HANDONG GLOBAL UNIVERSITY

Chapter 15
Dynamic Programming

Algorithm Analysis

School of CSEE

Dynamic Programming

- Not a specific algorithm, but a design paradigm.
- Design paradigms discussed so far.
  - Divide and Conquer
  - Incremental
  - Backtracking, etc
- Used for optimization problems:
  - Find *a* solution with *the* optimal value.
  - Minimization or maximization.

Optimization = Making Choices

- Optimization: Determine which are the optimal choices to make.
- The choice we make determines the subproblems we need to solve.
- We do not know the optimal choice.
- So we try all of them and determine the quality of each choice based on the (already known) cost of the resulting subproblems.

Examples of Optimization Problems

- Shortest-path:

– There are many routes from Pohang to Seoul.

– We want to compute the shortest route, the most scenic route, the fastest route, …

- Job scheduling:

– There are many ways to schedule a set of jobs on multiple processors.

– Given different capabilities of the processors and different resource requirements of the jobs, we want to make sure that we can complete as many jobs as possible in as little time as possible.

Dynamic programming

- A dynamic programming is a design strategy that involves constructing a solution S to a given problem by building it up dynamically from the solutions of smaller (or simpler) problems S_1, S_2, …, S_m of the same type, i.e.,

$$S = combine(S_1, S_2, \ldots, S_m)$$

- The solution to any given smaller problem S_i is itself built up from solutions to even smaller subproblems, and so forth.
- We start with the known solutions to the smallest problem instances and build from there in a bottom-up fashion.

Fibonacci Number Example

```
Fib (n){
if (n < 2)
    return n;
else
    return Fib(n-1) + Fib(n-2);
}
```

Divide and Conquer

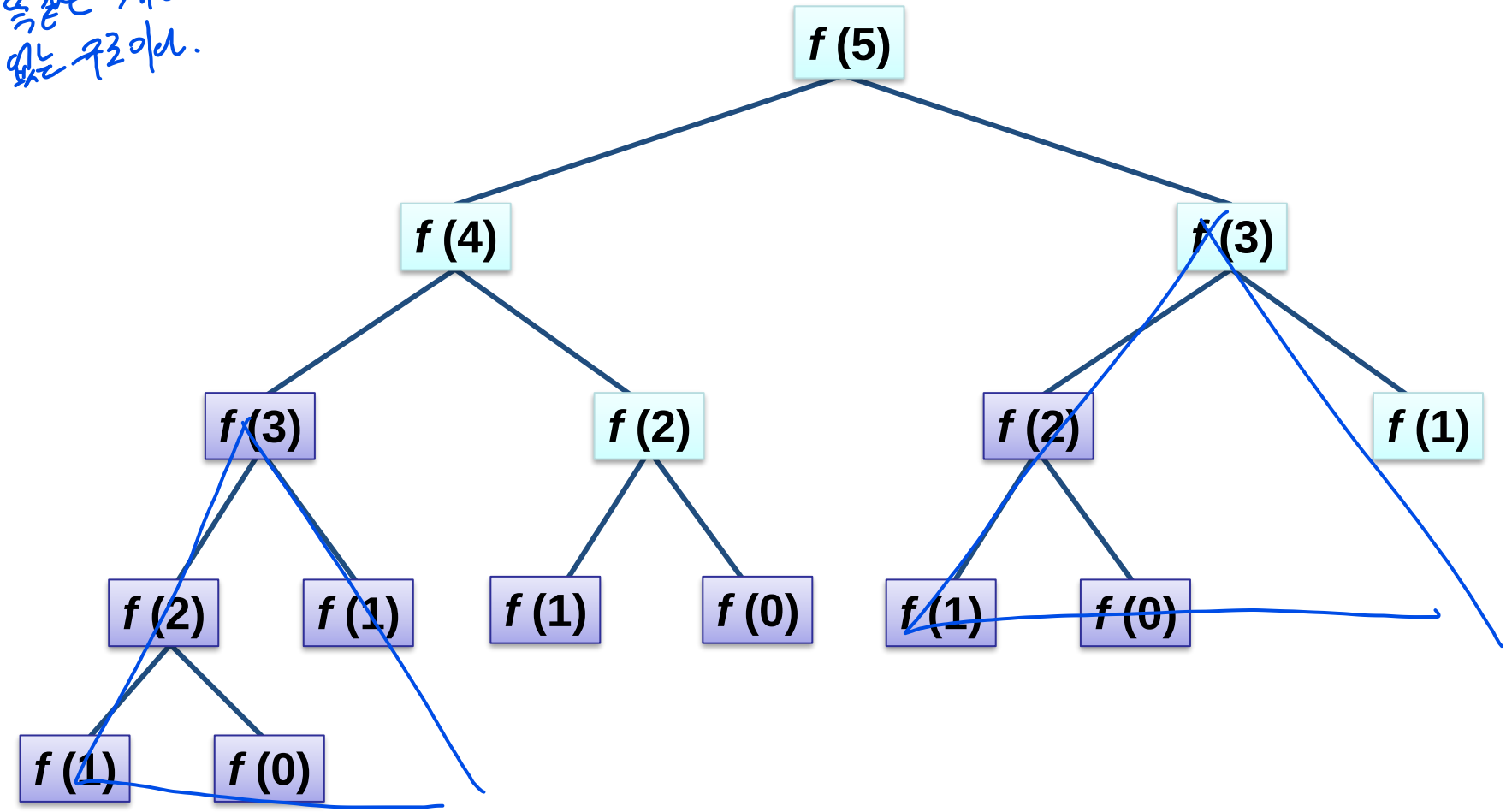
```
Fib (n){
int fib[n+1], i;
fib[0] = 0;
fib[1] = 1;
for i=2 to n
   fib[i] = fib[i-1] + fib[i-2];
return fib[n];
}
```

