AAI2007 Introduction to Algorithms

Week 11: Dynamic Programming 1

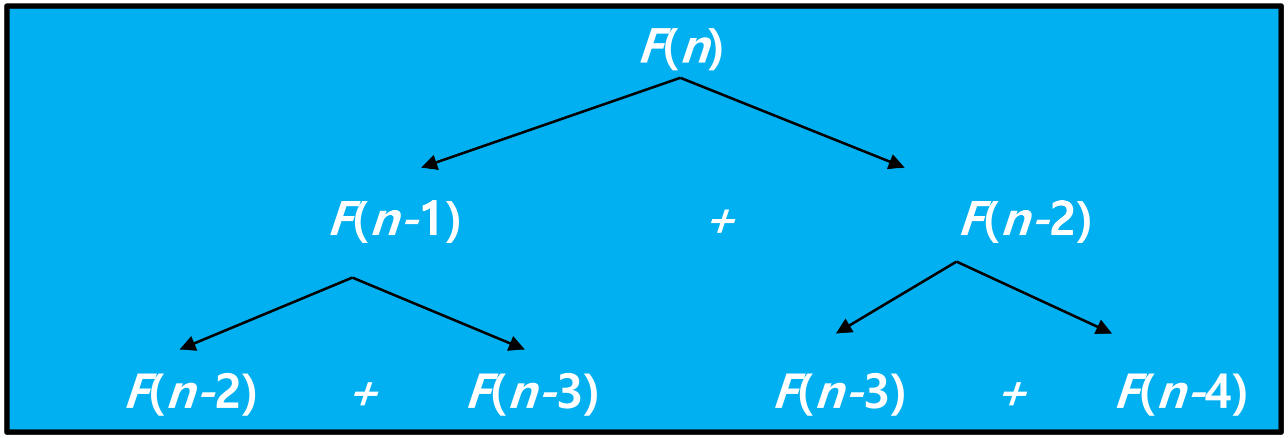
Instructor: Jinyoung Han (jinyounghan@skku.edu)

Fibonacci Numbers

F-Numbers

- Computing the nth Fibonacci number recursively:
  - $F(n) = F(n-1) + F(n-2)$
  - $F(0) = 0$
  - $F(1) = 1$
  - **A top-down approach**

