# Algorithms

## Ch.15
## 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.

# Four-step method

- 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

- 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 n

**Input:** A length n and table of prices pi , 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.

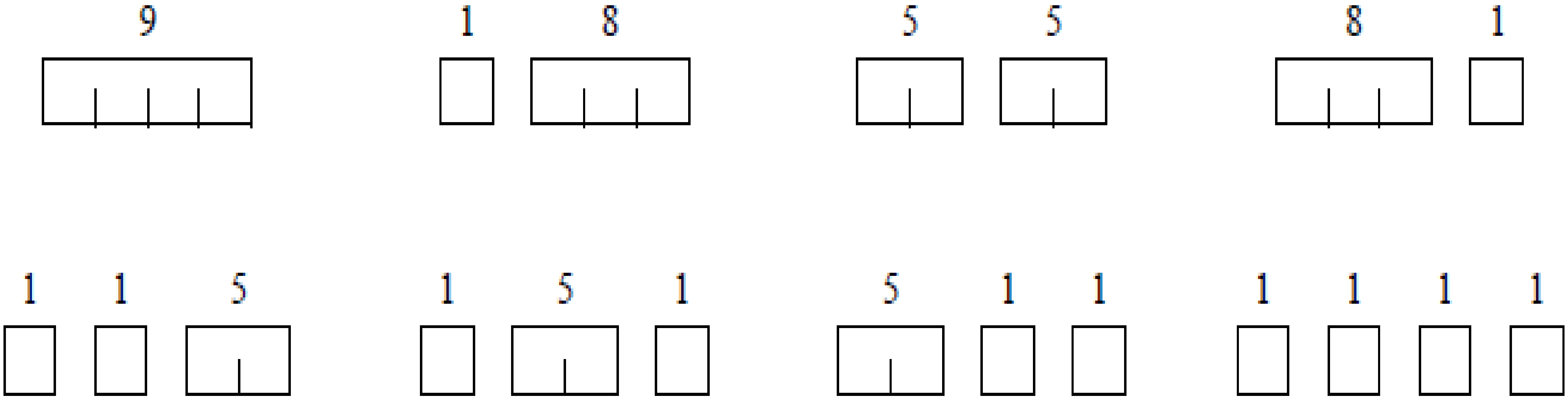
- If $p_n$ is large enough, an optimal solution might require no cuts, i.e., just leave the rod as n inches long.

# Example

| length $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 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:

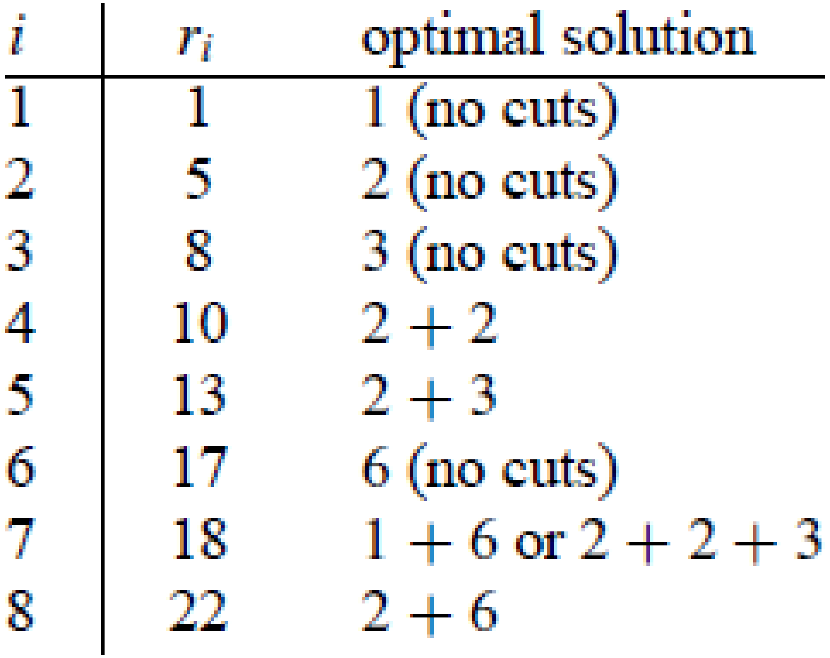


# Example…

- 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:



| $i$ | $r_i$ | 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 | $2 + 6$ |

# Example…

- 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, . . .
- $r_{n-1}$+ $r_1$.
- That is,

