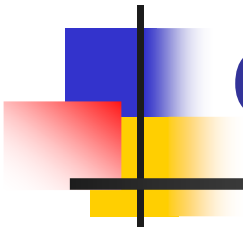
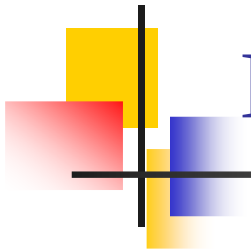


Ch20. Dynamic Programming

SNU Internet DataBase Lab.



BIRD'S-EYE VIEW

- Dynamic programming
  - The most difficult one of the five design methods
  - Has its foundation in the principle of optimality

- Elegant and efficient solutions to many hard problems
  - Knapsack
  - Matrix multiplication chains
  - Shortest-path

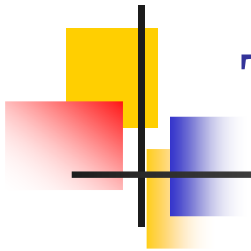


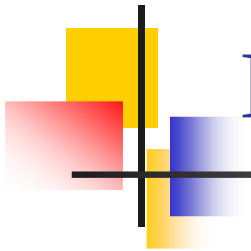
Table of Contents

SNU Internet DataBase Lab.

Dynamic Programming (1)

- The solution to a problem is viewed as the result of a sequence of decisions
- Unlike the greedy method, decisions are not made in a greedy and binding manner
- The principle of optimality
  - No matter what the first decision, the remaining decisions must be optimal with respect to the state that results from this first decision
  - Dynamic programming may be used only when the principle of optimality holds

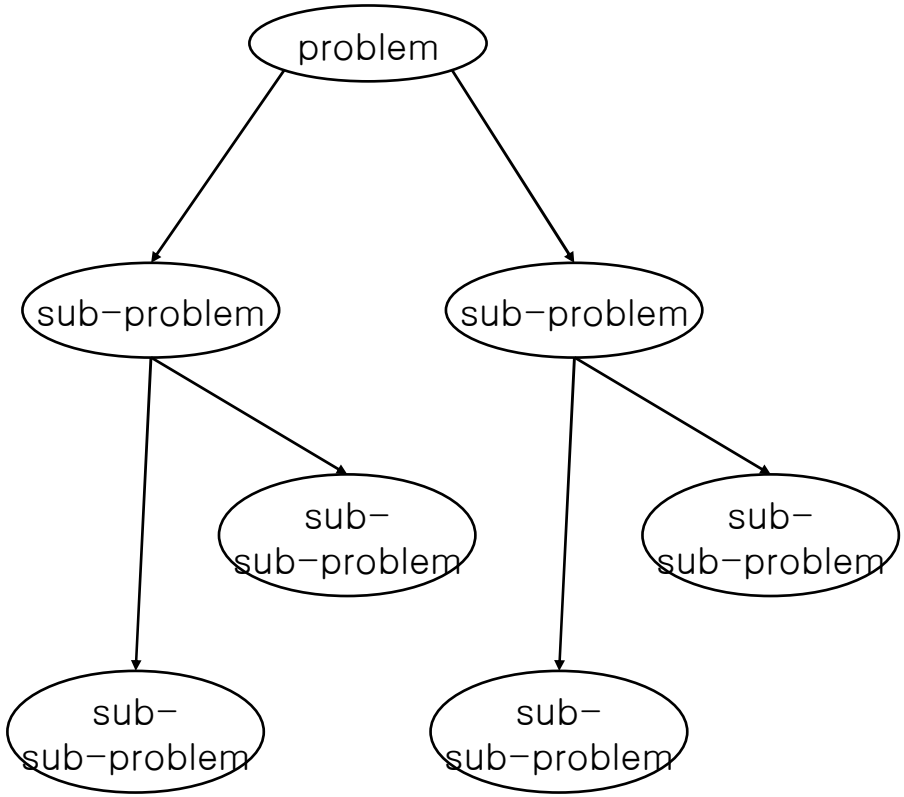
- Divide and Conquer method
  - solves a problem by combining the solutions to sub-problems
    1. Partition the problem into *non-overlapping* sub-problems
    2. Solve the sub-problems recursively
    3. Combine their solutions to solve the original problem
  - does more work than necessary
  - repeatedly solve the common sub-problems

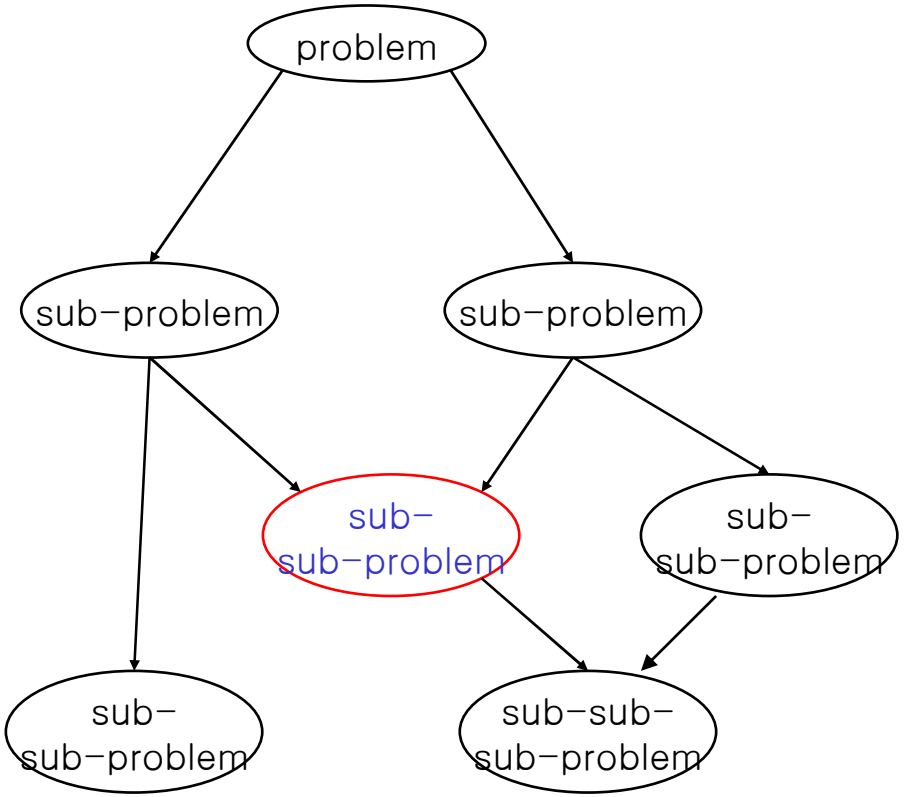


Dynamic Programming (2)

- Dynamic programming method
  - Useful when the sub-problems are *overlapping*
  - Solve an optimization problem by caching sub-problem solutions rather than recomputing them
    1. Verify that the principle of optimality holds
    2. Set up the dynamic-programming recurrence equations
    3. Solve the dynamic-programming recurrence equations for the value of the optimal solution
    4. Perform a traceback step in which the solution itself is constructed

Sub-Problem Structures





Sub-problems are not overlapping

Solve by Divide & Conquer
Ex. Merge sort

Sub-problems are overlapping

Solve by Dynamic Programming
Ex. Matrix multiplication chain

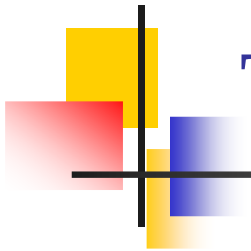


Table of Contents

SNU Internet DataBase Lab.

Matrix Multiplication Chains

- Consider matrix product $M_1 \times M_2 \times M_3 \times \ldots \times M_q$
  - Let the dimensions of M_i be $r_i \times r_{i+1}$
  - q-1 matrix multiplications are to be done.

- An optimal solution minimizes the number of the multiplications

- Consider four matrix multiplication: $A \times B \times C \times D$
  - $A \times ((B \times C) \times D)$
  - $A \times (B \times (C \times D))$
  - $(A \times B) \times (C \times D)$
  - $((A \times B) \times C) \times D$
  - $(A \times (B \times C) \times D)$

SNU Internet DataBase Lab.

Example (1)

- Multiply Matrices A, B, C
  - A:100×1, B:1×100, C:100×1
    - One method ➔ ((A X B) X C)

(A×B) : A : 100 × 1 → 100 × 100 matrix
B : 1 × 100
100 × 1 × 100 = 10000 multiplications

((A×B)×C) : A×B : 100 × 100 → 100 × 1 matrix
C : 100 × 1
100 × 100 × 1 = 10000 multiplications

▶ 10000 + 10000 = 20000 multiplications
10000 units of space to store A × B

Example (2)

- Multiply Matrices A, B, C (Cont.)
  - A:100×1, B:1×100, C:100×1
    - Another Method ➔ (A X (B X C))

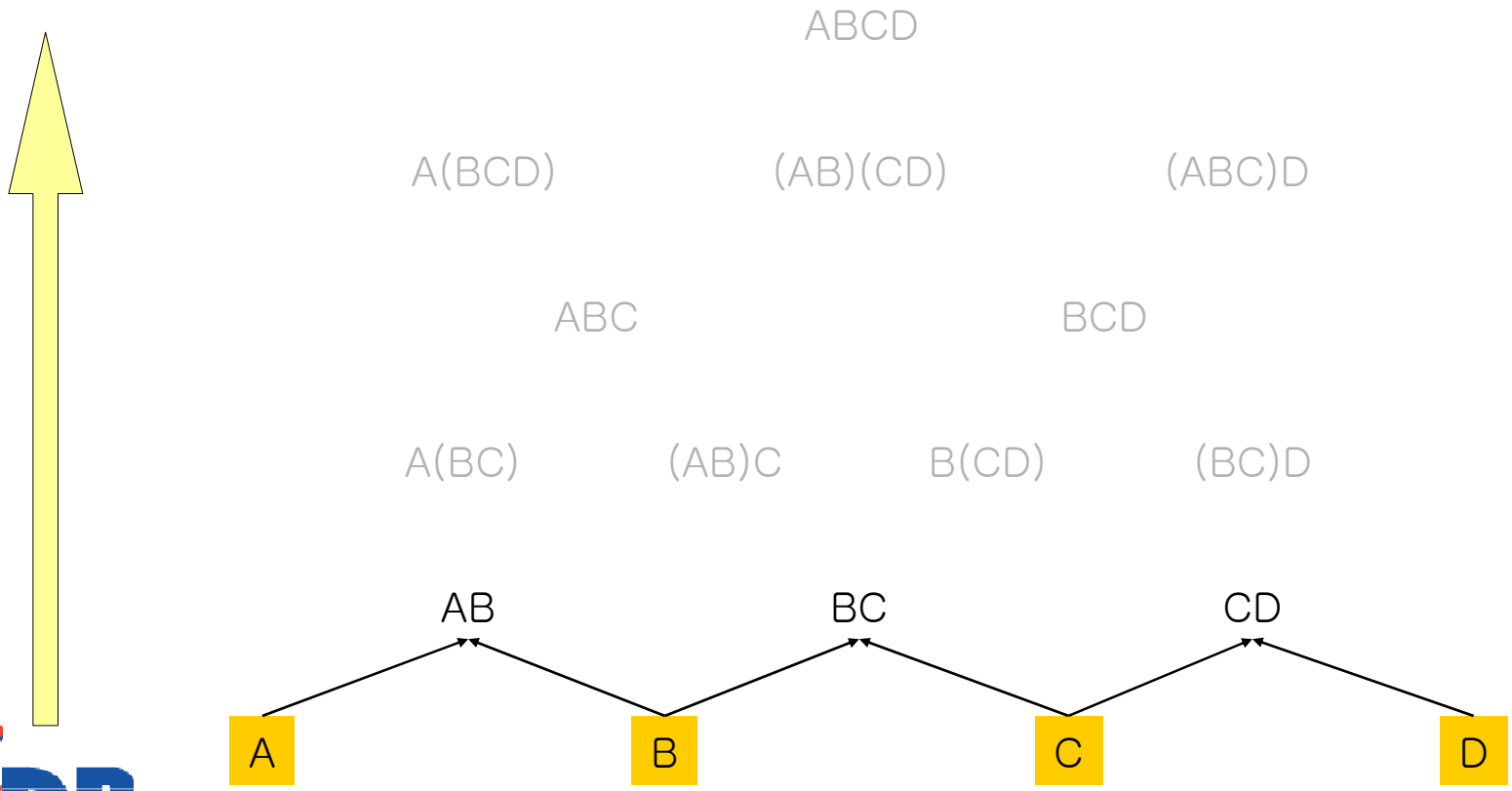
(B×C) : B : 1 × 100 → 1 × 1 matrix
C : 100 × 1
1 × 100 × 1 = 100 multiplications