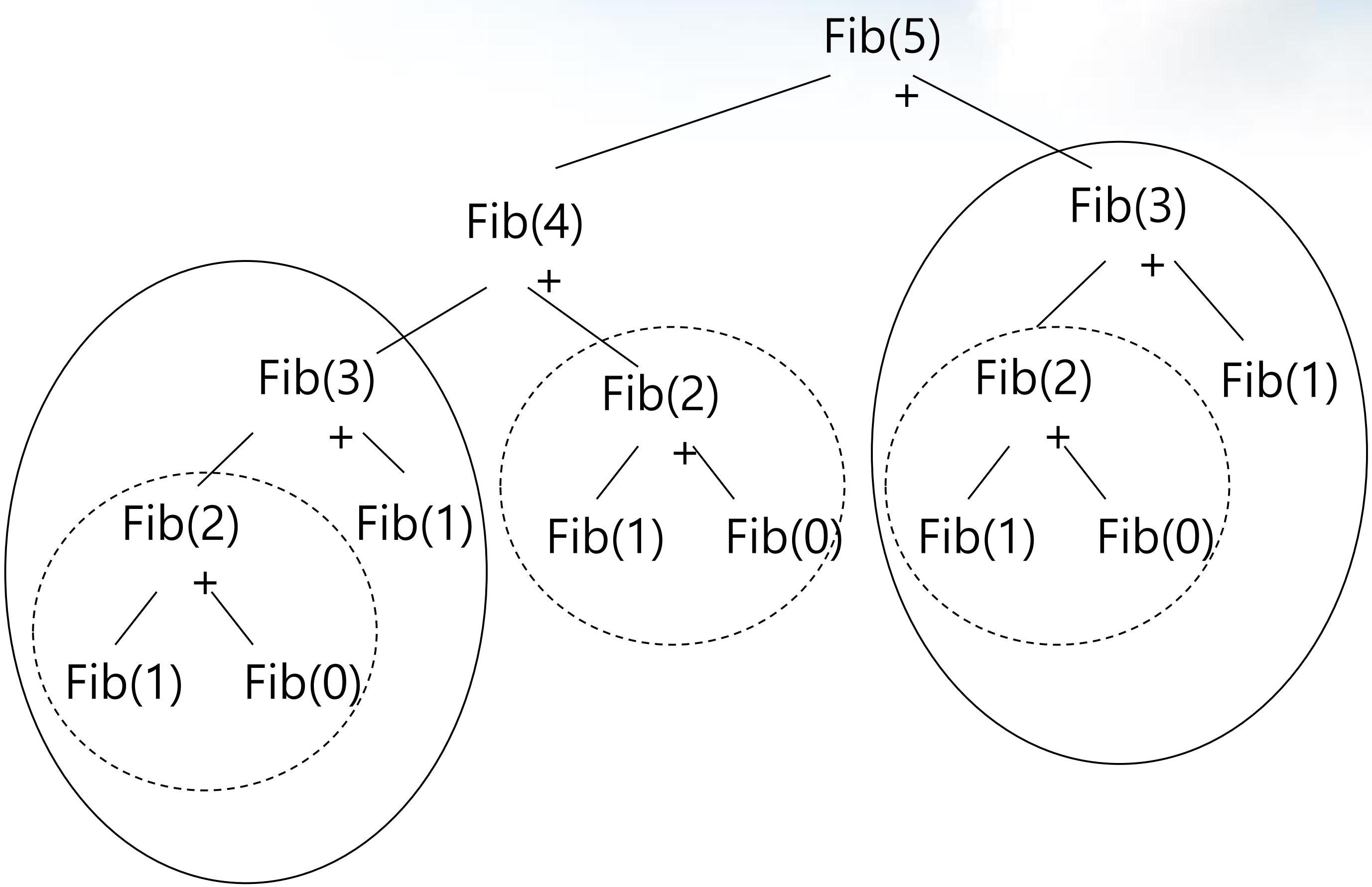
```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```



Fibonacci Numbers

F-Numbers

- This top-down approach is not so inefficient
  - Re-compute many sub-problems



Fibonacci Numbers

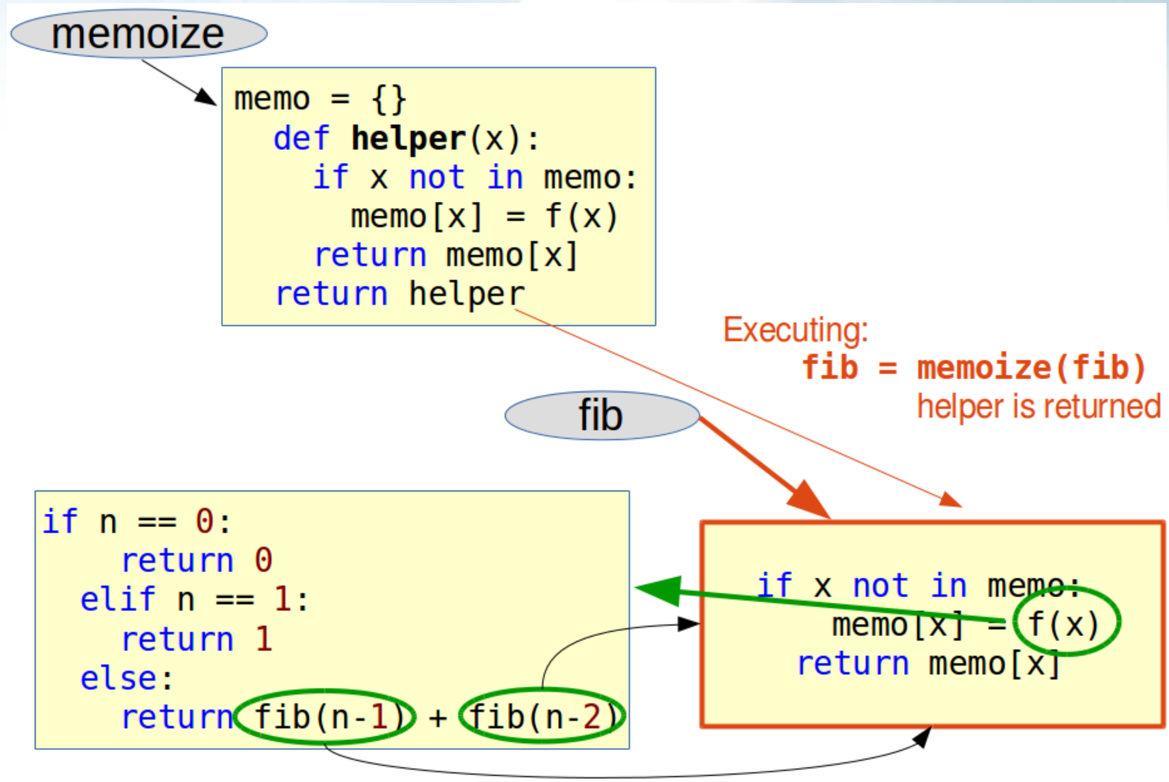
F-Numbers

- Alternative approach: memoization
  - Cache the internal results to avoid repetition

