# CSC 317: Data Structures and Algorithm Analysis

Dilip Sarkar

Department of Computer Science
University of Miami



## Outline I

## Divide-and-Conquer or **Dynamic-programming**

- Divide-and-Conquer method
  - Partition the input into **disjoint** subsets and solve them (recursively)
  - Combine solutions of the subproblems
- If recursive calls share input data, then same problem is solved multiple time (we will see an example)
- **Dynamic programming method**
  - Solve problems in a bottom-up fashion and store the solution
  - Avoids solving same subproblems repeatedly
  - Find **multiple solution for same input-size**, and store the best solutions
- Typically dynamic-programming is applied to **optimization problems**



## Steps for developing dynamic-programming algorithm

**Characterize the structure of an optimal solution**

1. Recursively define the value of an optimal solution
2. Compute the value of an optimal solution, typically in a bottom-up fashion
3. Compute an optimal solution from previously computed solutions to smaller problems

   Steps 1 to 3 form the basis of a dynamic-programming solution to a problem

   These three steps are fine to know value of an optimal solution
4. Need a 4th step for reconstructing an optimal solution

# Rod Cutting Problem

Given a rod of length $n$ and price $p_k$, of a rod length $k$.

Price table for rods.

| Length $l_i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Price $p_i$ | 1 | 5 | 8 | 9 | 10 | 17 | 17 | 20 | 24 | 30 |

How many cuts are possible?

$2^{10-1}$,

Because there are $(10-1)$ locations and we can choose to cut or not to cut

- Example for a rod of length 4

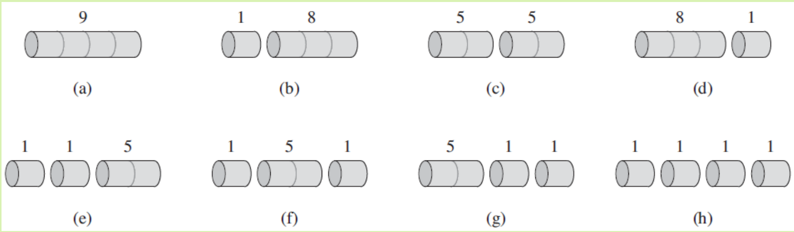



Figure: Possible cuts

- From the figure above, we can see that maximum revenue is 10

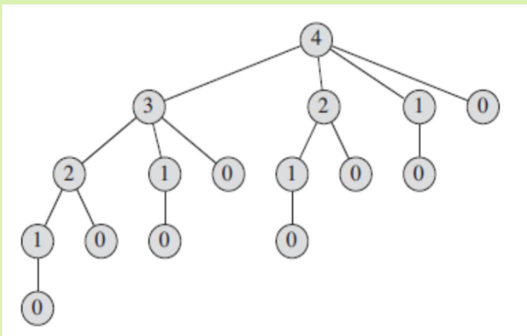


# Problem with top-down recursive approach

A top-down recursive algorithm

```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

Recursion tree



- We calculate max revenue for length 2 two times
- We calculate max revenue for length 1 four times
- We calculate max revenue for length 0 eight times
- We can **store max revenue for each length** in a table

# Bottom-up approach (dynamic programming)

A bottom-up algorithm using a table for optimal values

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        q = max(q, p[i] + r[j - i])
    r[j] = q
return r[n]
```

Trace of the bottom-up algorithm

| | *i* | 1 | 2 | 3 | 4 | 5 | Max revenue |
|---|---|---|---|---|---|---|---|
| j | | - | - | - | - | - | |
| 1a | | p(1)+r(0) | - | - | - | - | |
| 1b | | 1 | - | - | - | - | 1 |
| 2a | | p(1)+r(1) | p(2)+r(0) | - | - | - | |
| 2b | | 2 | 5 | - | - | - | 5 |
| 3a | | p(1)+r(2) | p(2)+r(1) | p(3)+r(0) | - | - | |
| 3b | | 6 | 6 | 8 | - | - | 8 |
| 4a | | p(1)+r(3) | p(2)+r(2) | p(3)+r(1) | p(4)+r(0) | - | |
| 4b | | 9 | 10 | 9 | 9 | - | 10 |
| 5a | | p(1)+r(4) | p(2)+r(3) | p(3)+r(2) | p(4)+r(1) | p(5)+r(0) | |
| 5b | | 11 | 13 | 13 | 10 | 10 | 13 |