(A×(B×C) : A : 100 × 1 → 100 × 1 matrix
B×C : 1 × 1
100 × 1 × 1 = 100 multiplications

▶ 100 + 100 = 200 multiplications
1 units of space to store B × C

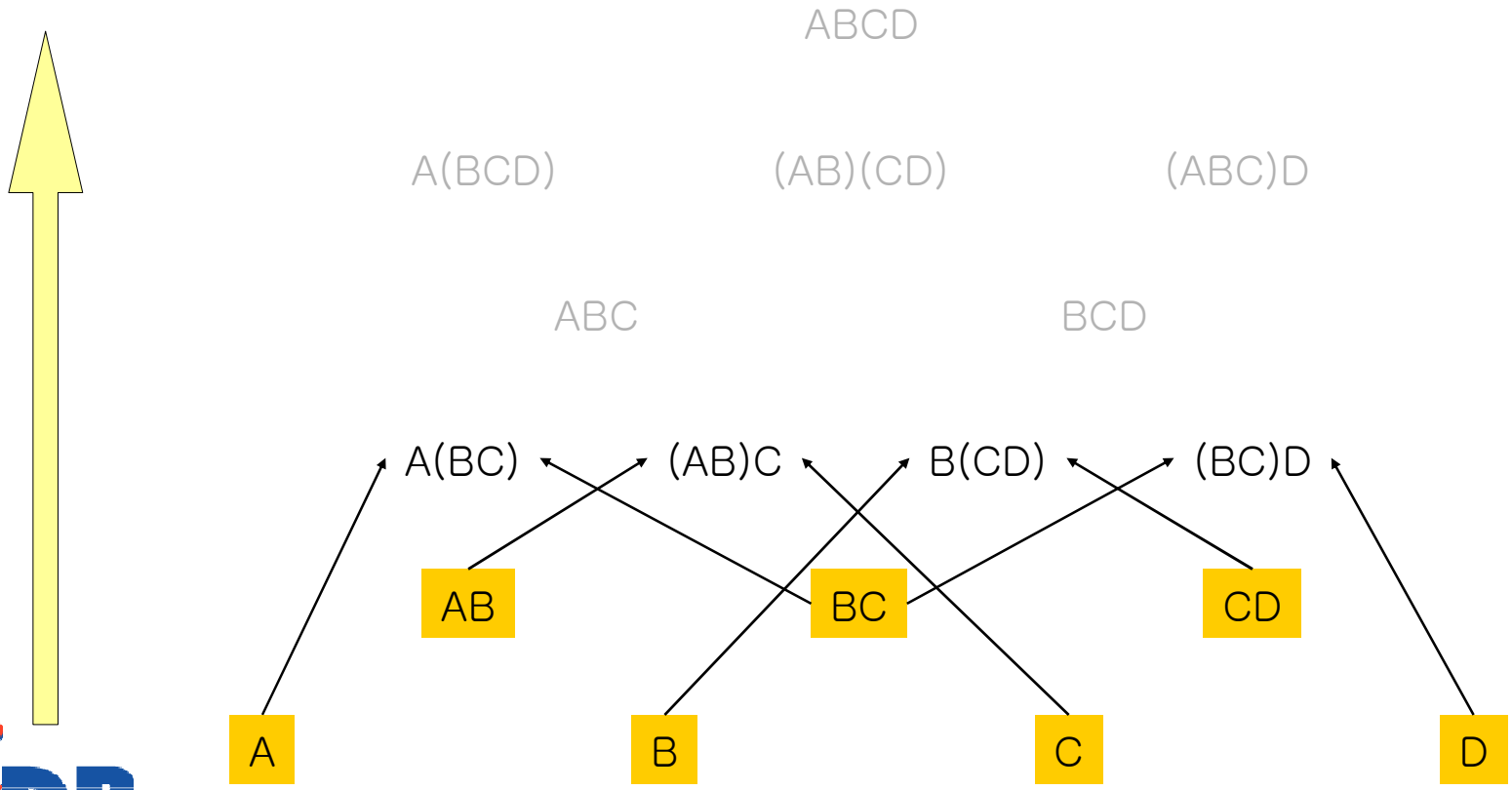
Multiply 4 Matrices: A×B×C×D (1)

- Compute the costs in the bottom-up manner
  - First we consider AB, BC, CD
  - No need to consider AC or BD



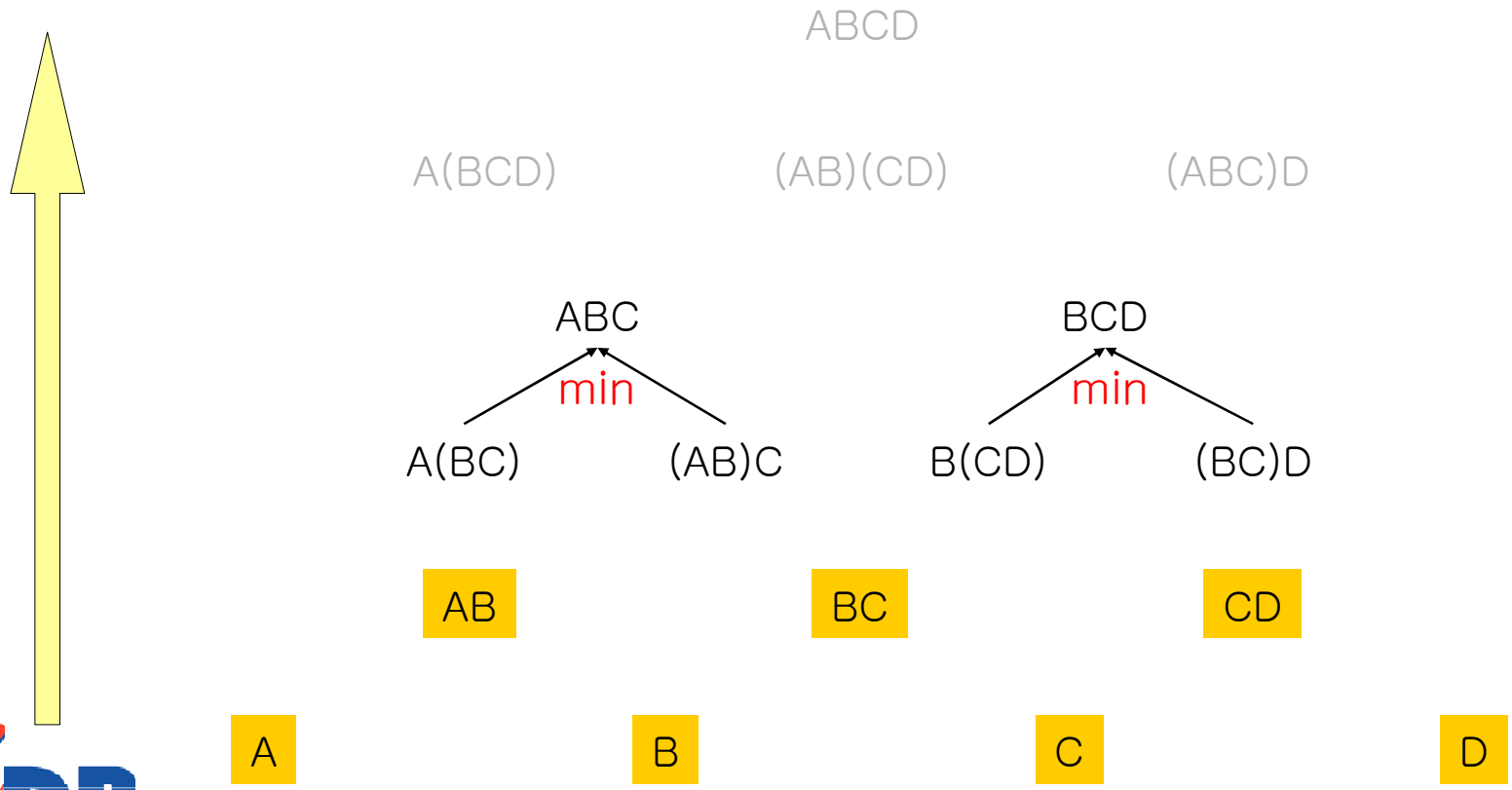
Multiply 4 Matrices: A×B×C×D (2)

- Compute the costs in the bottom-up manner
  - Then we consider A(BC), (AB)C, B(CD), (BC)D
  - No need to consider (AB)D, A(CD)



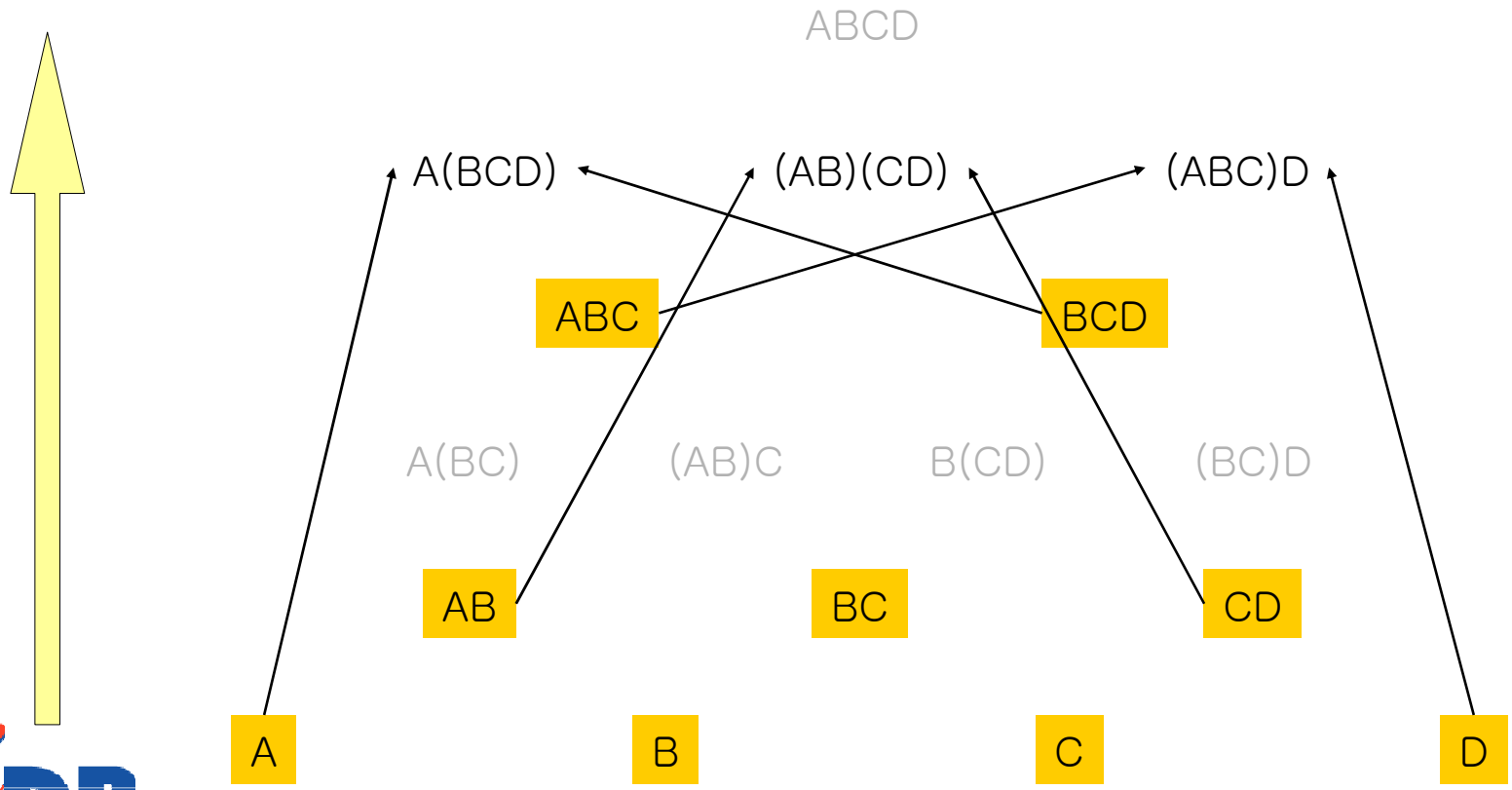
Multiply 4 Matrices: A×B×C×D (3)

