CS 6212 DESIGN AND ANALYSIS OF ALGORITHMS

LECTURE: DYNAMIC PROGRAMMING – PART I

Instructor: Abdou Youssef

OBJECTIVES OF THIS LECTURE (PART B)

By the end of Part B of this lecture, you will be able to:

- Describe how Dynamic Programming (DP) works, including its major design steps
- State and explain the *principle of optimality*
- Prove the principle of optimality holds in some problems, but not in others
- Begin to apply DP to derive powerful optimization algorithms
- Compare and contrast DP with Divide & Conquer and with the Greedy Method

OUTLINE (OF PART B)

- Perspective
  - DP as optimization
  - DP vs. Greedy
  - DP vs. Divide & Conquer
- Principle of Optimality
- Steps of Dynamic Programming
- First application: The Matrix Chain Problem

PERSPECTIVE
-- OPTIMIZATION --

- Dynamic programming is an optimization technique
- Recall what that is
  - What is an optimization problem?
  - What is an optimization algorithm?

PERSPECTIVE
-- DYNAMIC PROGRAMMING VS. GREEDY METHOD --

- Both techniques are optimization techniques
- Both build solutions from a collection of choices of individual elements.
- The greedy method
  - Computes its solution
    - by making its choices in a serial forward fashion,
    - never looking back, and never revising previous choices
  - There is **no litmus test** to quickly pre-check if the greedy solution will be optimal
- Dynamic programming
  - computes its solution bottom up by
    - synthesizing them from smaller subsolutions, and
    - by trying many possibilities and choices before it arrives at the optimal set of choices
  - There is **a litmus test** for DP, called the ***Principle of Optimality***, which can be applied relatively easily and quickly to tell if DP gives an optimal solution

PERSPECTIVE
-- DYNAMIC PROGRAMMING VS. DIVIDE & CONQUER --

- Both techniques split their input into parts, find subsolutions to the parts, and synthesize larger solutions from smaller ones
- Divide and Conquer splits its input at **a few pre-specified deterministic points** (e.g., always in the middle)
- Dynamic Programming
  - Splits its input at **every possible split points** rather than at a few pre-specified points
  - After trying all split points, it determines which split point is optimal
- DP is mainly for optimization, while D&C is mainly for non-optimization problems (there are exceptions)

PRINCIPLE OF OPTIMALITY (POO)

- **Definition**: A problem is said to satisfy the ***Principle of Optimality*** if **the subsolutions of any optimal solution of the problem are optimal** solutions of their subproblems

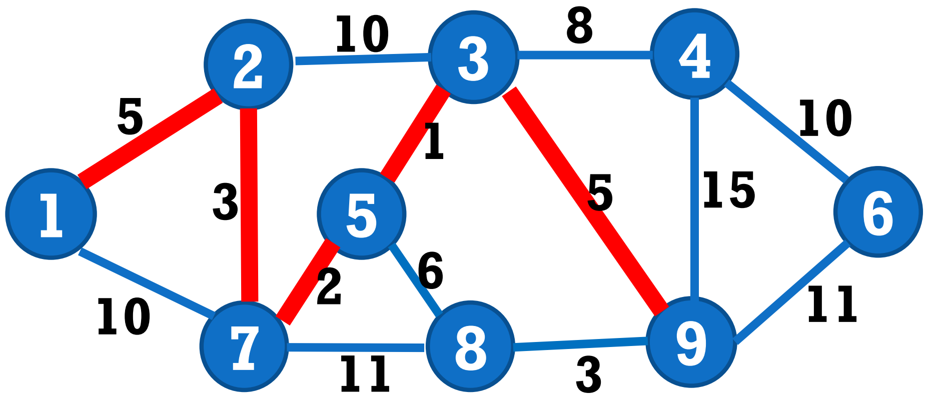
PRINCIPLE OF OPTIMALITY (POO)

-- AN EXAMPLE WHERE POO HOLDS --

- The shortest path problem satisfies the POO
- **Statement of the POO**: if $a, x_1, x_2, \dots, x_n, b$ is a shortest path from node a to node b in a graph, then the portion of x_i to x_j on that path is a shortest path from x_i to x_j, for any two intermediary nodes x_i and x_j
- **POO holds**: Proof by contradiction. If a portion is not shortest, then there is a shorter alternative ("a short cut") between x_i and x_j. Replace that portion by the short cut, we get a shorter path from a to b, contradicting that the original path was a shortest path.

