IS703:
Decision Support and Optimization

Week 3: Dynamic Programming & Greedy Method

Lau Hoong Chuin
School of Information Systems

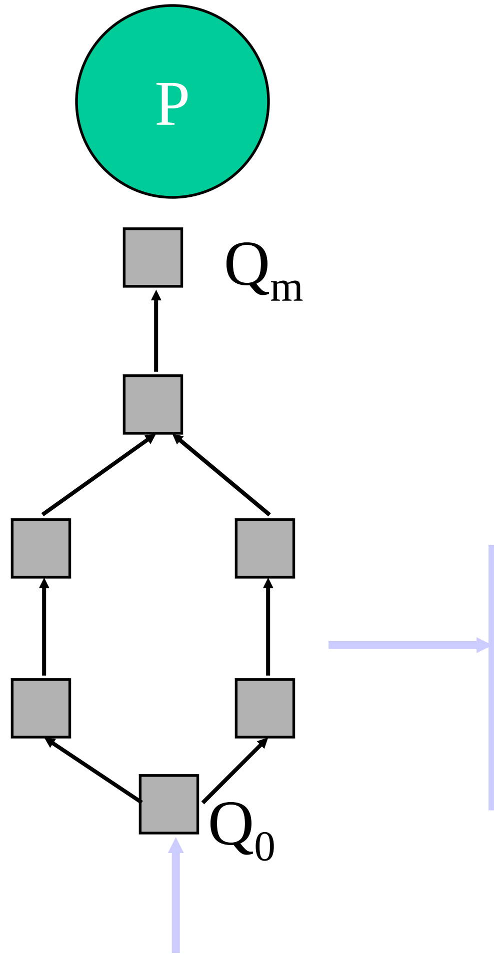
Dynamic Programming

- Richard Bellman coined the term dynamic programming in 1957
- Solves problems by combining the solutions to sub-problems that contain common sub-sub-problems.
- Difference between DP and Divide-and-Conquer:
  - Using Divide and Conquer to solve these problems is inefficient as the same common sub-sub-problems have to be solved many times.
  - DP will solve each of them once and their answers are stored in a table for future reference.

Intuitive Explanation

- Optimization Problem
  - Many solutions, each solution has a (objective) value
  - The goal is to find a solution with the optimal value
  - Minimization problems: e.g. Shortest path
  - Maximization problems: e.g. Tour planning
- Given a problem P, obtain a sequence of problems $Q_0, Q_1, \ldots, Q_m$, where:
  - You have a solution to Q_0
  - The solution to a problem Q_j, $j > 0$, can be obtained from solutions to problems Q_k, $k < j$, that appear earlier in the "sequence".

Intuitive Explanation



Find a way to compute the solution to Q_j from the solutions to Q_k $(k<j)$

You know how to compute solution to Q_0

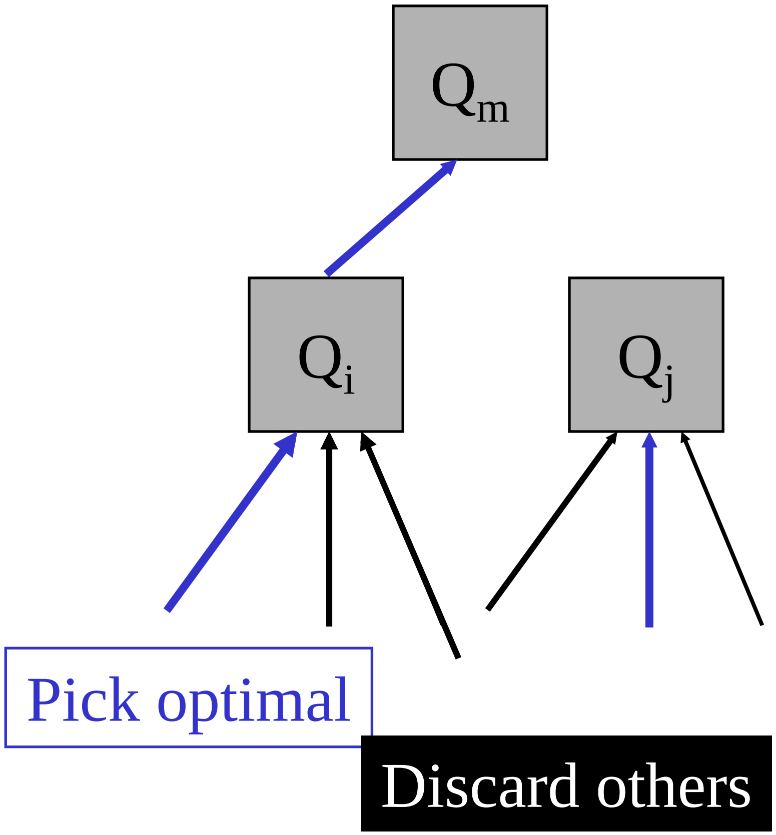
Elements of Dynamic Programming

DP is used to solve problems with the following characteristics:

- Optimal sub-structure (Principle of Optimality)
  - an optimal solution to the problem contains within it *optimal* solutions to sub-problems.
- Overlapping subproblems
  - there exist some places where we solve the same subproblem more than once

Optimal Sub-structure

Bellman's optimality principle

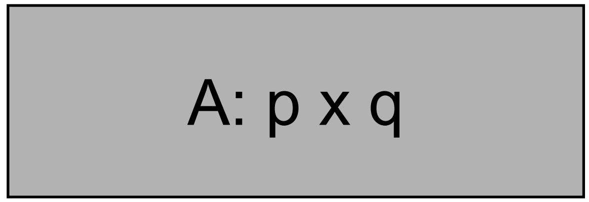


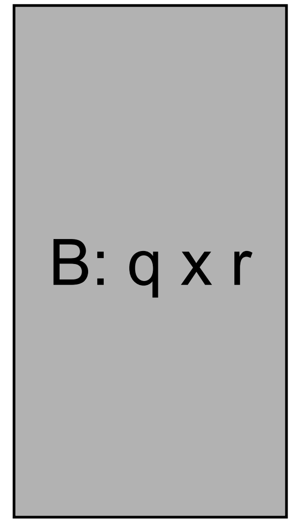
The discarded solutions for the smaller problem remain discarded because the optimal solution dominates them.

Steps to Designing a Dynamic Programming Algorithm

1. Characterize optimal sub-structure
2. Recursively define the value of an optimal solution
3. Compute the value bottom up
4. (if needed) Construct an optimal solution

Review: Matrix Multiplication






```
Matrix-Multiply(A,B):
    if columns[A] != rows[B] then
        error "incompatible dimensions"
    else for i = 1 to rows[A] do
        for j = 1 to columns[B] do
            C[i,j] = 0
            for k = 1 to columns[A] do
                C[i,j] = C[i,j]+A[i,k]*B[k,j]
    return C
```

Time complexity = O(pqr), where |A|=pxq and |B|=qxr

Matrix Chain Multiplication (**MCM**) Problem

Input: Matrices $A_1, A_2, \ldots, A_n$, each A_i of size $p_{i-1} x p_i$,

Output: Fully parenthesised product $A_1 A_2 \ldots A_n$ that minimizes the number of scalar multiplications.