- Compute the costs in the bottom-up manner
- Select minimum cost matrix calculations of ABC & BCD



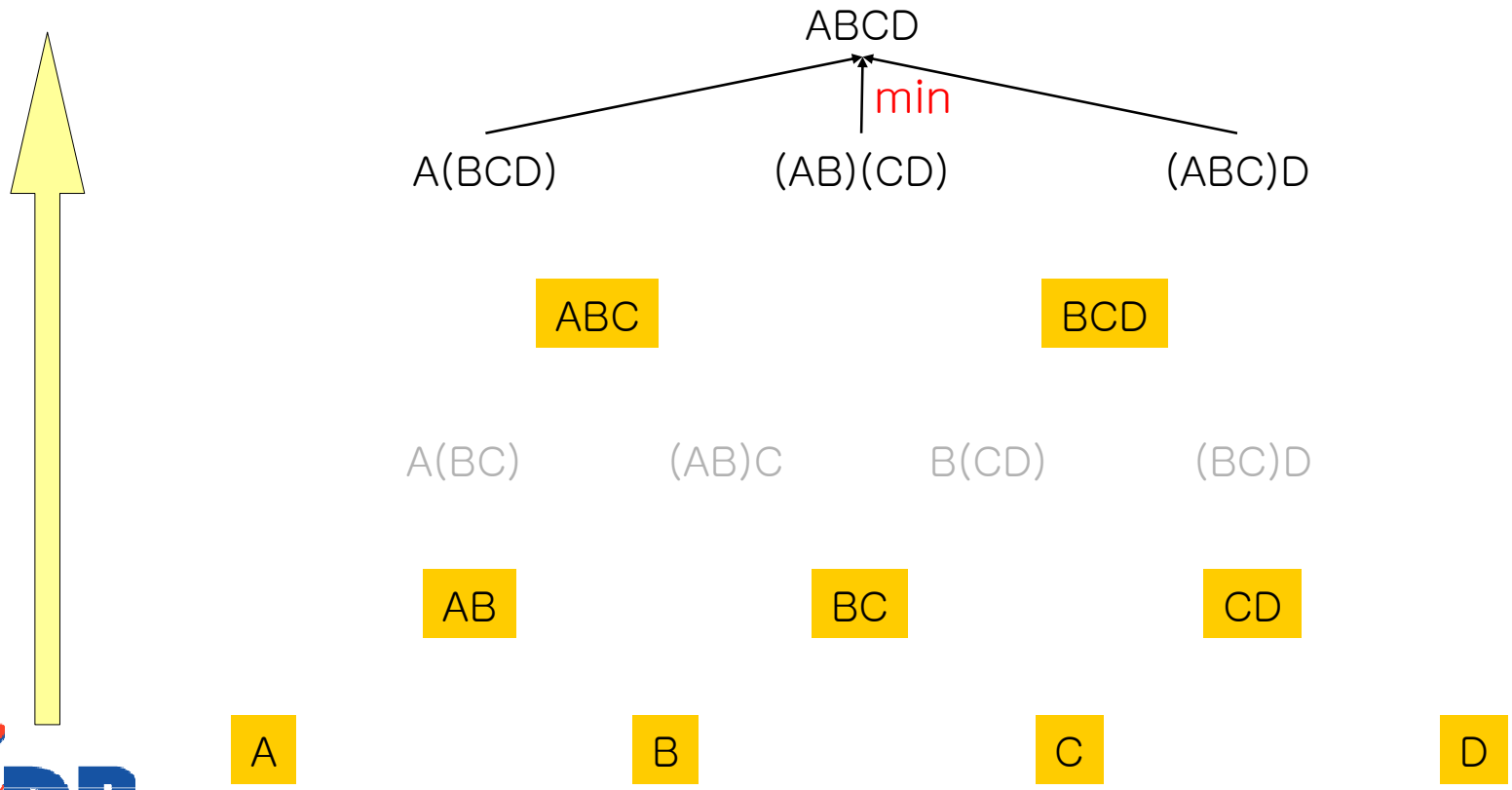
Multiply 4 Matrices: A×B×C×D (4)

- Compute the costs in the bottom-up manner
  - We now consider A(BCD), (AB)(CD), (ABC)D



Multiply 4 Matrices: A×B×C×D (5)

- Compute the costs in the bottom-up manner
  - Finally choose the cheapest cost plan for matrix calculations



Iterative Solution Code

```
private static int c(int i, int j)
{
  // check if already computed
  if (c[i][j] > 0)//c(i,j) was computed earlie
    return c[i][j];
  // c(i,j) not computed before,compute it now
  if (i == j)
    return 0; // one matrix
  if (i == j - 1)
  {// two matrices
    kay[i][i + 1] = i;
    c[i][j] = r[i] * r[i + 1] * r[i + 2];
    return c[i][j];
  }

  // more than two matrices
  // set u to min term for k = i
  int u = c(i,i) + c(i + 1, j) + r[i] *
                  r[i + 1] * r[j + 1];
  kay[i][j] = i;
  // compute remaining min terms and update u
  for (int k = i + 1; k < j; k++)
  {
    int t = c(i, k) + c(k + 1, j) + r[i] *
                    r[k + 1] * r[j + 1];
    if (t < u)
    {// smaller min term found, update u and
     // kay[i][j]
      u = t;
      kay[i][j] = k;
    }
  }

  c[i][j] = u;
  return c[i][j];
}
```

Program 20.6 Computing $c(i,j)$ without recomputations

Running time : $O(q^3)$

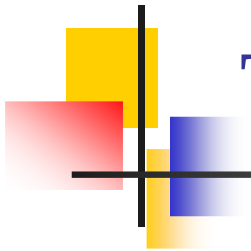
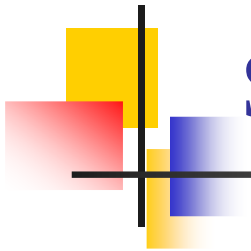


Table of Contents

SNU Internet DataBase Lab.



Shortest Path Problem

- To find a shortest path between two vertices in a directed graph G
- Assumption
  - Permit negative-length edges
  - No negative-length cycles
- Dijkstra's Greedy Single-Source Algorithm (18.3.5): O(n^3)
- Floyd's Dynamic Programming Solution: Θ (n^3)
  - c(i, j, k) denotes the length of a shortest path from i to j that has no intermediate vertex larger than k
    - C(i, j, 0): the length of the edge (i, j) in case this edge is in G
      - 0 (if i = j) or ∞ (otherwise)
    - C(i, j, n): the length of a shortest path from i to j

Floyd's Shortest Path by Dynamic Programming (1):

- c(1, 3, k) = ?
  - ∞ (for k = 0, 1, 2, 3)
  - 28 (for k = 4)
  - 10 (for k = 5, 6, 7)
  - 9 (for k = 8, 9, 10)
- Hence, the shortest 1-to-3 path has length 9

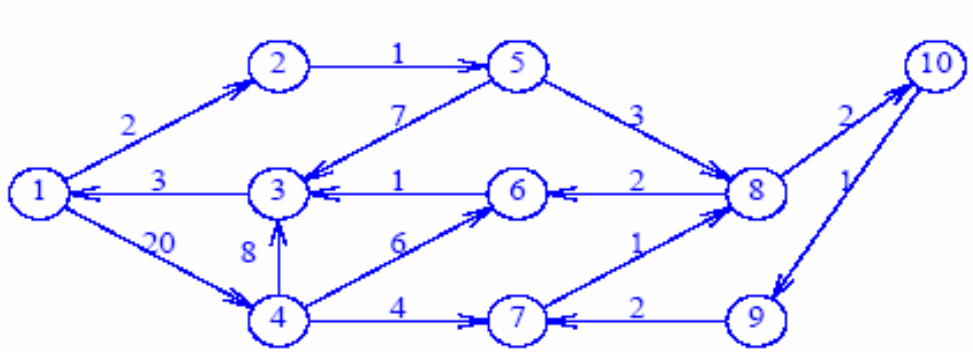


Figure 18.17 A weighted digraph

Floyd Shortest Path Algorithm

- To determine c(i, j, k) for any k (k > 0)
  - Two possibilities
    - The path may not have k as an intermediate vertex
      - c(i, j, k) = c(i, j, k-1)
    - The path may have k as an intermediate vertex
      - c(i, j, k) = c(i, k, k-1) + c(k, j, k-1)
  - So, recurrence equation is
    - c(i, j, k) = min{c(i, j, k-1), c(i, k, k-1) + c(k, j, k-1)}, k>0

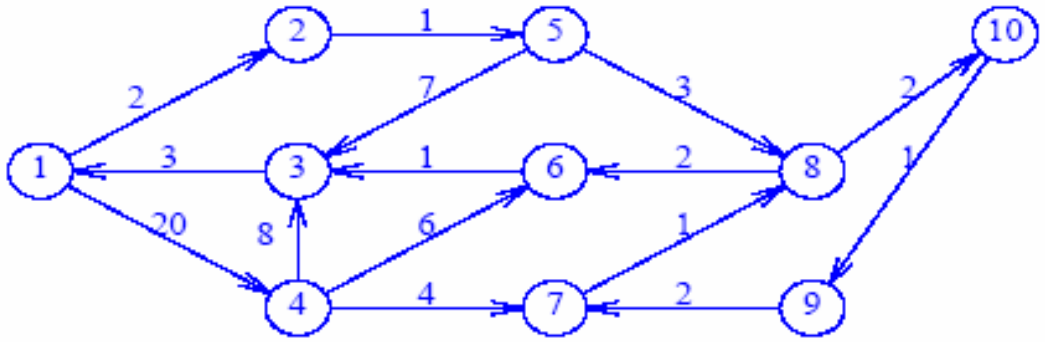


Figure 18.17 A weighted digraph

Floyd Iterative Solution Code for Shortest Path

- The value c(i, j, n) may be obtained efficiently by noticing that some c(i, j, k-1) values get used several times
  - All c values may be determined in Θ (n^3) time

k = 0에 대해서 미리 구해놓음

위에서 구해진 k = 0을 시작으로 k = 1, 2, ..., n 에 대해서 모든 c(i, j, k) 값을 구해 올라감

```
// Find the lengths of the shortest paths.
  // initialize c(i,j,0)
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
       // a is the cost-adjacency matrix
      c(i,j,0) = a(i,j);
  // compute c(i,j,k) for 0 < k <= n
  for (int k = 1; k <= n; k++)
    for (int i = 1; i <= n; i++)
      for (int j = 1; j <= n; j++)
        if (c(i,k,k-1) + c(k,j,k-1)
                       < c(i,j,k-1))
          c(i,j,k) = c(i,k,k-1)+c(k,j,k-1);
        else
          c(i,j,k) = c(i,j,k-1);
```

Figure 20.3 Initial shortest-paths algorithm

An Example for Floyd's Shortest Path (1)

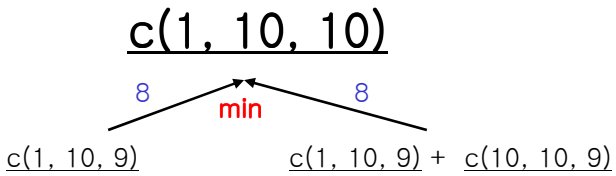
- **To determine c(1, 10, 10) = 8**

c(1, 10, 10)

$$c(i, j, k) = \min\{c(i, j, k-1), c(i, k, k-1) + c(k, j, k-1)\}$$

An Example for Floyd's Shortest Path (2)

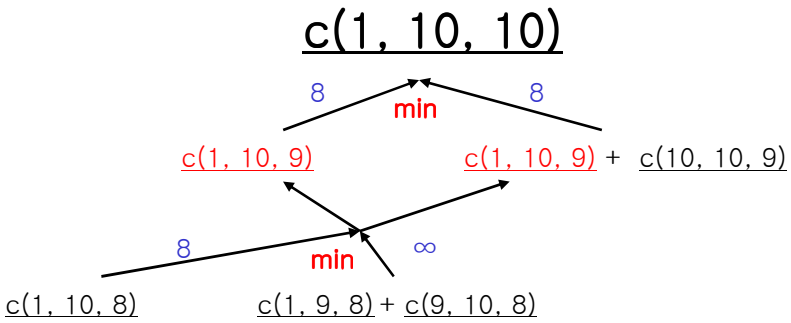
- **To determine c(1, 10, 10) = 8**



$$c(i, j, k) = \min\{c(i, j, k-1), c(i, k, k-1) + c(k, j, k-1)\}$$

An Example for Floyd's Shortest Path (3)

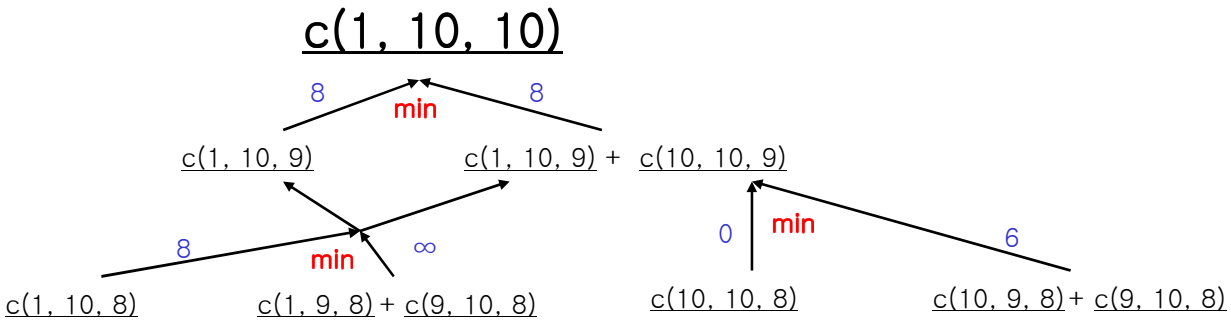
- **To determine c(1, 10, 10) = 8**



$$c(i, j, k) = \min\{c(i, j, k-1), c(i, k, k-1) + c(k, j, k-1)\}$$

An Example for Floyd's Shortest Path (4)

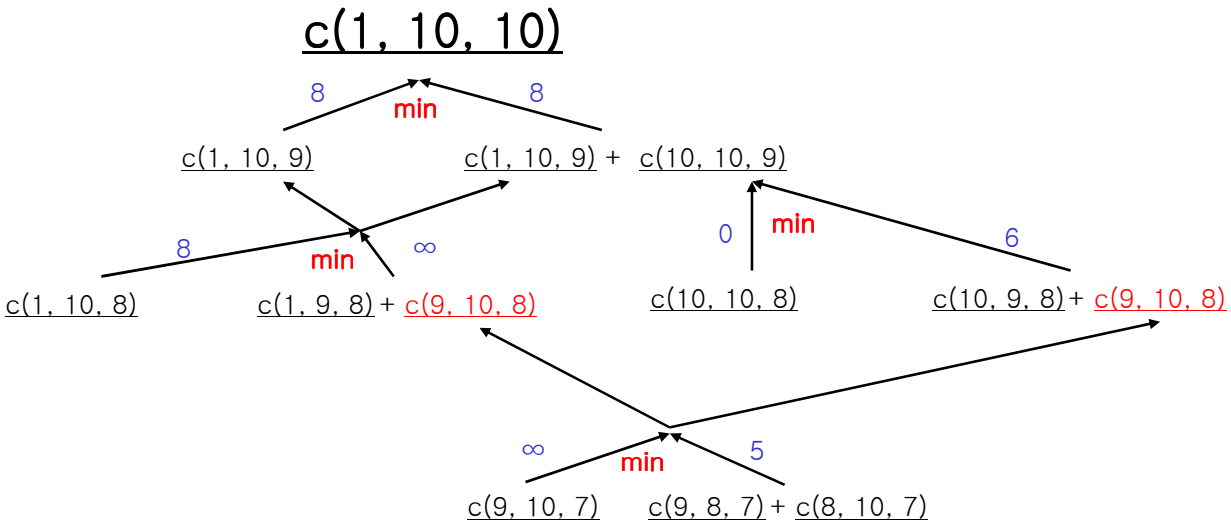
- **To determine c(1, 10, 10) = 8**



$$c(i, j, k) = \min\{c(i, j, k-1), c(i, k, k-1) + c(k, j, k-1)\}$$

An Example for Floyd's Shortest Path (5)

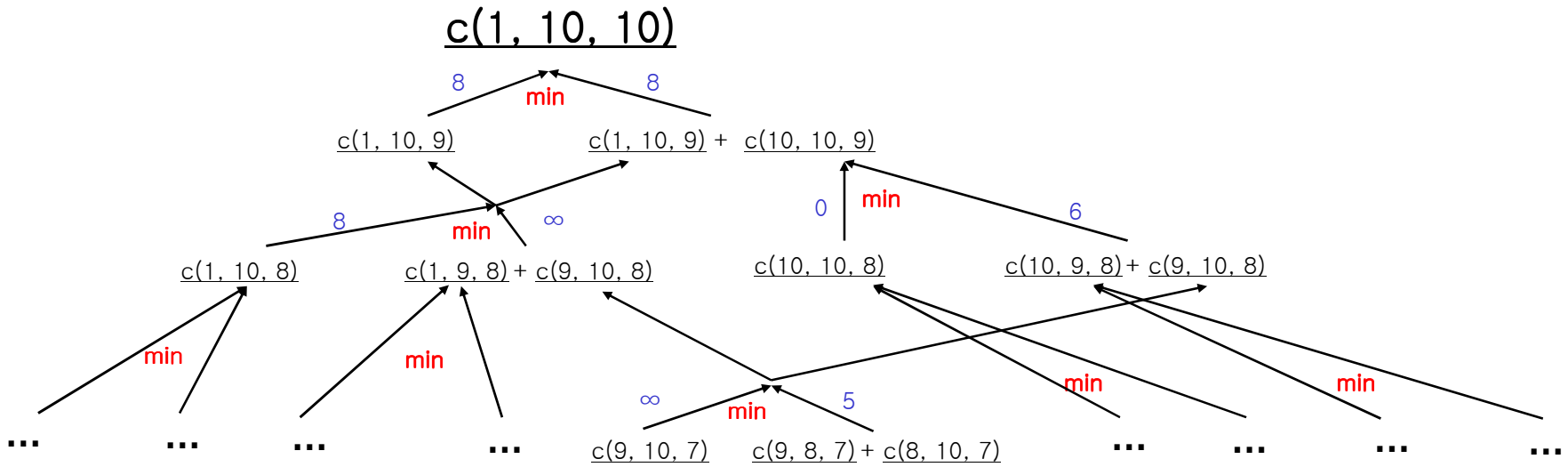
- **To determine c(1, 10, 10) = 8**



$$c(i, j, k) = \min\{c(i, j, k-1), c(i, k, k-1) + c(k, j, k-1)\}$$

An Example for Floyd's Shortest Path (6)

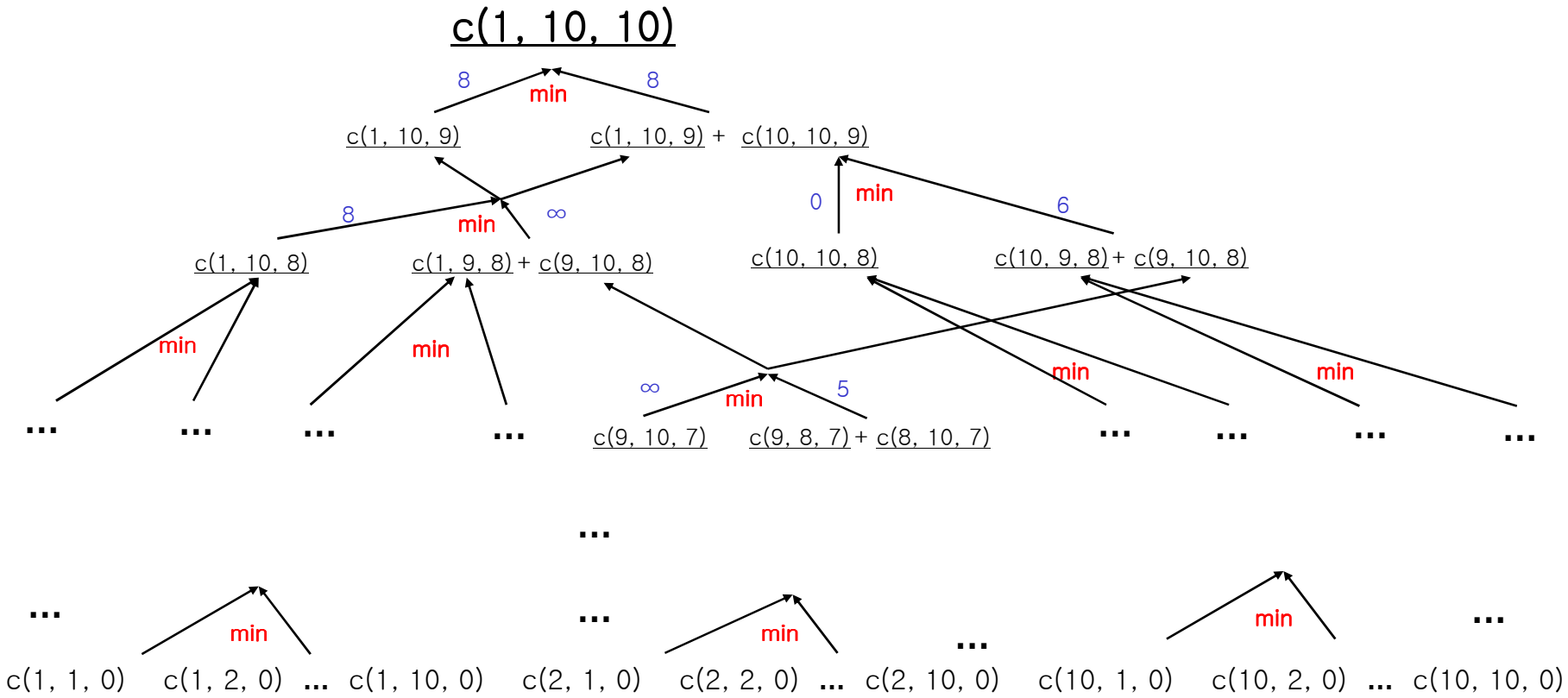
- **To determine c(1, 10, 10) = 8**



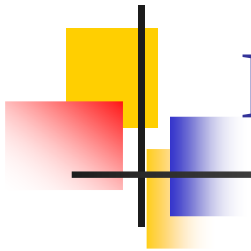
$$c(i, j, k) = \min\{c(i, j, k-1), c(i, k, k-1) + c(k, j, k-1)\}$$

An Example for Floyd's Shortest Path (7)

- **To determine c(1, 10, 10) = 8**



$$c(i, j, k) = \min\{c(i, j, k-1), c(i, k, k-1) + c(k, j, k-1)\}$$



BIRD'S-EYE VIEW

- Dynamic programming
  - Is the most difficult one of the five design methods
  - Has its foundation in the principle of optimality

- Elegant and efficient solutions to many problems
  - Knapsack
  - Matrix multiplication chains
  - Shortest-path

- 참고사항

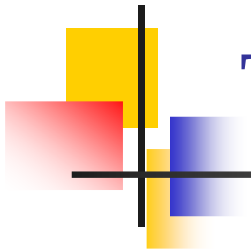


Table of Contents

SNU Internet DataBase Lab.