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

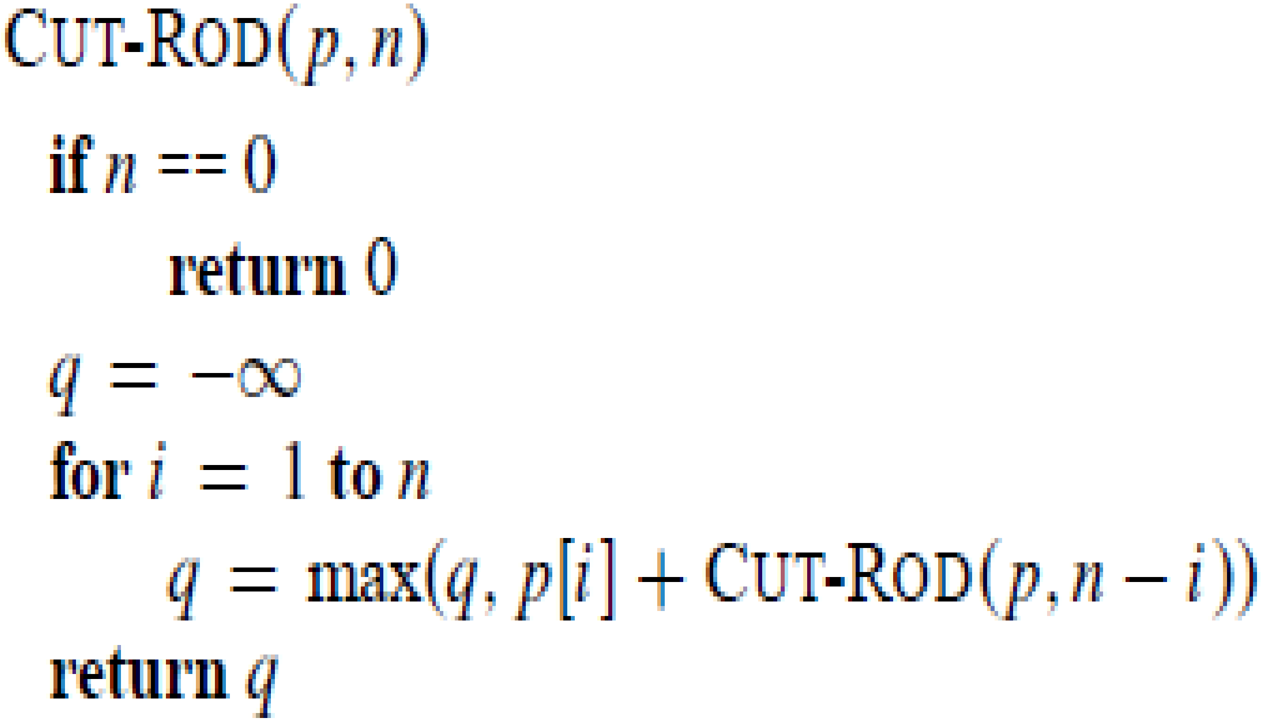
# *Optimal substructure*

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

- ***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.
- Need to divide only the remainder, not the first piece.
- Leaves only one sub-problem 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 $r_0 = 0$.
- Gives a simpler version of the equation for $r_n$:

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

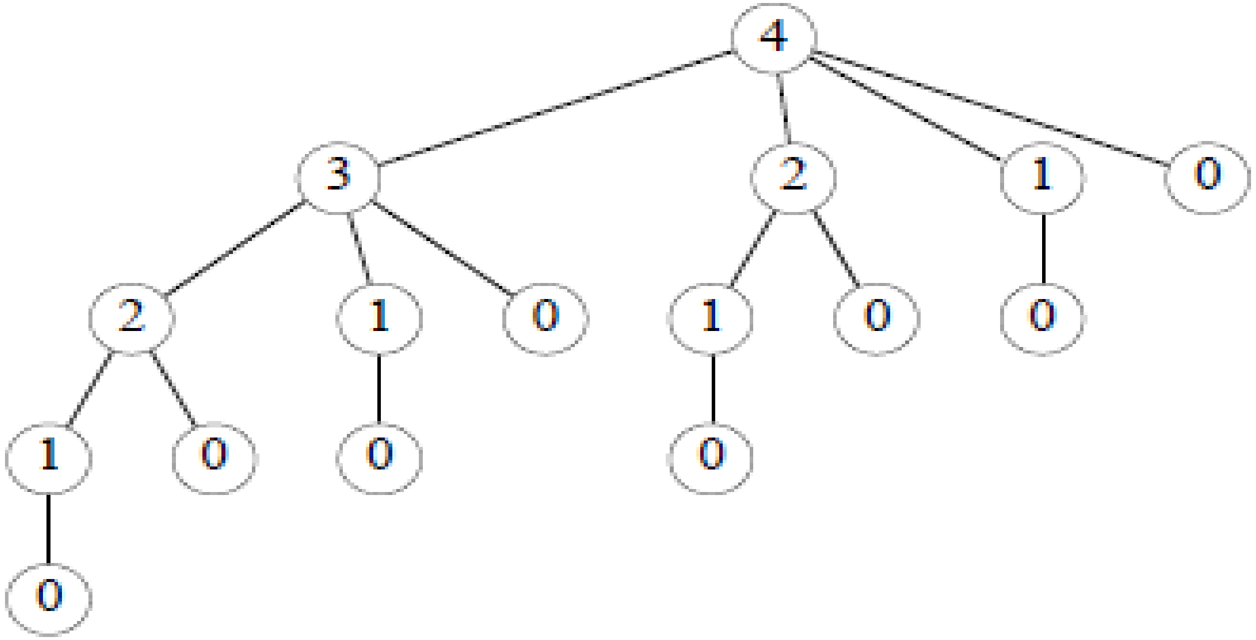
# 1-Recursive top-down solution



```
CUT-ROD($p, n$)
    if $n == 0$
        return 0
    $q = -\infty$
    for $i = 1$ to $n$
        $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$
    return $q$
```

This procedure works, but it is *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.

- ***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:



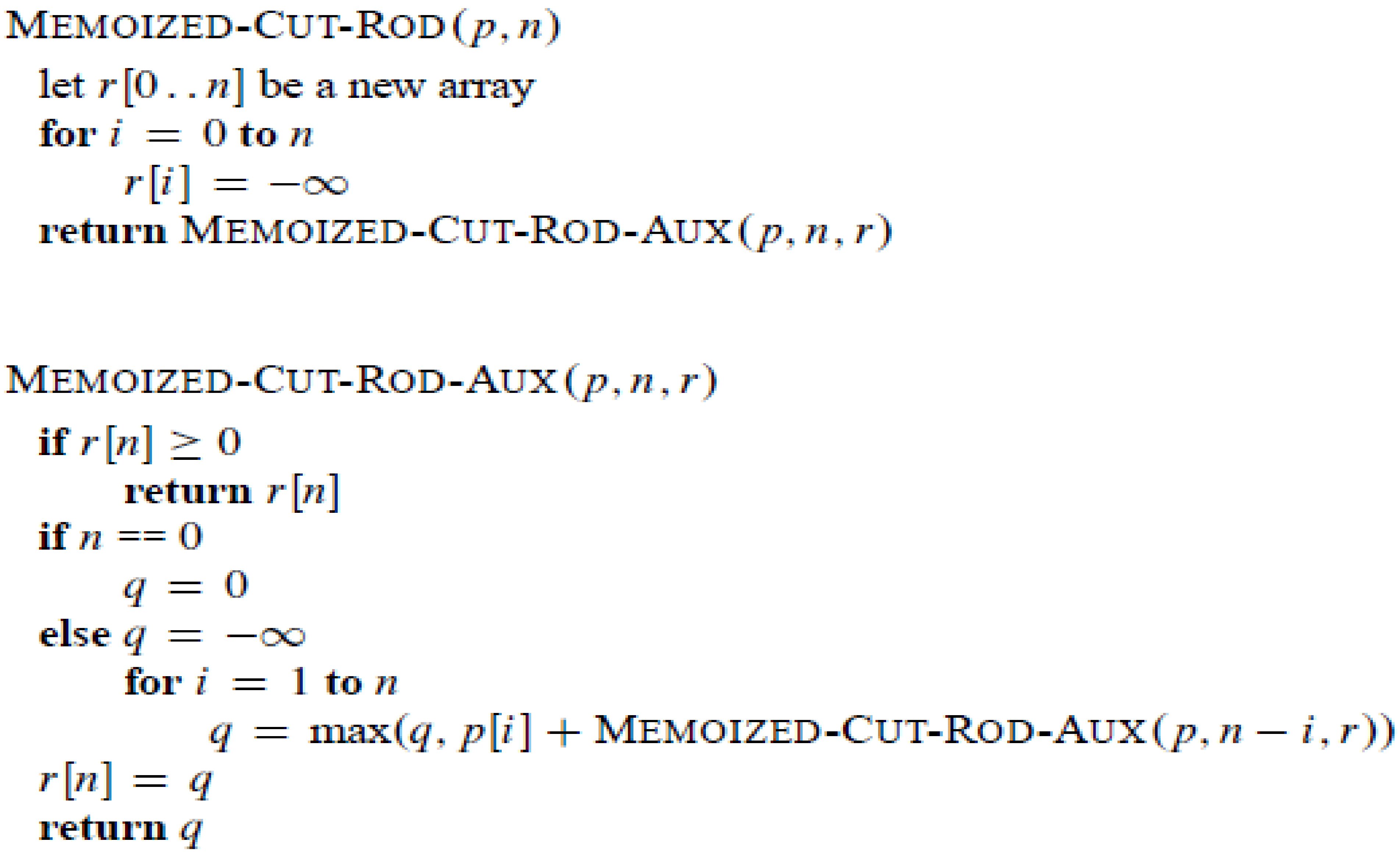
Lots of repeated subproblems. Solve the sub-problem for size 2 twice, for size 1 four times, and for size 0 eight times.

# 2-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, 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*

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

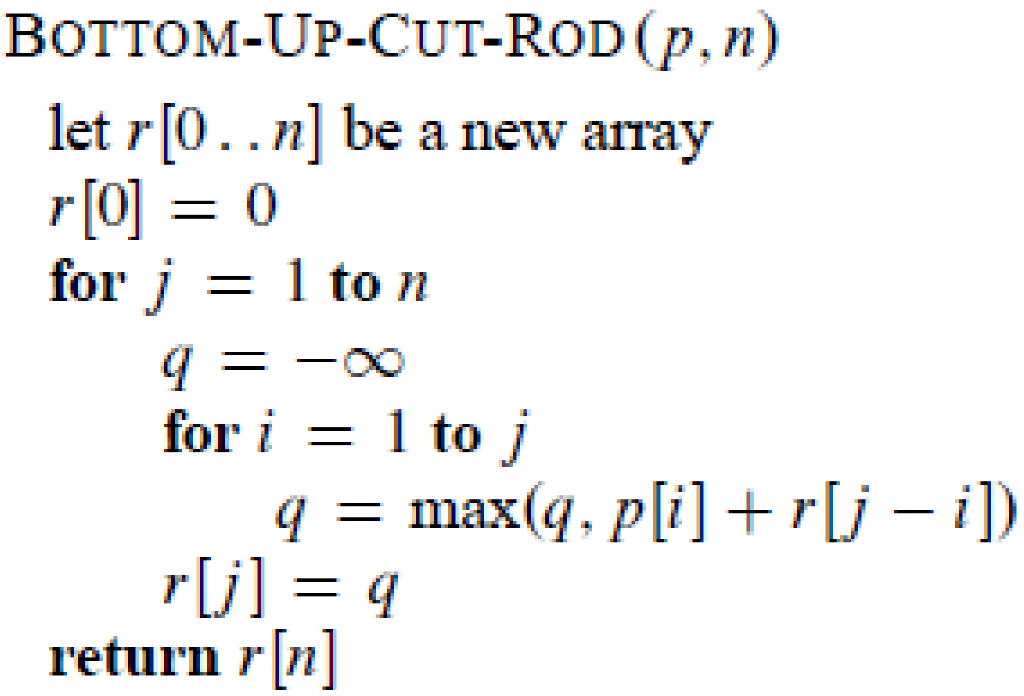


```
MEMOIZED-CUT-ROD$(p, n)$
    let $r[0..n]$ be a new array
    for $i = 0$ to $n$
        $r[i] = -\infty$
    return MEMOIZED-CUT-ROD-AUX$(p, n, r)$
```

```
MEMOIZED-CUT-ROD-AUX$(p, n, r)$
    if $r[n] \geq 0$
        return $r[n]$
    if $n == 0$
        $q = 0$
    else $q = -\infty$
        for $i = 1$ to $n$
            $q = \max(q, p[i] +$ MEMOIZED-CUT-ROD-AUX$(p, n - i, r))$
    $r[n] = q$
    return $q$
```

# *Bottom-up*

- 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.



