A_j$ becomes

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

Dynamic Programming Approach

- To keep track of how to construct an optimal solution, let us define $s[i, j]$ = value of k at which we can split the product $A_i\ A_{i+1}\ \dots\ A_j$ to obtain an optimal parenthesization.
- That is, s[I, j] equals a value k such that $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1}p_kp_j$

- Step 3: Computing the optimal cost
  - Algorithm: next slide
  - First computes costs for chains of length $l=1$
  - Then for chains of length $l=2,3, \dots$ and so on
  - Computes the optimal cost bottom-up

Dynamic Programming Approach

- **Input**: Array *p*[0...*n*] containing matrix dimensions and *n*
- **Result**: Minimum-cost table *m* and split table *s*

```
Algorithm MatrixChainOrder(p[ ], n)
{
    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] := ∞;
      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
    }
```

Example

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimensions	10x20	20x5	5x15	15x50	50x10	10x15

- P = [10, 20, 5, 15, 50, 10, 15]
- The problem therefore can be phrased as one of filling in the following table representing the values m.

i\j	1	2	3	4	5	6
1	0					
2		0				
3			0			
4				0		
5					0	
6						0

Example

- Chains of length 2 are easy, as there is no minimization required
- $m[i, i+1] = p_{i-1}p_ip_{i+1}$
- m[1, 2] = 10x20x5 = 1000
- m[2, 3] = 20x5x15 = 1500
- m[3, 4] = 5x15x50 = 3750
- m[4, 5] = 15x50x10 = 7500
- m[5, 6] = 50x10x15 = 7500

i\j	1	2	3	4	5	6
1	0	1000				
2		0	1500			
3			0	3750		
4				0	7500	
5					0	7500
6						0

Example

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

- Chains of length 3 require some minimization – but only one each.

- $m[1,3]=\min\{(m[1,1]+m[2,3]+p_2p_1p_3),(m[1,2]+m[3,3]+p_2p_2p_3)\}$

  $= \min\{(0+1500+10 \times 20 \times 5), (1000+0+20 \times 20 \times 5)\}$

  $= \min \{ 2500, 3000 \} = 1750$

Solve it. It has some error.

- $m[2,4]=\min\{(m[2,2]+m[3,4]+p_3p_2p_4),(m[2,3]+m[4,4]+p_1p_3p_4)\}$

  $= \min\{(0+3750+20 \times 5 \times 50), (1500+0+20 \times 15 \times 50)\}$

  $= \min \{ 8750, 16500 \} = 8750$

- $m[3,5]=\min\{(m[3,3]+m[4,5]+p_2p_3p_5),(m[3,4]+m[5,5]+p_2p_4p_5)\}$

  $= \min\{(0+7500+5 \times 15 \times 10), (3750+0+5 \times 50 \times 10)\}$

  $= \min \{ 8250, 6250 \} = 6250$

- $m[4,6]=\min\{(m[4,4]+m[5,6]+p_3p_4p_6),(m[4,5]+m[6,6]+p_3p_5p_6)\}$

  $= \min\{(0+7500+15 \times 50 \times 15), (7500+0+15 \times 10 \times 15)\}$

  $= \min \{ 18750, 9750 \} = 9750$

- P = [10, 20, 5, 15, 50, 10, 15]

Example

i\j	1	2	3	4	5	6
1	0	1000	1750			
2		0	1500	8750		
3			0	3750	6250	
4				0	7500	9750
5					0	7500
6						0

Contd

i\j	1	2	3	4	5	6
1	0	1000	1750	7250	7750	8750
2		0	1500	8750	7250	8500
3			0	3750	6250	7000
4				0	7500	9750
5					0	7500
6						0

Contd

- Step 4: Constructing an optimal solution
  - Our algorithm computes the minimum-cost table *m* and the split table *s*.
  - The optimal solution can be constructed from the split table *s*.
  - Each entry $s[i, j]=k$ shows where to split the product $A_i A_{i+1} \ldots A_j$ for the minimum cost.
  - The following recursive procedure prints an optimal parenthesization.

Contd

```
Algorithm PrintOptimalPerens(s, i, j)
    {       if (i=j) then
                Print "A"i;
            else
            {   Print "(";
                PrintOptimalPerens(s, i, s[i,j]);
                PrintOptimalPerens(s, s[i,j]+1, j);
                Print ")";
            }
    }
```

Contd

- So far we have decided that the best way to parenthesize the expression results in 8750 multiplication.
- But we have not addressed how we should actually DO the multiplication to achieve the value.
- However, look at the last computation we did – the minimum value came from computing

$$A = (A_1A_2)(A_3A_4A_5A_6)$$

(A1 A2) (((A3 A4) A5) A6)

Contd

- Therefore in an auxiliary array, we store value s[1,6]=2.
- In general, as we proceed with the algorithm, if we find that the best way to compute $A_{i..j}$ is as

$$A_{i..j} = A_{i..k}A_{(k+1)..j}$$

then we set

$$s[i, j] = k.$$

- Then from the values of k we can reconstruct the optimal way to parenthesize the expression.

Contd

- If we do this then we find that the s array looks like this:

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

Contd

- We already know that we must compute $A_{1..2}$ and $A_{3..6}$.
- By looking at s[3,6] = 5, we discover that we should compute $A_{3..6}$ as $A_{3..5}A_{6..6}$ and then by seeing that s[3,5] = 4, we get the final parenthesization

$$A = ((A_1A_2)(((A_3A_4)A_5)A_6)).$$

- And quick check reveals that this indeed takes the required 8750 multiplications.