Dynamic Programming

謝仁偉 副教授
jenwei@mail.ntust.edu.tw
國立台灣科技大學 資訊工程系
2017 Spring

Introduction

- "Programming" in this context refers to a tabular method, not to writing computer code.
- Divide-and-conquer:
  - Partition the problem into disjoint subproblems.
  - Solve the subproblems recursively.
  - Combine their solutions to solve the original problem.
- Dynamic programming:
  - Apply when the subproblems overlap.
  - Solve each subsubproblem just once and then saves its answer in a table.
  - Typically apply dynamic programming to optimization problems.

Dynamic Programming: 4 Steps

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Outine

- ***Rod Cutting***
- Matrix-Chain Multiplication
- Elements of Dynamic Programming
- Longest Common Subsequence
- Optimal Binary Search Trees

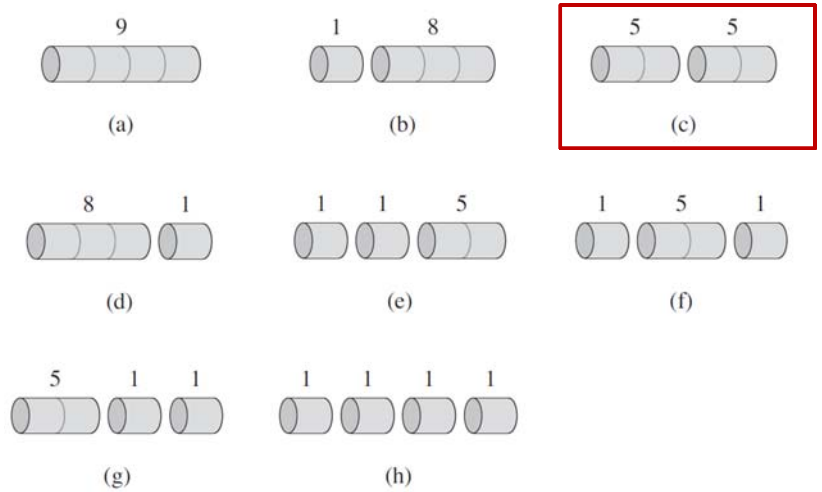
Rod-Cutting Problem (1/7)

- Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the 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

- Suppose $n = 4$, what is the maximum revenue r_n?

Rod-Cutting Problem (2/7)



Rod-Cutting Problem (3/7)

- How many ways can we cut up a rod of length n?
  - ➡ 2^{n-1}
  - ➡ $e^{\pi\sqrt{2n/3}}/4n\sqrt{3}$
    (in order of non-decreasing size)
- We denote a decomposition into pieces using ordinary additive notation, e.g., 7 = 2 + 2 + 3.

Rod-Cutting Problem (4/7)

- If an optimal solution cuts the rod into k pieces, for some $1 \le k \le n$, then an optimal decomposition
  $n = i_1 + i_2 + \cdots + i_k$
  of the rod into pieces of lengths $i_1, i_2, \ldots, i_k$ provides maximum corresponding revenue
  $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$.

Rod-Cutting Problem (5/7)

- We can determine the optimal revenue figures r_i, for $i = 1, 2, \ldots, 10$, by inspection, with the corresponding optimal decompositions

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

$$
\begin{aligned}
r_1 &= 1 && \text{from solution } 1 = 1 && \text{(no cuts)}\,, \\
r_2 &= 5 && \text{from solution } 2 = 2 && \text{(no cuts)}\,, \\
r_3 &= \\
r_4 &= \\
r_5 &= \\
r_6 &=
\end{aligned}
$$

Rod-Cutting Problem (6/7)

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

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

Rod-Cutting Problem (7/7)

- We can frame the values r_n for $n \geq 1$ in terms of optimal revenues from shorter rods:
  $r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$.
- We can also view a decomposition as consisting of a first piece of length i cut off the left-hand end, and then a right-hand remainder of length $n - i$. Only the remainder may be further divided.
  $r_n = \max\limits_{1 \leq i \leq n} (p_i + r_{n-i})$.

➡ In this formulation, an optimal solution embodies the solution to only 1 related subproblem (rather than 2).

Recursive Top-Down Implementation

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

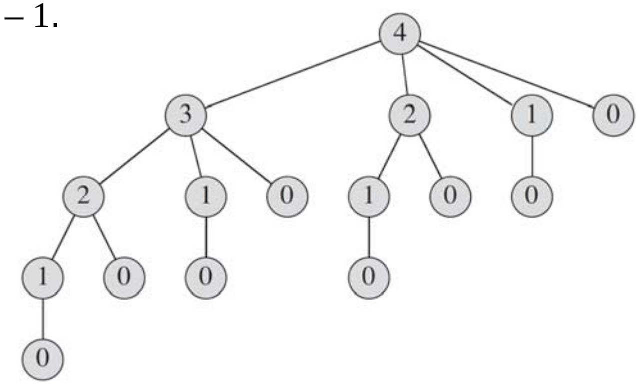
```
CUT-ROD(p, n)
    if n == 0
        return 0
    q = −∞
    for i = 1 to n
        q = max(q, p[i] + CUT-ROD(p, n − i))
    return q
```

- Once the input size becomes moderately large, your program would take a long time to run!

What Happen for n = 4?

- CUT-ROD(p, n) calls CUT-ROD(p, n – i) for $i = 1, 2, \ldots, n$.

➡ CUT-ROD(p, n) calls CUT-ROD(p, j) for $j = 0, 1, 2, \ldots, n - 1$.

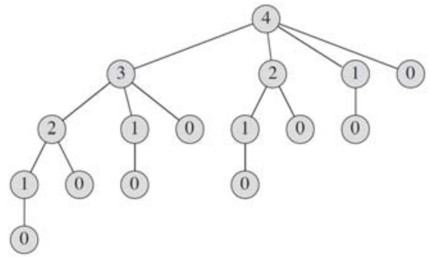


Running Time Analysis

- Let $T(n)$ denote the total number of calls made to CUT-ROD when called with its second parameter equal to n.

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

➡ $T(n) = 2^n$ (the running time of CUT-ROD is exponential in n)



Using Dynamic Programming

- A naive recursive solution is inefficient because it solves the same subproblems repeatedly.

➡ Dynamic programming uses additional memory to save computation time:

  - We arrange for each subproblem to be solved only once, saving its solution.
  - If we need to refer to this subproblem's solution again later, we can just look it up, rather than recompute it.
  - There are usually two equivalent ways to implement a dynamic-programming approach: top-down with memoization and bottom-up method.

Top-Down with Memoization

```
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)
```

```
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
```