- **Red path**: shortest from 1 to 9

  1, 2, 7, 5, 3, 9
- The portion from 2 to 3 (2, 7, 5, 3) is a shortest path from 2 to 3



PRINCIPLE OF OPTIMALITY (POO)

-- AN EXAMPLE WHERE POO DOESN'T HOLD --

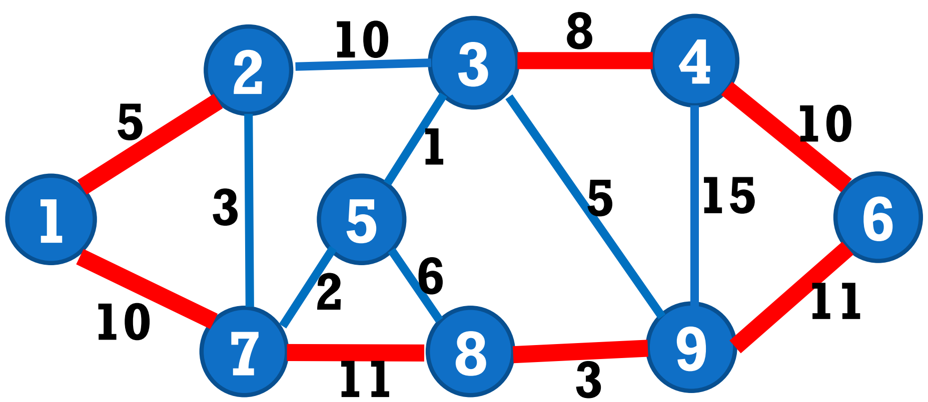
- The longest path problem **doesn't** satisfy the POO
- **Statement of the POO:** Every sub-path of longest simple path is a longest simple path between its end-points.
- **POO does not hold**: Proof by a counter-example. See the counter-example on the right.

- **Red path**: Longest from 2 to 3

  2, 1, 7, 8, 9, 6, 4

- The portion from 7 to 8 (of length 11) is not the longest simple path from 7 to 8 because:

  7, 1, 2, 3, 4, 6, 9, 8 is longer (57>11)



LESSONS LEARNED SO FAR

- DP is an optimization design technique
- It has a litmus test, called the principle of optimality, which holds in some problems but not in all problems
- More lessons later

STEPS OF DYNAMIC PROGRAMMING

Dynamic programming design involves 4 major steps:

1. **Notation**: Develop a mathematical notation that can express every solution and subsolution for the problem at hand
2. **POO**: Prove that the Principle of Optimality holds
3. **Recurrence Relation (RR)**: Develop a recurrence relation that relates a solution to its subsolutions, using the math notation of step 1. Indicate what the initial values are for that recurrence relation, and which term signifies the final solution.
4. **Algorithm for the RR**: Write an algorithm to compute the recurrence relation bottom up

COMMENTS ABOUT THE STEPS OF DP

- The steps in the previous slide are not called a template.
- That is because the steps are guidelines rather than algorithmic instructions
- Steps 1 and 2 need not be in that order
  - Do what makes sense in each problem
  - If you don't need notation to state (and prove) the principle of optimality, then do step 2 first
  - Otherwise, do step 1 first
- Step 3 is the heart of the design process
  - In high level algorithmic design situations, one can stop at step 3.
  - In this course, however, we will carry out step 4 as well.

MORE COMMENTS ABOUT THE STEPS OF DP

- Without the Principle of Optimality, it won't be possible to derive a sensible recurrence relation in step 3
  - This will be seen when we start using POO a little later

- When the Principle of Optimality holds, the 4 steps of DP are **guaranteed** to yield an **optimal solution**
  - No separate proof of optimality is needed
  - The proof that POO holds is sufficient!

