Design and Analysis of Algorithms

CAIRO UNIVERSITY

Lecture 04: Dynamic Programming

Ahmed Hamdy

What is Dynamic Programming?

- Similar in divide-and-conquer in dividing the problem into smaller problems to obtain solution.

- Different than divide-and-conquer in 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 (similar to bottom-up approach)
  - 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

Memoization

- Using memoization, the code simply becomes:

```
Pow(x, p)
    if xp == NULL
        xp[1] = x
    if p <= size(xp)
        return xp[p]

    xp[p] = pow(x, p/2) * pow(x, p/2)

    return xp[p]
```

- Trace the recursion tree??

Dynamic programming

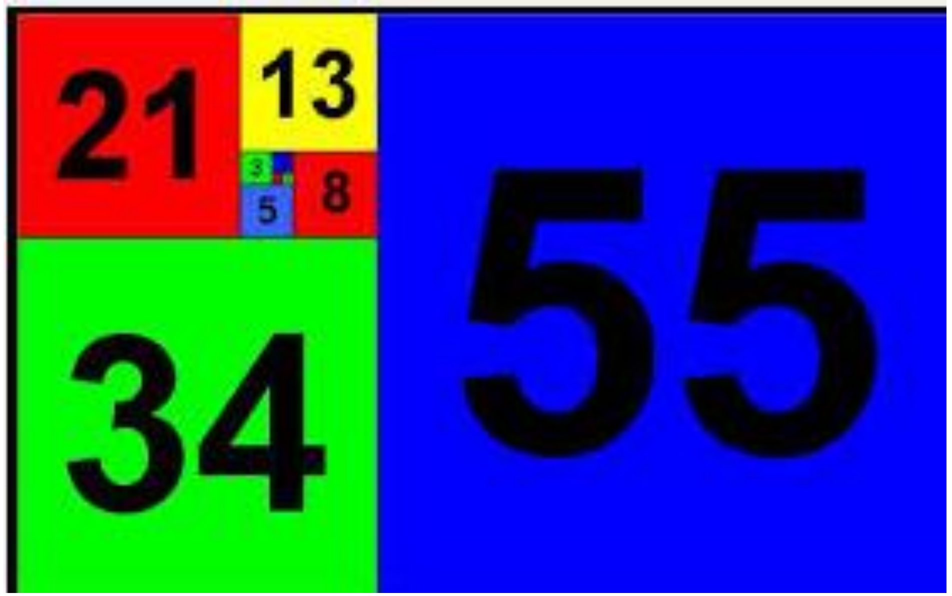
- Simplify recursion by memoization of subproblems
- Solves optimization problems
  - Finding shortest path
  - Best matrix parenthesization
  - Longest common subsequence
  - …etc.

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.

Fibonacci Numbers

- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 114, …
- $F_1 = 1$
- $F_2 = 1$
- $F_N = F_{N-1} + F_{N-2}$
- Recursive Solution?



Failing Spectacularly

- Naïve recursive method

```
// pre: n > 0
// post: return the nth Fibo      number
public int fib(int n) {
    if(n <= 2)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}
```

- Order of this method?

A. $O(1)$ B. $O(\log N)$ C. $O(N)$ D. $O(N^2)$ E. $O(2^N)$

Complexity of Fibonacci

- $T_{Fib}(n) = T_{Fib}(n-1) + T_{Fib}(n-2) + \Theta(1)$, Master doesn't hold!!
- Can be bounded by two recurrences $T_L(n)$ and $T_U(n)$, where $T_L(n) < T_{Fib}(n) < T_U(n)$ if defined as:
- $T_L(n) = 2\,T_L(n-2) + \Theta(1)$: recursion tree has height $n/2$, so number of nodes in the tree = $\left(2^{\frac{n}{2}+1} - 1\right) * \Theta(1) = \mathrm{T_L(n)} = \Theta(2^{n/2})$
- $T_U(n) = 2\,T_U(n-1) + \Theta(1)$: recursion tree has height n, so number of nodes in the tree = $(2^{n+1} - 1) * \Theta(1) = \mathrm{T_U(n)} = \Theta(2^n)$
- So $T_{Fib}(n) = O(2^n)$

Failing Spectacularly

```
1th fibonnaci number: 1 -  Time: 4.467E-6
2th fibonnaci number: 1 -  Time: 4.47E-7
3th fibonnaci number: 2 -  Time: 4.46E-7
4th fibonnaci number: 3 -  Time: 4.46E-7
5th fibonnaci number: 5 -  Time: 4.47E-7
6th fibonnaci number: 8 -  Time: 4.47E-7
7th fibonnaci number: 13 -  Time: 1.34E-6
8th fibonnaci number: 21 -  Time: 1.787E-6
9th fibonnaci number: 34 -  Time: 2.233E-6
10th fibonnaci number: 55 -  Time: 3.573E-6
11th fibonnaci number: 89 -  Time: 1.2953E-5
12th fibonnaci number: 144 -  Time: 8.934E-6
13th fibonnaci number: 233 -  Time: 2.9033E-5
14th fibonnaci number: 377 -  Time: 3.7966E-5
15th fibonnaci number: 610 -  Time: 5.0919E-5
16th fibonnaci number: 987 -  Time: 7.1464E-5
17th fibonnaci number: 1597 -  Time: 1.08984E-4
```

Failing Spectacularly

```
36th fibonnaci number: 14930352 -  Time: 0.045372057
37th fibonnaci number: 24157817 -  Time: 0.071195386
38th fibonnaci number: 39088169 -  Time: 0.116922086
39th fibonnaci number: 63245986 -  Time: 0.186926245
40th fibonnaci number: 102334155 -  Time: 0.308602967
41th fibonnaci number: 165580141 -  Time: 0.498588795
42th fibonnaci number: 267914296 -  Time: 0.793824734
43th fibonnaci number: 433494437 -  Time: 1.323325593
44th fibonnaci number: 701408733 -  Time: 2.098209943
45th fibonnaci number: 1134903170 -  Time: 3.392917489
46th fibonnaci number: 1836311903 -  Time: 5.506675921
47th fibonnaci number: -1323752223 -  Time: 8.803592621
48th fibonnaci number: 512559680 -  Time: 14.295023778
49th fibonnaci number: -811192543 -  Time: 23.030062974
50th fibonnaci number: -298632863 -  Time: 37.217244704
51th fibonnaci number: -1109825406 -  Time: 60.224418869
```

Why neg.?

Factor = 1.618

Golden Ratio

- Called $phi = \Phi = 1.618$

- Appears in nature *(source national geographic)*:
  - Number of petals typically follows Fibonacci number
  - Seeds of sunflowers and pine cones twist in opposing spirals of Fibonacci numbers
  - Even the sides of an unpeeled banana will usually be a Fibonacci number

Failing Spectacularly

```
50th fibonnaci number: -298632863 -  Time: 37.217244704
```

- How long to calculate the 70th Fibonacci Number with this method?

A. 37 seconds

B. 74 seconds

C. 740 seconds

D. 14,800 seconds

E. None of these

Aside - Overflow

- at 47^{th} Fibonacci number overflows int
- Could use BigInteger class instead

```
private static final BigInteger one
    = new BigInteger("1");

private static final BigInteger two
    = new BigInteger("2");

public static BigInteger fib(BigInteger n) {
    if(n.compareTo(two) <= 0)
        return one;
    else {
        BigInteger firstTerm =
            fib(n.subtract(two));
        BigInteger secondTerm =
            fib(n.subtract(one));
        return firstTerm.add(secondTerm);
    }
}
```

Aside - BigInteger

- Answers correct beyond 46th Fibonacci number
- Even slower due to creation of so many objects

```
37th fibonnaci number: 24157817 -  Time: 2.406739213
38th fibonnaci number: 39088169 -  Time: 3.680196724
39th fibonnaci number: 63245986 -  Time: 5.941275208
40th fibonnaci number: 102334155 -  Time: 9.63855468
41th fibonnaci number: 165580141 -  Time: 15.659745756
42th fibonnaci number: 267914296 -  Time: 25.404417949
43th fibonnaci number: 433494437 -  Time: 40.867030512
44th fibonnaci number: 701408733 -  Time: 66.391845965
45th fibonnaci number: 1134903170 -  Time: 106.964369924
46th fibonnaci number: 1836311903 -  Time: 178.981819822
47th fibonnaci number: 2971215073 -  Time: 287.052365326
```

Slow Fibonacci

- Why so slow?
- Algorithm keeps calculating the same value over and over
- When calculating the 40th Fibonacci number the algorithm calculates the 4th Fibonacci number **24,157,817** times!!!