# Rod Cutting Problem: pseudo-code

A bottom-up algorithm using a table for optimal values and cut positions

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        if q < p[i] + r[j - i]
            q = p[i] + r[j - i]
            s[j] = i
    r[j] = q
return r and s
```

Figure: Array '*s*' stores locations of the cut.

## Complexity of matrix multiplication

Matrix multiplication algorithm

```
MATRIX-MULTIPLY(A, B)
if A.columns ≠ B.rows
    error "incompatible dimensions"
else let C be a new A.rows × B.columns matrix
    for i = 1 to A.rows
        for j = 1 to B.columns
            c_ij = 0
            for k = 1 to A.columns
                c_ij = c_ij + a_ik · b_kj
    return C
```

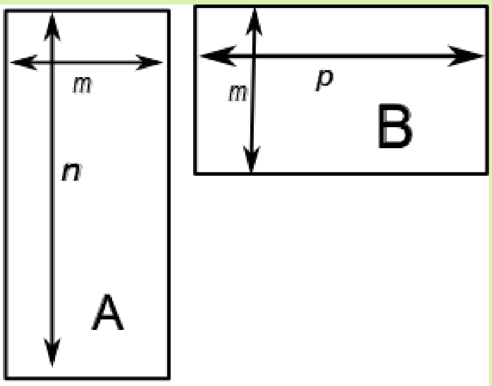



Figure: Matrix multiplication algorithm

If $A$ a $n$x$m$ matrix and $B$ is a $m$x$p$ matrix then

Number of operations $= O(mnp)$ and for $m = n = p$, we have $O(n^3)$.



## Matrix-chain multiplication

Let us consider four matrices:
$A_1, A_2, A_3$, and $A_4$
Let $A = A_1 \times A_2 \times A_3 \times A_4$

- How many ways can we compute $A$?
  - Recall the **Associative property** of matrix multiplication:
  - $A_1 \times (A_2 \times A_3) = (A_1 \times A_2) \times A_3$
- The answer is 5

$(A_1 \times (A_2 \times (A_3 \times A_4)))$,
$(A_1 \times ((A_2 \times A_3) \times A_4))$,
$((A_1 \times A_2) \times (A_3 \times A_4))$,
$((A_1 \times (A_2 \times A_3)) \times A_4)$, and
$(((A_1 \times A_2) \times A_3) \times A_4)$.

This relates to a counting problem: how many ways we can parenthesize a sequence of $n$ matrices?

$$C(n) = \sum_{k=1}^{n-1} C(k) \cdot C(n-k) = \frac{1}{n+1}\binom{2n}{n} = \Omega\left(\frac{4^n}{n^{3/2}}\right)$$

Too complex to find optimal order of multiplications exhaustively.

# Dynamic Programming formulation I

Let us develop a dynamic programming algorithm applying four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Computer the value of an optimal solution.
4. Construct an optimal solution from computed information.

Notations: $A = A_1 \times A_2 \times \cdots \times A_n$

Matrix $A_i$ is a $p_{i-1} \times p_i$ matrix.

Let $A_{i..j}$ be the product of sequence of matrix $(A_i, A_{i+1}, \cdots, A_j)$

