CMPN302: Algorithms Design and Analysis

Lecture 04: Dynamic Programming

Ahmed Hamdy

Computer Engineering Department

Cairo University

Fall 2018

What is Dynamic Programming?

- Similar in divide-and-conquer in dividing the problem into smaller problems to obtain solution.
- Different than divide-and-conquer is that the subproblems are typically overlapping with each other and with the bigger problem. Divide-and-conquer typically generates independent subproblems.
- Dynamic programming is more suited for *optimization problems*. It achieves optimal solutions for them

Power calculation

- Compute 3^{16}:
  - Loop 16 times to compute result
  - Recursively:

```
Pow(x, p)
    if p == 1
         return x
    return pow(x, p/2) * pow(x, p/2)
```

    - $3^{16} = 3^8 \times 3^8$
    - $3^8 = 3^4 \times 3^4$
    - $3^4 = 3^2 \times 3^2$
    - $3^2 = 3 \times 3$
  - How many calls to function Pow?
  - Better way? Memoization

Dynamic programming

- Simplify recursion by memoization of subproblems
- Solves optimization problems
  - Finding shortest path
  - Best matrix parenthesization
  - Longest common subsequence
  - …etc.

Dynamic programming

- Two approaches:
  - Top-down with memoization
    - Execute recursively in normal manner.
    - Just check first if the solution was computed and stored before. If so, return the solution.
    - Otherwise compute normally and store new solution.
  - Bottom-up method
    - It is proper for problems where every problem relies on smaller ones.
    - Sort problems according to size.
    - Solve them in order.

Dynamic programming

- Rod-cutting problem: cut rod of length n to maximize revenue based on following table

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Figure 15.1 A sample price table for rods. Each rod of length i inches earns the company p_i dollars of revenue.

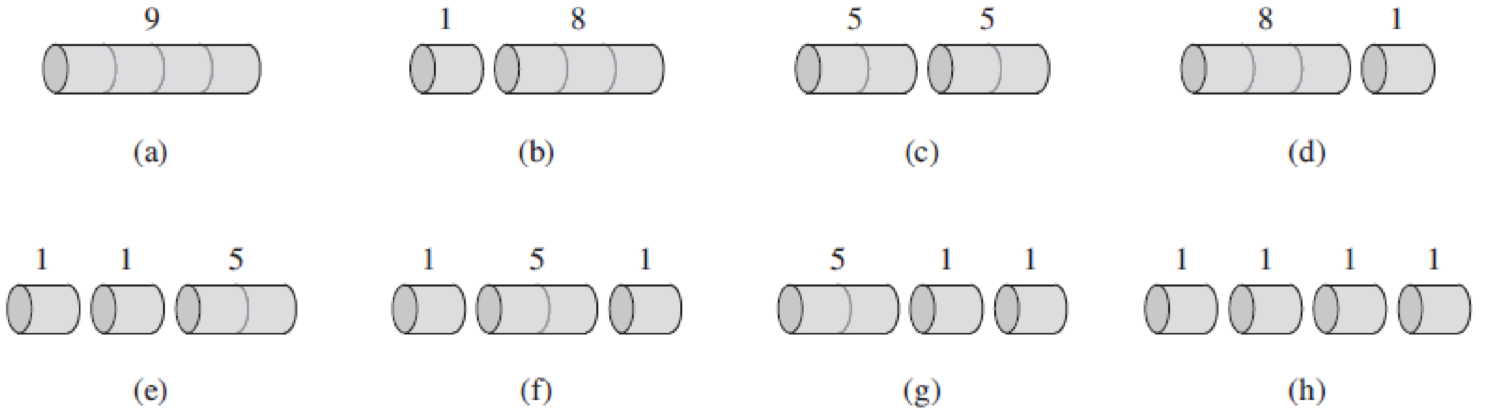


Figure 15.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 15.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