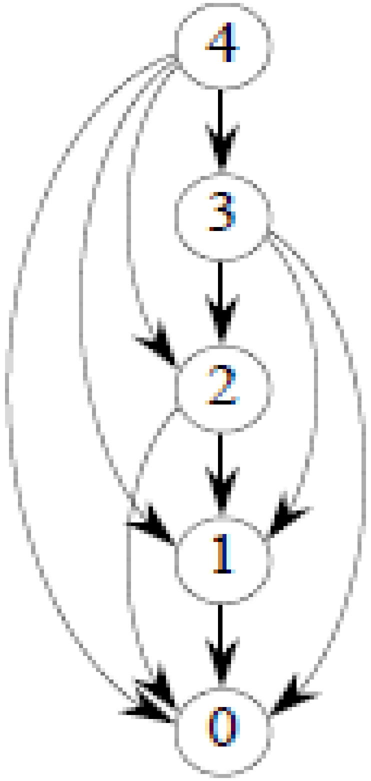
```
BOTTOM-UP-CUT-ROD$(p, n)$
  let $r[0\,..\,n]$ be a new array
  $r[0] = 0$
  for $j = 1$ to $n$
      $q = -\infty$
      for $i = 1$ to $j$
          $q = \max(q, p[i] + r[j - i])$
      $r[j] = q$
  return $r[n]$
```

# *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, ……, n. To solve a subproblem of size n, the **for** loop iterates n times
  - over all recursive calls, total number of iterations forms an arithmetic series

# Subproblem graphs

*Example:* For rod-cutting problem with $n = 4$:



# Subproblem graphs…

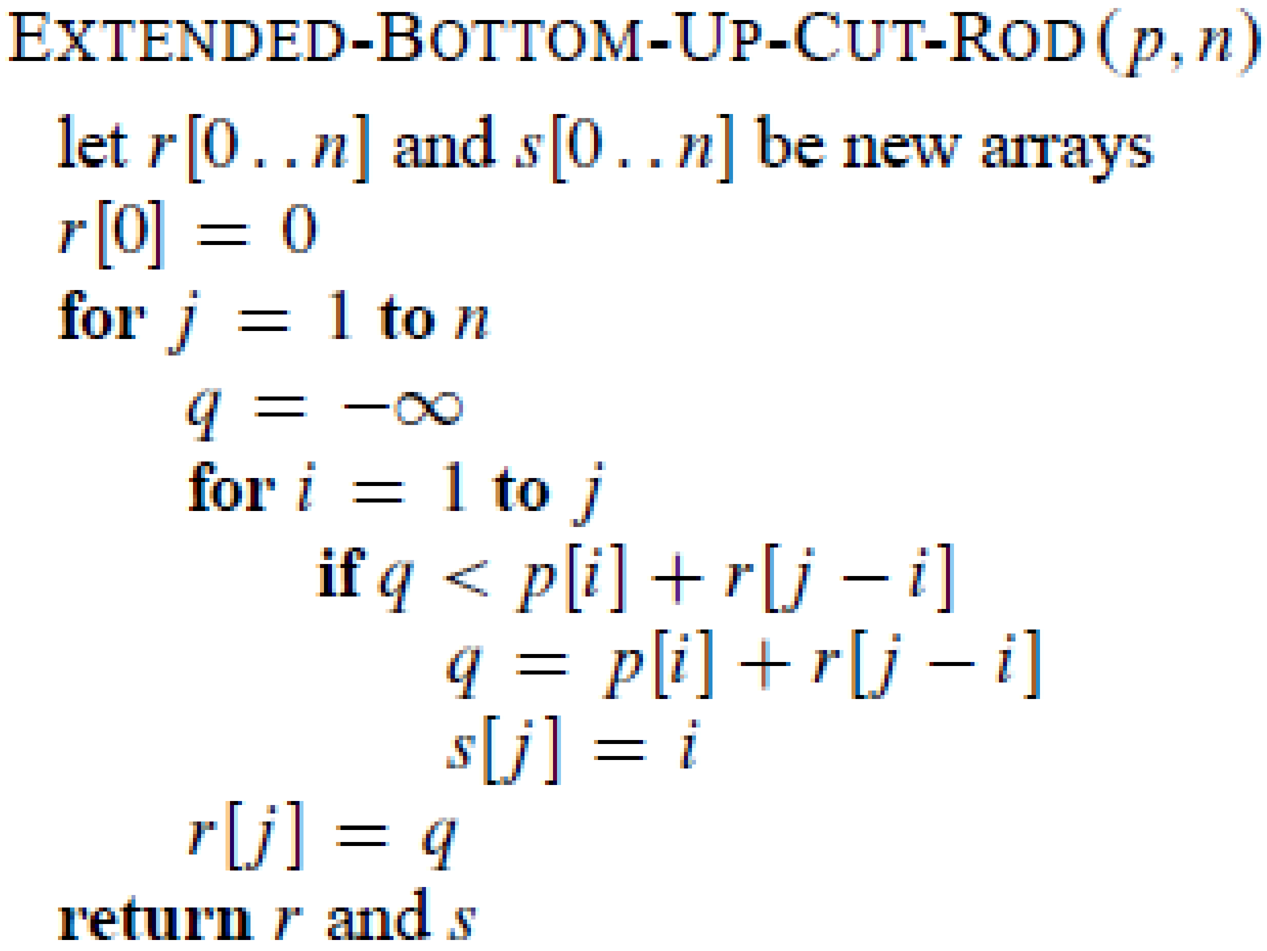
- Subproblem graph can help determine running time. Because we solve each subproblem just once, running time is sum of times needed to solve each subproblem.
- Time to compute solution to a subproblem is typically linear in the out-degree (number of outgoing edges) of its vertex.
- Number of subproblems equals number of vertices.