FIRST APPLICATION OF DP
-- THE MATRIX CHAIN PROBLEM --

The matrix Chain Problem:

- **Input**: n matrices $A_1, \quad A_2, \quad \dots \quad, A_i, \quad \dots, \quad A_n$ of dimensions

$$p_1 \times p_2, p_2 \times p_3, \dots, p_i \times p_{i+1}, \dots, p_n \times p_{n+1}$$

- **Output**: A sequence of pairings of the matrices in order to multiply them in the least amount of time
- **Task**: Design a DP algorithm for solving this problem

THE MATRIX CHAIN PROBLEM
-- PRELIMINARIES--

- A matrix is a table of numbers, i.e., a 2-dimensional array
  - A matrix $A = A[1{:}p, 1{:}q]$ is referred to as a $p \times q$ matrix
  - Its lines are called *rows* (so this A has p rows) labeled 1, 2, …
  - The columns are labeled 1, 2, … (A has q columns)
  - The element $A[i, j]$ in row i and column j is denoted A_{ij}
- Matrix multiplication
  - Let A be a $p \times q$ matrix, and B be a $q \times r$ matrix
  - The product AB is a $p \times r$ matrix C where
    - $C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} + \cdots A_{iq}B_{qj}$ (inner product of row i of A and column j of B)
  - Time of C_{ij} is q multiplications and $q - 1$ additions: simplify that to q
  - Times of computing all of C: $p \times r \times q = pqr$

- $AB \neq BA$
- $(AB)C = A(BC)$

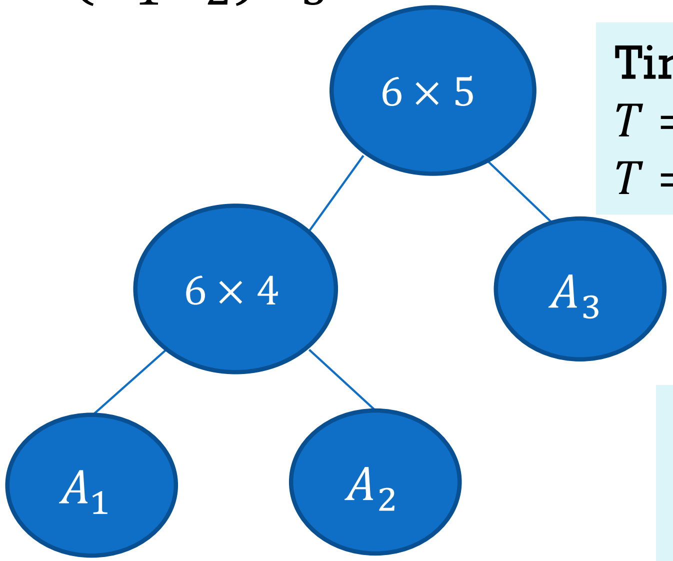
THE MATRIX CHAIN PROBLEM
-- EXAMPLE--

- Suppose we have 3 matrices A_1, A_2, A_3 of dimensions

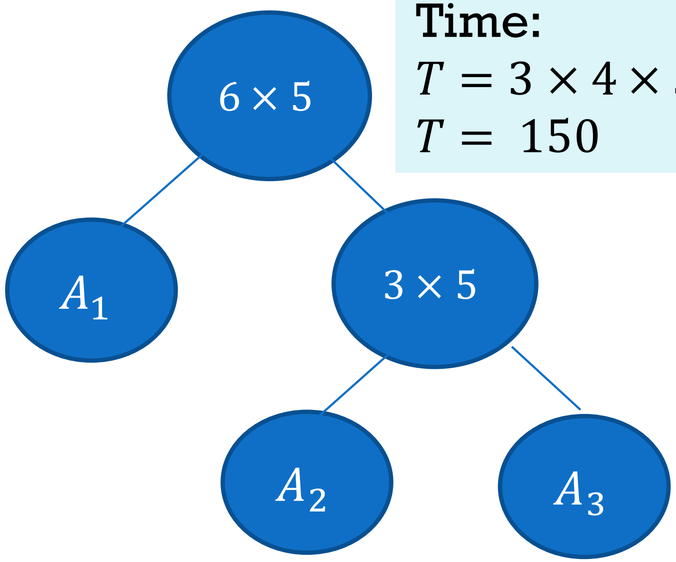
$$6 \times 3,\ \ 3 \times 4,\ \ 4 \times 5$$