```
def memoize(f):
    memo = {}
    def helper(x):
        if x not in memo:
            memo[x] = f(x)
        return memo[x]
    return helper

def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

fib = memoize(fib)
```



Fibonacci Numbers

F-Numbers

- Alternative bottom-up approach – dynamic programming (O(n)!)
  - F(0) = 0
  - F(1) = 1
  - F(2) = 1+0 = 1
  - …
  - F(n-2) =
  - F(n-1) =
  - F(n) = F(n-1) + F(n-2)

```
def fib(n):
  fibValues = [0,1]
  for i in range(2,n+1):
    fibValues.append(fibValues[i-1] + fibValues[i-2])
  return fibValues[n]
```

0	**1**	**1**	**. . .**	***F*(*n*-2)**	***F*(*n*-1)**	***F*(*n*)**

Binomial Coefficient

B-Coefficient

- Binomial coefficient
  - $C(n, k) = C(n-1, k-1) + C(n-1, k)$
  - $C(n, 0) = C(n, n) = 1$

```
def binomialCoeff(n , k):

    if k==0 or k ==n :
        return 1
    return binomialCoeff(n-1 , k-1) + binomialCoeff(n-1 , k)
```

- Not efficient: many repetition!

Binomial Coefficient

B-Coefficient

- Binomial coefficient – dynamic programming
  - $C(n, k) = C(n-1, k-1) + C(n-1, k)$
  - $C(n, 0) = C(n, n) = 1$

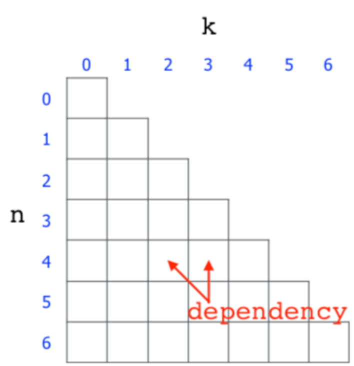
Value of $C(n,k)$ can be computed by filling a table:

	0	1	2	. . .	k-1	k
0	1					
1	1	1				
.						
.						
.						
n-1					$C(n-1,k-1)$	$C(n-1,k)$
n						$C(n,k)$

```
def binomialCoef(n, k):
    C = [[0 for x in range(k+1)] for x in range(n+1)]

    for i in range(n+1):
        for j in range(min(i, k)+1):
            if j == 0 or j == i:
                C[i][j] = 1
            else:
                C[i][j] = C[i-1][j-1] + C[i-1][j]

