Chapter 14-1 : Dynamic Programming I

About this lecture

- Divide-and-conquer strategy allows us to solve a big problem by handling only smaller sub-problems
- Some problems may be solved using a stronger strategy: dynamic programming

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). Doesn't really refer to computer programming.
- Used for optimization problems:
  - Find *a* solution with *the* optimal value.
  - Minimization or maximization. (We'll see both.)

Rod Cutting

- How to cut steel rods into pieces to maximize the revenue you can get?
- Each cut is free. Rod lengths are always an integer number of inches.
- **Input:** A length n and table of prices p_i , for i = 1, 2, …, n.
- **Output:** The maximum revenue obtainable for rods whose lengths sum to n, computed as the sum of the prices for the individual rods.

Rod Cutting

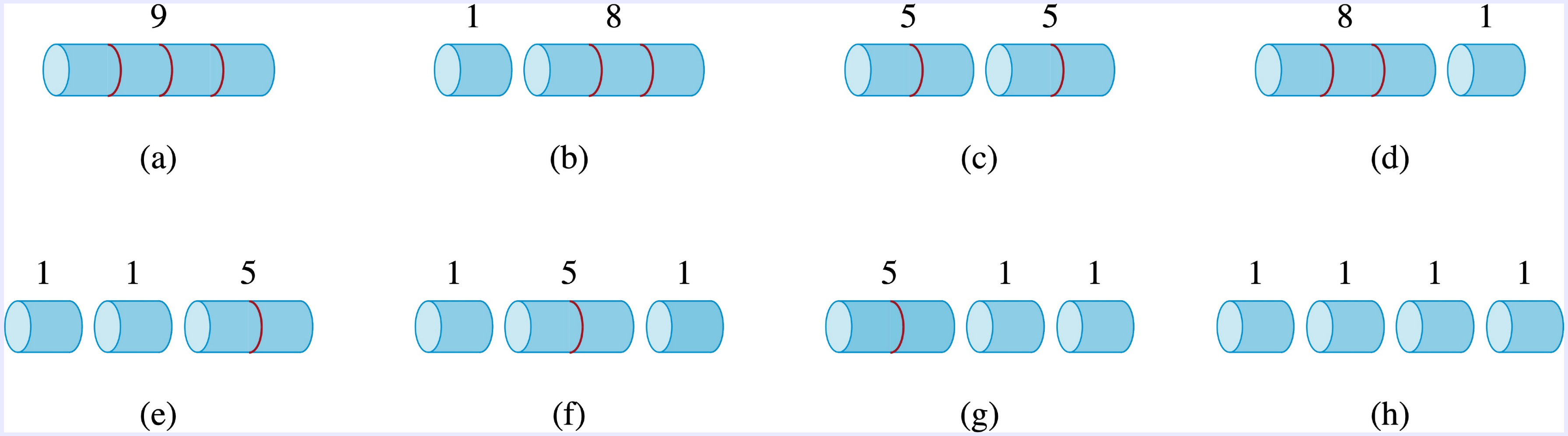
- If p_n is large enough, an optimal solution might require no cuts, i.e., just leave the rod as n inches long.

- Example:

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

Rod Cutting

- 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 revenue from the example:

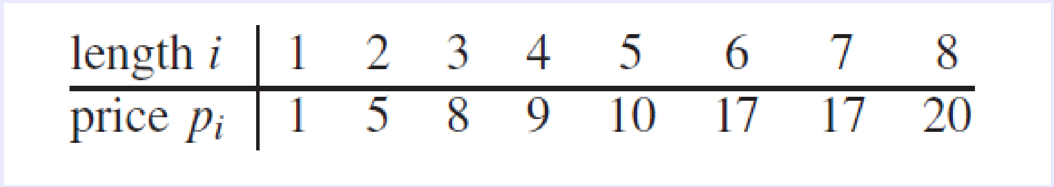


Rod Cutting

- 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. Can express a solution as a sum of individual rod lengths.
- Can determine optimal revenues r_i for the example, by inspection:

Rod Cutting

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



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

Rod Cutting

- Can determine optimal revenue r_n by taking the maximum of
  - p_n: the revenue from 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, . . .,
  - $r_{n-1} + r_1$.
- 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

- A simpler way to decompose the problem: Every optimal solution has a leftmost cut.
- In other words, there's 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.
- Gives a simpler version of the equation for r_n:
  - $r_n = \max\{p_{i,} + r_{n-i} : 1 \leq i \leq n\}$

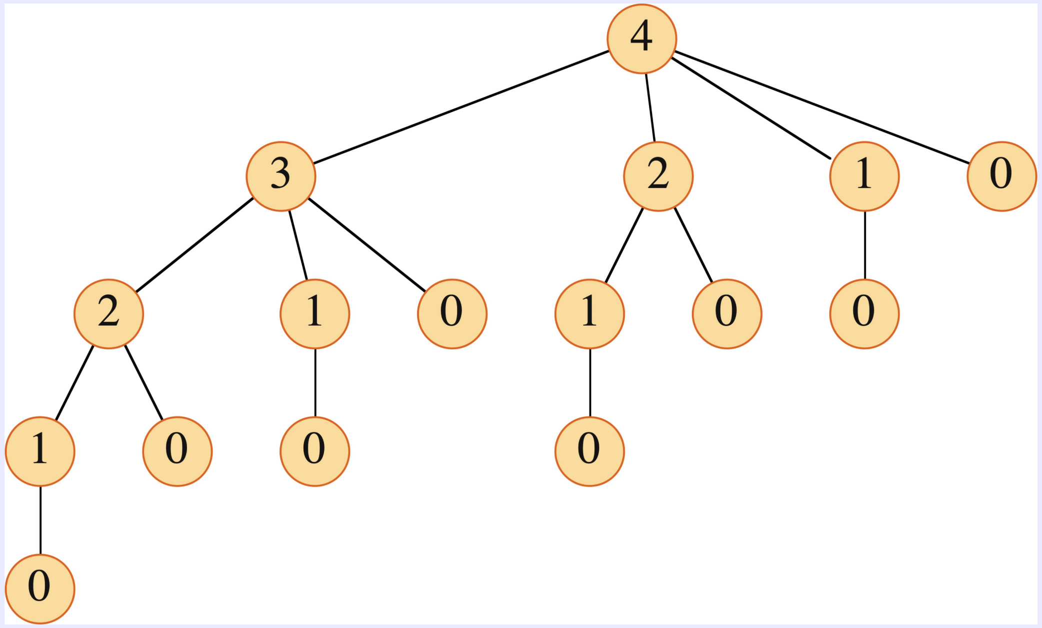
Recursive top-down solution

- ✓ 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 approximately doubles each time n increases by 1.