Matrix Multiplication Chains

- Consider matrix product $M_1 \times M_2 \times M_3 \times \ldots \times M_q$
  - Let the dimensions of M_i be $r_i \times r_{i+1}$
  - q-1 matrix multiplications are to be done.

- An optimal solution minimizes the number of the multiplications

- Consider four matrix multiplication: $A \times B \times C \times D$
  - $A \times ((B \times C) \times D)$
  - $A \times (B \times (C \times D))$
  - $(A \times B) \times (C \times D)$
  - $((A \times B) \times C) \times D$
  - $(A \times (B \times C) \times D)$

SNU Internet DataBase Lab.

Example (1)

- Multiply Matrices A, B, C
  - A:100×1, B:1×100, C:100×1
    - One method ➔ ((A X B) X C)

(A×B) : A : 100 × 1 → 100 × 100 matrix
B : 1 × 100
100 × 1 × 100 = 10000 multiplications

((A×B)×C) : A×B : 100 × 100 → 100 × 1 matrix
C : 100 × 1
100 × 100 × 1 = 10000 multiplications

▶ 10000 + 10000 = 20000 multiplications
10000 units of space to store A × B

Example (2)

- Multiply Matrices A, B, C (Cont.)
  - A:100×1, B:1×100, C:100×1
    - Another Method ➔ (A X (B X C))

(B×C) : B : 1 × 100 → 1 × 1 matrix
C : 100 × 1
1 × 100 × 1 = 100 multiplications

(A×(B×C) : A : 100 × 1 → 100 × 1 matrix
B×C : 1 × 1
100 × 1 × 1 = 100 multiplications

▶ 100 + 100 = 200 multiplications
1 units of space to store B × C

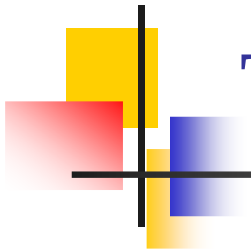


Table of Contents

Verify Principle of Optimality

- Consider $M_{ij} \leftarrow M_i \times M_{i+1} \times \ldots \times M_j$, $i \leq j$
- An optimal solution of M_{ij} can be obtained by combining optimal solutions of two matrices M_{ik} and $M_{k+1,j}$, $i \leq k < j$.
- Let OP(i,j) be an optimal solution of M_{ij}, then
  - $M_{ij} \leftarrow (M_i \times M_{i+1} \times \ldots \times M_k) \times (M_{k+1} \times M_{k+2} \times \ldots \times M_j)$
    $\leftarrow M_{ik} \times M_{(k+1)j}$
  - $OP(i,j) = OP(i,k) + OP(k+1,j) + r_i r_{k+1} r_{j+1}$
    - OP(i,k): an optimal solution to $(M_i \times M_{i+1} \times \ldots \times M_k)$
    - OP(k+1,j): an optimal solution to $(M_{k+1} \times M_{k+2} \times \ldots \times M_j)$
    - $r_i\, r_{k+1}\, r_{j+1}$: the # of multiplications of $M_{ik} \times M_{(k+1)j}$
- Suppose there is OP'(i,k) < OP(i,k) for a subchain M_{ik}, then
  - $OP'(i,j) = OP'(i,k) + OP(k+1,j) + r_i r_{k+1} r_{i+s+1} < OP(i,j)$
  - Contradiction because this violates our original hypothesis that OP(i,j) is an optimal solution of M_{ij}
- Therefore, the principle of optimality holds

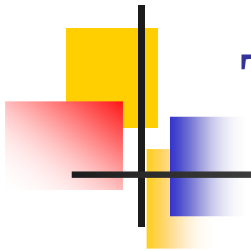


Table of Contents

SNU Internet DataBase Lab.

Recurrence Equations

- $M_{ij} \leftarrow M_i \times M_{i+1} \times \ldots \times M_j$, $i \leq j$
- $c(i,j) \leftarrow$ cost of an optimal way to compute M_{ij}
- kay(i,j) = k be such that the optimal computation of M_{ij} computes $M_{ik} \times M_{k+1,j}$

$$c(i, i) = 0, \quad 1 \leq i \leq q \quad (M_{ii} = M_i)$$

$$c(i, i+1) = r_i r_{i+1} r_{i+2}, \quad 1 \leq i < q \quad (M_{ii+1} = M_i \times M_{i+1})$$

$$c(i, i+s) = \min_{i \leq k < i+s} \{ c(i, k) + c(k+1, i+s) + r_i r_{k+1} r_{i+s+1} \}$$

kay(i,i+s) is the value of k that yields the above min.

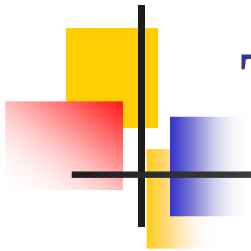
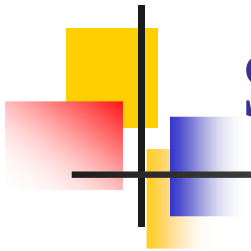


Table of Contents

SNU Internet DataBase Lab.



Solution Approaches

- Recursive
  - Simple, intuitive code
  - Unless care is taken to avoid recomputing previously computed values, the recursive program will have prohibitive complexity

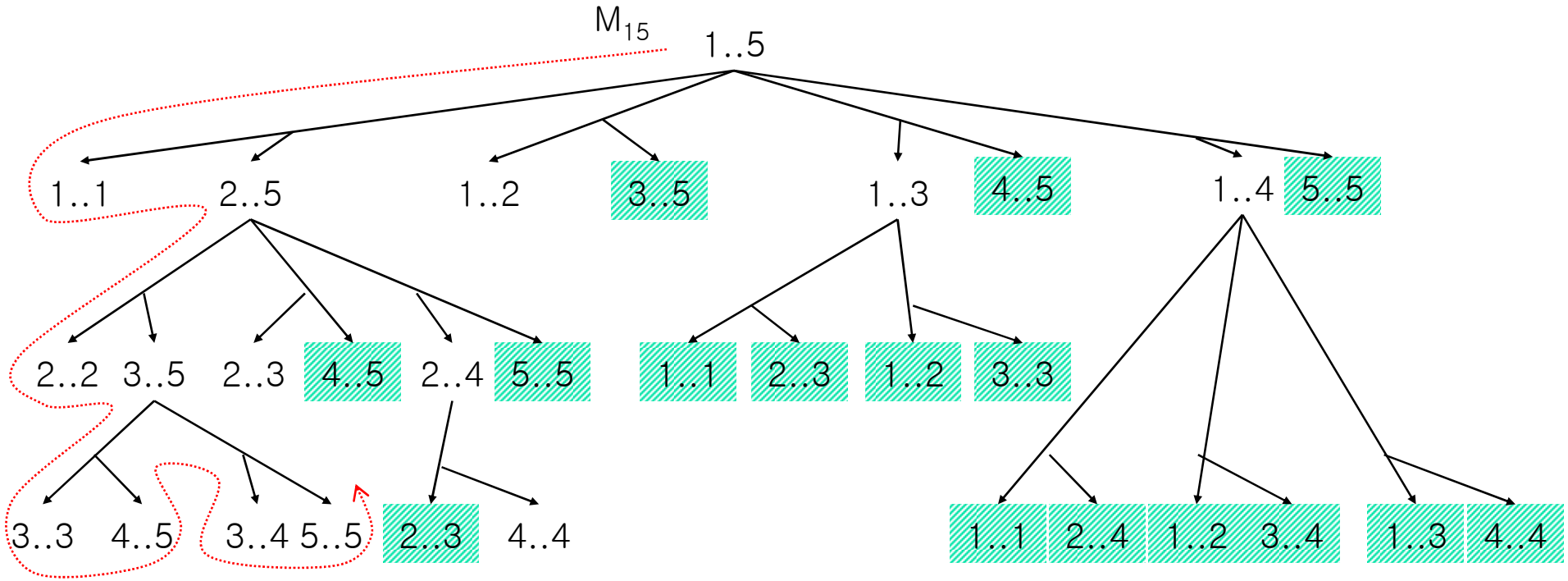
- Iterative
  - Not require additional space for the recursion stack
  - To minimize run time overheads, and hence to reduce actual run time, dynamic programming recurrences are almost always solved iteratively (no recursion).

Recursive Solution (1)

```
/**return c(i,j) and compute kay[i][j]=kay(i,j)*/
private static int c(int i, int j)
{
  if (i == j) return 0; // one matrix
  if (i == j - 1)
  {// two matrices
    kay[i][i + 1] = i;
    return r[i] * r[i + 1] * r[i + 2];
  }
  // more than two matrices
  // set u to min term for k = i
  int u = c(i,i)+c(i+1,j)+r[i]*r[i+1]*r[j+1];
  kay[i][j] = i;
  //compute remaining min terms and update u
  for (int k = i + 1; k < j; k++)
  {
    int t = c(i,k)+c(k+1,j)+r[i]*r[k+1]*r[j+1];
    if (t < u)
    {// smaller min term found, update u
     // and kay[i][j]
      u = t;
      kay[i][j] = k;
    }
  }
  return u;
}
```

Running time : $\Omega(2^n)$

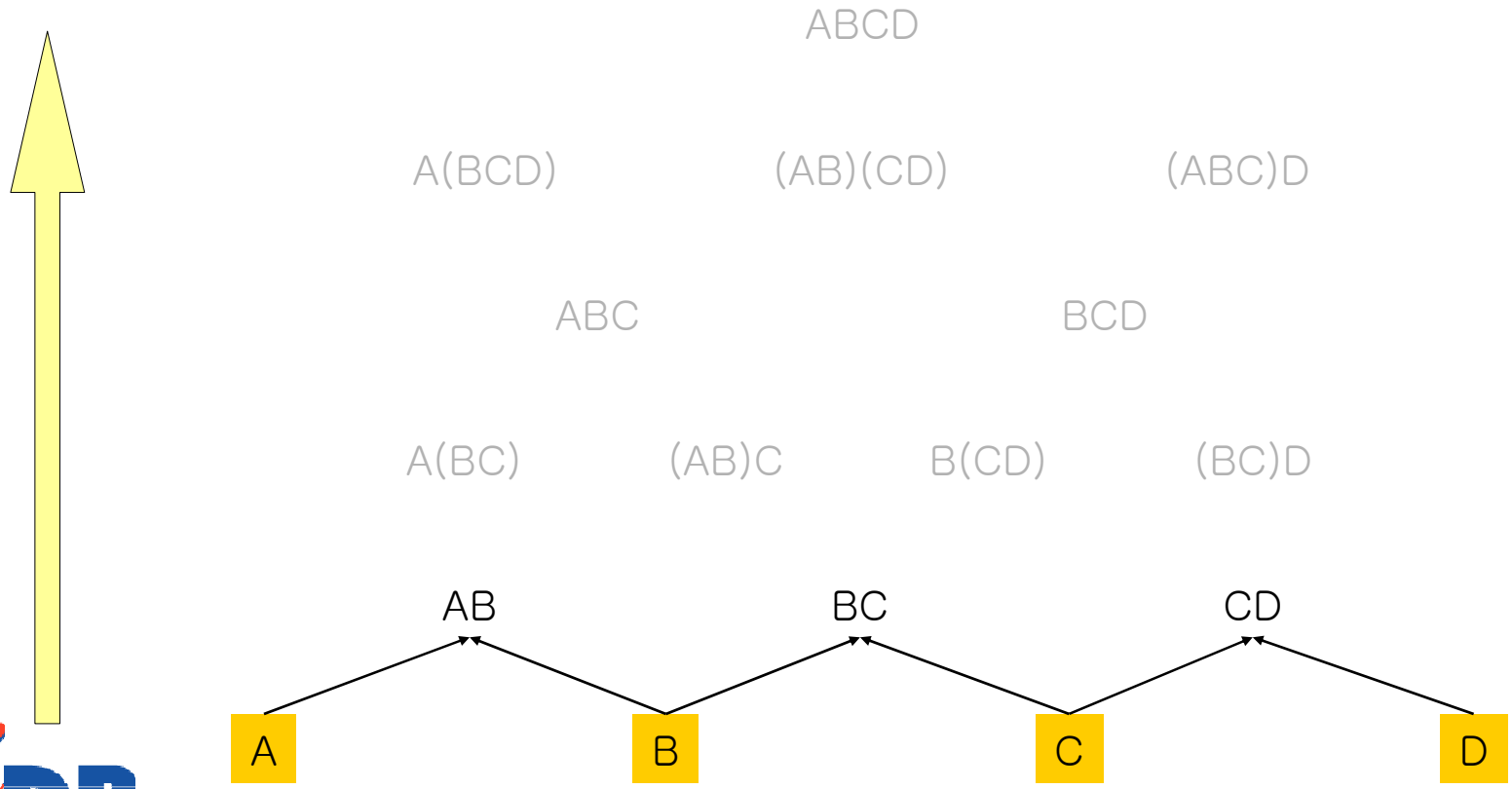
Recursive Solution (2)