Fast Fibonacci

- Instead of starting with the big problem and working down to the small problems
- ... start with the small problem and work up to the big problem
- This is bottom-up
- Write as top-down?

```
public static BigInteger fastFib(int n) {
    BigInteger smallTerm = one;
    BigInteger largeTerm = one;
    for(int i = 3; i <= n; i++) {
        BigInteger temp = largeTerm;
        largeTerm = largeTerm.add(smallTerm);
        smallTerm = temp;
    }
    return largeTerm;
}
```

Fast Fibonacci

```
1th fibonnaci number: 1 -  Time: 4.467E-6
2th fibonnaci number: 1 -  Time: 4.47E-7
3th fibonnaci number: 2 -  Time: 7.146E-6
4th fibonnaci number: 3 -  Time: 2.68E-6
5th fibonnaci number: 5 -  Time: 2.68E-6
6th fibonnaci number: 8 -  Time: 2.679E-6
7th fibonnaci number: 13 -  Time: 3.573E-6
8th fibonnaci number: 21 -  Time: 4.02E-6
9th fibonnaci number: 34 -  Time: 4.466E-6
10th fibonnaci number: 55 -  Time: 4.467E-6
11th fibonnaci number: 89 -  Time: 4.913E-6
12th fibonnaci number: 144 -  Time: 6.253E-6
13th fibonnaci number: 233 -  Time: 6.253E-6
14th fibonnaci number: 377 -  Time: 5.806E-6
15th fibonnaci number: 610 -  Time: 6.7E-6
16th fibonnaci number: 987 -  Time: 7.146E-6
17th fibonnaci number: 1597 -  Time: 7.146E-6
```

Fast Fibonacci

```
45th fibonnaci number: 1134903170 -  Time: 1.7419E-5
46th fibonnaci number: 1836311903 -  Time: 1.6972E-5
47th fibonnaci number: 2971215073 -  Time: 1.6973E-5
48th fibonnaci number: 4807526976 -  Time: 2.3673E-5
49th fibonnaci number: 7778742049 -  Time: 1.9653E-5
50th fibonnaci number: 12586269025 -  Time: 2.01E-5
51th fibonnaci number: 20365011074 -  Time: 1.9207E-5
52th fibonnaci number: 32951280099 -  Time: 2.0546E-5
```

```
67th fibonnaci number: 44945570212853 -  Time: 2.3673E-5
68th fibonnaci number: 72723460248141 -  Time: 2.3673E-5
69th fibonnaci number: 117669030460994 -  Time: 2.412E-5
70th fibonnaci number: 190392490709135 -  Time: 2.4566E-5
71th fibonnaci number: 308061521170129 -  Time: 2.4566E-5
72th fibonnaci number: 498454011879264 -  Time: 2.5906E-5
73th fibonnaci number: 806515533049393 -  Time: 2.5459E-5
74th fibonnaci number: 1304969544928657 -  Time: 2.546E-5
```

```
200th fibonnaci number: 280571172992510140037611932413038677189525 -  Time: 1.0273E-5
```

