Design & Analysis of Algorithms -I

Dynamic Programming

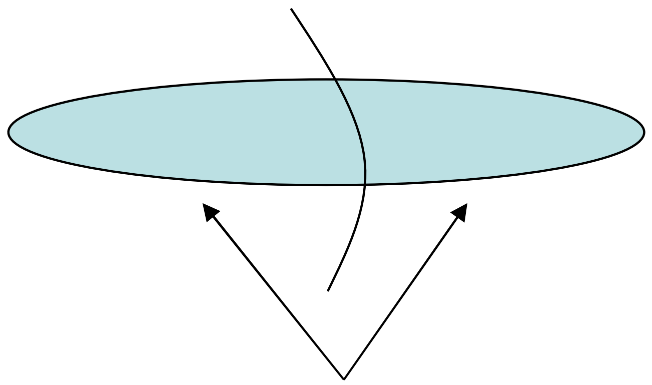
Dynamic Programming

- An algorithm design technique (like divide and conquer)
- Divide and conquer
  - Partition the problem into independent/disjoint subproblems
  - Solve the subproblems recursively
  - Combine the solutions to solve the original problem

DP - Two key ingredients

- Two key ingredients for an optimization problem to be suitable for a dynamic-programming solution:

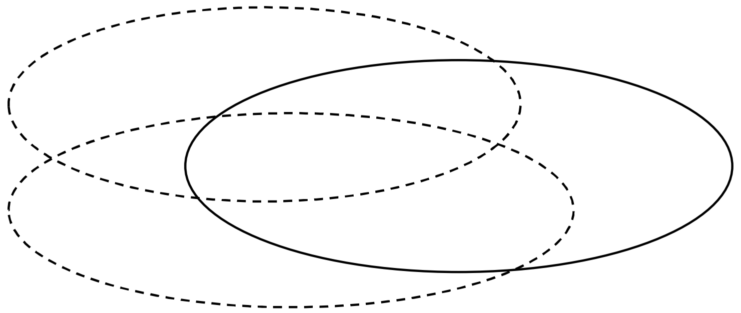
1. optimal substructures



Each substructure is optimal.

(Principle of optimality)

2. overlapping subproblems



Subproblems are dependent.

(otherwise, a divide-and-conquer approach is the choice.)

Three basic components

- The development of a dynamic-programming algorithm has three basic components:
  - The recurrence relation (for defining the value of an optimal solution);
  - The tabular computation (for computing the value of an optimal solution);
  - The traceback (for delivering an optimal solution).

Fibonacci numbers

The *Fibonacci numbers* are defined by the following recurrence:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2} \quad \text{for } i > 1.$$

How to compute F_{10}?

$$\begin{tikzcd}
 & & F_8 & \\
 & F_9 \arrow[ur, no head] \arrow[dr, no head] & & \\
 & & F_7 & \\
F_{10} \arrow[uur, no head] \arrow[ddr, no head] & & & \cdots\cdots \\
 & & F_7 & \\
 & F_8 \arrow[ur, no head] \arrow[dr, no head] & & \\
 & & F_6 &
\end{tikzcd}$$

Dynamic Programming

- Applicable when subproblems are not independent
  - Subproblems share sub-subproblems

E.g.: Fibonacci numbers:

- Recurrence: $F(n) = F(n-1) + F(n-2)$
- Boundary conditions: $F(1) = 0$, $F(2) = 1$
- Compute: $F(5) = 3$, $F(3) = 1$, $F(4) = 2$

  - A divide and conquer approach would **repeatedly** solve the common subproblems
  - Dynamic programming solves every subproblem just once and stores the answer in a table

