Rod Cutting Problem

Problem Statement

Rod Cutting Problem:
Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots, n$, determine the maximum revenue obtainable by cutting the rod and selling the pieces.

Example Price Table:

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

Problem Statement

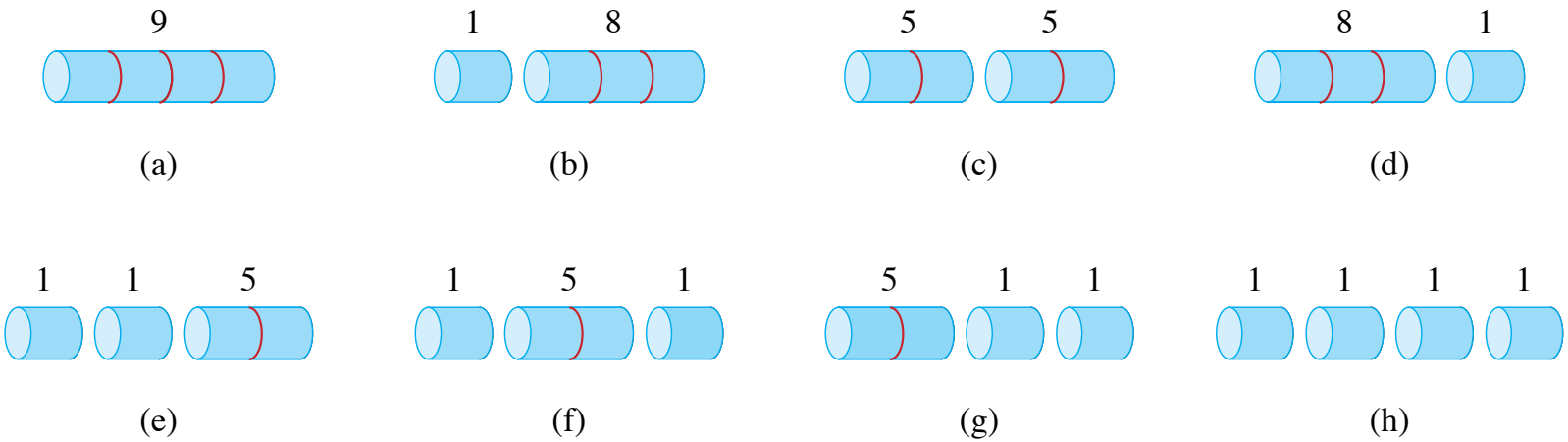


Figure 14.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 14.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

Recursive Formulation

Let r_n be the maximum revenue for a rod of length n. Then:

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

Base case: $r_0 = 0$

Exponential Time Algorithm:

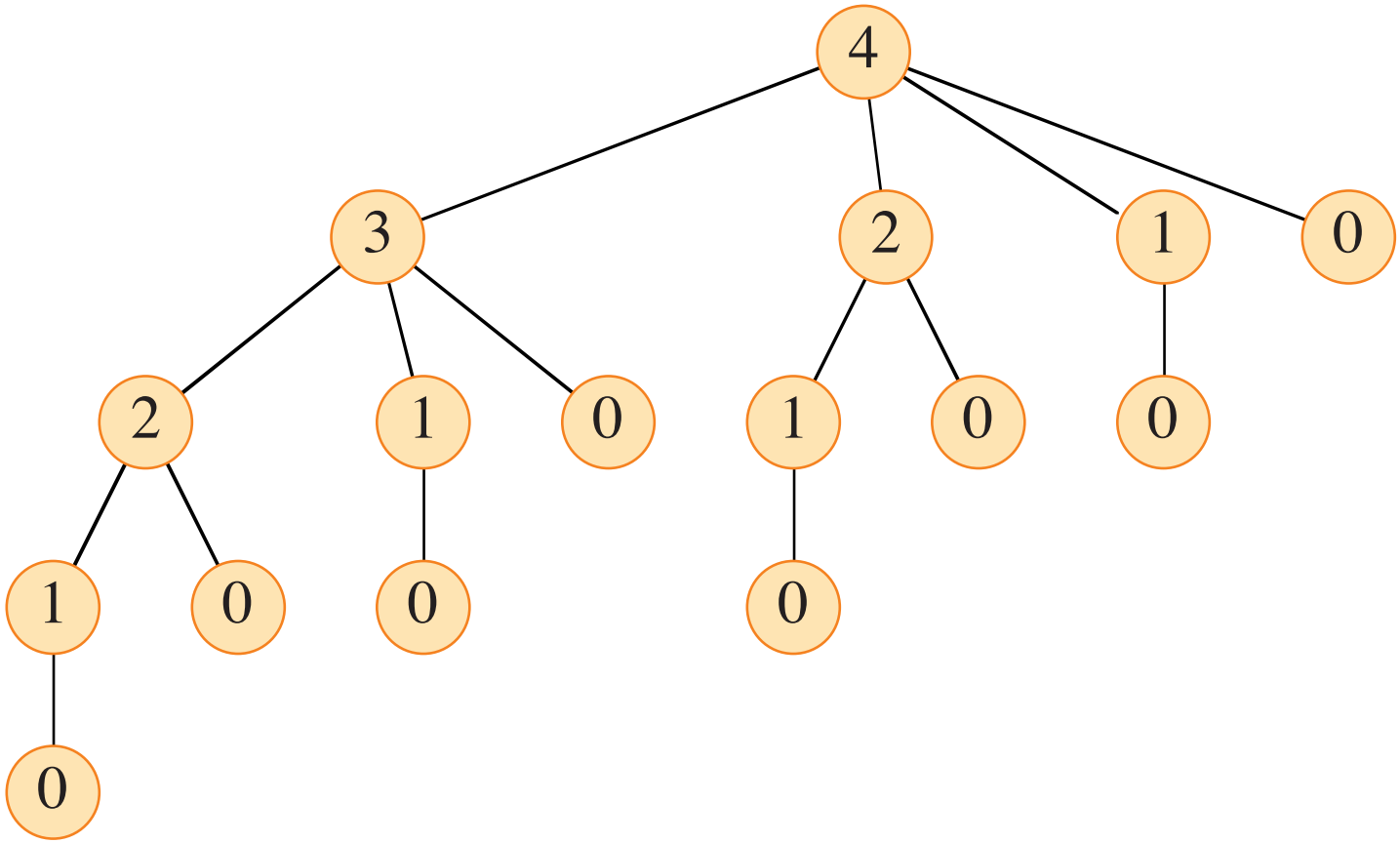
```
CUT-ROD(p, n)
1  if n == 0
2       return 0
3  q = -∞
4  for i = 1 to n
5       q = max {q, p[i] + CUT-ROD(p, n - i)}
6  return q
```

Recursive Formulation

- Let $T(n)$ denote the total number of calls made to $\text{CUT-ROD}(p, n)$ for a particular value of n.
- The number of nodes in a subtree whose root is labeled n in the recursion tree.

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

Recursive Formulation



Memoized Version

Avoids recomputation by storing already computed values.