Courtesy of Mike Scott CS314 @ UTexas

Dynamic programming

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

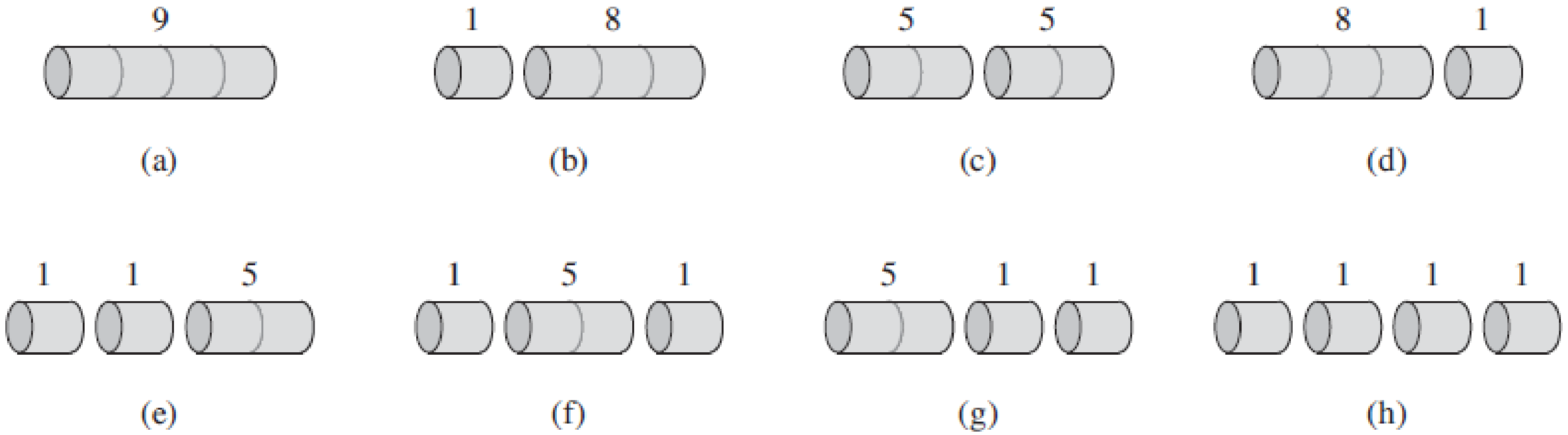


Figure 15.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 15.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

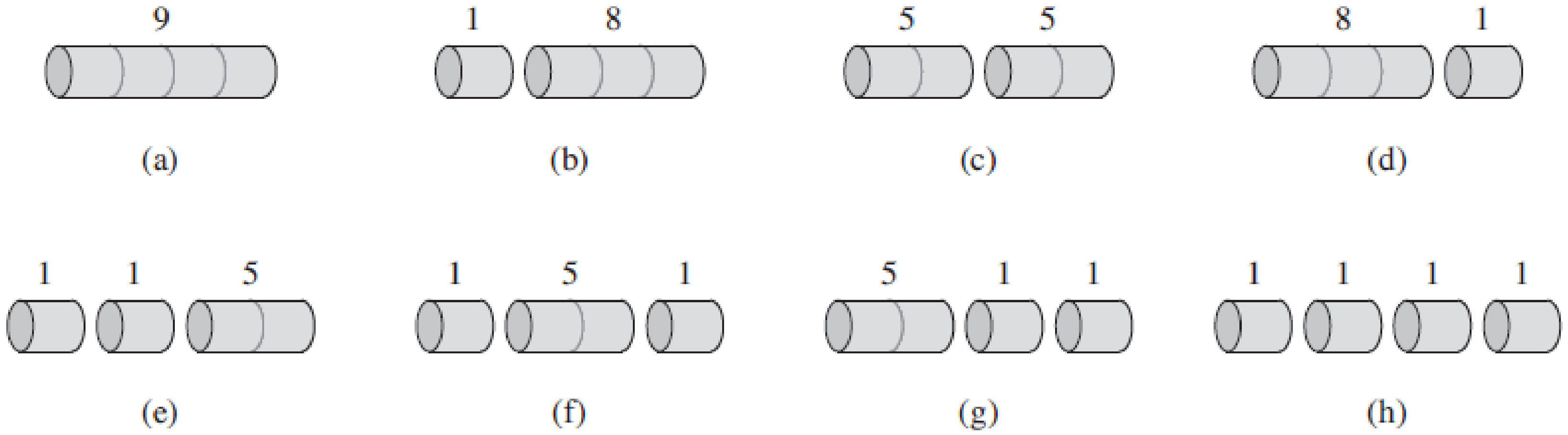
length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