    return C[n][k]
```



In This Lecture

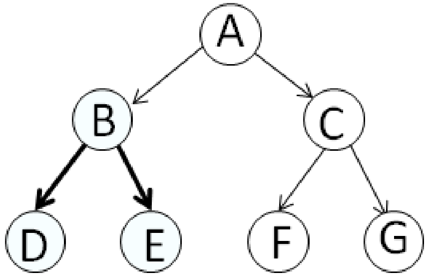
Outline

1. Dynamic Programming
2. Minimum Cost Path Problem
3. Matrix Chain-Products

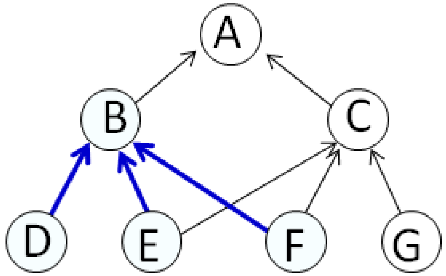
01. Dynamic Programming

DP

- Dynamic Programming (DP)
  - An algorithm design technique for optimization problems
    - often minimizing or maximizing
- Like "divide and conquer", DP solves problems by combining solutions to sub-problems
- Unlike divide and conquer, sub-problems are not independent
  - Sub-problems may share sub-sub-problems



DC



DP

01. Dynamic Programming

DP

- The term Dynamic Programming comes from Control Theory, not computer science
    - Programming refers to the use of tables (arrays) to construct a solution
- In dynamic programming, we usually reduce time by increasing the amount of space
- We solve the problem by solving sub-problems of increasing size and saving each optimal solution in a table (usually)
- The table is then used for finding the optimal solution to larger problems
- Time is saved since each sub-problem is solved only once

01. Dynamic Programming

DP

- When do we consider to apply DP?
  - Optimal substructure
    - Optimal solution of a bigger problem includes the optimal solution of a smaller problem
  - Overlapping recursive calls
    - Recursive solution results in many repetitions
- Memoization vs. DP
  - Both often considered as a DP
  - Memoization: top-down approach, solve mandatory subproblems
  - DP: bottom-up approach, no overhead with recursion

01. Dynamic Programming

Examples

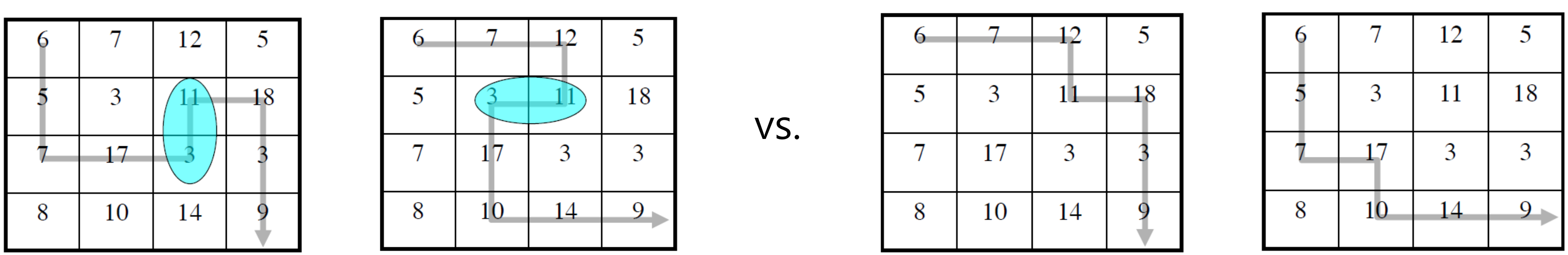
- Minimum cost path problem
- Matrix chaining optimization
- Longest common subsequence
- Knapsack problem
- Traveling salesman problem
- Warshall's algorithm for Transitive Closure
- Floyd's algorithm for all-pairs shortest paths
- Computing a binomial coefficient
- Constructing an optimal binary search tree

Minimum Cost Path Problem

02. Minimum Cost Path Problem

Problem

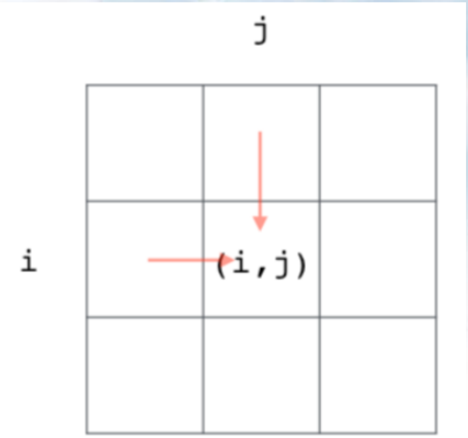
- Given an n x n matrix (with positive elements), move from the left top to the right bottom
- Constrains
  - Only move to right or down
  - Not allow to move left, up, or diagonal directions
- Goal
  - Find the minimum cost path



02. Minimum Cost Path Problem

Idea

- To reach (i, j),
  - we go through the (i, j-1) or (i-1, j)
- Also, to (i, j-1) or (i-1, j),
  - We move with the optimal way



02. Minimum Cost Path Problem

Recursive Approach

- minCost(m, n)
  - minCost(m, n) = min (minCost(m-1, n), minCost(m, n-1)) + cost[m][n]

```
def minCost(cost, m, n):
    if (n < 0 or m < 0):
        return sys.maxsize
    elif (m == 0 and n == 0):
        return cost[m][n]
    else:
        return cost[m][n] + min(minCost(cost, m-1, n),
                                minCost(cost, m, n-1) )
```

Many subproblems repetition!

02. Minimum Cost Path Problem

DP

- minCost(m, n)
  - minCost(m, n) = min (minCost(m-1, n), minCost(m, n-1)) + cost[m][n]

```
def minCost(cost, m, n):
    tc = [[0 for x in range(C)] for x in range(R)]
    tc[0][0] = cost[0][0]