A product of matrices is fully parenthesised if it is either

a) a single matrix, or

b) the product of 2 fully parenthesised matrix products surrounded by parentheses.

Example: $A_1 A_2 A_3 A_4$ can be fully parenthesised as:

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

Note: Matrix multiplication is associative

Matrix Chain Multiplication Problem

Example: 3 matrices:

A_1 : 10x100

A_2 : 100x5

A_3 : 5x50

Q: What is the cost of multiplying matrices of these sizes?

For $((A_1A_2)A_3)$,

number of multiplications = 10x100x5 + 10x5x50 = 7500

For $(A_1(A_2A_3))$, it is 75000

Matrix Chain Multiplication Problem

Let the number of different parenthesizations be $P(n)$. Then

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$

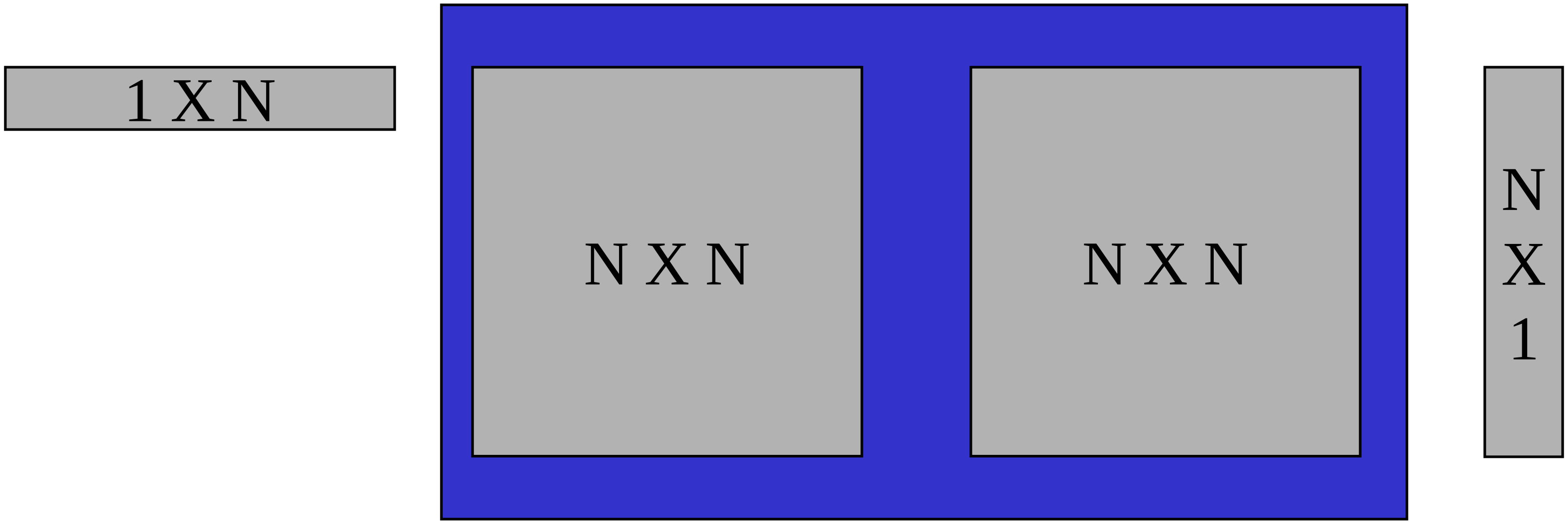
Using generating function, we have

$P(n)=C(n\text{-}1)$, the n-1[th] Catalan number where

$$C(n) = 1/(n+1)C_n^{2n} = \Omega(4^n / n^{3/2})$$

Exhaustively checking all possible parenthesizations take exponential time!

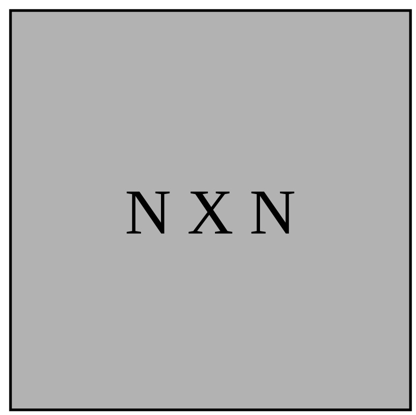
Parenthesization



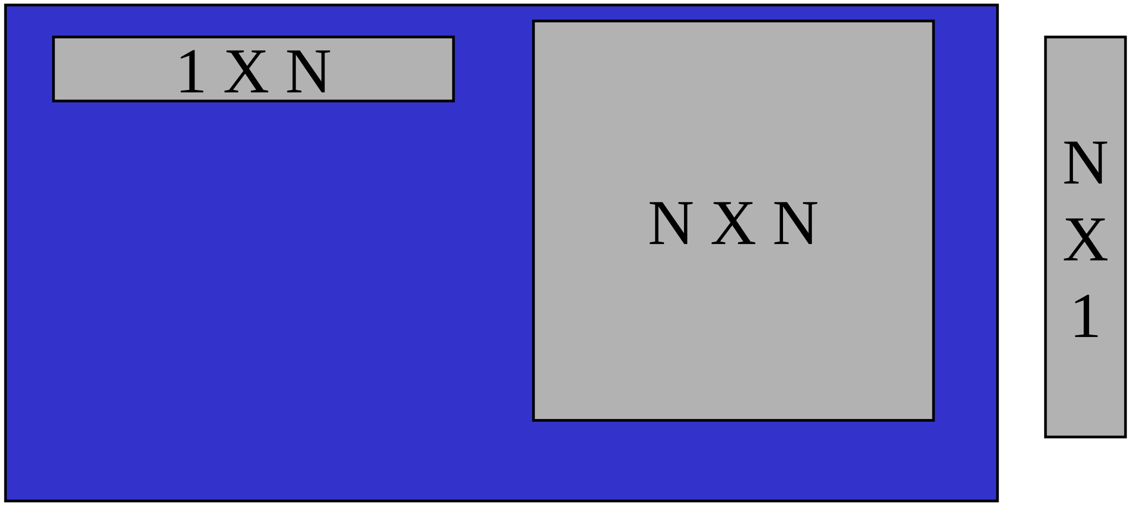
If we multiply these matrices first the cost is $2N^3$

(N^3 multiplications and N^3 additions).