- Two ways to multiply $A_1 A_2 A_3$ (both give the same answer):

$(A_1 A_2) A_3$ or $A_1 (A_2 A_3)$



Time:
$T = 6 \times 3 \times 4 \ + \ 6 \times 4 \times 5$
$T = 192$



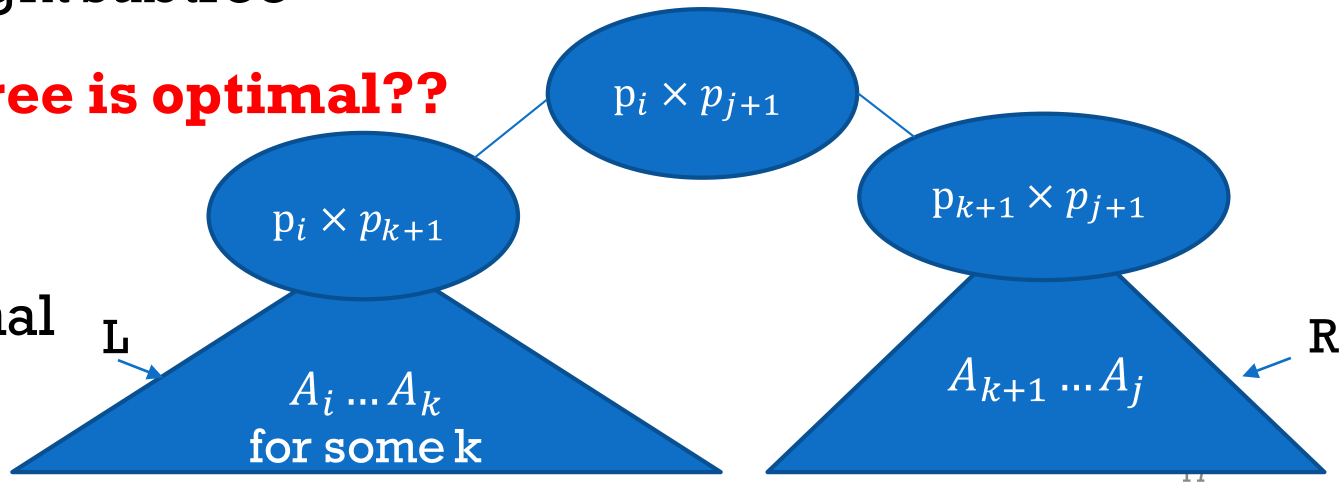
Time:
$T = 3 \times 4 \times 5 \ + \ 6 \times 3 \times 5$
$T = \ 150$

The second way is better

THE MATRIX CHAIN PROBLEM

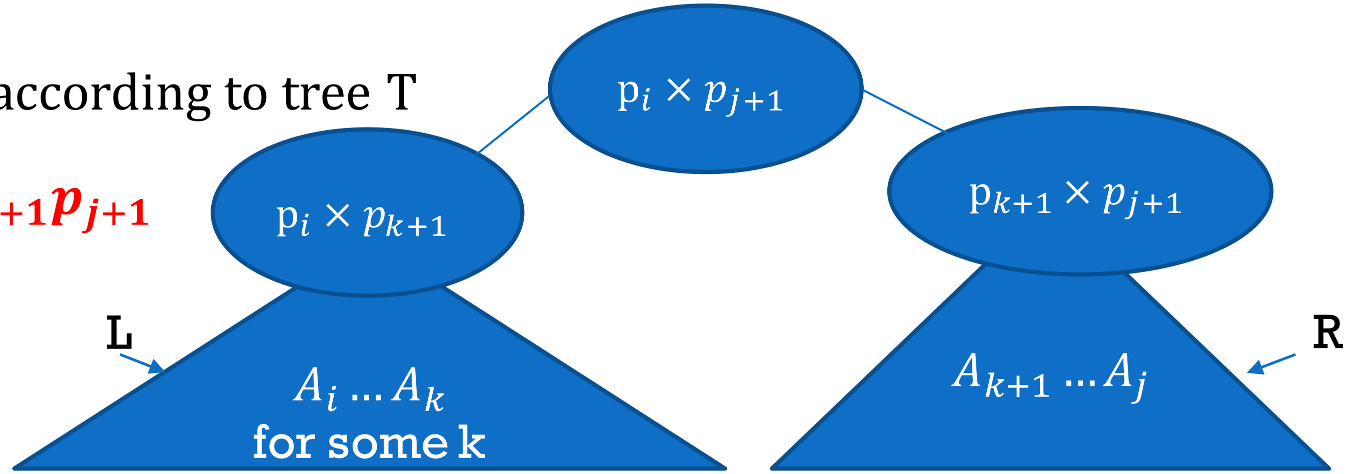
-- (1) NOTATION AND STATEMENT OF THE POO --