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0:n] be a new array          // will remember solution values in r
2  for i = 0 to n
3      r[i] = −∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0                       // already have a solution for length n?
2      return r[n]
3  if n == 0
4      q = 0
5  else q = −∞
6      for i = 1 to n                // i is the position of the first cut
7          q = max{q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r)}
8  r[n] = q                          // remember the solution value for length n
9  return q
```

Bottom-Up Dynamic Programming

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

Time complexity: $\mathcal{O}(n^2)$

Subproblem graphs

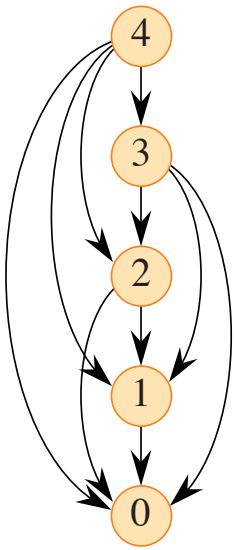


Figure 14.4 The subproblem graph for the rod-cutting problem with $n = 4$. The vertex labels give the sizes of the corresponding subproblems. A directed edge (x, y) indicates that solving subproblem x requires a solution to subproblem y. This graph is a reduced version of the recursion tree of Figure 14.3, in which all nodes with the same label are collapsed into a single vertex and all edges go from parent to child.

Reconstructing the Solution

Store the optimal cuts as well:

- Maintain a second array $s[1..n]$.
- $s[j]$ stores the length of the first piece to cut from a rod of length j.

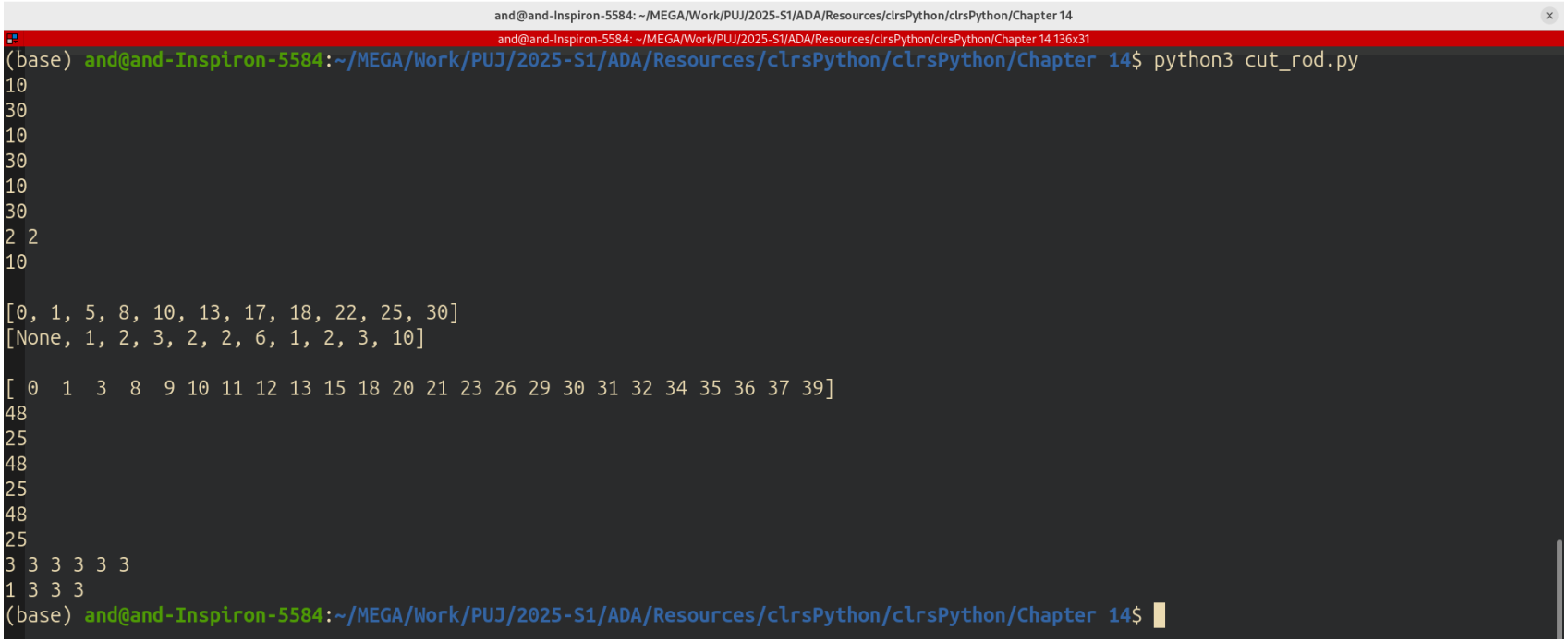
Then backtrack using $s[n]$ to print the actual cuts.

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
$s[i]$		1	2	3	2	2	6	1	2	3	10

Reconstructing a solution

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  let r[0:n] and s[1:n] be new arrays
 2  r[0] = 0
 3  for j = 1 to n                    // for increasing rod length j
 4      q = -∞
 5      for i = 1 to j                // i is the position of the first cut
 6          if q < p[i] + r[j - i]
 7              q = p[i] + r[j - i]
 8              s[j] = i              // best cut location so far for length j
 9      r[j] = q                      // remember the solution value for length j
10  return r and s
```

Reconstructing a solution



Key Takeaways

- Naive recursive solution has exponential time complexity
- Dynamic programming reduces it to $\mathcal{O}(n^2)$
- Optimal substructure and overlapping subproblems make DP suitable