Dynamic Programming

Recalculation in Divide and Conquer

Divide and Conquer는 같은 계산을 반복적으로 할 수밖에 없는 구조이다.

$$f(i) = f(i\text{-}1) + f(i\text{-}2)$$



Divide and Conquer vs. DP

A divide-and-conquer algorithm may do more work than necessary, repeatedly solving the common subproblems. A dynamic-programming algorithm solves every subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subproblem is encountered. Dynamic programming is typically applied to ***optimization problems.***

Matrix-Chain Multiplication

- Given a sequence $A_1, A_2, \ldots, A_n$ of matrices, we want to compute their product

$$A_1 \cdot A_2 \cdot \ldots \cdot A_n$$

곱하기의 순서에 초점을 맞춘다.
결과는 똑같지만,
곱하는 순서에 따라 곱하는 횟수가 달라진다.

- We do this by parenthesizing and thus computing products of matrix pairs:

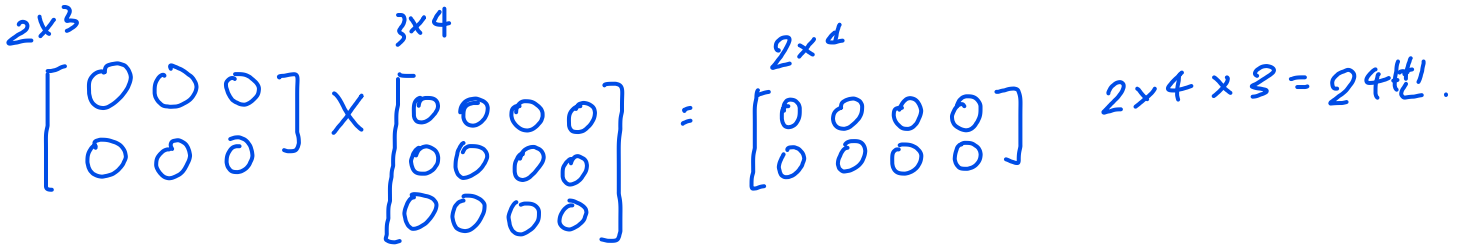
$$A_1 \cdot A_2 \cdot A_3 \cdot A_4 = (A_1 \cdot (A_2 \cdot A_3)) \cdot A_4$$

The Cost of Multiplying Two Matrices

- Given a $p \times q$ matrix A and a $q \times r$ matrix B, their product is a $p \times r$ matrix C defined by

$$C_{i,j} = \sum_{k=1}^{q} A_{i,k} B_{k,j}$$

- The cost of computing C is $p \cdot q \cdot r$.



The Cost of Multiplying Three Matrices

- Given three matrices:

  $A = p \times q$ matrix

  $B = q \times r$ matrix

  $C = r \times s$ matrix

- Two ways to calculate $A \cdot B \cdot C$:

  $(A \cdot B) \cdot C$ or $A \cdot (B \cdot C)$

- The first costs $p \cdot q \cdot r + p \cdot r \cdot s = p \cdot r \cdot (q + s)$.
- The second costs $q \cdot r \cdot s + p \cdot q \cdot s = (p + r) \cdot q \cdot s$.

More Matrices, More Ways to Parenthesize

- There are many ways to parenthesize:

$$(A_1 \cdot (A_2 \cdot (A_3 \cdot A_4)))$$
$$(A_1 \cdot ((A_2 \cdot A_3) \cdot A_4))$$
$$((A_1 \cdot A_2) \cdot (A_3 \cdot A_4))$$
$$((A_1 \cdot (A_2 \cdot A_3)) \cdot A_4)$$
$$(((A_1 \cdot A_2) \cdot A_3) \cdot A_4)$$

- The ways of parenthesizing make a rather dramatic difference in the number of multiplications performed.

Example

Let A_1, A_2, A_3, and A_4 have dimensions 20X10, 10X50, 50X5, and 5X30, respectively.

Parenthesizations	Number of multiplications
$(A_1(A_2(A_3A_4)))$	28500
$(A_1((A_2A_3)A_4))$	10000
$((A_1A_2)(A_3A_4))$	47500
$((A_1(A_2A_3))A_4)$	6500
$(((A_1A_2)A_3)\ A_4)$	18000

← optimal

Number of parenthesizations

- The number of alternative parenthesizations P_n for sequence of n matrices.

$$P_n = \begin{cases} 1 & \text{If } n = 1 \\ \sum_{k=1}^{n-1} P_k P_{n-k} & \text{If } n > 1 \end{cases}$$

$$P_n = P_1 * P_{n-1} + P_2 * P_{n-2} + \cdots\cdots + P_{n-1} * P_1$$

$$\geq 2 * P_{n-1}$$

Therefore, $P_n = \Omega(2^n)$ 곱셈을 할 수 있는 방법의 최소 2^n 개가 있다.

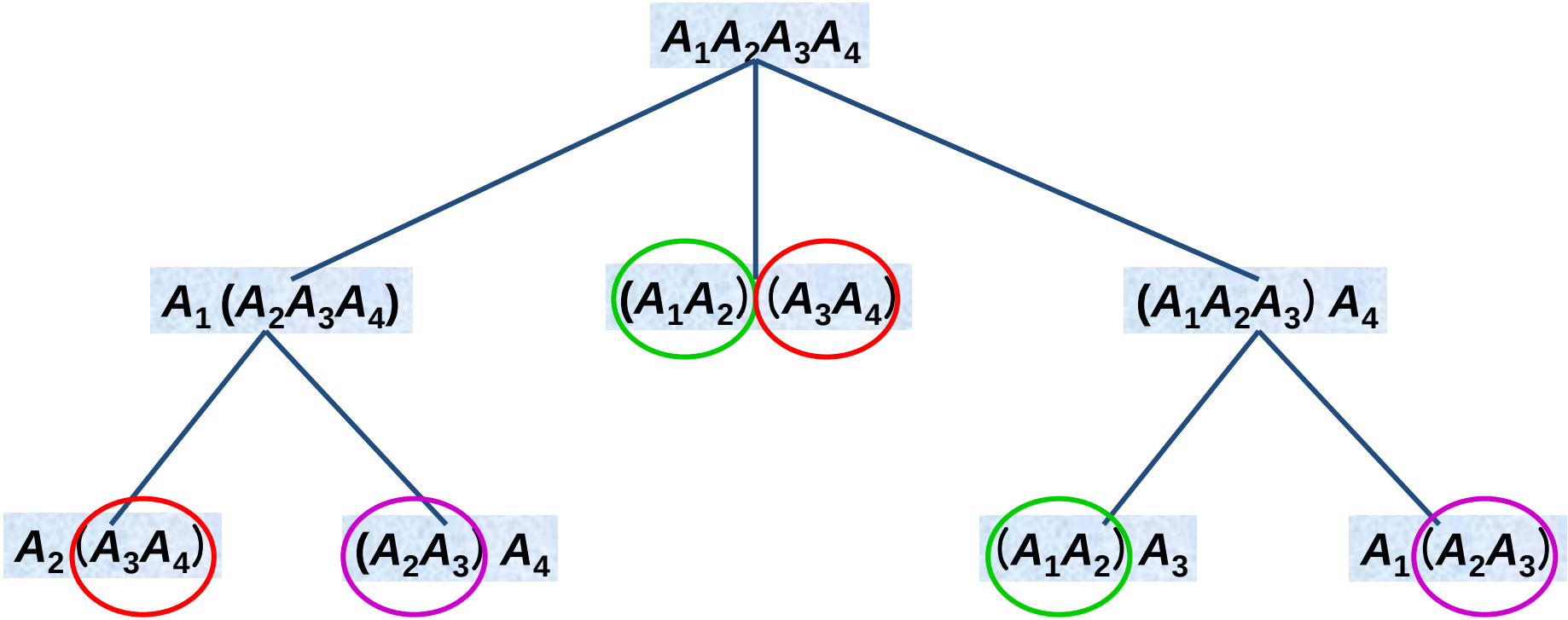
Optimal matrix-chain multiplication

- Problem: Parenthesize a chained matrix product so as to minimize the number of multiplications performed when computing the product.

A brute-force algorithm that examines all possible parenthesizations is computationally infeasible, since the time complexity is

$$\Omega(2^n)$$

Recalculation: The Source of All Evil



Structure of Optimal Solution

Key Observation : Given optimal sequence $A_i A_{i+1} \ldots A_j$, suppose the sequence is split between A_k and A_{k+1}. Then the sequence $A_i \ldots A_k$, and $A_{k+1} \ldots A_j$ are optimal.

Why? If there were a less costly way to parenthesize $A_iA_{i+1}\ldots A_k$, substituting that parenthesization in the optimal parenthesization of $A_iA_{i+1}\ldots A_j$ would produce another parenthesization of $A_iA_{i+1}\ldots A_j$ whose cost was lower than the optimum: a contradiction.

Structure of Optimal Solution

Generally, an optimal solution to a problem (optimal sequence $A_i A_{i+1} \ldots A_j$) contains within it an optimal solution to subproblems (the sequence $A_i \ldots A_k$ and $A_{k+1} \ldots A_j$). This is *optimal substructure*.

Use optimal substructure to construct optimal solution to problem from optimal solutions to subproblems.

When Can We Apply Dynamic Programming?

- The optimization problem must have two properties:

– Optimal Substructure: An optimal solution to an instance contains within it optimal solutions to smaller instances of the same problem.

– Optimal Overlapping Subproblems: A recursive solution to the problem solves certain smaller instances of the same problem over and over again, rather than new subproblems.

Principle of optimality

- Given an optimization problem and an associated function *combine*, the Principle of Optimality holds if the following is always true:

If $S=combine(S_1,S_2,\ldots,S_m)$ and S is an optimal solution to the problem, then $S_1, S_2, \ldots, S_m$ are optimal solutions to their associated subproblems.

Principle of optimality

The method of dynamic programming is most effective in solving optimization problems when the

Principle of Optimality

holds.

The principle of optimality holds for optimal parenthesizing.

Memoization: Avoiding Recalculation

- Memoization: A variation of dynamic programming that often offers the efficiency of the usual dynamic-programming approach while maintaining a top-down strategy. As in ordinary dynamic programming, we maintain a table with subproblem solutions, but the control structure for filling in the table is more like the recursive algorithm. *Use a table to remember previously calculated values.* (Store a memo for oneself.)

Fibonacci Number

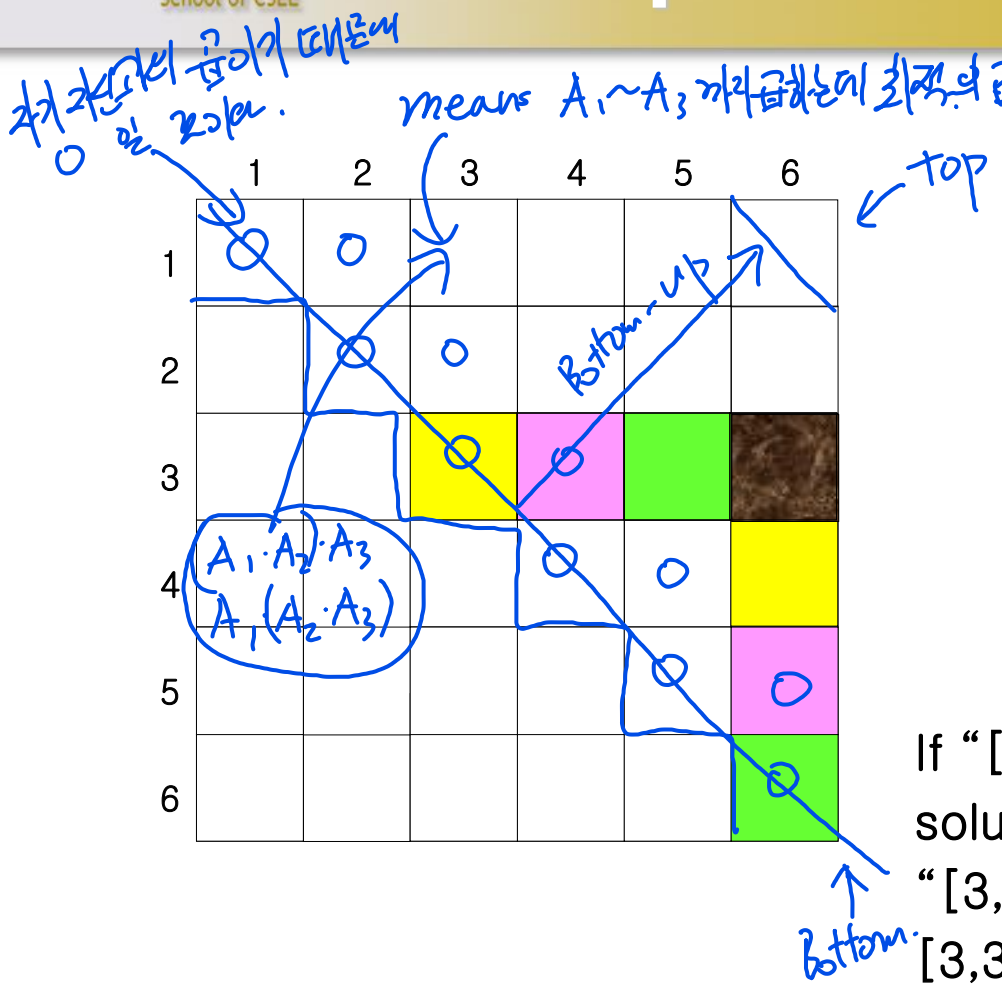
```
Fib (n){
   int fib[n+1], i;
   fib[0] = 0;
   fib[1] = 1;
   for i=2 to n
      fib[i] = fib[i-1] + fib[i-2];
   return fib[n];
}
```