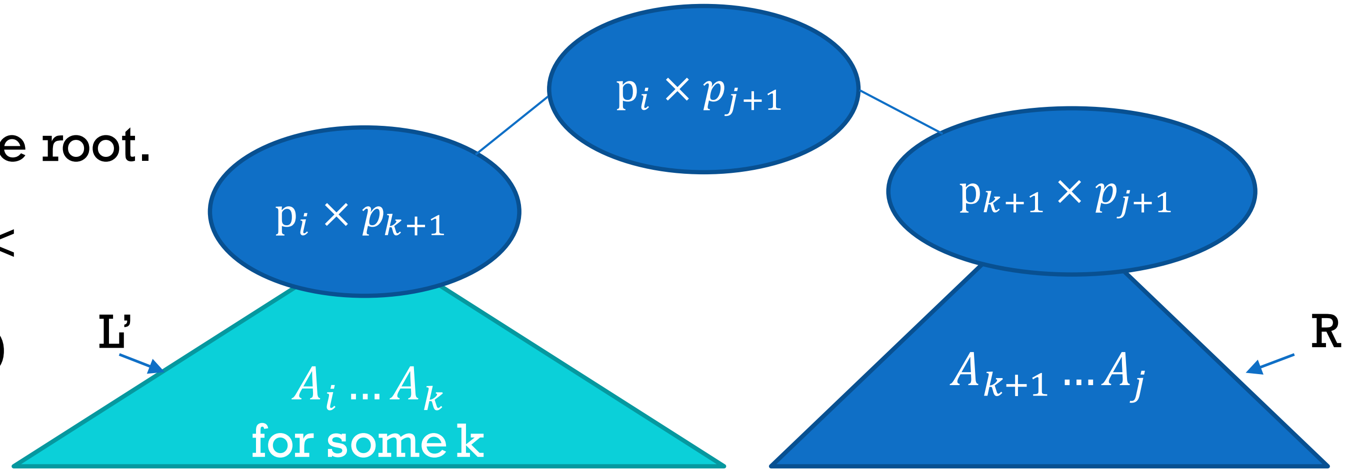
- **Input**: $A_1, A_2, \ldots, A_n$ where A_i is of dimension $p_i \times p_{i+1}$
- Every way of multiplying a sequence of matrices can be represented by a binary (infix) tree, where the leaves are the matrices, and the internal nodes are intermediary products (as was illustrated on the previous slide)
- Let T_{ij} be the optimal tree of multiplying $A_i \times \cdots \times A_j$
  - Let L be its left subtree, R be its right subtree
- POO: **Every subtree of an optimal tree is optimal??**
- POO: **L and R are optimal**
- Proof of POO: prove L and R are optimal