Dynamic programming

- Rod-cutting problem

$$
\begin{aligned}
r_1 &= 1 && \text{from solution } 1 = 1 \quad \text{(no cuts)}\,, \\
r_2 &= 5 && \text{from solution } 2 = 2 \quad \text{(no cuts)}\,, \\
r_3 &= 8 && \text{from solution } 3 = 3 \quad \text{(no cuts)}\,, \\
r_4 &= 10 && \text{from solution } 4 = 2 + 2\,, \\
r_5 &= 13 && \text{from solution } 5 = 2 + 3\,, \\
r_6 &= 17 && \text{from solution } 6 = 6 \quad \text{(no cuts)}\,, \\
r_7 &= 18 && \text{from solution } 7 = 1 + 6 \ \text{ or } \ 7 = 2 + 2 + 3\,, \\
r_8 &= 22 && \text{from solution } 8 = 2 + 6\,, \\
r_9 &= 25 && \text{from solution } 9 = 3 + 6\,, \\
r_{10} &= 30 && \text{from solution } 10 = 10 \quad \text{(no cuts)}\,.
\end{aligned}
$$

More generally, we can frame the values r_n for $n \geq 1$ in terms of optimal revenues from shorter rods:

$$r_n = \max\left(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\right) \,. \qquad (15.1)$$

Dynamic programming

- Rod-cutting problem

```
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
```

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) .$$

$$T(n) = 2^n ,$$

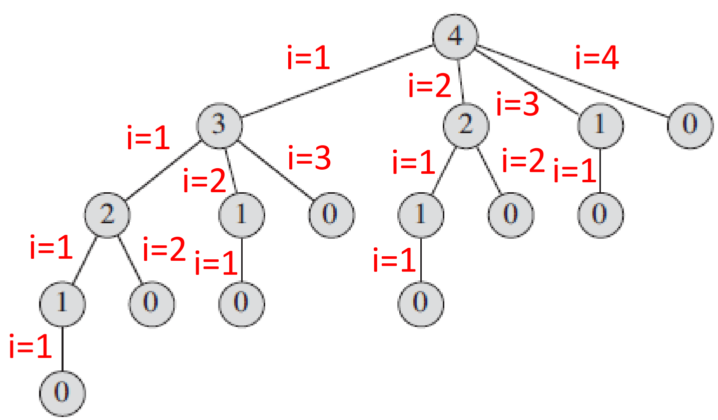


Figure 15.3 The recursion tree showing recursive calls resulting from a call CUT-ROD(p, n) for $n = 4$. Each node label gives the size n of the corresponding subproblem, so that an edge from a parent with label s to a child with label t corresponds to cutting off an initial piece of size $s - t$ and leaving a remaining subproblem of size t. A path from the root to a leaf corresponds to one of the 2^{n-1} ways of cutting up a rod of length n. In general, this recursion tree has 2^n nodes and 2^{n-1} leaves.

Dynamic programming

- Rod-cutting problem: Top-down memoized approach

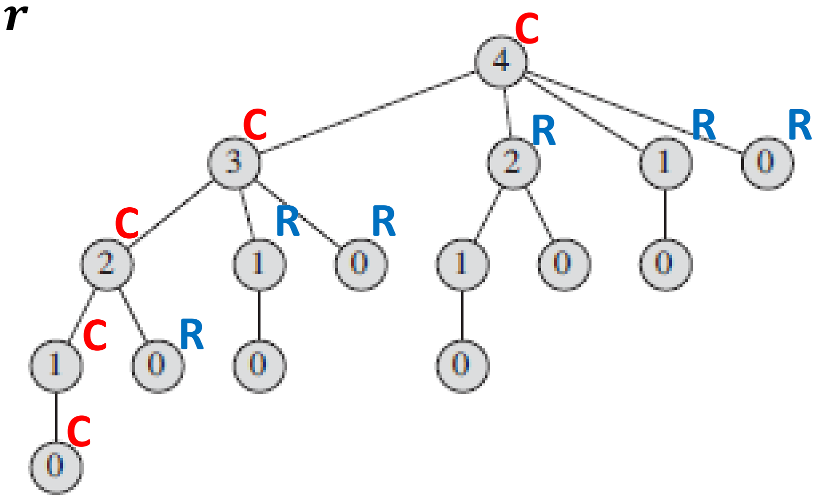
```
MEMOIZED-CUT-ROD(p, n)
let r[0..n] be a new array
for i = 0 to n
    r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

Initialization r

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

Retrieving from $r[n]$

Computing



4				
	3	2	1	0
		2	1	0
			1	0
				0

- Complexity: $\Theta(n^2)$

Dynamic programming

- Rod-cutting problem: Bottom-up approach
- Simpler when problem is a good fit

```
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]
```

Dynamic programming

- Rod-cutting problem
- How to print optimal cuts??

```
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
```

```
PRINT-CUT-ROD-SOLUTION(p, n)
(r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while n > 0
    print s[n]
    n = n - s[n]
```

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Dynamic programming

- Matrix-chain multiplication: $A_1 A_2 \cdots A_n$
- Example:

To illustrate the different costs incurred by different parenthesizations of a matrix product, consider the problem of a chain $\langle A_1, A_2, A_3 \rangle$ of three matrices. Suppose that the dimensions of the matrices are 10×100, 100×5, and 5×50, respectively. If we multiply according to the parenthesization $((A_1 A_2) A_3)$, we perform $10 \cdot 100 \cdot 5 = 5000$ scalar multiplications to compute the 10×5 matrix product $A_1 A_2$, plus another $10 \cdot 5 \cdot 50 = 2500$ scalar multiplications to multiply this matrix by A_3, for a total of 7500 scalar multiplications. If instead we multiply according to the parenthesization $(A_1(A_2 A_3))$, we perform $100 \cdot 5 \cdot 50 = 25{,}000$ scalar multiplications to compute the 100×50 matrix product $A_2 A_3$, plus another $10 \cdot 100 \cdot 50 = 50{,}000$ scalar multiplications to multiply A_1 by this matrix, for a total of 75,000 scalar multiplications. Thus, computing the product according to the first parenthesization is 10 times faster.

Dynamic programming

- Matrix-chain multiplication:
- Recurrence:

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

- Grows as $\Omega(2^n)$

Dynamic programming

- Matrix-chain multiplication
- Example:

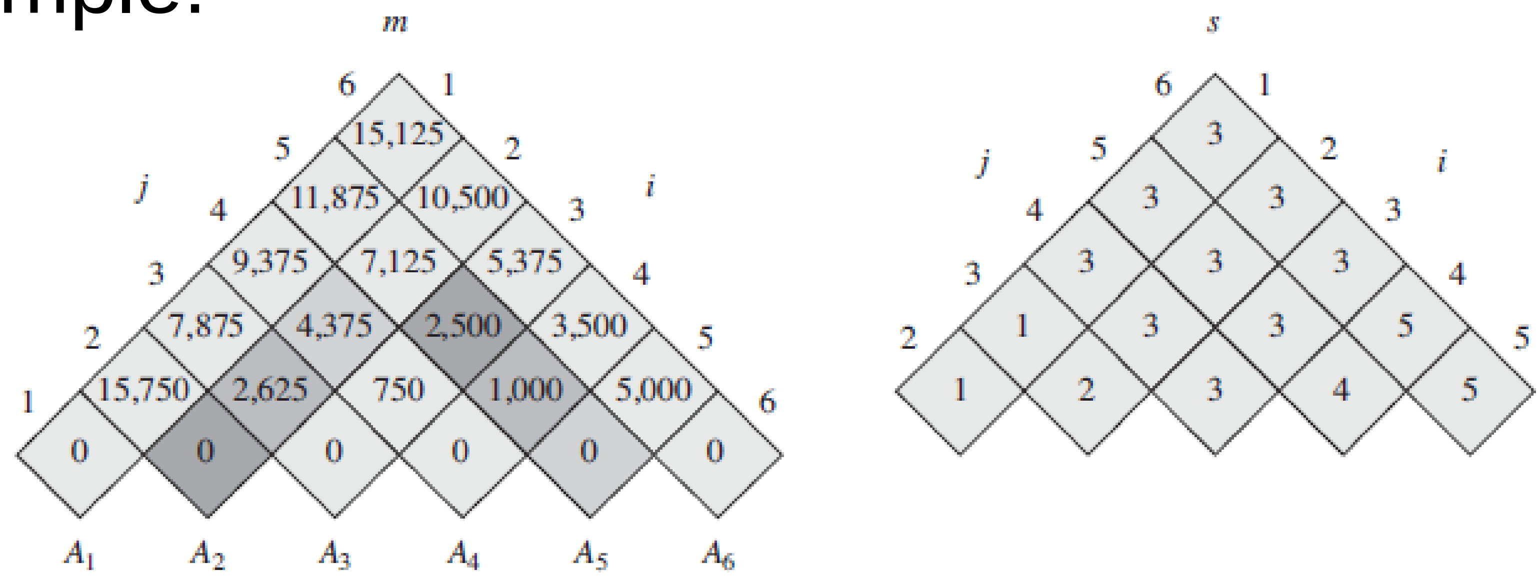


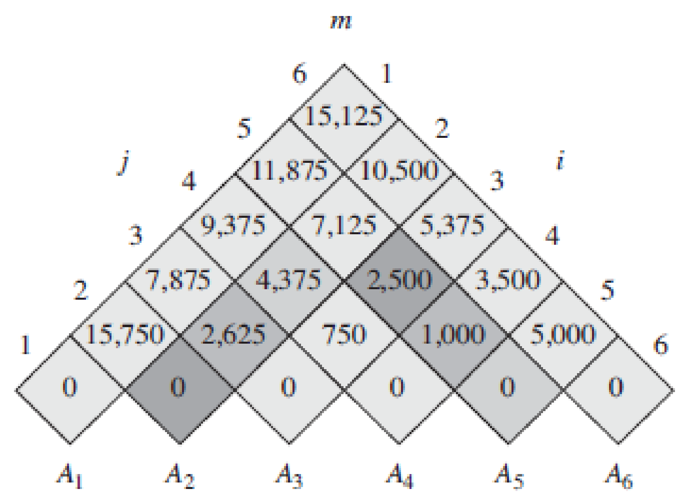
Figure 15.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

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

Dynamic programming

- Matrix-chain multiplication
- Code:

```
MATRIX-CHAIN-ORDER(p)
n = p.length - 1
let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n                  // l is the chain length
    for i = 1 to n - l + 1
        j = i + l - 1
        m[i, j] = ∞
        for k = i to j - 1
            q = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```



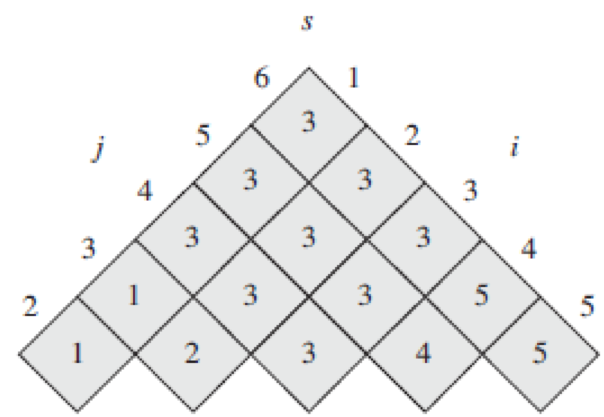
- Complexity: $O(n^3)$

Dynamic programming

- Matrix-chain multiplication
- Display solution:

```
PRINT-OPTIMAL-PARENS(s, i, j)
if i == j
    print "A"_i
else print "("
    PRINT-OPTIMAL-PARENS(s, i, s[i, j])
    PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
    print ")"
```

In the example of Figure 15.5, the call PRINT-OPTIMAL-PARENS$(s, 1, 6)$ prints the parenthesization $((A_1(A_2A_3))((A_4A_5)A_6))$.



Dynamic programming

- Matrix-chain multiplication
- Recursive code:

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
if i == j
    return 0
m[i, j] = ∞
for k = i to j − 1
    q = RECURSIVE-MATRIX-CHAIN(p, i, k)
        + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
        + p_{i−1} p_k p_j
    if q < m[i, j]
        m[i, j] = q
return m[i, j]
```

Dynamic programming

- Matrix-chain multiplication
- Memoized version:

```
LOOKUP-CHAIN(m, p, i, j)
if m[i, j] < ∞
    return m[i, j]
if i == j
    m[i, j] = 0
else for k = i to j − 1
        q = LOOKUP-CHAIN(m, p, i, k)
            + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
        if q < m[i, j]
            m[i, j] = q
return m[i, j]
```

Dynamic programming

- Longest Common Subsequence
- Given two strings, find longest common subsequence of characters (NOT necessarily consecutive)
- Example:

```
 S1=ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
 S2=GTCGTTCGGAATGCCGTTGCTCTGTAAA
LCS=GTCGTCGGAAGCCGGCCGAA
```

Dynamic programming

- Longest Common Subsequence

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i-1, j-1] + 1
            b[i, j] = "↖"
        elseif c[i-1, j] ≥ c[i, j-1]
            c[i, j] = c[i-1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j-1]
            b[i, j] = "←"
return c and b
```

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4