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = −∞
4  for i = 1 to n
5      q = max{q, p[i] + CUT-ROD(p, n − i)}
6  return q
```

Why so inefficient?

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



Optima Substructure

- Optimal substructure: To solve the original problem of size n, solve subproblems on smaller sizes.
- After making a cut, two subproblems remain. The optimal solution to the original problem incorporates optimal solutions to the subproblems.
- Solve the subproblems independently.

Exponential growth

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

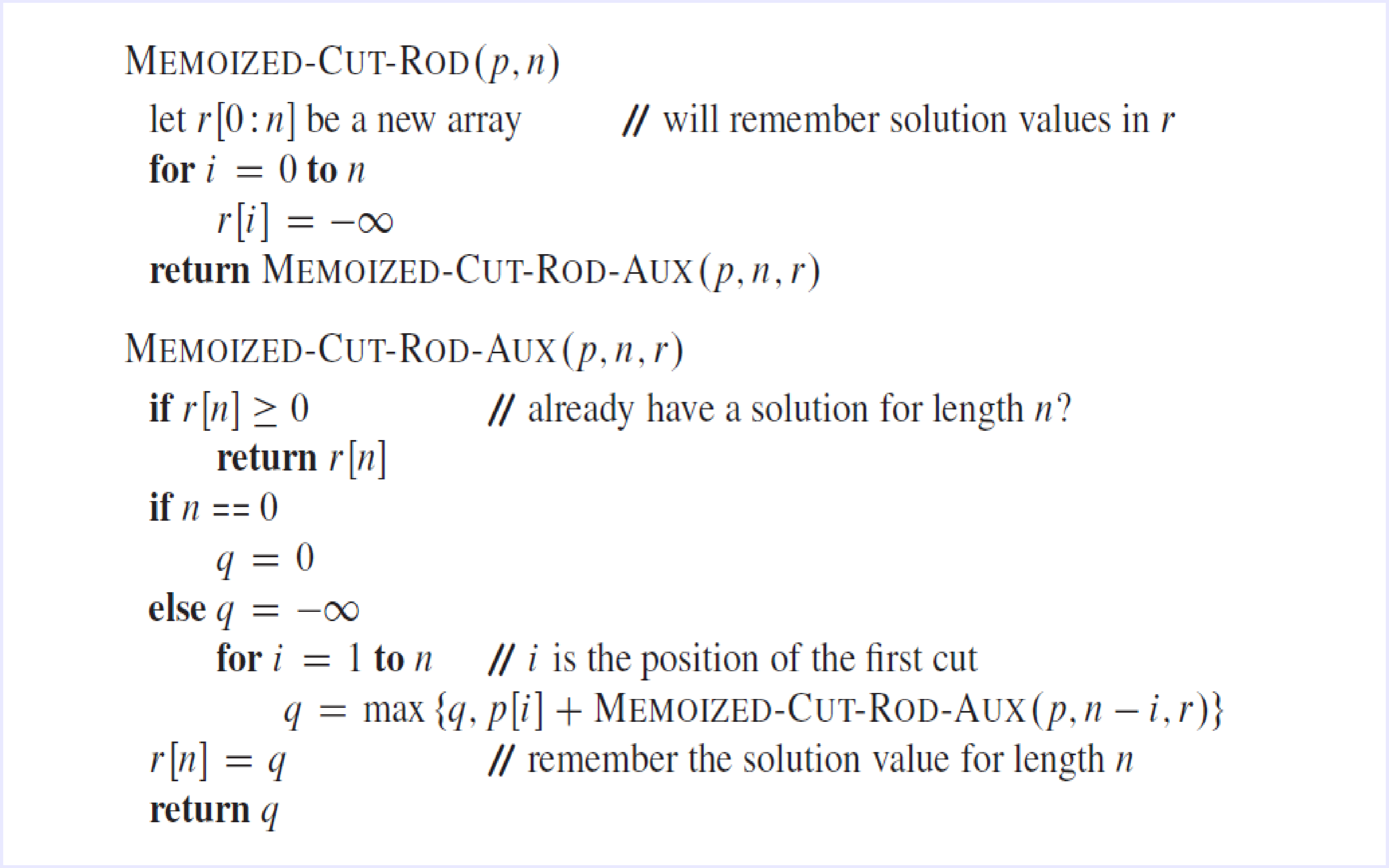
$$T(n) = \begin{cases} 1 & \text{if } n = 0\,, \\ 1 + \sum_{j=0}^{n-1} T(j) & \text{if } n \geq 1\,. \end{cases}$$

Exercise: $T(n) = 2^n$

Dynamic-programming solution

- Instead of solving the same subproblems repeatedly, arrange to solve each sub-problem just once.
- Save the solution to a subproblem in a table, and refer back to the table whenever we revisit the subproblem.
- "Store, don't recompute" → 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.

Top-down with memoization



```
MEMOIZED-CUT-ROD$(p, n)$
  let $r[0:n]$ be a new array        // will remember solution values in $r$
  for $i = 0$ to $n$
      $r[i] = -\infty$
  return MEMOIZED-CUT-ROD-AUX$(p, n, r)$

MEMOIZED-CUT-ROD-AUX$(p, n, r)$
  if $r[n] \geq 0$                   // already have a solution for length $n$?
      return $r[n]$
  if $n == 0$
      $q = 0$
  else $q = -\infty$
      for $i = 1$ to $n$             // $i$ is the position of the first cut
          $q = \max\{q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n - i, r)\}$
  $r[n] = q$                         // remember the solution value for length $n$
  return $q$
```

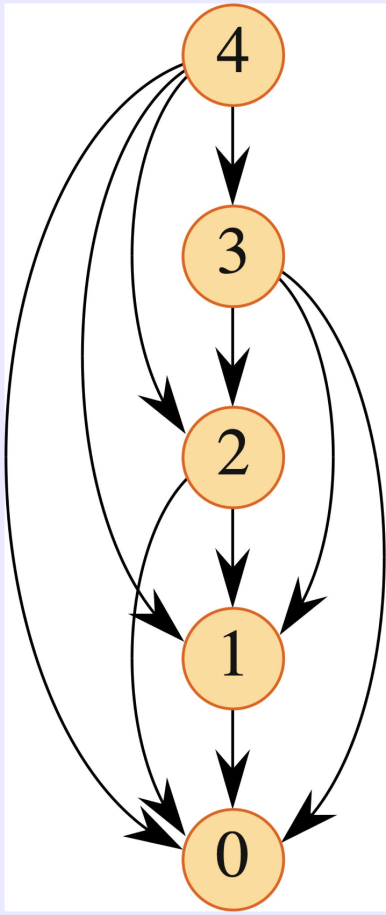
Bottom-Up Solution

```
BOTTOM-UP-CUT-ROD$(p, n)$
  let $r[0:n]$ be a new array          // will remember solution values in $r$
  $r[0] = 0$
  for $j = 1$ to $n$                   // for increasing rod length $j$
      $q = -\infty$
      for $i = 1$ to $j$               // $i$ is the position of the first cut
          $q = \max\{q, p[i] + r[j-i]\}$
      $r[j] = q$                       // remember the solution value for length $j$
  return $r[n]$
```

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

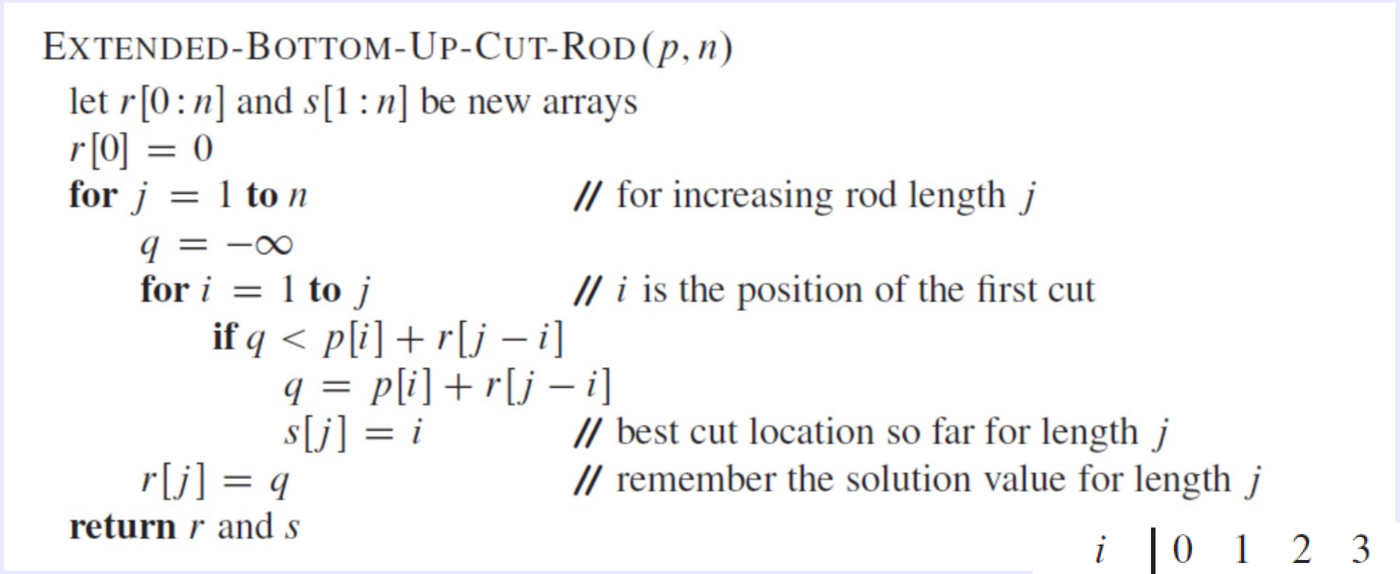
Subproblem graphs

- Can think of the subproblem graph as a collapsed version of the tree of recursive calls, where all nodes for the same subproblem are collapsed into a single vertex, and all edges go from parent to child.
- Subproblem graph can help determine running time. Because each subproblem is solved just once, running time is the sum of times needed to solve each subproblem.
- Time to compute the solution to a subproblem is typically linear in the out-degree (number of outgoing edges) of its vertex.
- Number of subproblems equals number of vertices.

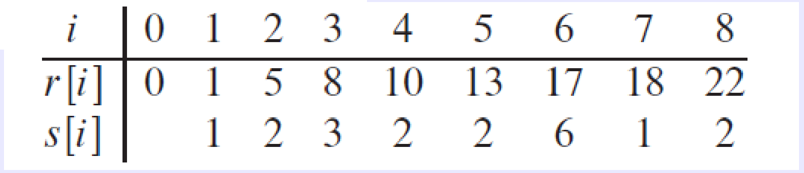


EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

- Extend the bottom-up approach to record not just optimal values, but optimal choices.
- Save the optimal choices in a separate table. Then, use a separate procedure to print the optimal choices.