Resulting matrix



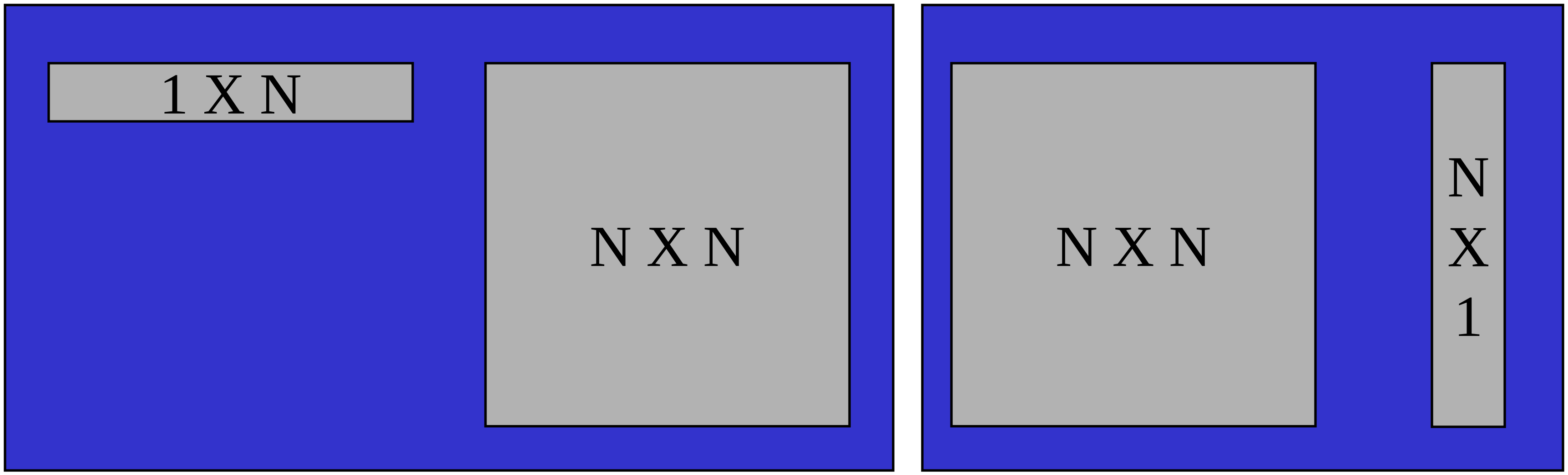
Parenthesization



Cost of multiplication is N^2.

Thus, total cost is proportional to $N^3 + N^2 + N$ if we parenthesize the expression in this way.

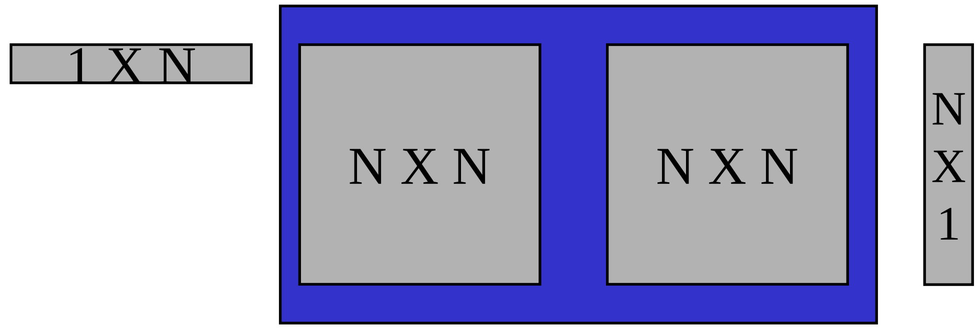
Different Ordering



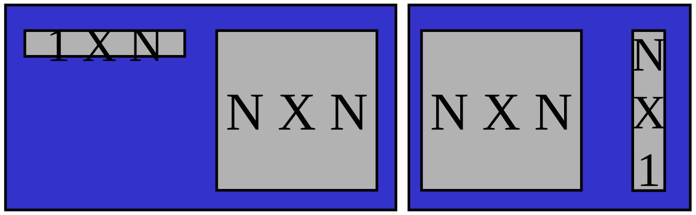
Cost is proportional to N^2

The Ordering Matters!

One ordering costs $O(N^3)$



The other ordering costs $O(N^2)$



Cost depends on parameters of the operands.

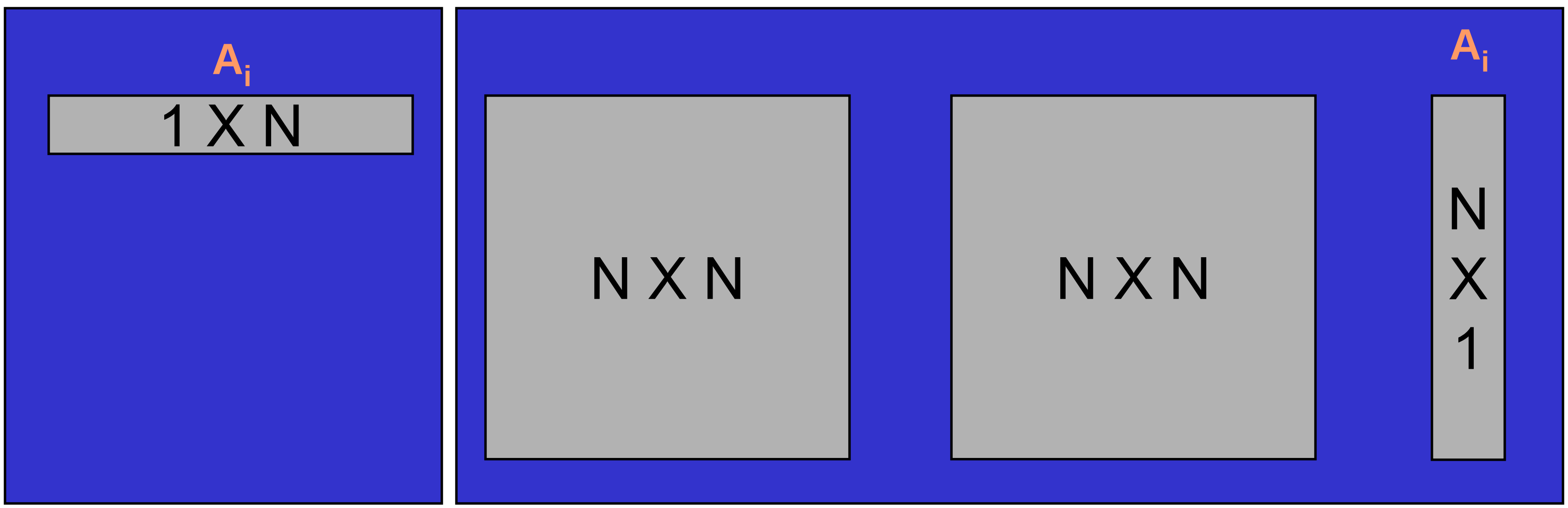
How to parenthesize to minimize total cost?

Step 1: Characterize Optimal Sub-structure

Let $A_{i..j}$ (i<j) denote the result of multiplying $A_i A_{i+1} \ldots A_j$.

$A_{i..j}$ can be obtained by splitting it into $A_{i..k}$ and $A_{k+1..j}$ and then multiplying the sub-products.

There are j-i possible splits (i.e. k=i,…, j-1)

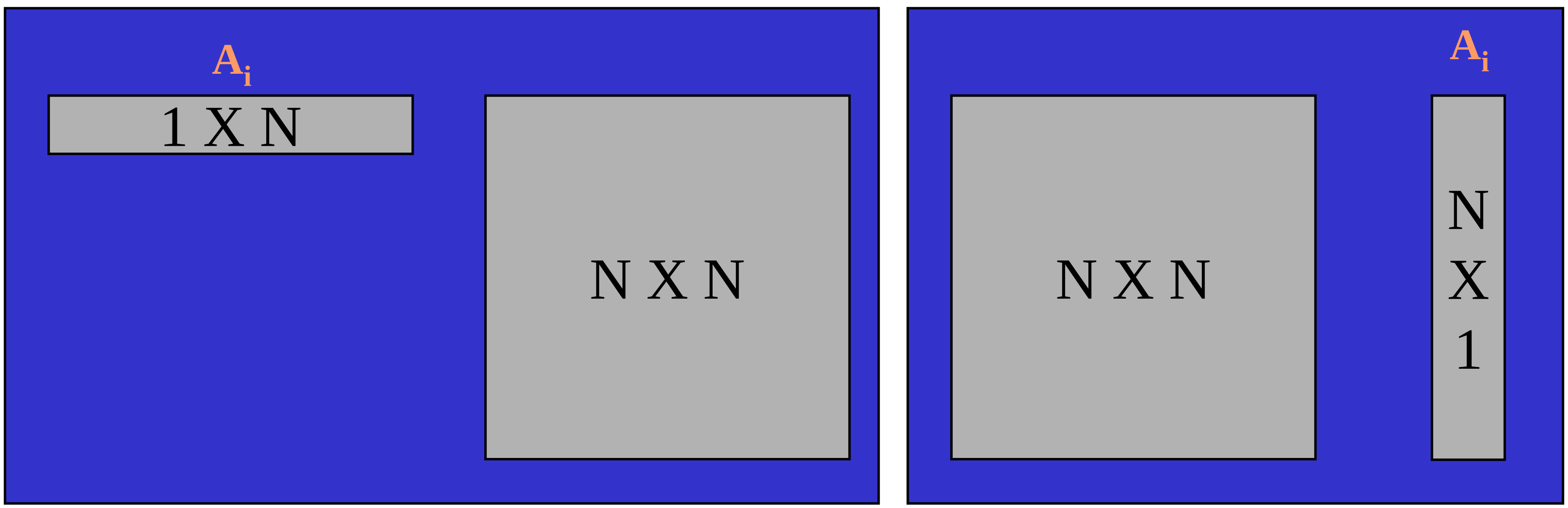


Step 1: Characterize Optimal Sub-structure

Let $A_{i..j}$ (i<j) denote the result of multiplying $A_iA_{i+1}\ldots A_j$.

$A_{i..j}$ can be obtained by splitting it into $A_{i..k}$ and $A_{k+1..j}$ and then multiplying the sub-products.

There are j-i possible splits (i.e. k=i,…, j-1)

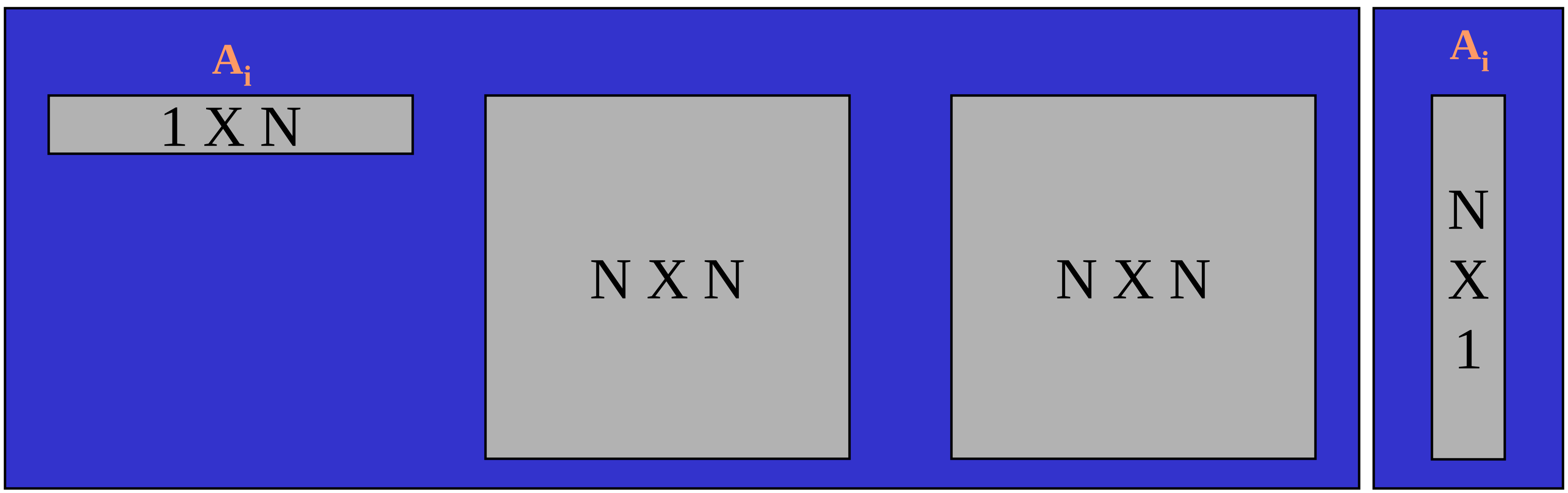


Step 1: Characterize Optimal Sub-structure

Let $A_{i..j}$ (i<j) denote the result of multiplying $A_i A_{i+1} \ldots A_j$.

$A_{i..j}$ can be obtained by splitting it into $A_{i..k}$ and $A_{k+1..j}$ and then multiplying the sub-products.

There are j-i possible splits (i.e. k=i,…, j-1)



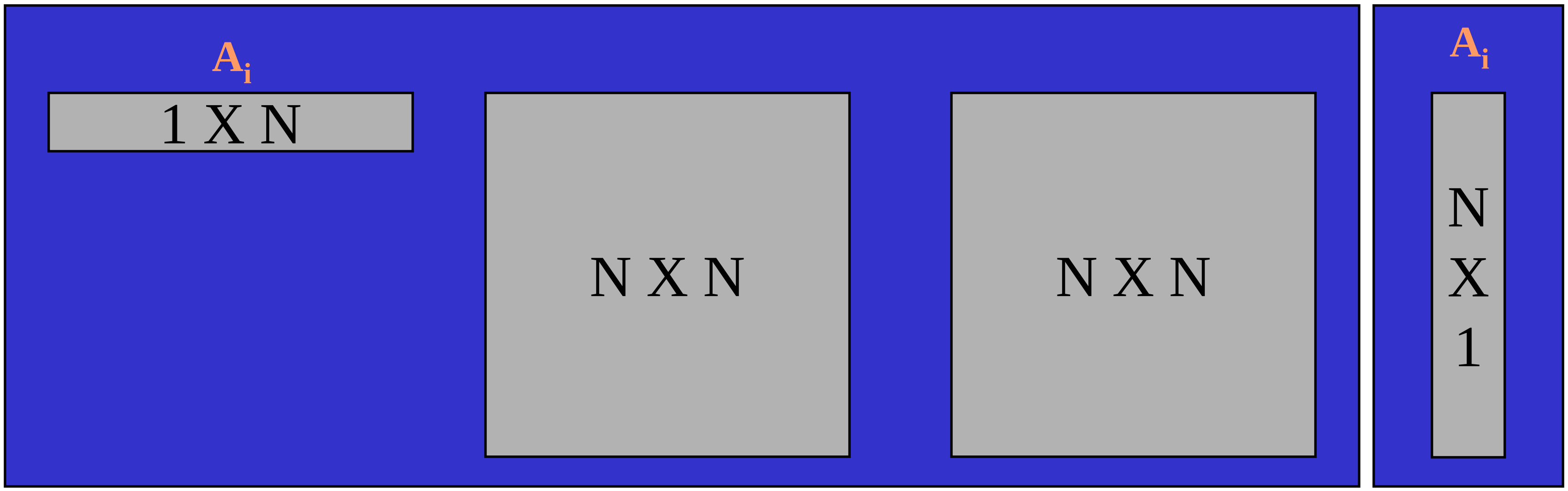
Step 1: Characterize Optimal Sub-structure

Within the optimal parenthesization of $A_{i..j}$,

(a) the parenthesization of $A_{i..k}$ must be optimal

(b) the parenthesization of $A_{k+1..j}$ must be optimal

Why?



Step 2: Recursive (Recurrence) Formulation

Need to find $A_{1..n}$

Let m[i,j] = min # of scalar multiplications needed to compute $A_{i..j}$

Since $A_{i..j}$ can be obtained by breaking it into $A_{i..k}$ $A_{k+1..j}$, we have

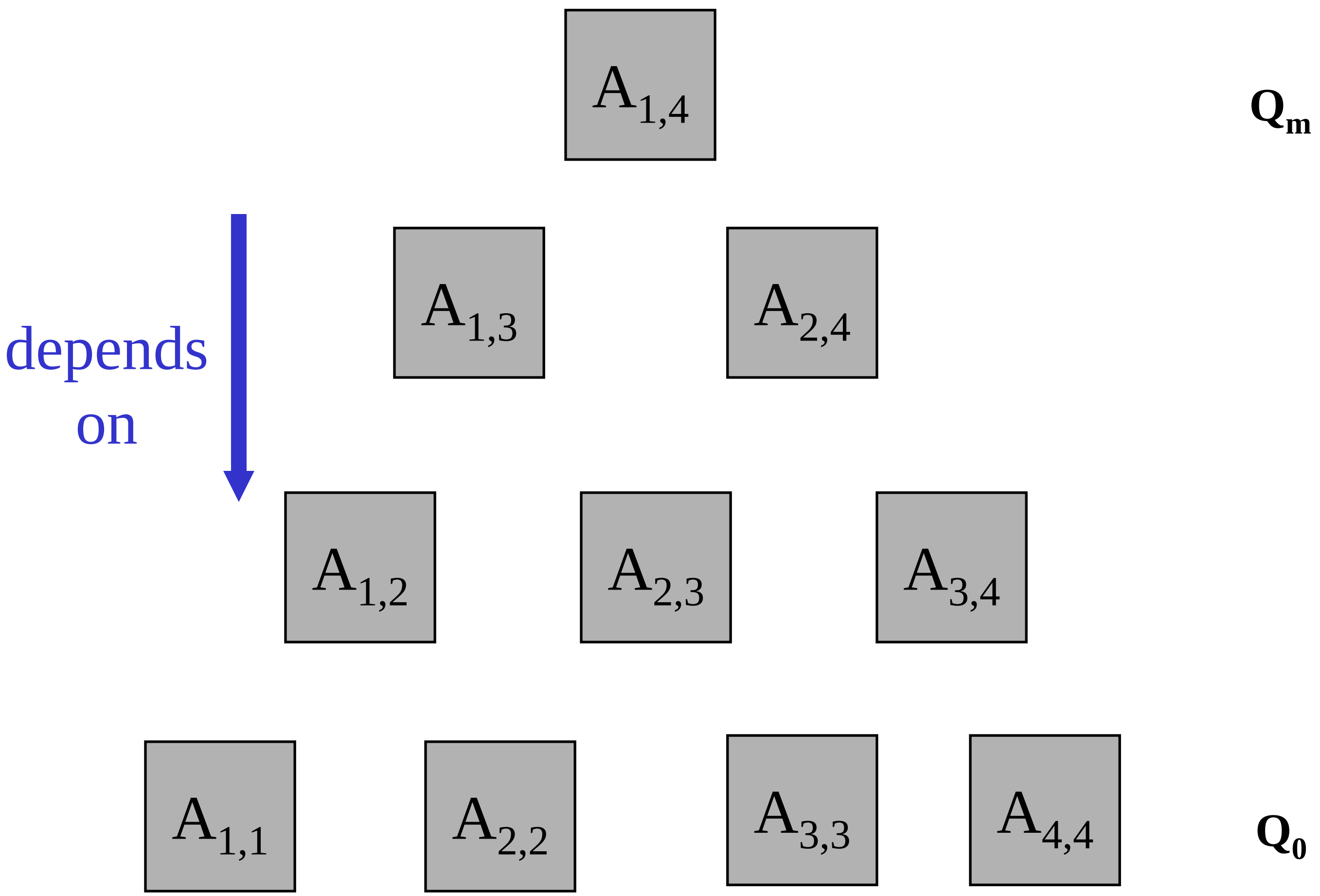
$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j \end{cases}$$

Note: The sizes of $A_{i..k}$ is p_{i-1} p_k, $A_{k+1..j}$ is p_k p_j , and

$A_{i..k}$ $A_{k+1..j}$ is p_{i-1} p_j after p_{i-1} p_k p_j scalar multiplications.

Let s[i,j] be the value k where the optimal split occurs

Step 3: Computing the Optimal Costs



Step 3: Computing the Optimal Costs

Matrix-Chain-Order(p)

```
n = length[p]-1 //p is the array of matrix sizes
for i = 1 to n do
    m[i,i] = 0  // no multiplication for 1 matrix
for len = 2 to n do // len is length of sub-chain
  for i = 1 to n-len+1 do // i: start of sub-chain
    j = i+len-1           // j: end of sub-chain
    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
```

Time complexity = $O(n^3)$

Example

Solve the following MCM instance:

Matrix	Dimension
A_1	30x35
A_2	35x15
A_3	15x5
A_4	5x10
A_5	10x20
A_6	20x25

p=[30,35,15,5,10,20,25]

See CLRS Figure 15.3

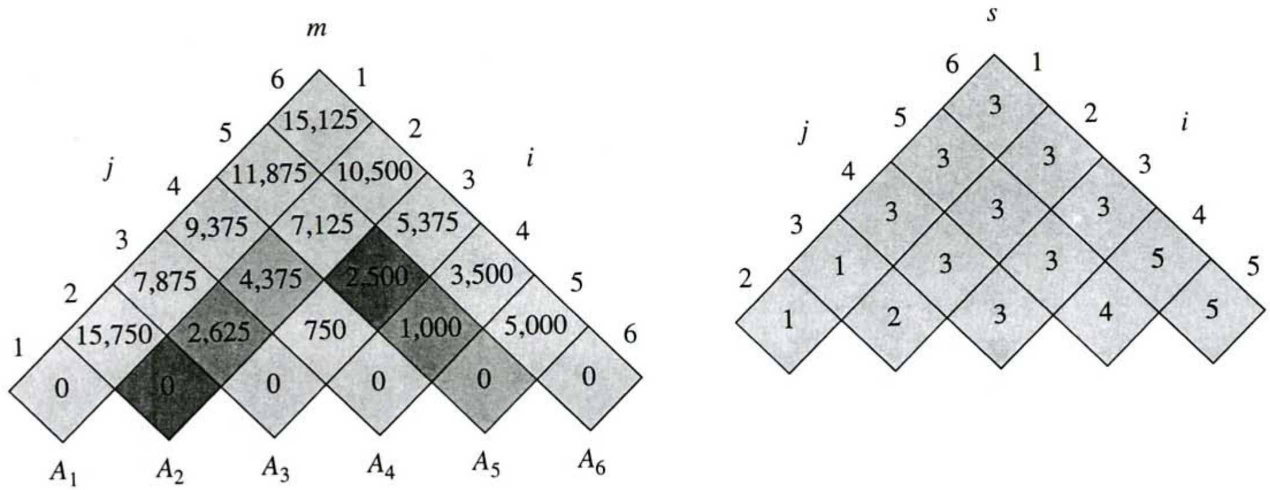


Figure 15.3 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

The tables are rotated so that the main diagonal runs horizontally. Only the main diagonal and upper triangle are used in the m table, and only the upper triangle is used in the s table. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1, 6] = 15{,}125$. Of the darker entries, the pairs that have the same shading are taken together in line 9 when computing

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 & = 13000 , \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 & = 7125 , \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 & = 11375 \end{cases}$$

$$= 7125 .$$

$\therefore S[2,5] = 3$

Step 4: Constructing an Optimal Solution

To get the optimal solution $A_{1..6}$, s[ ] is used as follows:

$A_{1..6}$

$= (A_{1..3}\ A_{4..6})$ since s[1,6] = 3

$= ((A_{1..1}\ A_{2..3})(A_{4..5}\ A_{6..6}))$ since s[1,3] =1 and s[4,6]=5

$=((A_1\ (A_2\ A_3))((A_4\ A_5)A_6))$

MCM can be solved in $O(n^3)$ time

Recap: Elements of Dynamic Programming

DP is used to solve problems with the following characteristics:

- Optimal substructure (Principle of Optimality)
  - Example. In MCM, $A_{1..6} = A_{1..3} A_{4..6}$
- Overlapping subproblems
  - there exist some places where we solve the same subproblem more than once
  - Example. In MCM, $A_{2..3}$ is common to the sub-problems $A_{1..3}$ and $A_{2..4}$
  - Effort wasted in solving common sub-problems repeatedly

Overlapping Subproblems

```
Recursive-Matrix-Chain(p,i,j)

if i = j
   then return 0
m[i,j] = ∞
for k = i to j-1 do
   q = Recursive-Matrix-Chain(p,i,k)+
       Recursive-Matrix-Chain(p,k,j)+p_{i-1}p_kp_j
   if q < m[i,j]
     then m[i,j] = q
return m[i,j]
```

See CLRS Figure 15.5

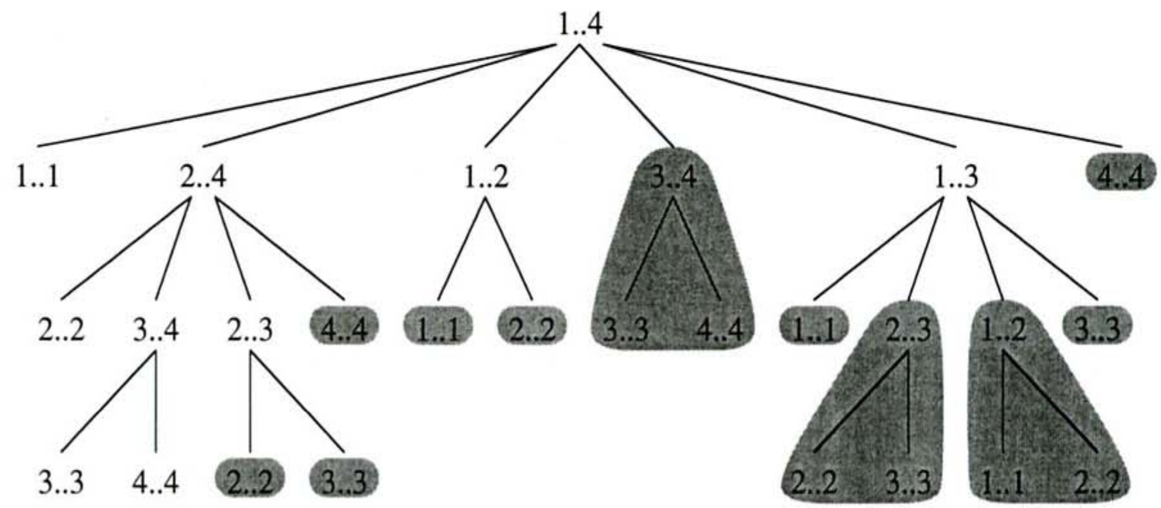


Figure 15.5 The recursion tree for the computation of RECURSIVE-MATRIX-CHAIN(p, 1, 4). Each node contains the parameters i and j. The computations performed in a shaded subtree are replaced by a single table lookup in MEMOIZED-MATRIX-CHAIN(p, 1, 4).

Overlapping Subproblems

Let T(n) be the time complexity of
`Recursive-Matrix-Chain(p,1,n)`

For n > 1, we have

$$T(n) = 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + \underline{1})$$

a) 1 is used to cover the cost of lines 1-3, and 8

b) <u>1</u> is used to cover the cost of lines 6-7

Using substitution, we can show that $T(n) \geq 2^{n-1}$

Hence $T(n) = \Omega(2^n)$

Memoization

- *Memoization* is one way to deal with overlapping subproblems
  - After computing the solution to a subproblem, store it in a table
  - Subsequent calls just do a table lookup
- Can modify recursive algo to use memoziation

Memoization

Memoized-Matrix-Chain(p) // Compare with Matrix-Chain-Order

```
Memoized-Matrix-Chain(p)
      n = length[p] - 1
      for i = 1 to n do
          for j = i to n do
              m[i,j] = ∞
      return Lookup-Chain(p,1,n)
```

```
Lookup-Chain(p,i,j)
   if m[i,j]< ∞  // m[i,j] has been computed
   then return m[i,j]
   if i = j       // only one matrix
      then m[i,j] = 0
      else for k = i to j - 1 do
         q = Lookup-Chain(p,i,k) +
               Lookup-Chain(p,k+1,j) + p_{i-1}p_kp_j
         if q < m[i,j]
            then m[i,j] = q
   return m[i,j]
```

Time complexity: $O(n^3)$ Why?

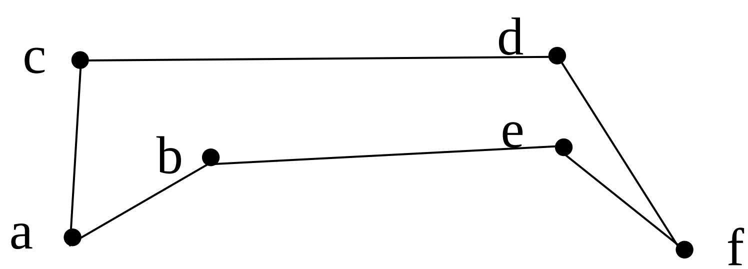
Example: Traveling Salesman Problem

Given: A set of *n* cities V={$x_1, x_2, \ldots, x_n$} and distance matrix c, containing cost to travel between cities, find a minimum-cost tour.