    # Initialize first column of total cost(tc) array
    for i in range(1, m+1):
        tc[i][0] = tc[i-1][0] + cost[i][0]

    # Initialize first row of tc array
    for j in range(1, n+1):
        tc[0][j] = tc[0][j-1] + cost[0][j]

    # Construct rest of the tc array
    for i in range(1, m+1):
        for j in range(1, n+1):
            tc[i][j] = min(tc[i-1][j], tc[i][j-1]) + cost[i][j]

    return tc[m][n]
```

m

6	7	12	5
5	3	11	18
7	17	3	3
8	10	14	9

L	1	2	3	4
1	6	13	25	30
2	11	14	25	43
3	18	31	28	31
4	26	36	42	40

P

-	←	←	←
↑	←	←	←
↑	↑	↑	←
↑	←	↑	↑

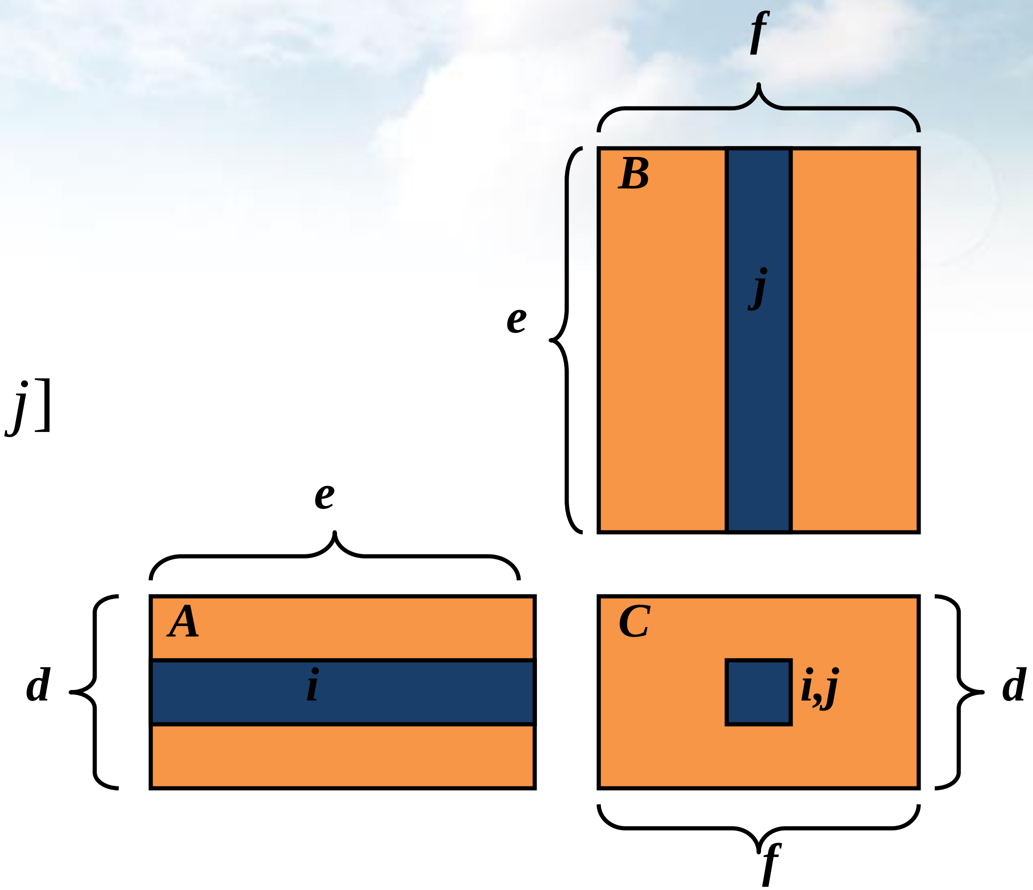
Matrix Chain-Products

03. Matrix Chain-Products

Matrix Multiplication

- Review: Matrix Multiplication
  - C = A*B
  - A is $d \times e$ and B is $e \times f$
  - $O(d \cdot e \cdot f)$ time

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



03. Matrix Chain-Products

Matrix Chain-Product

- Matrix Chain-Product
  - Compute $A=A_0*A_1*\ldots*A_{n-1}$
  - A_i is $d_i \times d_{i+1}$
  - Problem: How to parenthesize?
- Example
  - B is 3 × 100
  - C is 100 × 5
  - D is 5 × 5
  - (B*C)*D takes 1500 + 75 = 1575 ops
  - B*(C*D) takes 1500 + 2500 = 4000 ops

03. Matrix Chain-Products

Matrix Chain-Product

- Matrix Chain-Product algorithm
  - Try all possible ways to parenthesize $A=A_0*A_1*\ldots*A_{n-1}$
  - Calculate number of ops for each one
  - Pick the one that is best
- Running time
  - The number of parenthesizations is equal to the number of binary trees with n nodes
  - This is **exponential**!
  - It is called the Catalan number, and it is almost 4^n
  - This is a terrible algorithm!

03. Matrix Chain-Products

Greedy Approach

- Idea #1: repeatedly select the product that uses the fewest operations
- Counter-example
  - A is 101 × 11
  - B is 11 × 9
  - C is 9 × 100
  - D is 100 × 99
  - Greedy idea #1 gives A*((B*C)*D)), which takes 109989+9900+108900=228789 ops
  - (A*B)*(C*D) takes 9999+89991+89100=189090 ops
- The greedy approach is not giving us the optimal value

03. Matrix Chain-Products

Optimal Solution

- The optimal solution can be defined in terms of optimal sub-problems
  - There has to be a final multiplication (root of the expression tree) for the optimal solution
  - Say, the final multiplication is at index k: $(A_0 * \ldots * A_k) * (A_{k+1} * \ldots * A_{n-1})$
- Let us consider all possible places for that final multiplication