It has many overlappings (recomputations)

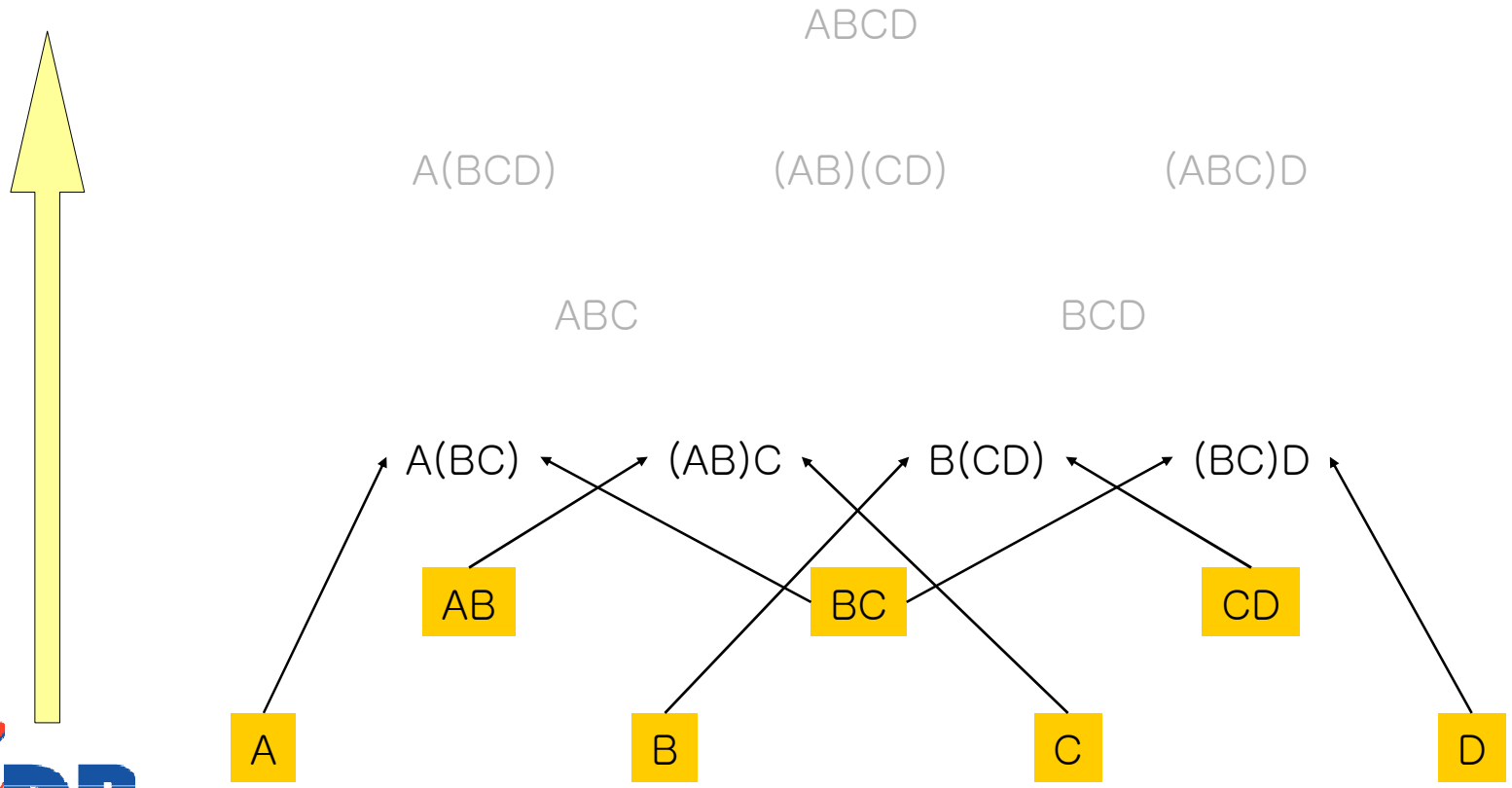
Iterative Solution (1)

- Compute the costs in the bottom-up manner
  - A×B×C×D



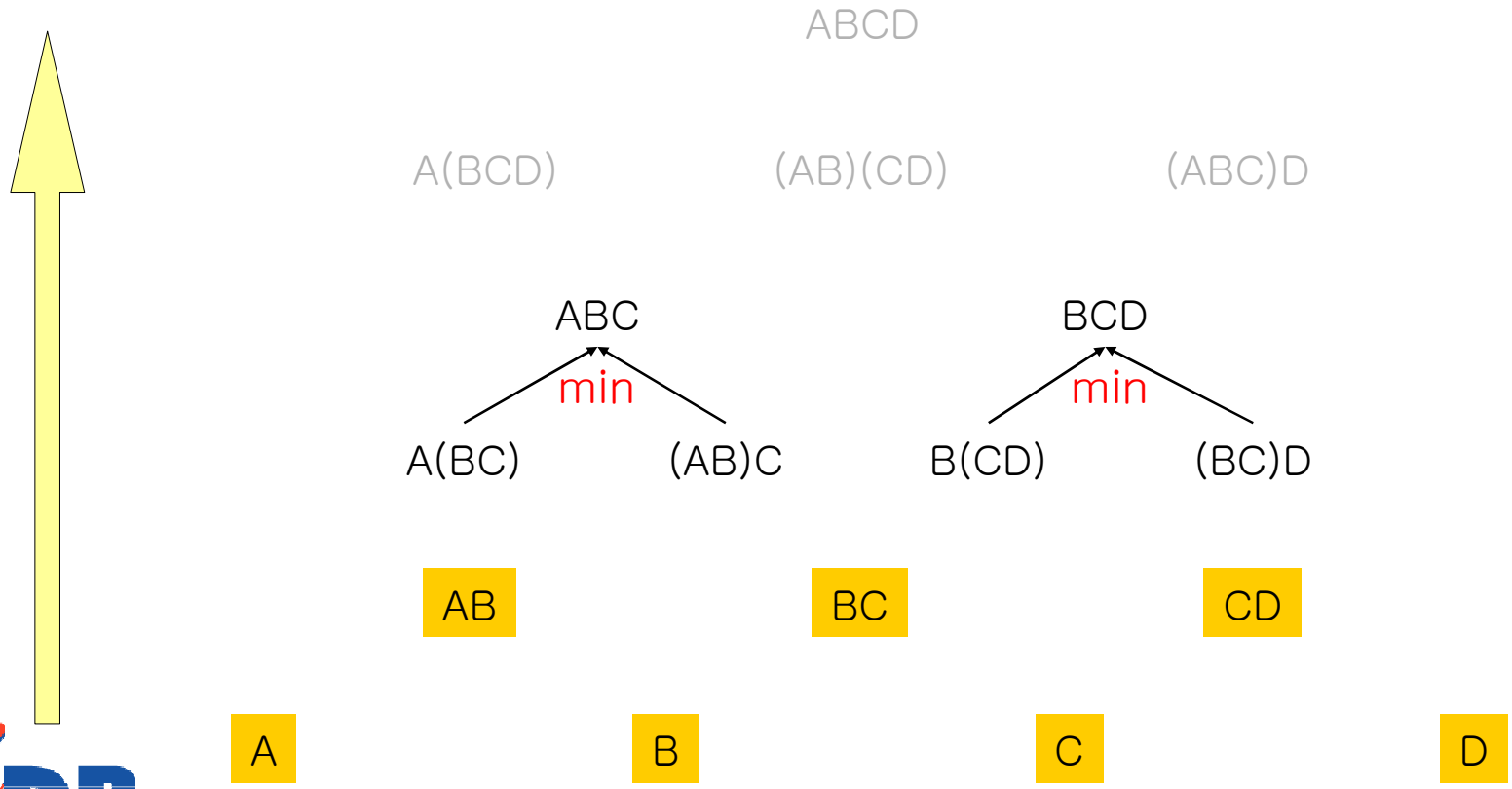
Iterative Solution (2)

- Compute the costs in the bottom-up manner
  - A×B×C×D



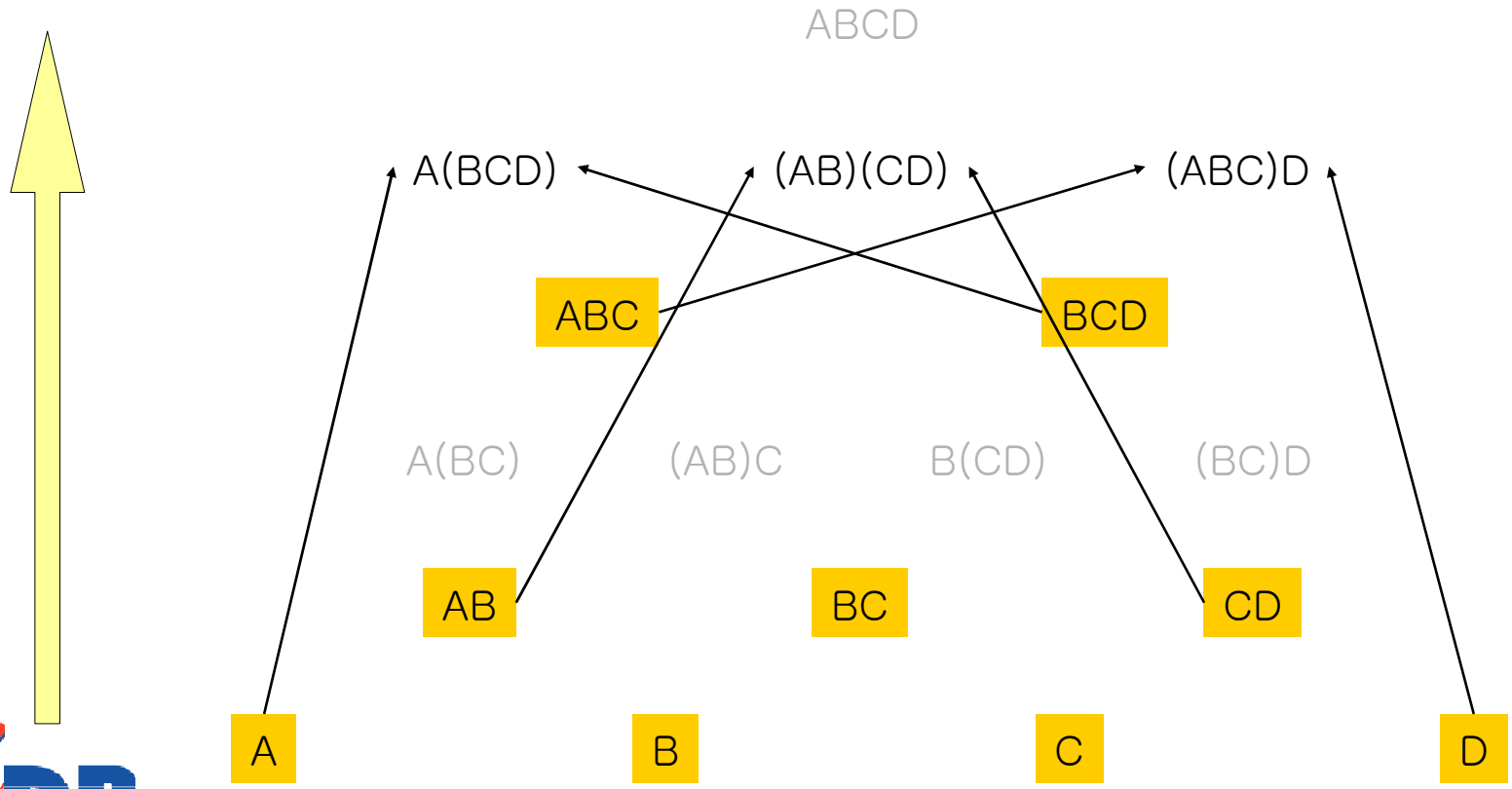
Iterative Solution (3)

- Compute the costs in the bottom-up manner
  - A×B×C×D



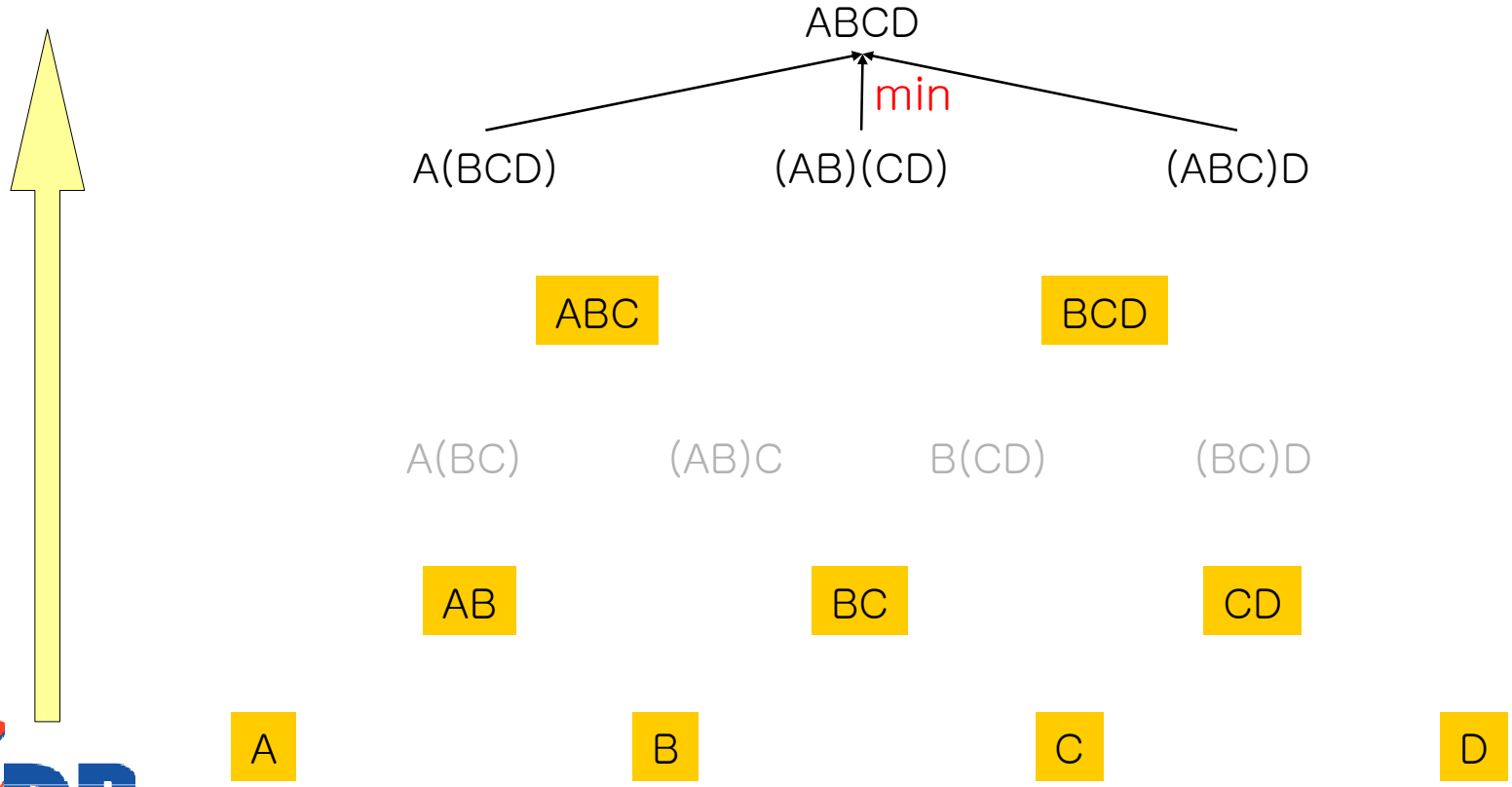
Iterative Solution (4)

- Compute the costs in the bottom-up manner
  - A×B×C×D



Iterative Solution (5)

- Compute the costs in the bottom-up manner
  - A×B×C×D



Iterative Solution (6)

```
private static int c(int i, int j)
{
  // check if already computed
  if (c[i][j] > 0)//c(i,j) was computed earlie
    return c[i][j];
  // c(i,j) not computed before,compute it now
  if (i == j)
    return 0; // one matrix
  if (i == j - 1)
  {// two matrices
    kay[i][i + 1] = i;
    c[i][j] = r[i] * r[i + 1] * r[i + 2];
    return c[i][j];
  }

  // more than two matrices
  // set u to min term for k = i
  int u = c(i,i) + c(i + 1, j) + r[i] *
                  r[i + 1] * r[j + 1];
  kay[i][j] = i;
  // compute remaining min terms and update u
  for (int k = i + 1; k < j; k++)
  {
    int t = c(i, k) + c(k + 1, j) + r[i] *
                    r[k + 1] * r[j + 1];
    if (t < u)
    {// smaller min term found, update u and
     // kay[i][j]
      u = t;
      kay[i][j] = k;
    }
  }

  c[i][j] = u;
  return c[i][j];
}
```

Program 20.6 Computing $c(i,j)$ without recomputations

Running time : $O(q^3)$

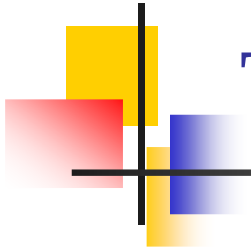
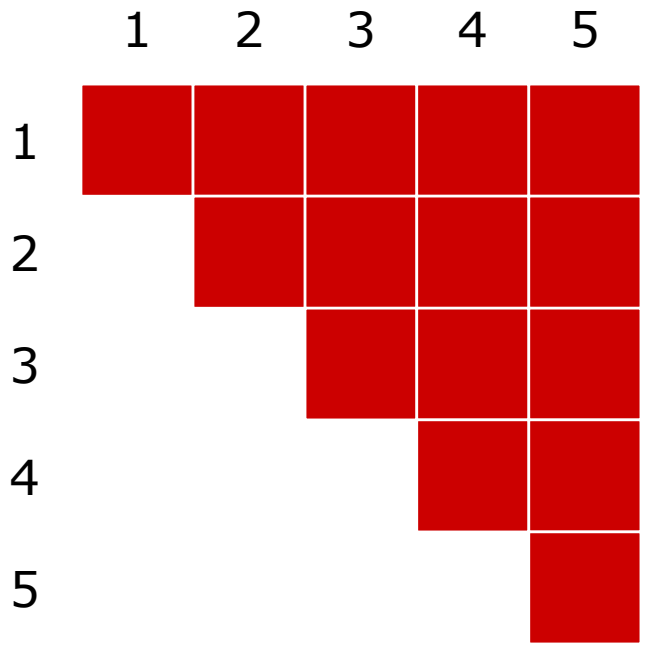


Table of Contents

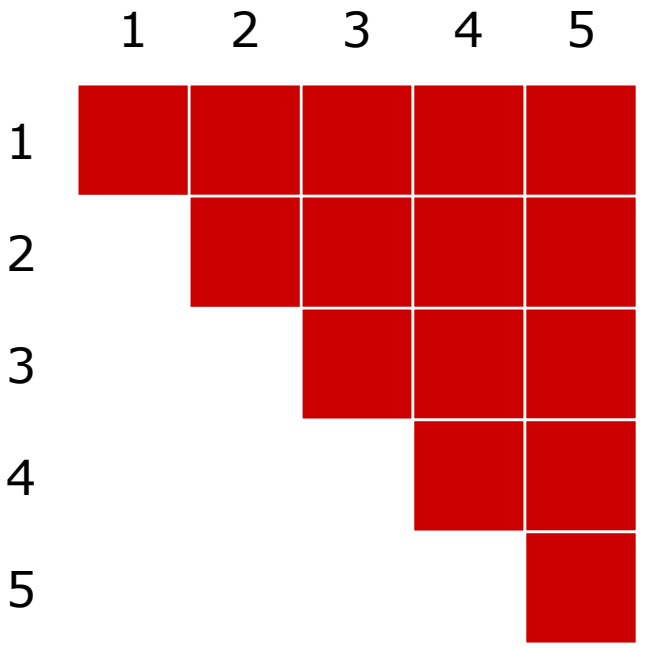
SNU Internet DataBase Lab.

Example 20.11 (1)

- q = 5, r = (10, 5, 1, 10, 2, 10)
  - [10×5]×[5×1]×[1×10]×[10×2]×[2×10]



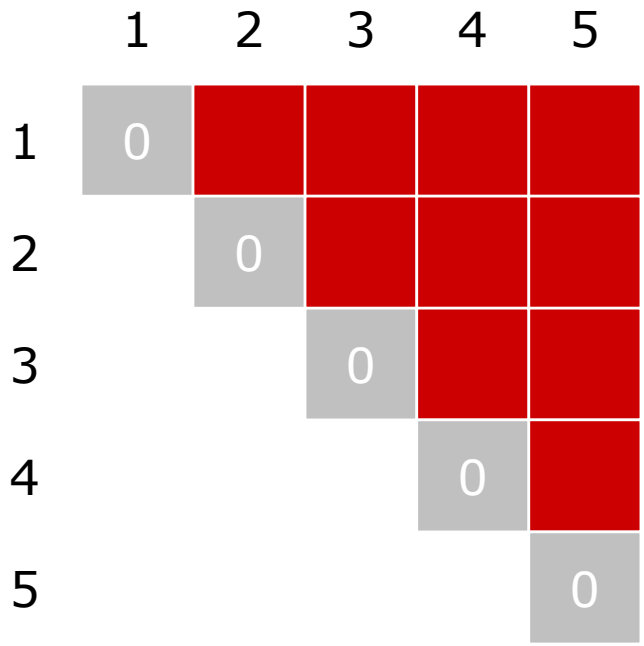
c(i,j), i ≤ j



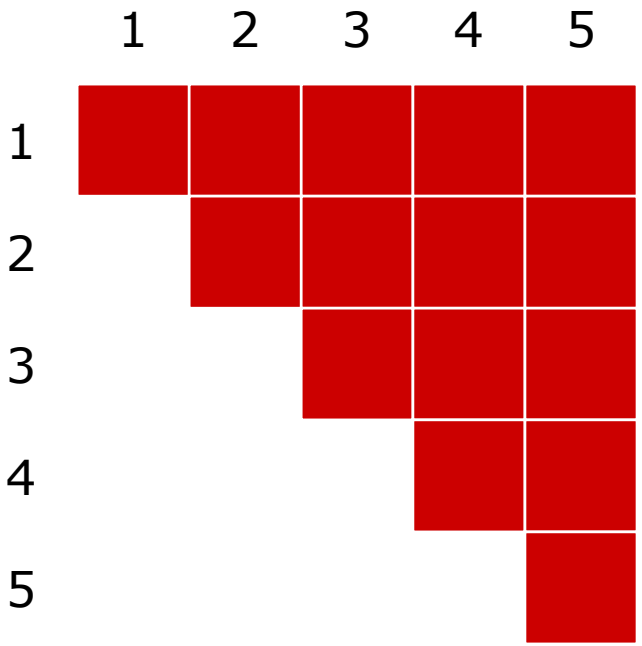
kay(i,j), i ≤ j

Example 20.11 (2)

- s = 0, [10×5]×[5×1]×[1×10]×[10×2]×[2×10]
- c(i,i) = 0



c(i,j), i ≤ j



kay(i,j), i ≤ j

Example 20.11 (3)

- s = 0, $[10\times5]\times[5\times1]\times[1\times10]\times[10\times2]\times[2\times10]$
- $c(i,i+1) = r_i r_{i+1} r_{i+2}$
- kay(i,i+1) = i

	1	2	3	4	5
1	0	50			
2		0	50		
3			0	20	
4				0	200
5					0

c(i,j), i ≤ j

	1	2	3	4	5
1		1			
2			2		
3				3	
4					4
5					

kay(i,j), i ≤ j

Example 20.11 (4)

- s = 1, $[10\times5]\times[5\times1]\times[1\times10]\times[10\times2]\times[2\times10]$
- $c(i,i+2) = \min\{c(i,i) + c(i+1,i+2) + r_i r_{i+1} r_{i+3},$
  $c(i,i+1) + c(i+2,i+2) + r_i r_{i+2} r_{i+3}\}$

	1	2	3	4	5
1	0	50			
2		0	50		
3			0	20	
4				0	200
5					0

c(i,j), i ≤ j

	1	2	3	4	5
1		1			
2			2		
3				3	
4					4
5					

kay(i,j), i ≤ j

SNU Internet DataBase Lab.

Example 20.11 (5)

- s = 1, $[10\times5]\times[5\times1]\times[1\times10]\times[10\times2]\times[2\times10]$
- $c(1,3) = \min\{c(1,1) + c(2,3) + r_1r_2r_4,\ \underline{c(1,2) + c(3,3) + r_1r_3r_4}\}$
  $= \min\{0+50+500, \underline{50+0+100}\} = 150$

	1	2	3	4	5
1	0	50	150		
2		0	50		
3			0	20	
4				0	200
5					0

c(i,j), i ≤ j

	1	2	3	4	5
1		1	2		
2			2		
3				3	
4					4
5					

kay(i,j), i ≤ j

Example 20.11 (6)

- s = 1, $[10\times5]\times[5\times1]\times[1\times10]\times[10\times2]\times[2\times10]$
- $c(2,4) = \min\{c(2,2) + c(3,4) + r_2r_3r_5,\ \ c(2,3) + c(4,4) + r_2r_4r_5\}$
- $c(3,5) = \ldots$

	1	2	3	4	5
1	0	50	150		
2		0	50	30	
3			0	20	40
4				0	200
5					0

c(i,j), i ≤ j

	1	2	3	4	5
1		1	2		
2			2	2	
3				3	3
4					4
5					

kay(i,j), i ≤ j

Example 20.11 (7)

- s = 2, $[10\times5]\times[5\times1]\times[1\times10]\times[10\times2]\times[2\times10]$
- $c(i,i+3) = \min\{c(i,i) + c(i+1,i+3) + r_i r_{i+1} r_{i+4},$
  $c(i,i+1) + c(i+2,i+3) + r_i r_{i+2} r_{i+4},$
  $c(i,i+2) + c(i+3,i+3) + r_i r_{i+3} r_{i+4}\}$

	1	2	3	4	5
1	0	50	150	90	
2		0	50	30	90
3			0	20	40
4				0	200
5					0

c(i,j), i ≤ j

	1	2	3	4	5
1		1	2	2	
2			2	2	2
3				3	3
4					4
5					

kay(i,j), i ≤ j

SNU Internet DataBase Lab.

Example 20.11 (8)

- s = 3, $[10\times5]\times[5\times1]\times[1\times10]\times[10\times2]\times[2\times10]$
- $c(i,i+4) = \min\{c(i,i) + c(i+1,i+4) + r_i r_{i+1} r_{i+5},$
  $c(i,i+1) + c(i+2,i+4) + r_i r_{i+2} r_{i+5},\ c(i,i+2) + c(i+3,i+4) + r_i r_{i+3} r_{i+5},$
  $c(i,i+3) + c(i+4,i+4) + r_i r_{i+4} r_{i+5}\}$

	1	2	3	4	5
1	0	50	150	90	190
2		0	50	30	90
3			0	20	40
4				0	200
5					0

c(i,j), i ≤ j

	1	2	3	4	5
1		1	2	2	2
2			2	2	2
3				3	3
4					4
5					

kay(i,j), i ≤ j

Example 20.11 (9)

- Optimal multiplication sequence
  - kay(1,5) = 2
    - $M_{15} = M_{12} \times M_{35}$

	1	2	3	4	5
1	0	50	150	90	190
2		0	50	30	90
3			0	20	40
4				0	200
5					0

c(i,j), i ≤ j

	1	2	3	4	5
1		1	2	2	2
2			2	2	2
3				3	3
4					4
5					

kay(i,j), i ≤ j

Example 20.11 (10)

- $M_{15} = M_{12} \times M_{35}$
- kay(1,2) = 1 ▶ $M_{12} = M_{11} \times M_{22}$

→ $M_{15} = (M_{11} \times M_{22}) \times M_{35}$

	1	2	3	4	5
1	0	50	150	90	190
2		0	50	30	90
3			0	20	40
4				0	200
5					0

c(i,j), i ≤ j

	1	2	3	4	5
1		1	2	2	2
2			2	2	2
3				3	4
4					4
5					

kay(i,j), i ≤ j

Example 20.11 (11)

- $M_{15} = (M_{11} \times M_{22}) \times M_{35}$
- Kay(3,5) = 4 ▶ $M_{35} = M_{34} \times M_{55}$

→ $M_{15} = (M_{11} \times M_{22}) \times (M_{34} \times M_{55})$

	1	2	3	4	5
1	0	50	150	90	190
2		0	50	30	90
3			0	20	40
4				0	200
5					0

c(i,j), i ≤ j

	1	2	3	4	5
1		1	2	2	2
2			2	2	2
3				3	4
4					4
5					

kay(i,j), i ≤ j

Example 20.11 (12)

- $M_{15} = (M_{11} \times M_{22}) \times (M_{34} \times M_{55})$
- Kay(3,4) = 3 ▶ $M_{34} = M_{33} \times M_{44}$

→ $M_{15} = (M_{11} \times M_{22}) \times ((M_{33} \times M_{44}) \times M_{55})$

	1	2	3	4	5
1	0	50	150	90	190
2		0	50	30	90
3			0	20	40
4				0	200
5					0

c(i,j), i ≤ j

	1	2	3	4	5
1		1	2	2	2
2			2	2	2
3				3	4
4					4
5					

kay(i,j), i ≤ j

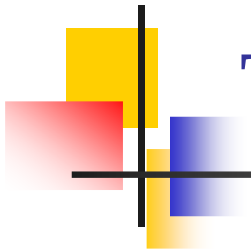


Table of Contents

SNU Internet DataBase Lab.

All-Pairs Shortest Path Problem

- To find a shortest path between every pair of vertices in a directed graph G
  - That is, for every pair of vertices (i, j), to find a shortest path from i to j as well as j to i
- Assumption
  - Permit negative-length edges
  - No negative-length cycles

⇒ Every pair of vertices (i, j) always has a shortest path that has no cycle

- Dijkstra's Greedy Single-Source Algorithm (18.3.5): O(n^3)
- Floyd's Dynamic Programming Solution: Θ (n^3)
  - c(i, j, k) denotes the length of a shortest path from i to j that has no intermediate vertex larger than k
    - C(i, j, 0): the length of the edge (i, j) in case this edge is in G
      - 0 (if i = j) or ∞ (otherwise)
    - C(i, j, n): the length of a shortest path from i to j

Shortest Path by Dynamic Programming (1):

- c(1, 3, k) = ?
  - ∞ (for k = 0, 1, 2, 3)
  - 28 (for k = 4)
  - 10 (for k = 5, 6, 7)
  - 9 (for k = 8, 9, 10)
- Hence, the shortest 1-to-3 path has length 9

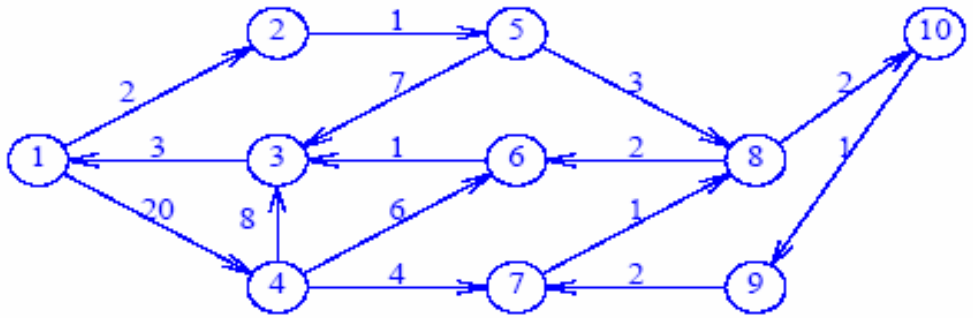


Figure 18.17 A weighted digraph

Shortest Path by Dynamic Programming (2)

- To determine c(i, j, k) for any k (k > 0)
  - Two possibilities
    - The path may not have k as an intermediate vertex
      - c(i, j, k) = c(i, j, k-1)
    - The path may have k as an intermediate vertex
      - c(i, j, k) = c(i, k, k-1) + c(k, j, k-1)
  - So, recurrence equation is
    - c(i, j, k) = min{c(i, j, k-1), c(i, k, k-1) + c(k, j, k-1)}, k>0

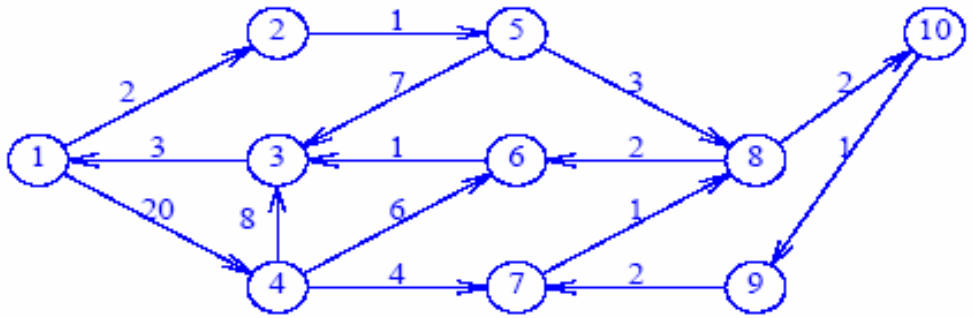


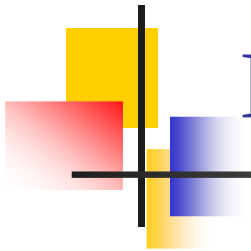
Figure 18.17 A weighted digraph

Iterative Solution for Shortest Path

- The value c(i, j, n) may be obtained efficiently by noticing that some c(i, j, k-1) values get used several times
  - All c values may be determined in Θ (n^3) time

```
// Find the lengths of the shortest paths.
  // initialize c(i,j,0)
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
      // a is the cost-adjacency matrix
      c(i,j,0) = a(i,j);
  // compute c(i,j,k) for 0 < k <= n
  for (int k = 1; k <= n; k++)
    for (int i = 1; i <= n; i++)
      for (int j = 1; j <= n; j++)
        if (c(i,k,k-1) + c(k,j,k-1)
                       < c(i,j,k-1))
          c(i,j,k) = c(i,k,k-1)+c(k,j,k-1);
        else
          c(i,j,k) = c(i,j,k-1);
```

Figure 20.3 Initial shortest-paths algorithm



BIRD'S-EYE VIEW

- Dynamic programming
  - Is the most difficult one of the five design methods
  - Has its foundation in the principle of optimality
- Elegant and efficient solutions to many problems
  - Knapsack
  - Matrix multiplication chains
  - Shortest-path

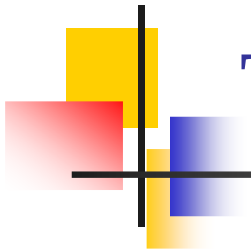


Table of Contents

SNU Internet DataBase Lab.