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

Reducing Number of Combinations

- p_i indicates price of one piece of size i (fixed to this size).
- r_i indicates price of optimal partitioning of a piece of size i. It can be split to whatever optimal.
- Many of the shown combinations can be reduced to:
  - p_1, r_3 (covers cases b, d, e, g, h)
  - p_2, r_2 (c, e, f, g)
  - p_3, r_1 (b, d)
  - p_4, r_0 (a)

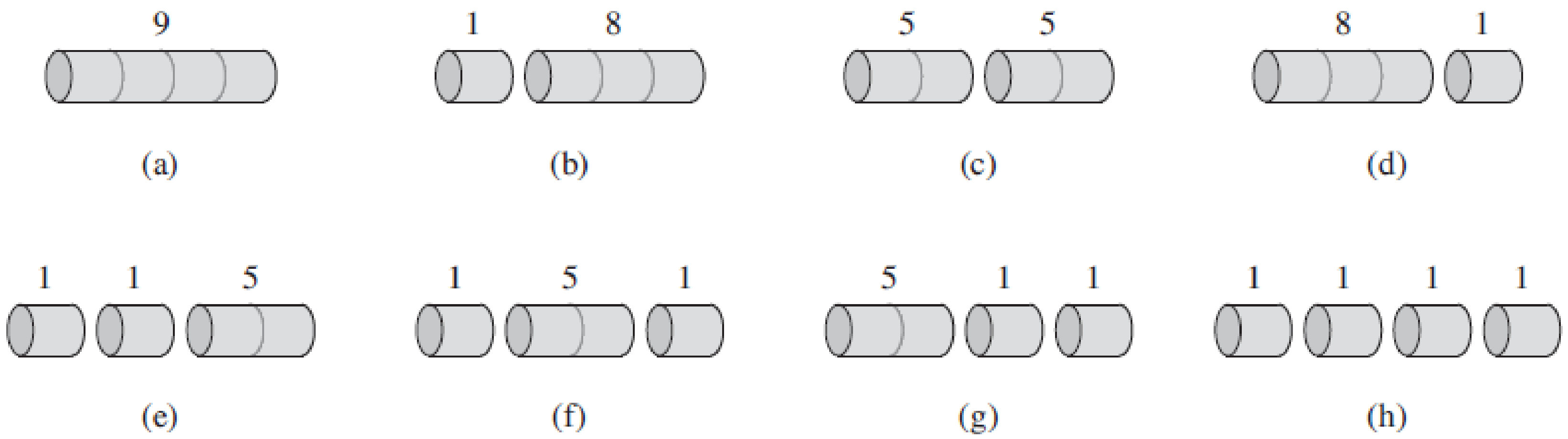
Works also with r_i instead of p_i



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

Reducing Number of Combinations

- Many of the shown combinations can be reduced to:
  - r_1, r_3 (covers cases b, d, e, g, h)
  - r_2, r_2 (c, e, f, g, h)
  - r_3, r_1 (b, d): *redundant*
  - p_4, r_0 (a): can't be r_4!!



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

Dynamic programming

- Rod-cutting problem

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.

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

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

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

Dynamic programming

- Rod-cutting problem

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

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

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
```

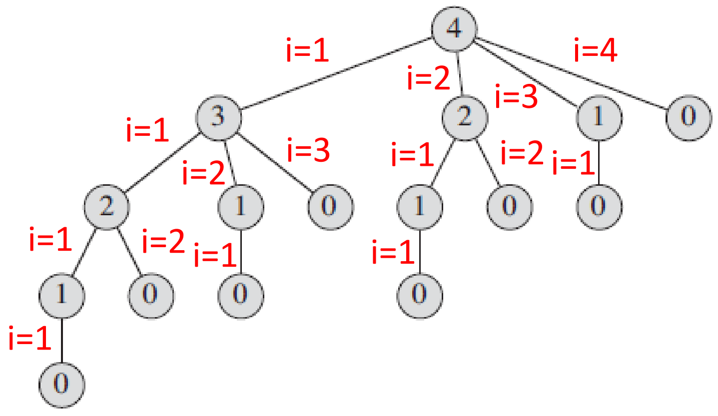


Figure 15.3 The recursion tree showing recursive calls resulting from a call CUT-ROD(p, n) for $n = 4$. Each node label gives the size n of the corresponding subproblem, so that an edge from a parent with label s to a child with label t corresponds to cutting off an initial piece of size $s - t$ and 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

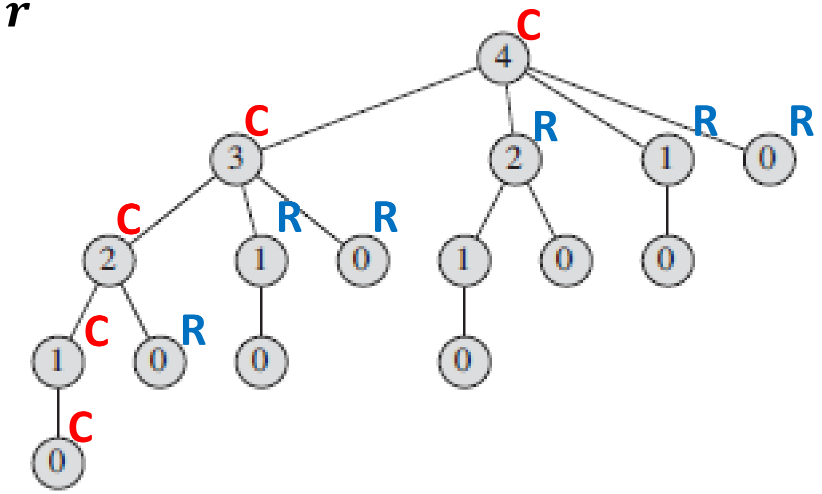
```
MEMOIZED-CUT-ROD(p, n)
let r[0..n] be a new array
for i = 0 to n
    r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