  - There are *n*-1 possible ***splits***. Assume we know the minimum cost of computing the matrix product of each combination $A_0 \ldots A_i$ and $A_{i+1} \ldots A_n$. Let's call these $N_{0,i}$ and $N_{i+1,n}$
- Recall that A_i is a $d_i \times d_{i+1}$ dimensional matrix, and the final product will be a $d_0 \times d_n$

03. Matrix Chain-Products

Optimal Solution

- Definition

$$\boxed{N_{0,n-1} = \min_{0 \le k < n-1} \{N_{0,k} + N_{k+1,n-1} + d_0 d_{k+1} d_n\}}$$

- Then the optimal solution $N_{0,n-1}$ is the sum of two optimal sub-problems, $N_{0,k}$ and $N_{k+1,n-1}$ plus the time for the last multiplication

03. Matrix Chain-Products

Optimal Solution

- Define **sub-problems**
  - Find the best parenthesization of an arbitrary set of consecutive products: $A_i * A_{i+1} * \ldots * A_j$
  - Let $N_{i,j}$ denote the **minimum** number of operations done by this sub-problem
    - Define $N_{k,k} = 0$ for all k
  - The optimal solution for the whole problem is then $N_{0,n-1}$

03. Matrix Chain-Products

Optimal Solution

- The characterizing equation for $N_{i,j}$ is:

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

- For example, $N_{2,6}$ and $N_{3,7}$, both need solutions to $N_{3,6}$, $N_{4,6}$, $N_{5,6}$, and $N_{6,6}$.
  - This is an example of high sub-problem overlap, and clearly pre-computing these will significantly speed up the algorithm

03. Matrix Chain-Products

Recursive Approach

- We could implement the calculation of these $N_{i,j}$'s using a straight-forward recursive implementation of the equation (aka not pre-compute them)

Algorithm ***RecursiveMatrixChain***(S, i, j):

Input: sequence S of n matrices to be multiplied

Output: number of operations in an optimal parenthesization of S

if i=j

  then return 0

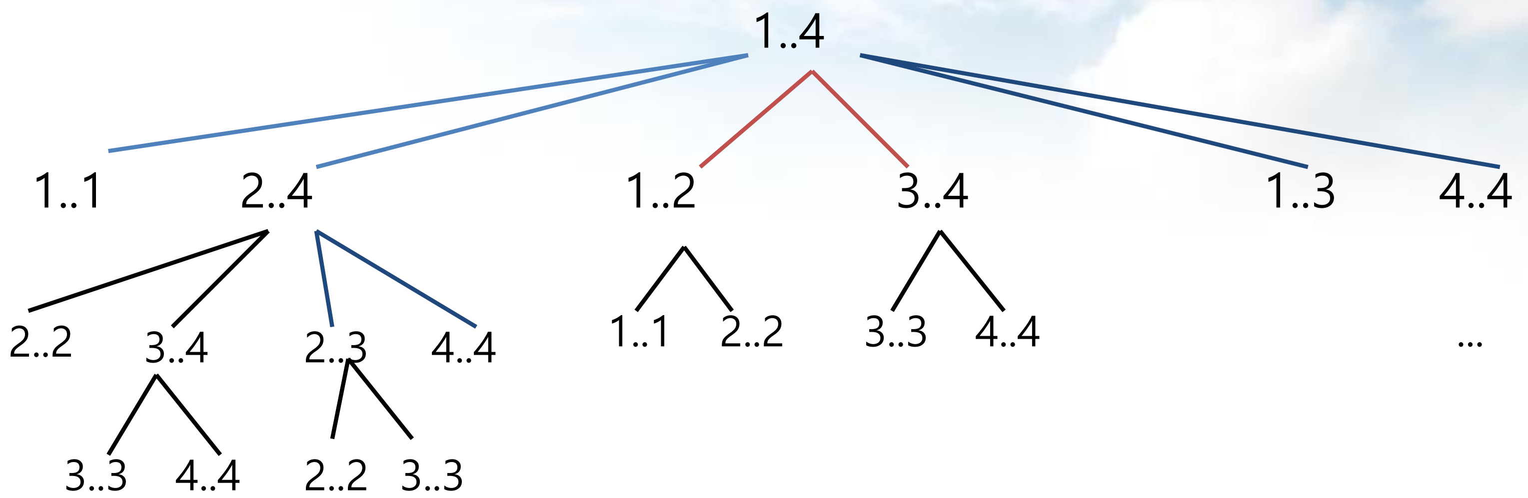
for $k \leftarrow i$ **to** j **do**

  $N_{i,j} \leftarrow \min\{N_{i,j},\ \textit{RecursiveMatrixChain}(S, i, k) + \textit{RecursiveMatrixChain}(S, k+1, j) + d_i d_{k+1} d_{j+1}\}$

return $N_{i,j}$

Subproblem Overlap

- Overlap



03. Matrix Chain-Products

DP

- High sub-problem overlap, with independent sub-problems indicate that a dynamic programming approach may work
- Construct optimal sub-problems “bottom-up” and remember them
- $N_{i,i}$’s are easy, so start with them
- Then do problems of *length* 2,3,… sub-problems, and so on
- Running time: $O(n^3)$

```
Algorithm matrixChain(S):
    Input: sequence S of n matrices to be multiplied
    Output: number of operations in an optimal parenthesization of S
    for i ← 1 to n – 1 do
        N_{i,i} ← 0
    for b ← 1 to n – 1 do
        { b = j – i is the length of the problem }
        for i ← 0 to n – b - 1  do
            j ← i + b
            N_{i,j} ← +∞
            for k ← i to j – 1 do
                N_{i,j} ← min{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}}
    return N_{0,n-1}
```

03. Matrix Chain-Products

DP

Algorithm *matrixChain(S)*:

Input: sequence S of n matrices to be multiplied

Output: number of operations in an optimal parenthesization of S

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

  $N_{i,i} \leftarrow 0$

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

  { $b = j - i$ is the length of the problem }

  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**

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

return $N_{0,n-1}$

(1) Initialize C[1,1], C[2,2], …, C[n-1,n-1] as 0
- No computation is needed for same matrix

03. Matrix Chain-Products

DP

Algorithm *matrixChain*(S):

Input: sequence S of n matrices to be multiplied

Output: number of operations in an optimal parenthesization of S

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

$N_{i,i} \leftarrow 0$

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

{ $b = j - i$ is the length of the problem }

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

$j \leftarrow i + b$

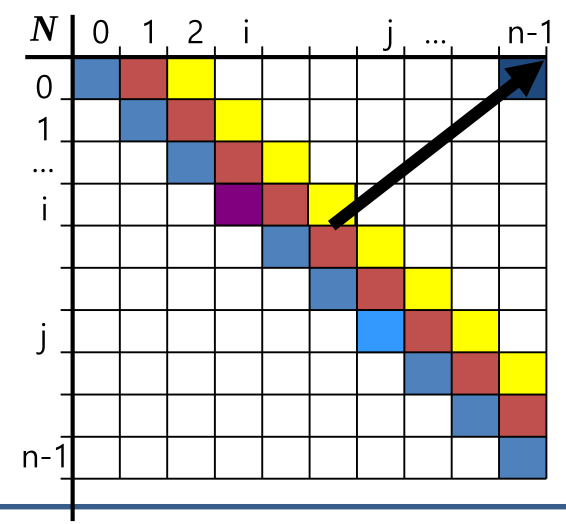
$N_{i,j} \leftarrow +\infty$

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

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

return $N_{0,n-1}$

(2) b: length of a sub-problem



03. Matrix Chain-Products

DP

Algorithm *matrixChain(S)*:

Input: sequence S of n matrices to be multiplied

Output: number of operations in an optimal parenthesization of S

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

$N_{i,i} \leftarrow 0$

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

{ $b = j - i$ is the length of the problem }

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**

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

return $N_{0,n-1}$

(3) b=1 -> subproblems with size 2

b=2 -> subproblems with size 3

(e.g., $A_1 x A_2 x A_3$, $A_2 x A_3 x A_4$, ...)

...

03. Matrix Chain-Products

DP

Algorithm *matrixChain*(S):

Input: sequence S of n matrices to be multiplied