Now let us split this sequence into two subsequences

$(A_i, A_{i+1}, \cdots, A_k)$ and $(A_{k+1}, A_{k+2}, \cdots, A_j)$

NOTE: $(A_i, A_{i+1}, \cdots, A_k)$ is a $p_{i-1} \times p_k$ matrix

NOTE: $(A_{k+1}, A_{k+2}, \cdots, A_j)$ is a $p_k \times p_j$ matrix

# of operations for $(A_i, A_{i+1}, \cdots, A_k) \times (A_{k+1}, A_{k+2}, \cdots, A_j)$?

$p_{i-1} p_k p_j$



# Dynamic Programming formulation II

Let us develop a dynamic programming algorithm applying four steps:

Let $A_{i..j}$ be the product of sequence of matrix $(A_i, A_{i+1}, \cdots, A_j)$

Now let us split this sequence into two subsequences

$(A_i, A_{i+1}, \cdots, A_k)$ and $(A_{k+1}, A_{k+2}, \cdots, A_j)$

- $\text{cost}(A_{i..j}) = \text{cost}(A_{i..k}) + \text{cost}(A_{k+1..j}) + \text{cost}(A_{i..k} \times A_{k+1..j})$

  Recall that the cost for the cost for $(A_{i..k} \times A_{k+1..j}) = p_{i-1} p_k p_j$
- Thus, $\text{cost}(A_{i..j}) = \text{cost}(A_{i..k}) + \text{cost}(A_{k+1..j}) + p_{i-1} p_k p_j$

  denoting $\text{cost}(A_{i..j})$ by $m([i, j])$, we have

  $\Rightarrow m([i, j]) = m(i, k) + m(k+1, j) + p_{i-1} p_k p_j$
- We want to find $k$ for which total cost is minimized

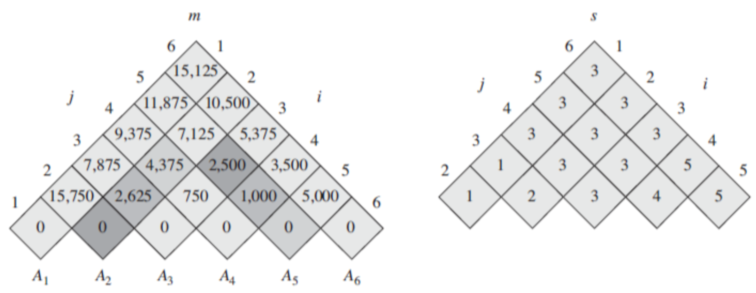
$$m([i, j]) = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \{ m([i, k] + m[k+1, j] + p_{i-1} p_k p_j & \text{if } i < j \end{cases}$$

## Example of matrix-chain multiplication I

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

Matrices for our example

| matrix | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ | $A_6$ |
|---|---|---|---|---|---|---|
| dimension | $30 \times 35$ | $35 \times 15$ | $15 \times 5$ | $5 \times 10$ | $10 \times 20$ | $20 \times 25$ |






## Pseudocode for the Dynamic Programming Algorithm

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

- $m$ stores the optimal cost
- $s$ stores location of partition
- Time complexity:

  initialization: lines 3 and 4

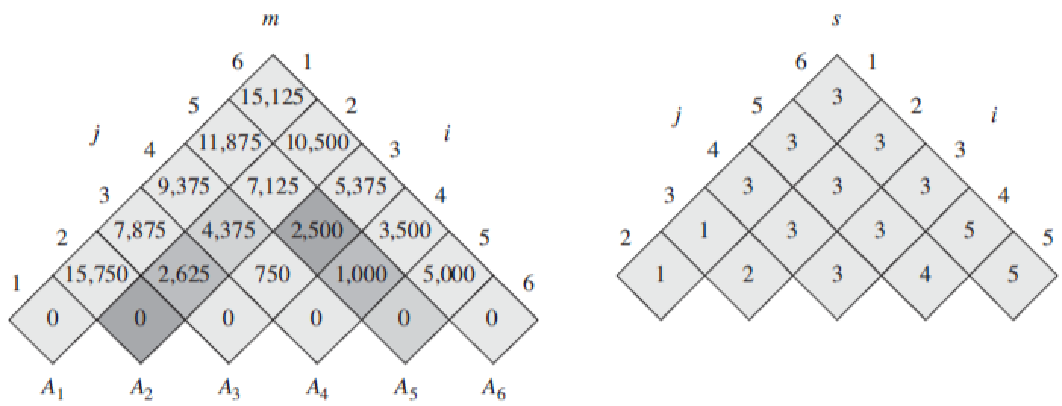
  Cost $n$.

  3 nested for loops

  Total is $O(n^3)$

# Example of matrix-chain multiplication I

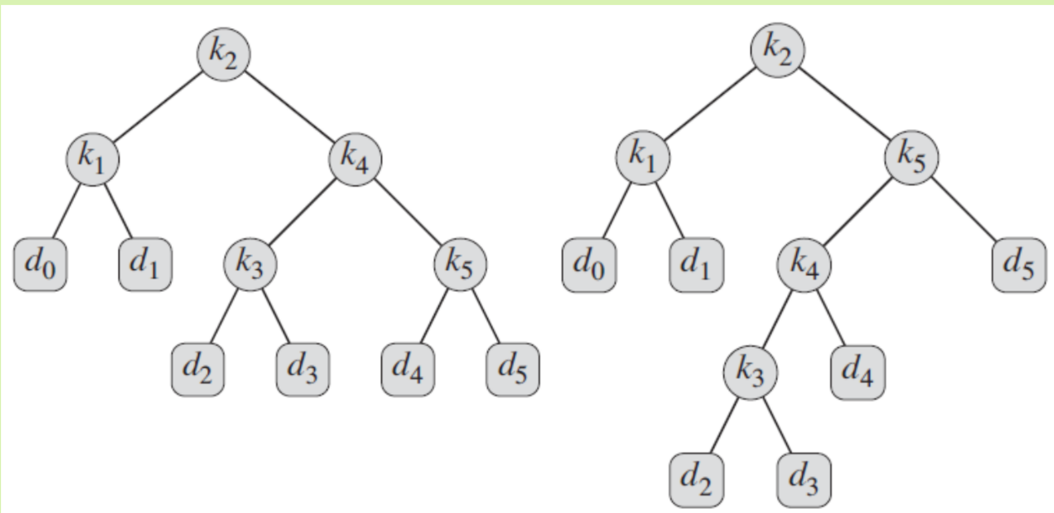
| matrix | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ | $A_6$ |
|---|---|---|---|---|---|---|