Initialization $\boldsymbol{r}$

```
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 $\boldsymbol{r[n]}$

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:

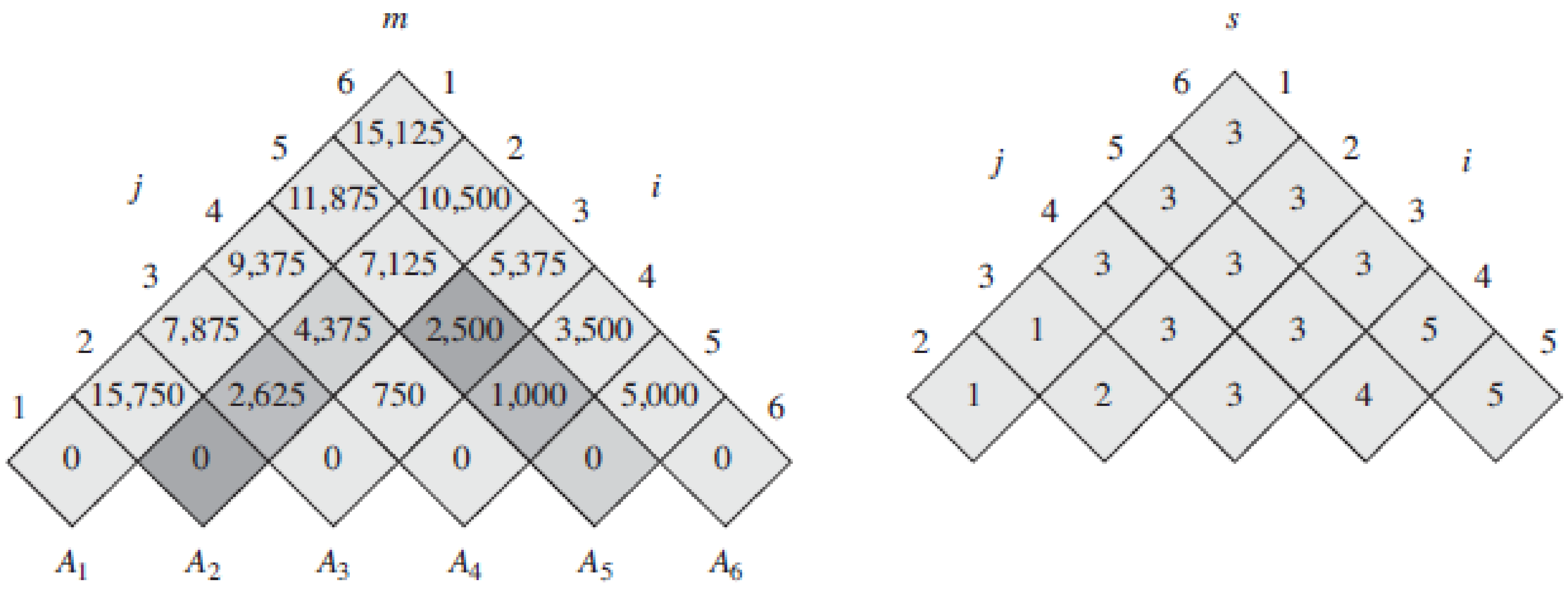


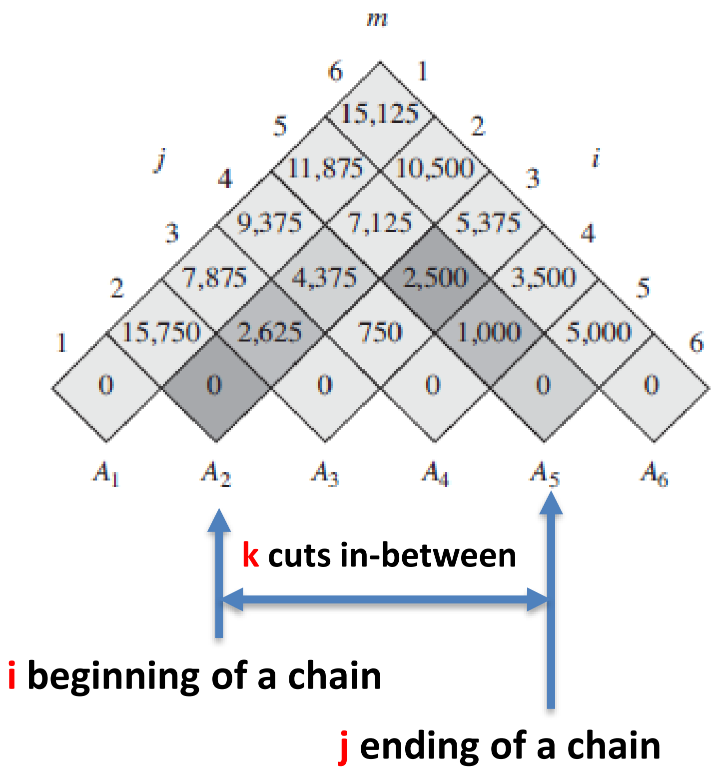
Figure 15.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

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
```



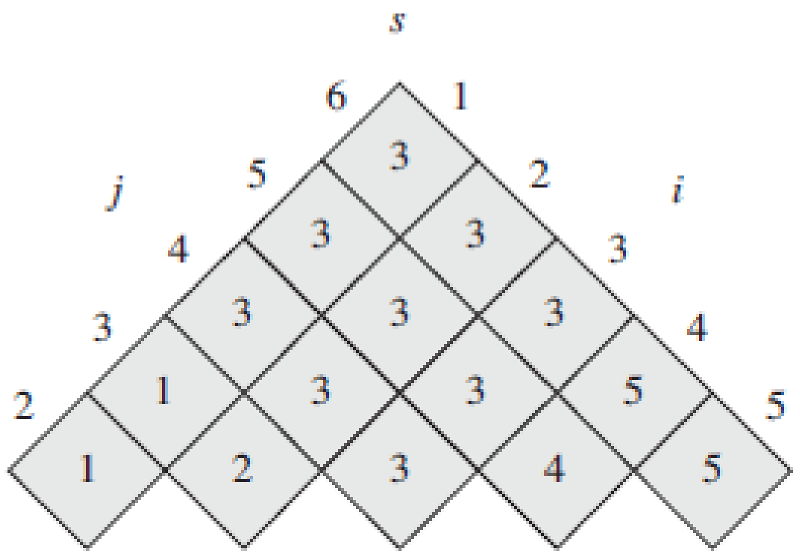
- Complexity: $O(n^3)$

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]
```