```
EXTENDED-BOTTOM-UP-CUT-ROD($p, n$)
  let $r[0:n]$ and $s[1:n]$ be new arrays
  $r[0] = 0$
  for $j = 1$ to $n$                    // for increasing rod length $j$
      $q = -\infty$
      for $i = 1$ to $j$                // $i$ is the position of the first cut
          if $q < p[i] + r[j-i]$
              $q = p[i] + r[j-i]$
              $s[j] = i$                // best cut location so far for length $j$
      $r[j] = q$                        // remember the solution value for length $j$
  return $r$ and $s$
```



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

Dynamic Programming

- The previous strategy that applies “tables” is called dynamic programming (DP) [Here, programming means: a good way to plan things/to optimize the steps]
- A problem that can be solved efficiently by DP often has the following properties:

1. Optimal Substructure (allows recursion)
   - ✓ solution to the problem contains optimal solutions to subproblems.
2. Overlapping Subproblems (allows speed up)
   - ✓ a recursive algorithm revisits the same subproblem over and over again.

Practice at Home

Exercises: 14.1-1, 14.1-2, 14.1-3, 14.1-4, 14.1-6

Practice at home

- Peter is an owner of a Japanese sushi restaurant and today he invites you for dinner at his restaurant. In front of you are n sushi dishes that are arranged in a line. All dishes are different and they have different costs. Peter hopes that you can select the dishes you want, starting from the left to the right. However, there is a further restriction: when you select a dish, say A, the next dish you can select must cost higher than A. Design an $O(n^2)$-time algorithm to select the dishes so as to maximize the total costs.

Practice at Home

- There is a staircase with n steps. Your friend, Jack, wants to count how many different ways he can walk up this staircase. Because John is quite tall, in one move, he can choose either to walk up 1 step, 2 steps, or 3 steps. Let F_k denote the number of ways John can walk up a staircase with k steps. Derive a recurrence for F_k, and show that F_n can be computed in $O(n)$ time.

Practice at Home

- Let A[1..n] be an array of n distinct integers. Give an algorithm to find the length of the longest increasing subsequence of entries in A. The subsequence is not required to be contiguous in the original sequence. For example, if the entries are 11, 17, 5, 8, 6, 4, 7, 12, 3, the longest increasing subsequence is 5, 6, 7, 12. Analyze the worst-case running time and space requirement of your algorithm.