| dimension | $30 \times 35$ | $35 \times 15$ | $15 \times 5$ | $5 \times 10$ | $10 \times 20$ | $20 \times 25$ |




$$m([2,5]) = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13,000 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7,125 \\ m[2,4] + m[5,5] + p_1 p_5 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11,375 \end{cases}$$

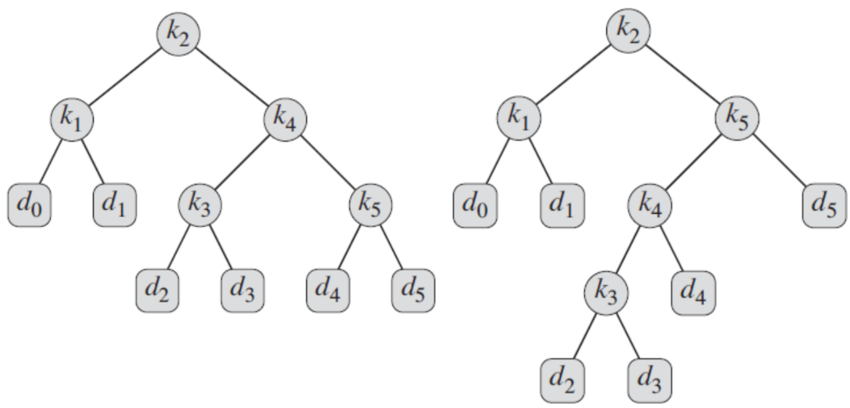


# Optimal BST when Keys have different search Probabilities I



$k_1$ to $k_5$ are five keys
$d_0$ to $d_5$ are 'dummy' nodes
A failed search ends at one the dummy nodes
Which is a better search tree?
It depends on the probability of search of keys
Let us see an example

# Optimal BST when Keys have different search Probabilities II




| $i$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| $p_i$ | | 0.15 | 0.10 | 0.05 | 0.10 | 0.20 |
| $q_i$ | 0.05 | 0.10 | 0.05 | 0.05 | 0.05 | 0.10 |

$p_i$ is the probability of searching $k_i$
$q_i$ is the probability of search ending at $d_i$, and $\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$
Expected search cost for tree to the left is 2.8
Expected search cost for tree to the right is 2.75



# Dynamic Programming Formulation

- Notations:
  - Let $e[i, j]$ be the cost of searching an optimal subtree with keys $k_i, k_{i+1}, \cdots, k_j$
  - Let $w[i, j]$ be the sum of probabilities of a subtree containing keys $k_i, k_{i+1}, \cdots, k_j$
  - That is, $w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$
- Computation of optimal cost:
  - A search tree can have any key from $k_i, k_{i+1}, \cdots, k_j$ as root.
  - To find an optimal tree we must consider all of them as potential root.
  - Let $k_l$ for $i \leq l \leq j$, be the root.
  - The left subtree will have keys $k_i$ to $k_{l-1}$ and the right subtree will have keys $k_{l+1}$ to $k_j$
  - Average cost for searching a tree rooted at $k_l$ is cost of searching left and right subtrees plus $(w(i, l) + w(l + 1, j) + p_l = w(i, j))$

For optimal cost

$e[i, j] = \min_{l=i}^{j}\{(e[i, l - 1] + w(i, l - 1)) + (e[l + 1, j] + w(l + 1, j)) + p_l\}$

That is, $e[i, j] = \min_{l=i}^{j}\{e[i, l - 1] + (e[l + 1, j] + w(i, j)\}$

# Dynamic Programming Algorithm for Optimal Binary Search Trees II