When these conditions hold, running time is linear in number of vertices and edges.

# Reconstructing a solution

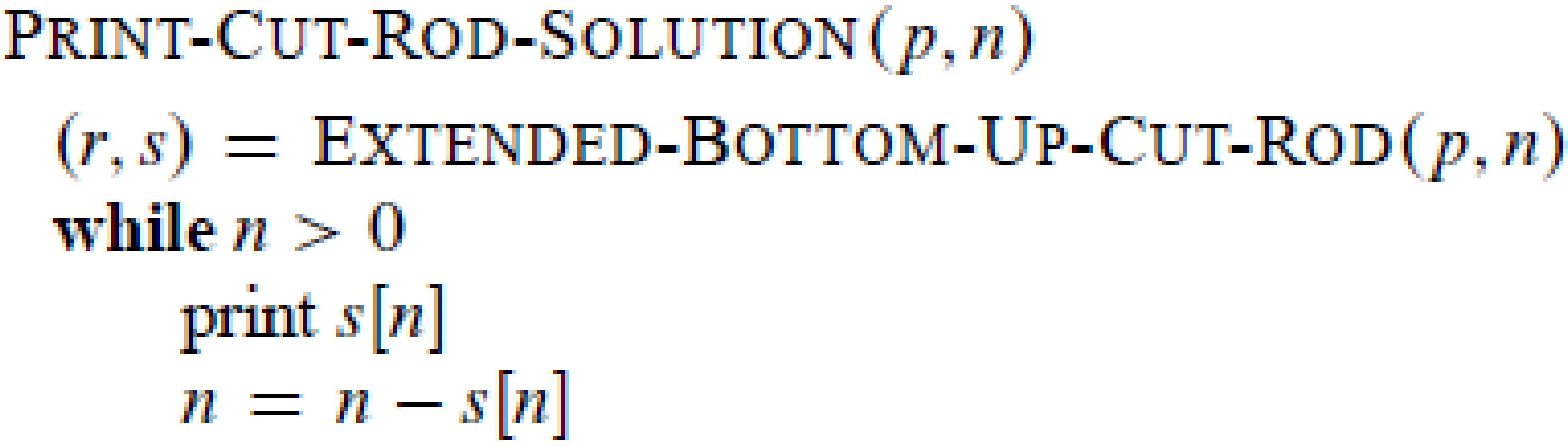
- So far, have focused on computing the *value* of an optimal solution, rather than the *choices* that produced an optimal solution.
- Extend the bottom-up approach to record not just optimal values, but optimal choices. Save the optimal choices in a separate table. Then use a separate procedure to print the optimal choices.

# Reconstructing a solution



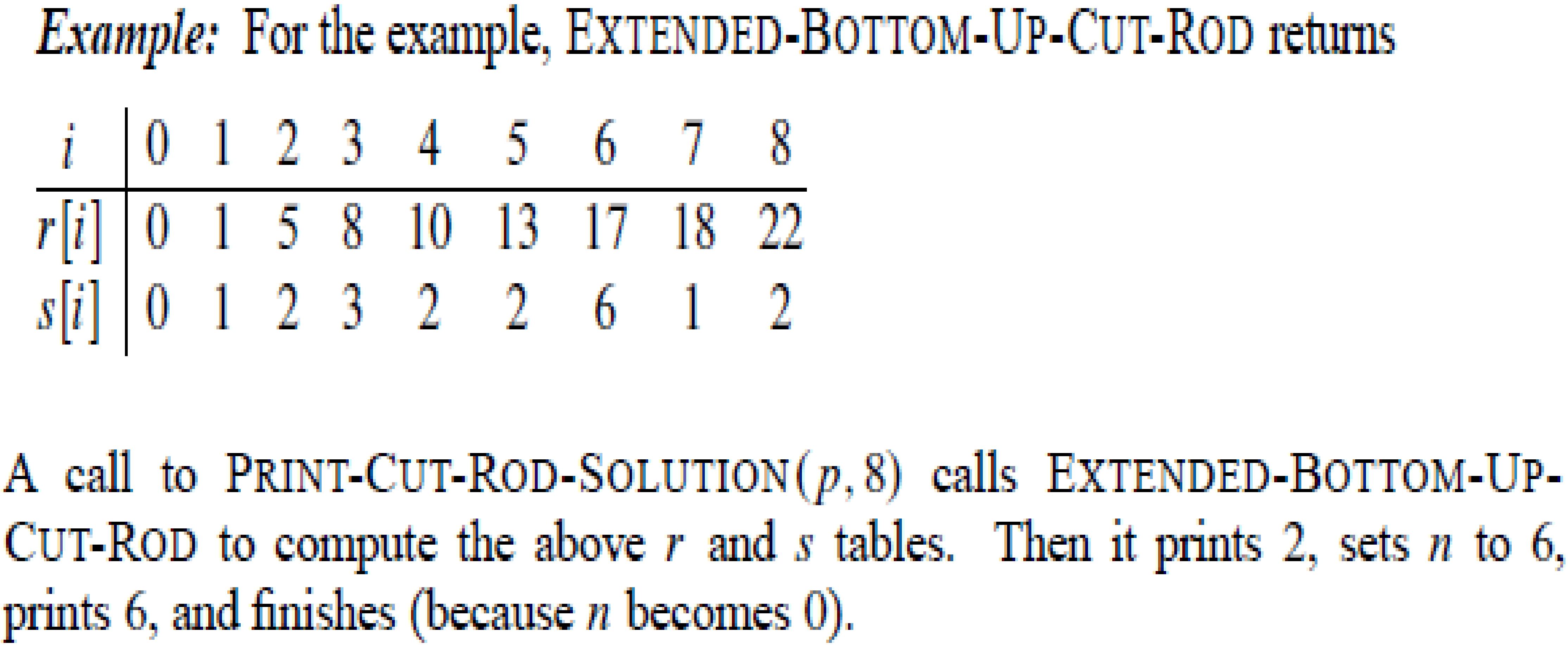
```
EXTENDED-BOTTOM-UP-CUT-ROD($p, n$)
  let $r[0..n]$ and $s[0..n]$ be new arrays
  $r[0] = 0$
  for $j = 1$ to $n$
      $q = -\infty$
      for $i = 1$ to $j$
          if $q < p[i] + r[j-i]$
              $q = p[i] + r[j-i]$
              $s[j] = i$
      $r[j] = q$
  return $r$ and $s$
```

# Reconstructing a solution



```
PRINT-CUT-ROD-SOLUTION$(p, n)$
  $(r, s) =$ EXTENDED-BOTTOM-UP-CUT-ROD$(p, n)$
  while $n > 0$
      print $s[n]$
      $n = n - s[n]$
```

# Example



*Example:* For the example, EXTENDED-BOTTOM-UP-CUT-ROD returns

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

A call to PRINT-CUT-ROD-SOLUTION$(p, 8)$ calls EXTENDED-BOTTOM-UP-CUT-ROD to compute the above $r$ and $s$ tables. Then it prints 2, sets $n$ to 6, prints 6, and finishes (because $n$ becomes 0).