CS375
Design & Analysis of Algorithms

Spring 2016

Lecture 12-13

Lei Yu

Department of Computer Science

State University of New York at Binghamton

lyu@cs.binghamton.edu

www.cs.binghamton.edu/~lyu

Dynamic Programming

- Not a specific algorithm, but a technique (like divide-and-conquer).
- Developed back in the day when "programming" meant "tabular method" (like linear programming). Does not really refer to computer programming.
- Used for optimization problems:
  - Find a solution with the optimal value.
  - Minimization or maximization.

Four-step Method

When developing a dynamic-programming algorithm, we follow a sequence of four 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.

Rod Cutting Problem

How to cut steel rods into pieces in order to maximize the revenue you can get? Each cut is free. Rod lengths are always an integral number of inches.

Input: A length n and table of prices p_i, for $i = 1, 2, \ldots, n$.

Output: The maximum revenue obtainable for rods whose lengths sum to n, computed as the sum of the prices for the individual rods.

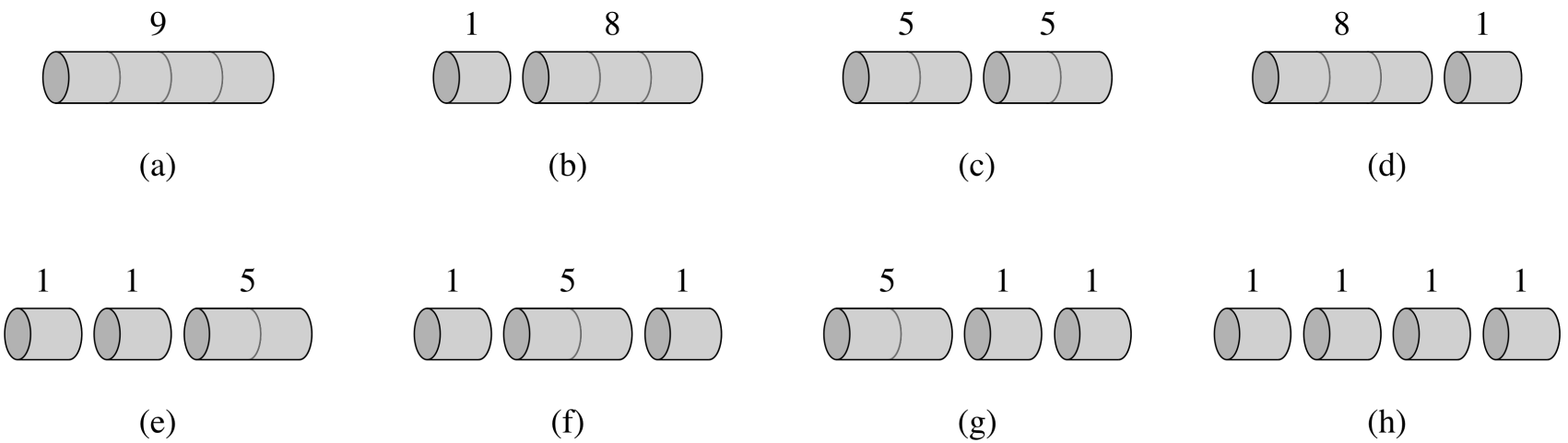
If p_n is large enough, an optimal solution might require no cuts, i.e., just leave the rod as n inches long.

Rod Cutting Example

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

Can cut up a rod in 2^{n-1} different ways, because can choose to cut or not cut after each of the first $n-1$ inches.

Here are all 8 ways to cut a rod of length 4, with the costs from the example:



Rod Cutting: Initial Analysis

The best way is to cut it into two 2-inch pieces, getting a revenue of $p_2 + p_2 = 5 + 5 = 10$

Let r_i be the maximum revenue for a rod of length i. We can express a solution as a sum of individual rod lengths. We further can determine optimal revenues r_i for the example, by inspection:

i	r_i	optimal solution
1	1	1 (no cuts)
2	5	2 (no cuts)
3	8	3 (no cuts)
4	10	$2 + 2$
5	13	$2 + 3$
6	17	6 (no cuts)
7	18	$1 + 6$ or $2 + 2 + 3$
8	22	$2 + 6$

Rod Cutting: Recurrent Formula

In general, we can determine optimal revenue r_n by taking the maximum of

- p_n: the price we get by not making a cut,
- $r_1 + r_{n-1}$: the maximum revenue from a rod of 1 inch and a rod of $n-1$ inches,
- $r_2 + r_{n-2}$: the maximum revenue from a rod of 2 inches and a rod of $n-2$ inches,
- $\ldots$
- $r_{n-1} + r_1$.

That is,

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

Rod Cutting: Optimal Substructure

To solve the original problem of size n, solve subproblems on smaller sizes. After making a cut, we have two subproblems, that we may solve the subproblems independently. The optimal solution to the original problem incorporates optimal solutions to the subproblems.

Example: For $n = 7$, one of the optimal solutions makes a cut at 3 inches, giving two subproblems, of lengths 3 and 4. We need to solve both of them optimally. The optimal solution for the problem of length 4, cutting into 2 pieces, each of length 2, is used in the optimal solution to the original problem with length 7.

Rod Cutting: Problem Decomposition

Every optimal solution has a leftmost cut. In other words, there is some cut that gives a first piece of length i cut off the left end, and a remaining piece of length $n - i$ on the right.

- We need to divide only the remainder, not the first piece.
- This leaves only one subproblem to solve, rather than two subproblems.
- Say that the solution with no cuts has first piece size $i = n$ with revenue p_n, and remainder size 0 with revenue $r_0 = 0$.
- This gives a simpler version of the equation for r_n:

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

Recursive Top-down Solution

Direct implementation of the simpler equation for r_n. The call $\textsc{Cut-Rod}(p, n)$ returns the optimal revenue r_n:

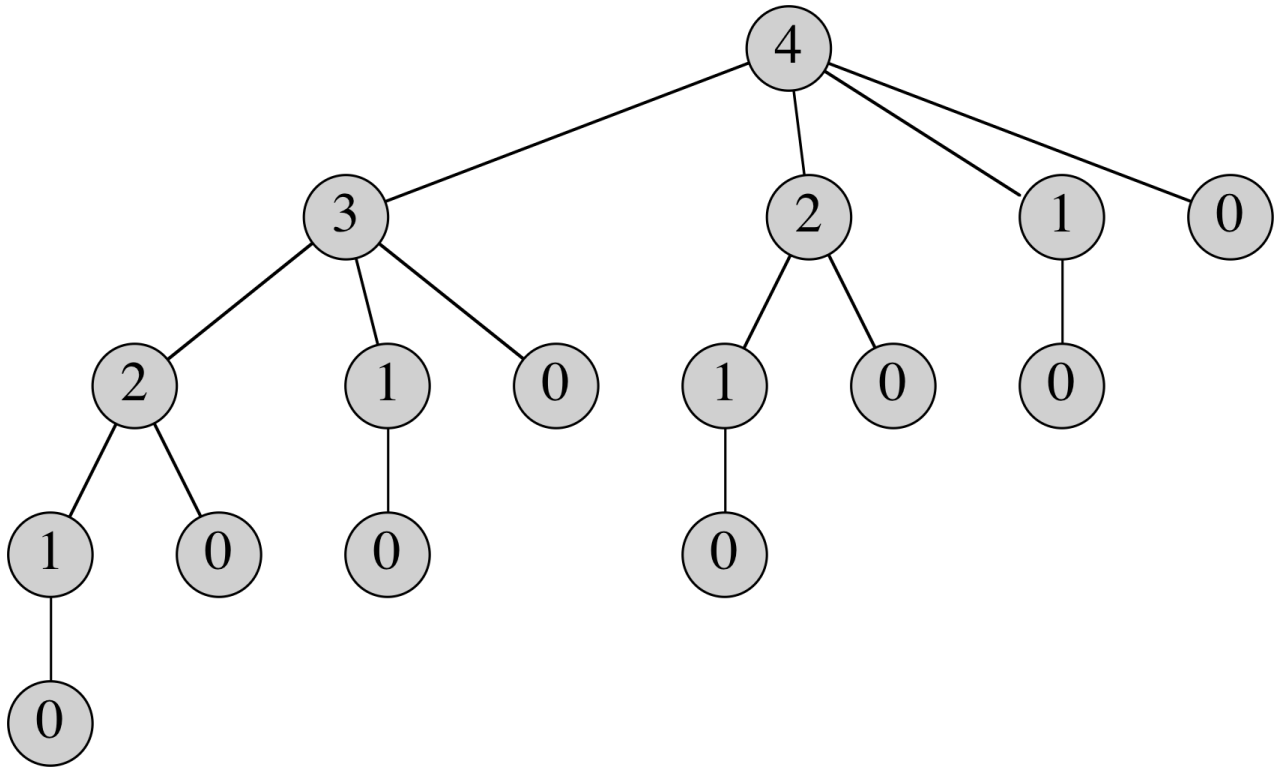
$\textsc{Cut-Rod}(p, n)$

1. **if** $n == 0$
2. **return** 0
3. $q = -\infty$
4. **for** $i = 1$ **to** n
5. $q = \max\{q, p[i] + \textsc{Cut-Rod}(p, n - i)\}$
6. **return** q

This procedure works, but it is terribly inefficient. If you code it up and run it, it could take more than an hour for $n = 40$. Running time almost doubles each time n increases by 1.

CUT-ROD Is Inefficient

CUT-ROD calls itself repeatedly, even on subproblems it has already solved. Here is a tree of recursive calls for $n = 4$. Inside each node is the value of n for the call represented by the node:



There are lots of repeated subproblems. It solves the subproblem for size 2 twice, for size 1 four times, and for size 0 eight times.

CUT-ROD Is Inefficient

Let $T(n)$ equal the number of calls to CUT-ROD with second parameter equal to n. Then

$$T(n) = \begin{cases} 1 & \text{if } n = 0, \\ 1 + \sum_{j=0}^{n-1} T(j) & \text{if } n > 1. \end{cases}$$

Summation counts calls where second parameter is $j = n - i$.
Solution to recurrence is $T(n) = 2^n$.

Dynamic-programming Solution

- Instead of solving the same subproblems repeatedly, arrange to solve each subproblem just once.
- Save the solution to a subproblem in a table, and refer back to the table whenever we revisit the subproblem.
- Store, do not recompute, using time-memory trade-off.
- Can turn an exponential-time solution into a polynomial-time solution.
- Two basic approaches: top-down with memoization, and bottom-up.

Dynamic Programming: Top-down with Memoization

- Solve recursively, but store each result in a table.
- To find the solution to a subproblem, first look in the table.
- If the answer is there, use it. Otherwise, compute the solution to the subproblem and then store the solution in the table for future use.
- *Memoizing* is remembering what we have computed previously.

MEMOIZED-CUT-ROD Pseudocode

Memoized version of the recursive solution, storing the solution to the subproblem of length i in array entry $r[i]$:

MEMOIZED-CUT-ROD(p, n)

1. let $r[0..n]$ be a new array
2. **for** $i = 0$ **to** n
3. $\quad r[i] = -\infty$
4. **return** MEM-CUT-ROD-AUX(p, n, r)

MEM-CUT-ROD-AUX(p, n, r)

1. **if** $r[n] \geq 0$
2. $\quad$ **return** $r[n]$
3. **if** $n == 0$
4. $\quad q = 0$
5. **else** $q = -\infty$
6. $\quad$ **for** $i = 1$ **to** n
7. $\quad\quad q = \max\{q, p[i] + \text{MEM-CUT-ROD-AUX}(p, n - i, r)\}$
8. $r[n] = q$
9. **return** q

Dynamic Programming: Bottom-up

In bottom-up dynamic programming, we sort the subproblems by size and solve the smaller ones first. That way, when solving a subproblem, have already solved the smaller subproblems we need.

BOTTOM-UP-CUT-ROD(p, n)

1. let $r[0..n]$ be a new array
2. $r[0] = 0$
3. **for** $j = 1$ **to** n
4. $\quad q = -\infty$
5. $\quad$ **for** $i = 1$ **to** j
6. $\quad\quad q = \max\{q, p[i] + r[j - i]\}$
7. $\quad r[j] = q$
8. **return** $r[n]$

Dynamic Programming: Running Time

Both the top-down and bottom-up versions run in $\Theta(n^2)$ time.

Bottom-up: Doubly nested loops. Number of iterations of inner for loop forms an arithmetic series.

Top-down: MEMOIZED-CUT-ROD solves each subproblem just once, and it solves subproblems for sizes $0, 1, \ldots, n$. To solve a subproblem of size k, the for loop iterates k times, implying that total number of iterations is $1 + 2 + \cdots + n$.