Bottom-up approach

```
Fib (n){
   int fib, f1, f2;
   if (n < 2)
      fib = n;
   else
      if (list[n-1] == false)  f1 = Fib(n-1);
      else  f1 = list[n-1];
      if (list[n-2] == false)  f2 = Fib(n-2);
      else  f2 = list[n-2];
      fib = f1 + f2;
   list[n] = fib;
}
```

Maintains a top-down approach

Optimal Substructure



$A_3A_4A_5A_6$: [3,6]

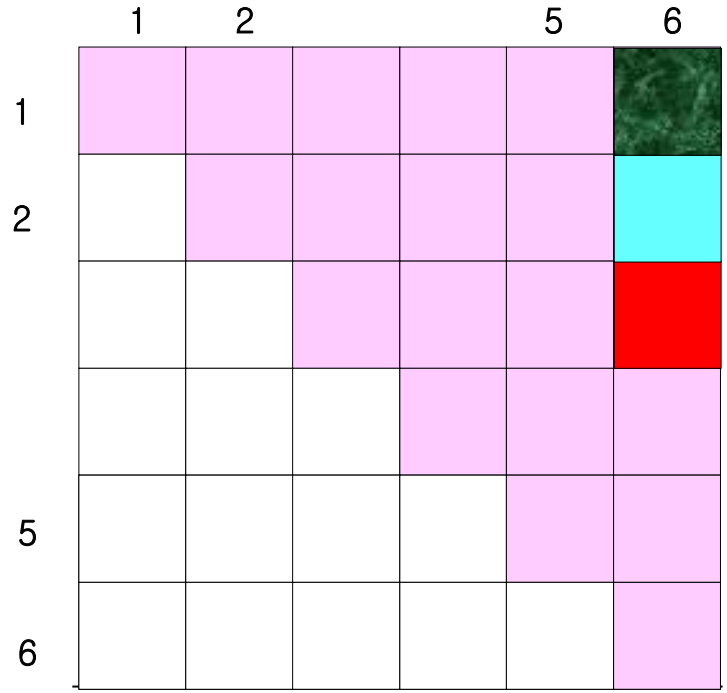
[3,6] = [3,5] X [6,6]

[3,6] = [3,4] X [5,6]

[3,6] = [3,3] X [4,6]

If "[3,6] = [3,4] X [5,6]" is optimal solution , then we don't have to consider "[3,6] = [3,5] X [6,6]" and "[3,6] = [3,3] X [4,6]" any more.

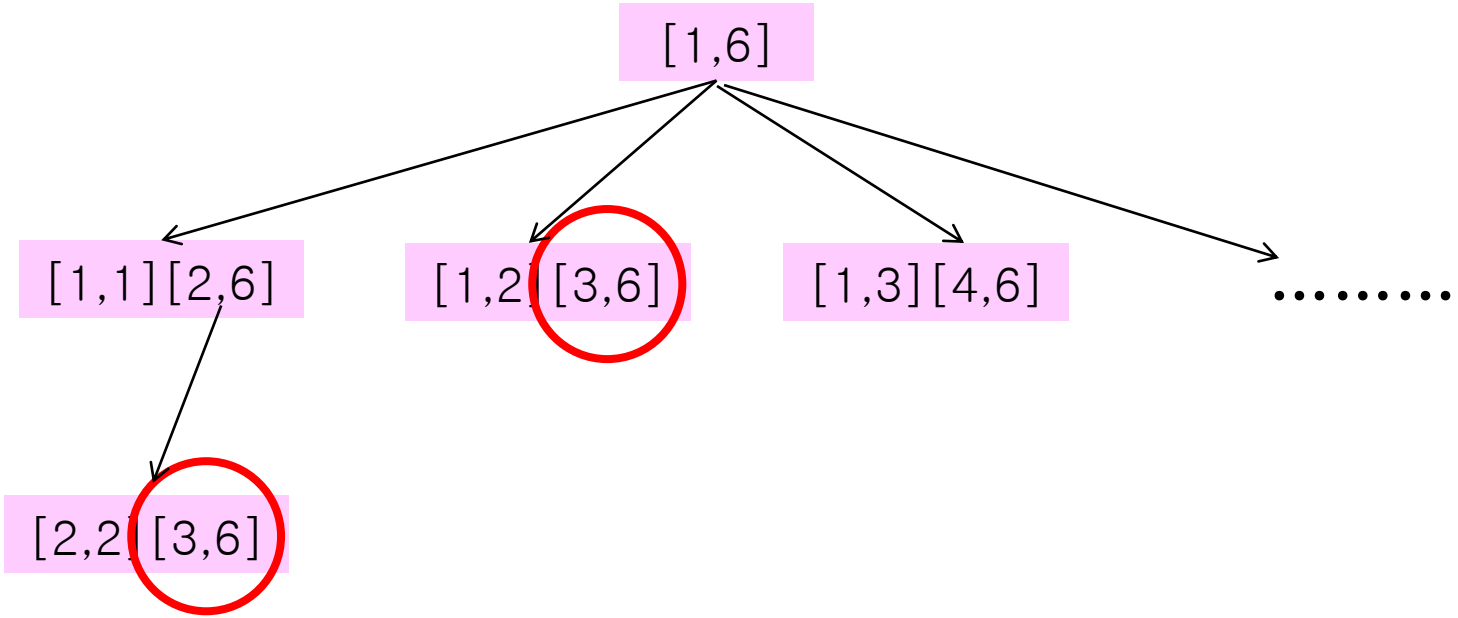
Optimal Substructure



[1,6] = [1,2] X [3,6]

[2,6] = [2,2] X [3,6]

Optimal Substructure



Development of DP algorithms

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Step 1

- Find the optimal substructure

  : From the key observation we identified optimal substructure.

- We can build an optimal solution to an instance of the matrix-chain multiplication problem by splitting the problem into two subproblems (optimally parenthesizing $A_iA_{i+1}A_k$ and $A_{k+1}A_{k+2}A_j$), finding optimal solutions to subproblem instances, and then combining these optimal subproblem solutions..

Step 2: A recursive solution

- $m[i,j]$: the minimum number of scalar multiplications needed to compute $A_i \ldots A_j$.
- final solution: $m[1,n]$
- Assume that the optimal parenthesization splits the product $A_iA_{i+1}...A_j$ between A_k and A_{k+1}, where $i \le k < j$.
- Recursively,
  - if $i=j$, $m[i,j]=0$. (trivial)
  - if $i<j$, $m[i,j]=m[i,k]+m[k+1,j]+p_{i-1}p_kp_j$ for some k.

    where size of matrix $A_i = p_{i-1}p_i$.

$A_1 : A_i \cdot A_{i+1} \cdots A_k$
$: P_{i-1} \cdot P_k$ (when $A_i = P_{i-1}P_i$)

$A_2 = A_{k+1} \cdot A_{k+2} \cdots A_j$
$: P_{k+1} P_j$ (when $A_i = P_{i-1}P_i$)

Number of multi for $A_1 \times A_2$ is $P_{i-1}P_kP_j$

A recursive solution

- The minimum cost of parenthesizing the product $A_iA_{i+1}\ldots A_j$
  - $m[i,j] = \begin{cases} 0 & \text{if } i=j \\ \min_{i \le k < j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & \text{if } i<j \end{cases}$

- $s[i,j]$: a value k s.t. $m[i,j]=m[i,k]+m[k+1,j]+p_{i-1}p_kp_j$

  Remembers where optimal parenthesization splits the product.

Computing the optimal costs

$i = 3,\ j = 6,\ k = 3, 4, 5$

$\min_{i \le k < j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\}$

$k = 3 : m[3,3] + m[4,6] + p_2p_3p_6$

$k = 4 : m[3,4] + m[5,6] + p_2p_4p_6$

$k = 5 : m[3,5] + m[6,6] + p_2p_5p_6$

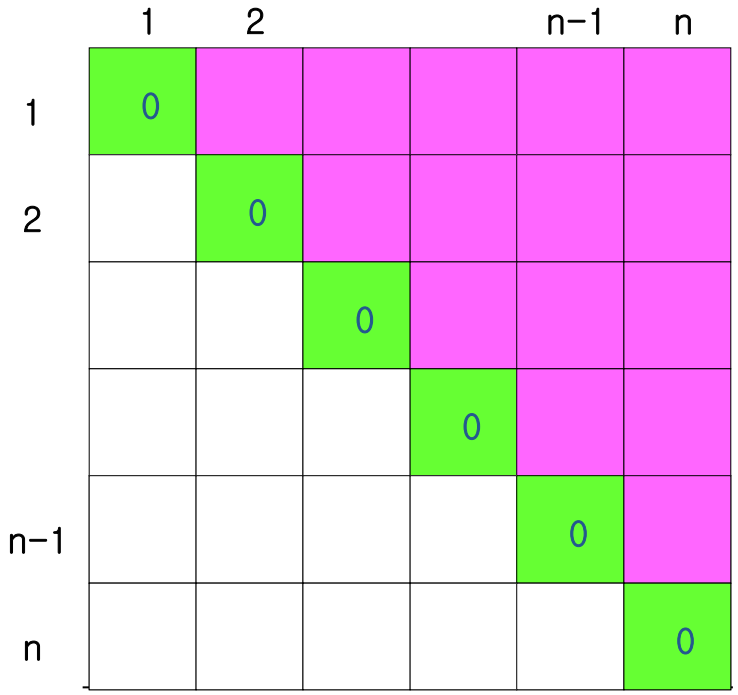
	1	2	3	4	5	6
1	0					
2		0				
3			0			
4				0		
5					0	
6						0

이중에 minimum (optimize)한 방법을 찾는다.

Question. what if more than one method is optimize one?

Step 3: Computing the optimal costs

- Resources
  - Input
    - $p = \langle p_0, p_1, \ldots, p_n \rangle$
  - Auxiliary table
    - $m[1..n, 1..n]$
    - $s[1..n, 1..n]$
- # of subproblems
  - ${}_nC_2 + n = \Theta(n^2)$



Computing the optimal costs

```
MATRIX-CHAIN-ORDER (p)
1.  n ← length[p] - 1
2.  for i ← 1 to n
3.      m[i,i] ← 0
4.  for r ← 2 to n      // r is the chain length.
5.      for i ← 1 to n-r+1
6.          j ← i + r - 1
7.          m[i,j] ← ∞
8.          for k ← i to j-1
9.              q ← m[i,k] + m[k+1,j] + p_{i-1} p_k p_j
10.             if q < m[i,j]
11.                 m[i,j] ← q
12.                 s[i,j] ← k
13. return m and s
```