THE MATRIX CHAIN PROBLEM
-- (2) PROOF OF THE POO --

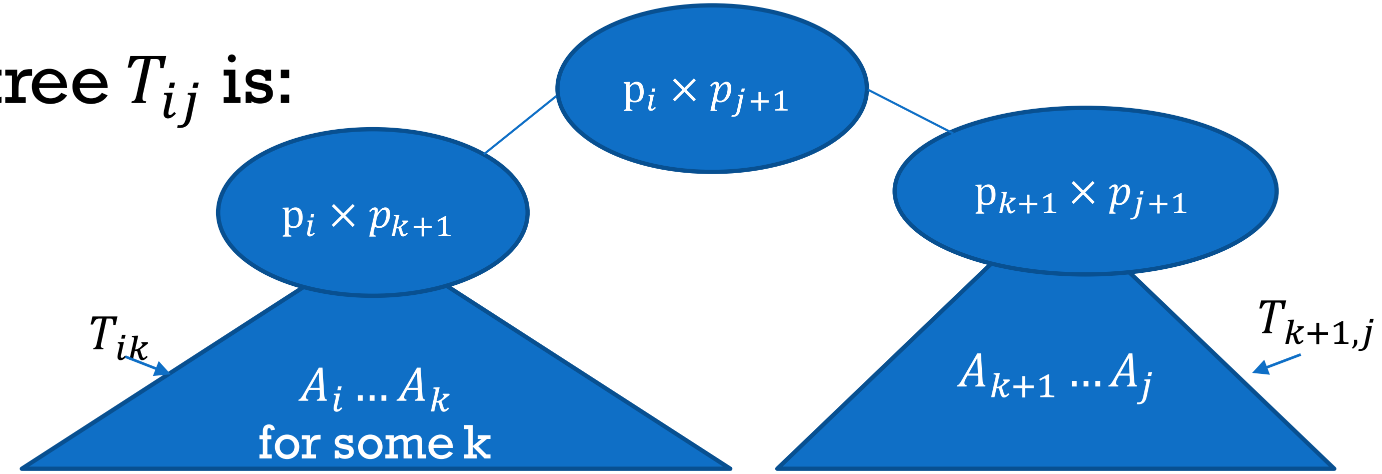
- Let $cost(\mathrm{T})$ the time to multiply the matrices according to tree T
- Thus: $\boldsymbol{cost(T_{ij}) = cost(L) + cost(R) + p_i p_{k+1} p_{j+1}}$
- Need to prove L (and R) is optimal
- By contradiction. If L is not optimal, there is a faster tree L' of multiplying its matrices $A_i \ldots A_k$, i.e., $cost(L') < cost(L)$
- Take the tree T' made of L', R, and the same root.
- $cost(T') = cost(L') + cost(R) + p_i p_{k+1} p_{j+1} <$ $cost(L) + cost(R) + p_i p_{k+1} p_{j+1} = cost(T_{ij})$
- $cost(T') < cost(T_{ij})$
- This implies that T' is better than optimal. Contradiction. Q.E.D.





THE MATRIX CHAIN PROBLEM
-- MORE ON NOTATION AND VISUALS--

- Now that L and R are optimal, they should be denoted T_{ik} and $T_{k+1,j}$, respectively.
- Visually, the optimal tree T_{ij} is:



- Let $M_{ij} = cost(T_{ij})$
- When $i = j$, T_{ii} is a single node (single matrix A_i) with no multiplication, so $M_{ii} = 0$ for all i
- The optimal cost of multiplying $A_1 A_2 \ldots A_n$ is M_{1n}, to be determined

THE MATRIX CHAIN PROBLEM
-- (3) RECURRENCE RELATION--

- We saw before that $cost(T_{ij}) = cost(L) + cost(R) + p_i p_{k+1} p_{j+1}$
- Therefore, $cost(T_{ij}) = cost(T_{ik}) + cost(T_{k+1,j}) + p_i p_{k+1} p_{j+1}$
- Using the M notation: $M_{ij} = M_{ik} + M_{k+1,j} + p_i p_{k+1} p_{j+1}$
- But what is k?
  - It is a split point
  - We don't know it, but we know that $i \leq k \leq j - 1$ and that it is the <u>best</u> split point in that it should be the split point that gives us the minimum cost
- So to find the best k, try all its possible values and take the best
- Therefore, the RR is:

$$\boldsymbol{M_{ij} = \min_{i \leq k \leq j-1} (M_{ik} + M_{k+1,j} + p_i p_{k+1} p_{j+1})}$$