- David Applegate, Robert Bixby, Vašek Chvátal, William Cook (http://www.math.princeton.edu/tsp/)
- Exhaustive search:
  - Find optimal tour by systematically examining all tours
  - enumerate all permutations of the cities and evaluate tour (given by particular vertex order)
  - Keep track of shortest tour
  - (n-1)! permutations, each takes O(n) time to evaluate
    - Don’t look at all n permutations, since we don’t care about starting point of tour: A,B,C,(A) is same tour as C,A,B,(C)
  - Unacceptable for large n

TSP

- Let S ={x_1, x_2, …, x_k} be a subset of the vertices in V
- A path P from v to w covers S if P=[v, x_1, x_2, … , x_k, w], where x_i may appear in any order but each must appear exactly once
- Example, path from a to a, covering {c, d, f, e, b}



Dynamic Programming

- Let d(v, w, S) be cost of shortest path from v to w covering S
- Need to find d(v, v, V-{v})
- Recurrence relation:

$$d(v, w, S) = \begin{cases} c(v, w) & \text{if } S=\{\} \\ \min_{\forall x} \big(c(v,x) + d(x, w, S-\{x\})\big) & \text{otherwise} \end{cases}$$

- Solve all subproblems where |S|=0, |S|=1, etc.
- How many subproblems d(x, y, S) are there? $(n-1)2^{n-1}$
  - S could be any of the 2^{n-1} distinct subsets of n-1 vertices
- Takes O(n) time to compute each d(v, w, S)

Dynamic Programming

- Total time $O(n^2 2^{n-1})$
- Much faster than O(n!)
- Example:
  - n=1, algorithm takes 1 micro sec.
  - n=20, running time about 3 minutes (vs. 1 million years)

Summary

- DP is suitable for problems with:
  - Optimal substructure: optimal solution to problem consists of optimal solutions to subproblems
  - Overlapping subproblems: few subproblems in total, many recurring instances of each
- Solve bottom-up, building a table of solved subproblems that are used to solve larger ones
- Dynamic Programming applications

Exercise (Knapsack Problem)

- You are the ops manager of an equipment which can be used to process one job at a time
- There are a set of jobs, each incurs a processing cost (weight) and reaps an associated profit (value), all numbers are non-negative integers
- Jobs may be processed in any order
- Your equipment has a processing capacity
- Question: What jobs should you take to maximize the profit?

Exercise (Knapsack Problem)

Design a dynamic programming algorithm to solve the Knapsack Problem.

Your algorithm should run in O(nW) time, where n is the number of jobs and W is the processing capacity.

Greedy Algorithms

Reference:

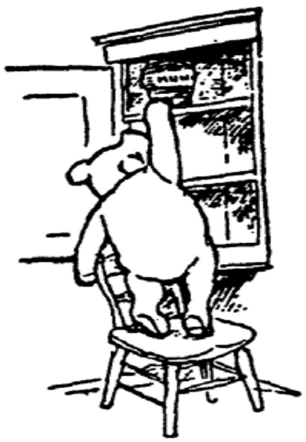
- CLRS Chapters 16.1-16.3, 23

Objectives:

- To learn the Greedy algorithmic paradigm
- To apply Greedy methods to solve several optimization problems
- To analyse the correctness of Greedy algorithms

Greedy Algorithms

- Key idea: Makes the choice that looks best at the moment
  - The hope: a locally optimal choice will lead to a globally optimal solution
- Everyday examples:
  - Driving
  - Shopping



Applications of Greedy Algorithms

- Scheduling
  - Activity Selection (Chap 16.1)
  - Scheduling of unit-time tasks with deadlines on single processor (Chap. 16.5)
- Graph Algorithms
  - Minimum Spanning Trees (Chap 23)
  - Dijkstra’s (shortest path) Algorithm (Chap 24)
- Other Combinatorial Optimization Problems
  - Knapsack (Chap 16.2)
  - Traveling Salesman (Chap 35.2)
  - Set-covering (Chap 35.3)

Greedy vs Dynamic

- Dynamic Programming
  - Bottom up (while Greedy is top-down)
- Dynamic programming can be overkill; greedy algorithms tend to be easier to code

Real-World Applications

- Get your $$ worth out of a carnival
  - Buy a passport that lets you onto any ride
  - Lots of rides, each starting and ending at different times
  - Your goal: ride as many rides as possible
- Tour planning
- Customer satisfaction planning
- Room scheduling

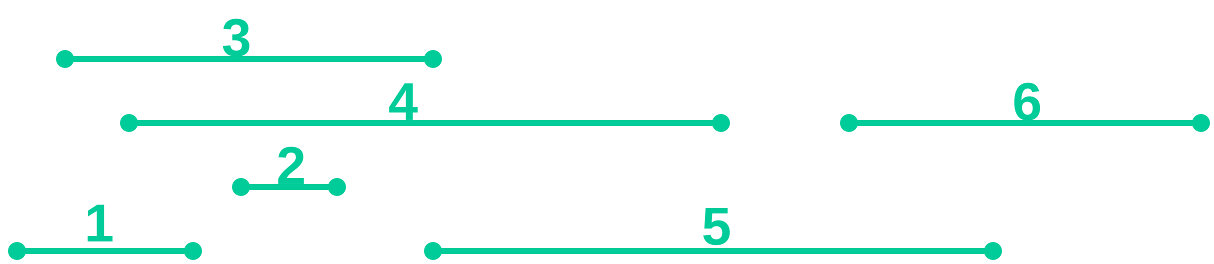
Application: Activity-Selection Problem

- Input: a list S of n activities = $\{a_1, a_2, \ldots, a_n\}$

  s_i = start time of activity i

  f_i = finish time of activity i

  S is sorted by finish time, i.e. $f_1 \leq f_2 \leq \ldots \leq f_n$
- Output: a subset A of compatible activities of maximum size
  - Activities are compatible if $[s_i, f_i) \cap [s_j, f_j)$ is null

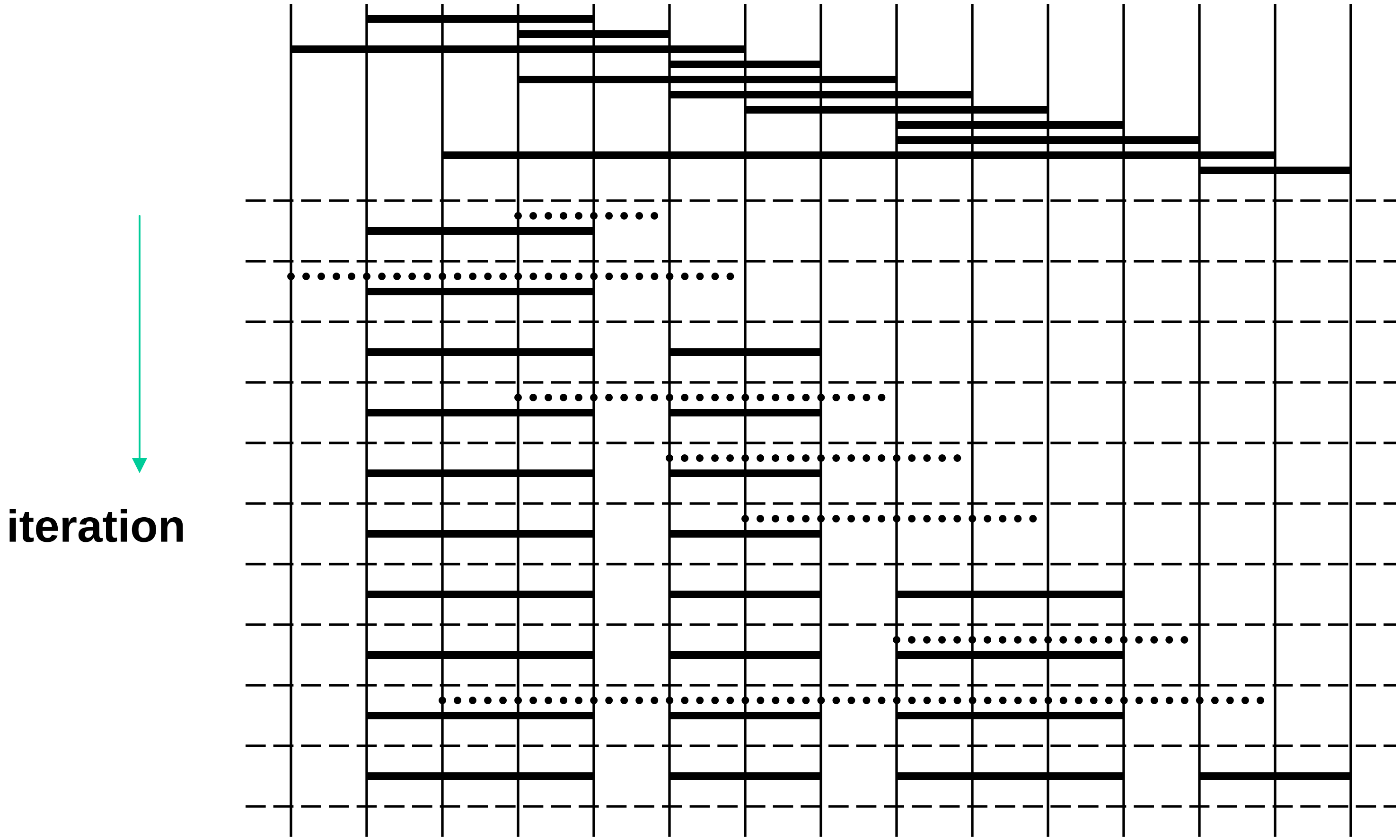


How many possible solutions are there?

Greedy Algorithm

```
Greedy-Activity-Selection(s,f)
n := length[s]
A := {a_1}
j := 1
for k:=2 to n do
  if s_k >= f_j  // compatible activity
  then A := A ∪ {a_k}
       j := k
Return A
```

Example Run



When does Greedy Work?

- Two key ingredients:

1. **Optimal sub-structure**

   An optimal solution to the entire problem contains within it optimal solutions to subproblems (this is also true of dynamic programming)

2. **Greedy choice property**

- Greedy choice + Optimal sub-structure establish the correctness of the greedy algorithm

Optimal Sub-structure

Let A be an optimal solution to problem with input S. Let a_k be the activity in A with the earliest finish time. Then $A - \{a_k\}$ is an optimal solution to the subproblem with input $S' = \{i \in S: s_i \geq f_k\}$

– In other words: the optimal solution S contains within it an optimal solution for the sub-problem on activities that are compatible with a_k

Proof by Contradiction (Cut-and-Paste Argument):

Suppose $A - \{a_k\}$ is not optimal to S'.

Then, $\exists$ optimal solution B to S' with $|B| > |A - \{a_k\}|$,

Clearly, $B \cup \{a_k\}$ is a solution for S.

But, $|B \cup \{a_k\}| > |A|$ (Contradiction)

Greedy Choice Property

- Locally optimal choice
  - Make best choice available at a given moment

- Locally optimal choice $\Rightarrow$ globally optimal solution
  - In other words, the greedy choice is always safe
  - How to prove? Use Exchange Argument usually.

- Contrast with dynamic programming
  - Choice at a given step may depend on solutions to subproblems (bottom-up)

Greedy Choice Property

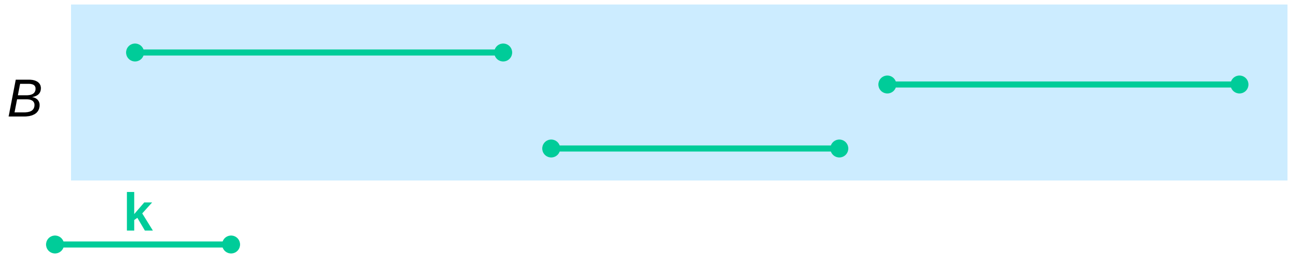
- Theorem: (paraphrased from CLRS Theorem 16.1)

  Let a_k be a compatible activity with the earliest finish time. Then, there exists an optimal solution that contains a_k.

- Proof by Exchange Argument:

  For any optimal solution *B* that does not contain a_k,

  we can always replace first activity in *B* with a_k (*Why?*). Same number of activities, thus optimal.



Application: Knapsack Problem

- Recall *0-1 Knapsack problem*:
  - choose among n items, where the ith item worth v_i dollars and weighs w_i pounds
  - knapsack carries at most W pounds
  - maximize value
    - Note: assume v_i, w_i, and W are all integers
    - “0-1”, since each item must be taken or left in entirety
  - solved by Dynamic Programming
- A variant - *Fractional Knapsack problem*:
  - can take fractions of items
  - can be solved by a Greedy algorithm

Knapsack Problem

- The optimal solution to the fractional knapsack problem can be found with a greedy algorithm
  - *How?*
- The optimal solution to the 0-1 problem cannot be found with the same greedy strategy
  - Proof by a counter example
  - Greedy strategy: take in order of dollars/kg
  - Example: 3 items weighing 10, 20, and 30 kg, knapsack can hold 50 kg
    - *Suppose item 2 is worth \$100. Assign values to the other items so that the greedy strategy will fail*

Knapsack Problem: Greedy vs Dynamic

- The fractional problem can be solved greedily
- The 0-1 problem cannot be solved with a greedy approach
  - It can, however, be solved with dynamic programming (recall previous lesson)

Summary

- Greedy algorithms works under:
  - Greedy choice property
  - Optimal sub-structure property
- Design of Greedy algorithms to solve:
  - Some scheduling problems
  - Fractional knapsack problem

Exercise (Traveling Salesman Problem)

Design a greedy algorithm to solve TSP.

Demonstrate that greedy fails by giving a counter example.

Exercise (Interval Coloring Problem)

Suppose that we have a set of activities to schedule among a large number of lecture halls. We wish to schedule *all* the activities using minimum number of lecture halls.

Give an efficient greedy algorithm to determine which activity should use which lecture hall.

Next Week

Read CLRS Chapters 22-26 (Graphs and Networks)

Do Assignment 2!