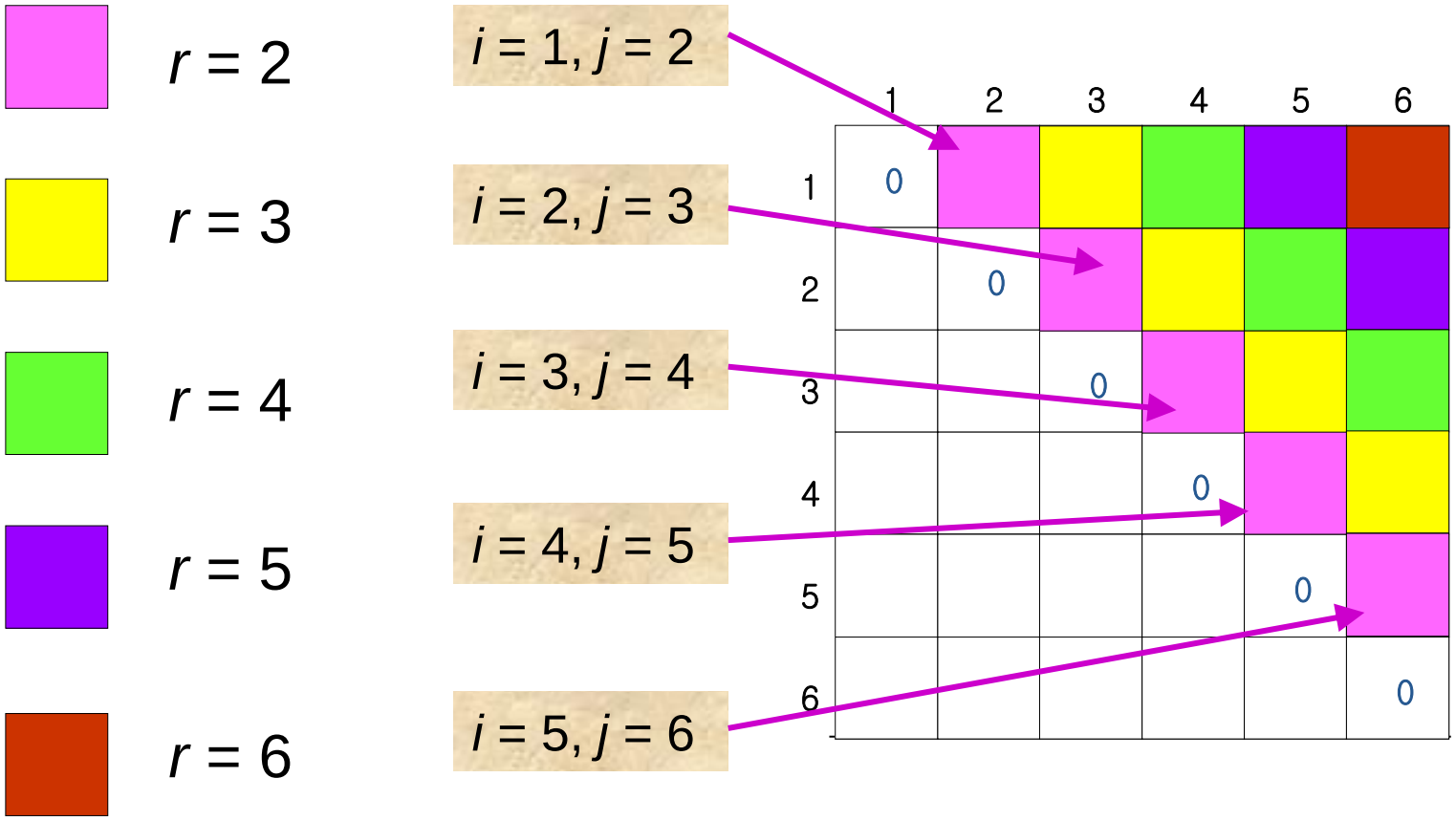
대각선을 구하는 경우는 0이다.

m[i,j]의 초기값 설정.

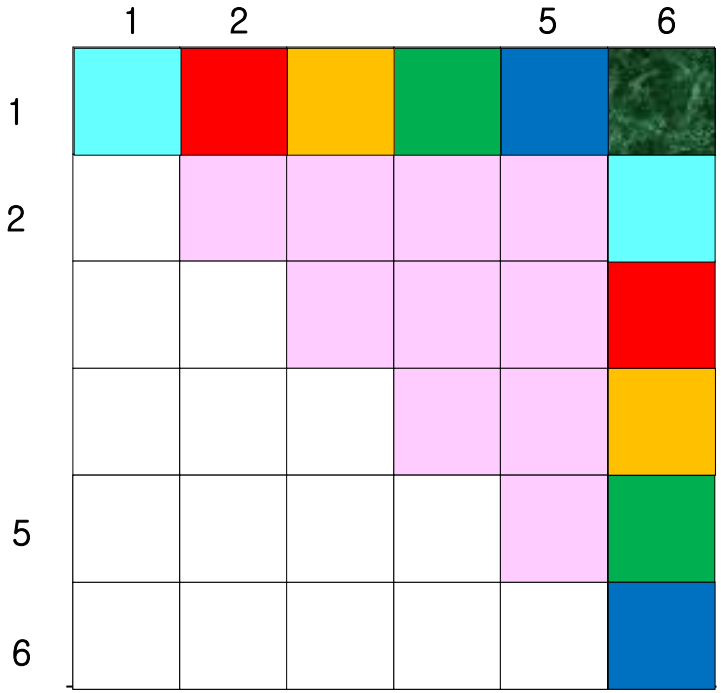
'값'을 구하는 식.

Minimum 값을 구하는 과정.
값 → m[i,j]
위치 → s[i,j]

Computing the optimal costs



Optimal Substructure



[1,6] = [1,5] X [6,6]

[1,6] = [1,4] X [5,6]

[1,6] = [1,3] X [4,6]

[1,6] = [1,2] X [3,6]

[1,6] = [1,1] X [2,6]

Step 4 : Constructing an optimal solution

Recall that $s[i,j]$ records the value of k s.t. the optimal parenthesization of $A_iA_{i+1}...A_j$ splits the product between A_k and A_{k+1}.

PRINT-OPTIMAL-PARENS(s,i,j)

1. if i=j
2. then print "A"$_i$
3. else print "("
4. PRINT-OPTIMAL-PARENS(s, i, $s[i,j]$)
5. PRINT-OPTIMAL-PARENS(s, $s[i,j]$+1, j)
6. print ")"