Lecture 24: Dynamic Programming I

2023/10/18

詹博华（中国科学院软件研究所）

Dynamic Programming（动态规划）

- General technique rather than solution to a specific problem.
- To solve a problem, solve its subproblems and store these intermediate results in a table.
- Look for *recurrence relations*.

Rod cutting

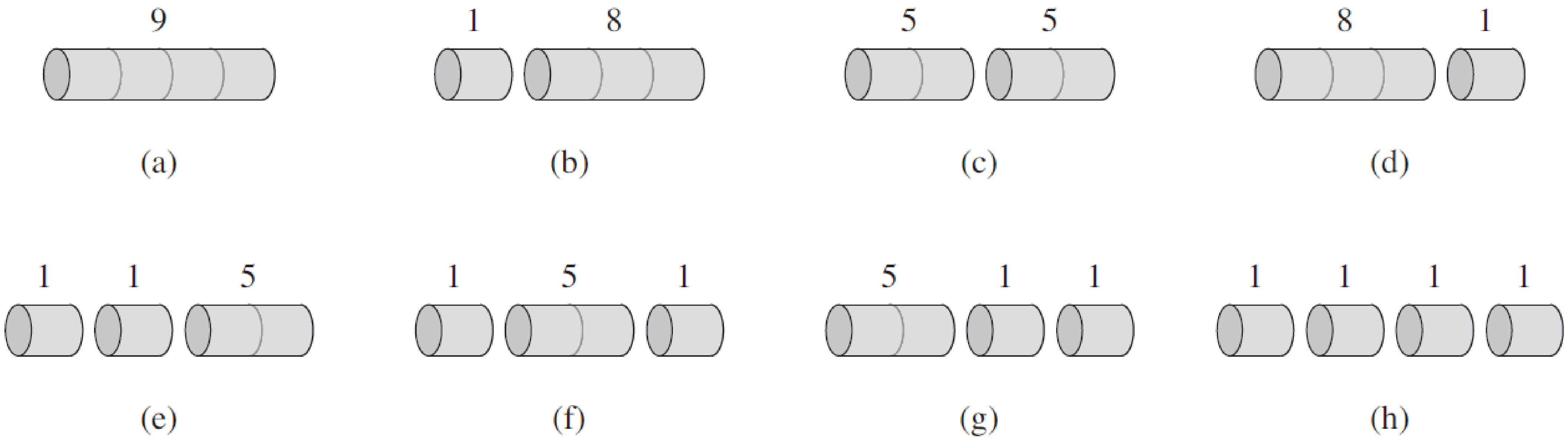
- We wish to cut up a rod of length n, and sell the pieces. We are given a table of the price of each piece as a function of its length:

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

- How should we cut up the rod to maximize the profit?

Rod cutting

- For a rod of length 4, it is best to cut into two length 2 pieces:



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

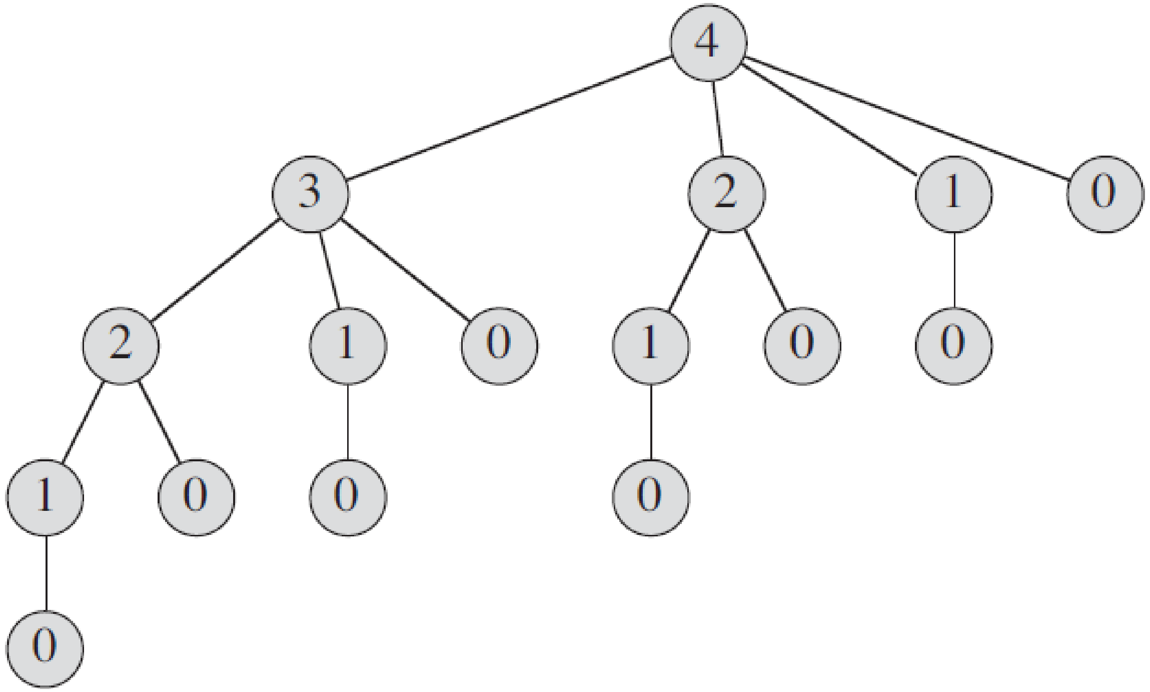
Naïve solution using recursion

- Top-down approach.
- Correct, but highly inefficient (exponential time).

```
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: call graph

- There are repeated calls to $\textsc{Cut-Rod}(p, n)$ for $n = 0, 1, 2$, but the answer returned each time should be same.



Recursion with **memoization**

- Keep a table $r[n]$ containing **memoized** results from previous calls to CUT-ROD(p, n).
- If an answer has already been computed, directly retrieve it from r.
- Initialization:

```
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)
```

Implementation of memoization

```
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
```

If already present, retrieve from r.

Write each result to r.

Bottom-up approach

- Solve the subproblems in turn, starting from the smallest.
- Maintain a table of solutions.

```
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]
```

Bottom-up approach: store solutions

- Maintain an additional array s, storing the optimal cut at each step.

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
```

Bottom-up approach: results

- Result up to $n = 10$:

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

- Exercise: use the above table to find optimal cuts for $n = 1 \ldots 10$, check the values of $r[i]$ is correct.

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

Matrix-chain multiplication

- Given a chain of matrices $A_1, A_2, \ldots, A_n$, wish to compute the product $A_1 A_2 \cdots A_n$.
- The dimensions of matrices can be quite different. E.g.

$$A_1: 10 \times 100$$
$$A_2: 100 \times 5$$
$$A_3: 5 \times 50$$

- If perform multiplication using $(A_1 A_2) A_3$, then
  - $10 \times 100 \times 5 = 5000$ scalar multiplications to compute $A_1 A_2: 10 \times 5$.
  - $10 \times 5 \times 50 = 2500$ scalar multiplications to compute $(A_1 A_2) A_3$.
  - Total 7500 scalar multiplications.

Matrix-chain multiplication

$$A_1: 10 \times 100$$
$$A_2: 100 \times 5$$
$$A_3: 5 \times 50$$

- If perform multiplication using $A_1(A_2A_3)$, then
  - $100 \times 5 \times 50 = 25000$ scalar multiplications to compute $A_2A_3: 100 \times 50$.
  - $10 \times 100 \times 50 = 50000$ scalar multiplications to compute $A_1(A_2A_3)$.
  - Total 75000 scalar multiplications.

7500 vs. 75000: a large difference!

Bottom-up approach

- Q: What are the subproblems of this problem?
- A: Number of scalar multiplications to compute $A_i A_{i+1} \cdots A_j$ for $i < j$. Let $A_{ij} = A_i A_{i+1} \cdots A_j$. Let $m[i,j]$ be the minimum number of scalar multiplications needed to compute A_{ij}.
- Suppose matrix A_i has dimension $p_{i-1} \times p_i$, then A_{ik} has dimension $p_{i-1} \times p_k$, and computing $A_{ik} A_{k+1,j}$ takes $p_{i-1} p_k p_j$ scalar multiplications.
- So computing A_{ij} through $A_{ik} A_{k+1,j}$ takes

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$$

steps.

Bottom-up approach

- Recurrence relation is:

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

- Next slide: implementation following this recurrence relation.

Implementation

- $m[i, j]$: minimum number of scalar multiplications.
- $s[i, j]$: choice of k that obtains the minimum.

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
```

Results

- Given problem

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

- Result of computation: