Bottom-Up Method

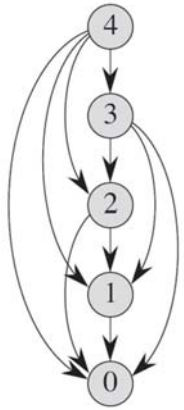
```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = −∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j − i])
7      r[j] = q
8  return r[n]
```

- Line 6 directly references array entry $r[j - i]$ instead of making a recursive call to solve the subproblem of size $j - i$.

Subproblem Graphs (1/4)

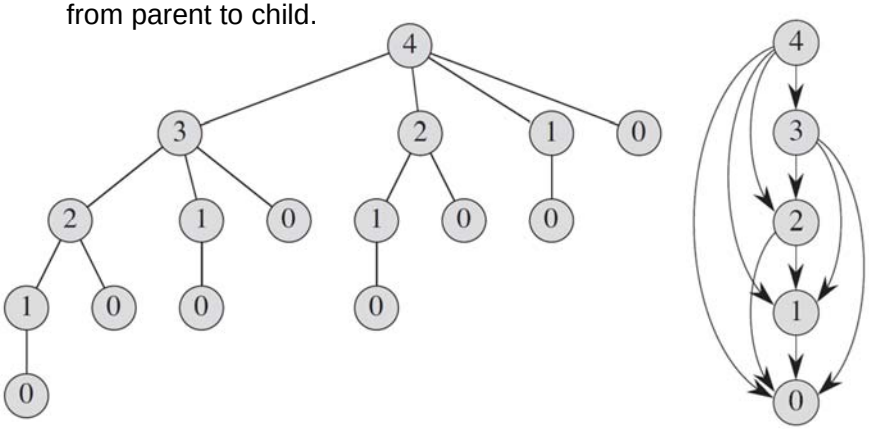
- When we think about a dynamic-programming problem, we should understand the set of subproblems involved and how subproblems depend on one another.

The subproblem graph for the rod-cutting problem with $n = 4$.



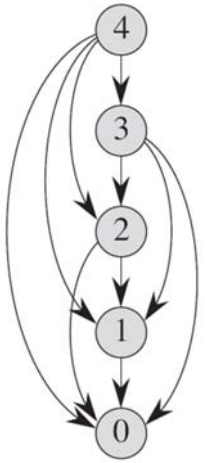
Subproblem Graphs (2/4)

- We can think of the subproblem graph as a "reduced" or "collapsed" version of the recursion tree for the top-down recursive method, in which we coalesce all nodes for the same subproblem into a single vertex and direct all edges from parent to child.



Subproblem Graphs (3/4)

- In a bottom-up dynamic-programming algorithm, we consider the vertices of the subproblem graph in an order that is a "reverse topological sort," or a "topological sort of the transpose" of the subproblem graph.



Subproblem Graphs (4/4)

- Typically, the time to compute the solution to a subproblem is proportional to the degree (number of outgoing edges) of the corresponding vertex in the subproblem graph, and the number of subproblems is equal to the number of vertices in the subproblem graph.

Reconstructing a Solution (1/3)

- Our dynamic-programming solutions to the rod-cutting problem return the value of an optimal solution, but they do not return an actual solution: a list of piece sizes.
- ➡ We can extend the dynamic-programming approach to record not only the optimal value computed for each subproblem, but also a choice that led to the optimal value.

Reconstructing a Solution (2/3)

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  let r[0..n] and s[0..n] be new arrays
 2  r[0] = 0
 3  for j = 1 to n
 4      q = -∞
 5      for i = 1 to j
 6          if q < p[i] + r[j - i]
 7              q = p[i] + r[j - i]
 8              s[j] = i
 9      r[j] = q
10  return r and s

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


Reconstructing a Solution (3/3)

- In our rod-cutting example, the call EXTENDED-BOTTOM-UP-CUT-ROD(*p*,10) would return the following arrays:

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
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

- What would PRINT-CUT-ROD-SOLUTION(*p*,10) print?
- What would PRINT-CUT-ROD-SOLUTION(*p*,7) print?