THE MATRIX CHAIN PROBLEM
-- (4) RECURRENCE RELATION IMPLEMENTATION --

- Now one needs to write an algorithm for computing all the M_{ij} for all $1 \leq i \leq j \leq n$
- To be able to construct the actual optimal solution, we need to record for each M_{ij} the k that produces its minimum
- Since the matrix chain problem is a "toy" problem, we will not produce the actual algorithm
- But we will carry out an example

DP FOR THE MATRIX CHAIN PROBLEM

-- RUNNING AN EXAMPLE: COMPUTING THE RR AS A TABLE --

- Take $n = 4, A_1$ is $3 \times 5, A_2$ is $5 \times 7, A_3$ is $7 \times 3,$ and A_4 *is* $3 \times 4,$ that is, $p_1 = 3, p_2 = 5, p_3 = 7, p_4 = 3, p_5 = 4$
- We will build a table to compute the M_{ij}'s bottom up. For each M_{ij}, we record the k that gives M_{ij} its min value

$M_{11}=0$	$M_{22}=0$	$M_{33}=0$	$M_{44}=0$
$M_{12}=105$	$M_{23}=105$	$M_{34}=84$	
$M_{13}=150$ k=1	$M_{24}=165$ k=3		
$M_{14}=186$ k=3			

- For the 2[nd] row, every $M_{i,i+1} = p_i p_{i+1} p_{i+2}$ (why?), so compute it quickly
- For the 3[rd] row, let's illustrate the computing of M_{13}:

$$M_{13} = \min_{1 \le k \le 3-1} (M_{1k} + M_{k+1,3} + p_1 p_{k+1} p_4)$$

$$M_{13} = \min(M_{11} + M_{23} + p_1 p_2 p_4, M_{12} + M_{33} + p_1 p_3 p_4)$$

$$M_{13} = \min(0 + 105 + 3 \times 5 \times 3, 105 + 0 + 3 \times 7 \times 3)$$

$M_{13} = \min(150, 168) = 150$, from the 1[st] k, i.e., k=1

RUNNING AN EXAMPLE
-- CONSTRUCTING THE ACTUAL OPTIMAL SOLUTION --

- $A_1$is $3 \times 5, A_2$ is $5 \times 7, A_3$ is $7 \times 3,$ and A_4 *is* $3 \times 4,$

  $p_1 = 3, p_2 = 5, p_3 = 7, p_4 = 3, p_5 = 4$

- Use the table to construct the optimal solution

M_{11}=0	**M_{22}=0**	**M_{33}=0**	**M_{44}=0**
M_{12}=105	M_{23}=105	M_{34}=84	
M_{13}=150 k=1	M_{24}=165 k=3		
M_{14}=186 k=3			

Construction of the actual optimal solution:

- Since for M_{14} the optimal k=3, the best splitting for $A_1A_2A_3A_4$ is $(A_1A_2A_3)A_4$
- Now, to find the best split of $A_1A_2A_3$, look for the best k that minimizes M_{13}: it is k=1
- Therefore, the best splitting of $A_1A_2A_3$ is $A_1(A_2A_3)$
- Hence, the optimal solution is: $(A_1(A_2A_3))A_4$

LESSONS LEARNED SO FAR

- DP is an optimization design technique
- It has a litmus test (the principle of optimality): DP applies if OOP holds
- Proof of the POO is almost always by contradiction:
    1. assume a subsolution is not optimal, so there is a better subsolution
    2. replace it by a better one
    3. This results in a new solution better than optimal, a contradiction
- Solving the recurrence relation is non-recursive, bottom up, from smallest subsolutions to the final solution, where the subsolutions are usually recorded by filling a table
- The actual optimal solution is constructed by using the optimal split points recorded in the table
- More lessons later

MORE TO COME

- Next lecture we complete our coverage of Dynamic programming
- Specifically, we use DP to solve two very serious problems:
  - The all-pairs shortest path problem
  - The optimal binary search tree problem