Output: number of operations in an optimal parenthesization of S

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

$N_{i,i} \leftarrow \mathbf{0}$

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

{ $b = j - i$ is the length of the problem }

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** (4) To find the minimum # operations

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

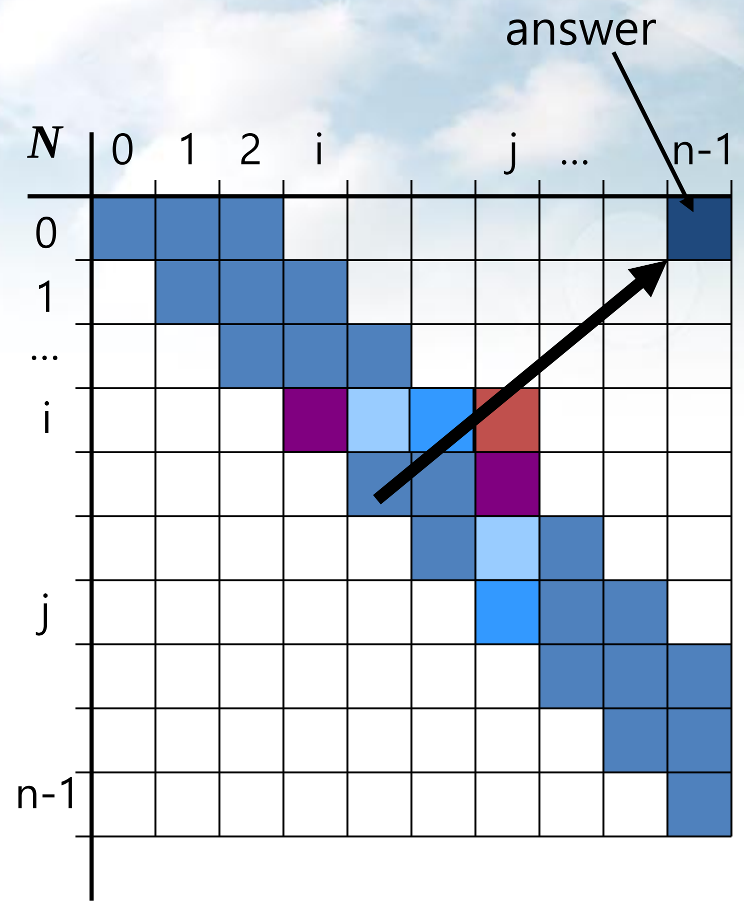
return $N_{0,n-1}$

03. Matrix Chain-Products

Visualization

- The bottom-up construction fills in the N array by diagonals
- $N_{i,j}$ gets values from previous entries in i-th row and j-th column
- Filling in each entry in the N table takes O(n) time
- Total run time: $O(n^3)$
- Getting actual parenthesization can be done by remembering "k" for each N entry

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



03. Matrix Chain-Products

Visualization

- A_0: 30 X 35; A_1: 35 X15; A_2: 15X5; A_3: 5X10; A_4: 10X20; A_5: 20 X 25

0	1	2	3	4	5	
0	15,750	7,875	9,375	11,875	15,125	0
	0	2,625	4,375	7,125	10,500	1
		0	750	2,500	5,375	2
			0	1,000	3,500	3
				0	5,000	4
					0	5

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

$$N_{1,4} = \min\{$$
$$N_{1,1} + N_{2,4} + d_1 d_2 d_5 = 0 + 2500 + 35*15*20 = 13000,$$
$$N_{1,2} + N_{3,4} + d_1 d_3 d_5 = 2625 + 1000 + 35*5*20 = 7125,$$
$$N_{1,3} + N_{4,4} + d_1 d_4 d_5 = 4375 + 0 + 35*10*20 = 11375$$
$$\}$$
$$= 7125$$

03. Matrix Chain-Products

Summary

- We ~~reduced~~ replaced a $\boldsymbol{O}(2^n)$ algorithm with a $\boldsymbol{O}(n^3)$ algorithm
- While the generic top-down recursive algorithm would have solved $\boldsymbol{O}(2^n)$ sub-problems, there are $\boldsymbol{O}(n^2)$ sub-problems
  - Implies a high overlap of sub-problems
- The sub-problems are independent:
  - Solution to $A_0A_1\ldots A_k$ is independent of the solution to $A_{k+1}\ldots A_n$
- Determine the cost of each pair-wise multiplication, then the minimum cost of multiplying three consecutive matrices, using the pre-computed costs for two matrices
- Repeat until we compute the minimum cost of all n matrices using the costs of the minimum n-1 matrix product costs

What You Need to Know

Summary

- Dynamic programming
  - Fibonacci, binomial coefficient examples
  - Memoization vs. DP, DC vs. DP
- Minimum cost path problem
- Matrix Chain-Products

Thanks

Week 11: Dynamic Programming 1
Instructor: Jinyoung Han (jinyounghan@skku.edu)