```
OPTIMAL-BST(p, q, n)
let e[1..n+1, 0..n], w[1..n+1, 0..n],
        and root[1..n, 1..n] be new tables
for i = 1 to n + 1
    e[i, i-1] = q_{i-1}
    w[i, i-1] = q_{i-1}
for l = 1 to n
    for i = 1 to n - l + 1
        j = i + l - 1
        e[i, j] = ∞
        w[i, j] = w[i, j-1] + p_j + q_j
        for r = i to j
            t = e[i, r-1] + e[r+1, j] + w[i, j]
            if t < e[i, j]
                e[i, j] = t
                root[i, j] = r
return e and root
```

Array $e$ stores optimal cost
Array $w$ stores weights of probabilities of the subtree
Array $r$ stores selected roots



# Optimal BST: An example

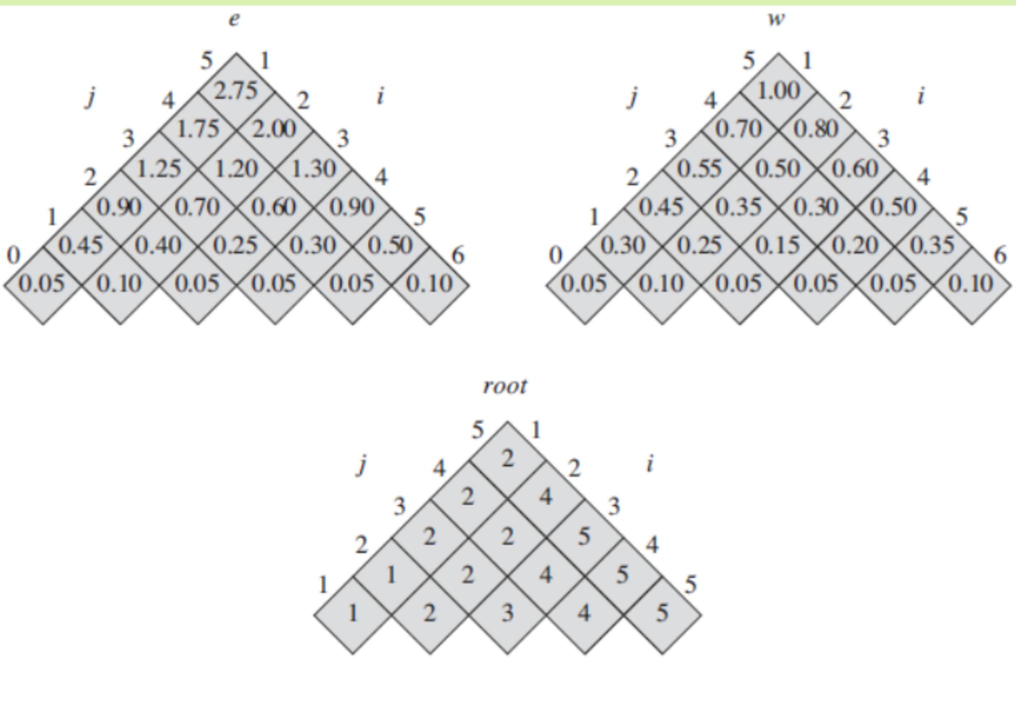



**Figure 15.10** The tables $e[i, j]$, $w[i, j]$, and $root[i, j]$ computed by OPTIMAL-BST on the key distribution shown in Figure 15.9. The tables are rotated so that the diagonals run horizontally.