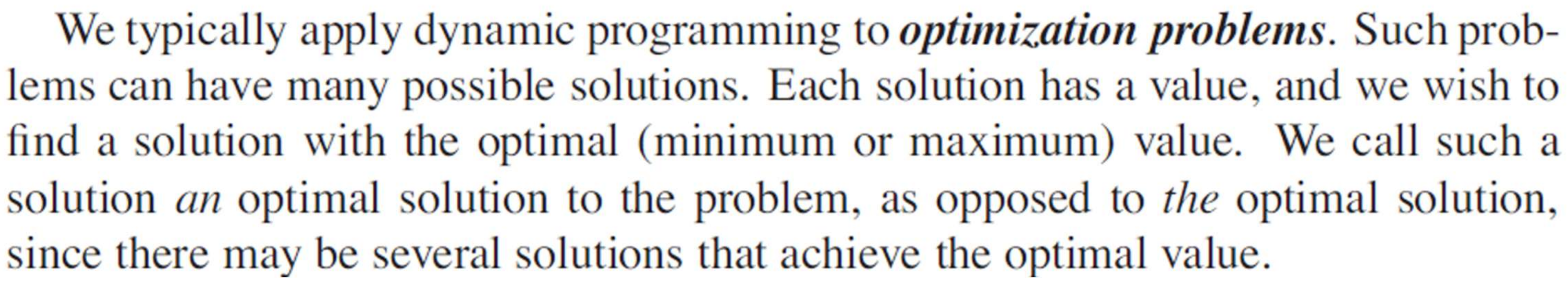
Tabular computation

- The tabular computation can avoid recompuation.

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
0	1	1	2	3	5	8	13	21	34	55

Result

The DP Methodology



We typically apply dynamic programming to ***optimization problems***. Such problems can have many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution *an* optimal solution to the problem, as opposed to *the* optimal solution, since there may be several solutions that achieve the optimal value.

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 in a bottom-up fashion
4. Construct an optimal solution from computed information

- Steps 1–3 form the basis of a dynamic-programming solution to a problem.
- For the value of an **optimal solution**, and not **the solution itself**, then we can omit step 4.
- When we do perform step 4, we sometimes maintain additional information during step 3 so that we can easily construct an optimal solution.

Rod Cutting

- Serling Enterprises buys long steel rods and cuts them into shorter rods
  - which it then sells.
- Each cut is free. The management of Serling enterprises wants to know the best way to cut up the rods.

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

The ***rod-cutting problem*** is the following. Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

We can cut up a rod of length n in 2^{n-1} different ways

Rod Cutting

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

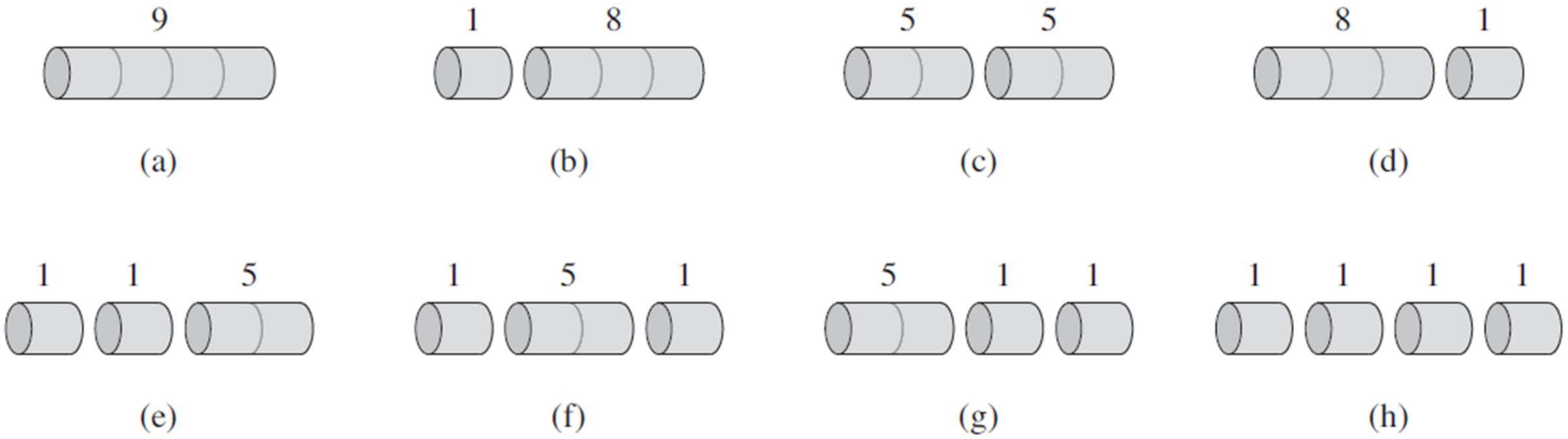


Figure 15.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 15.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

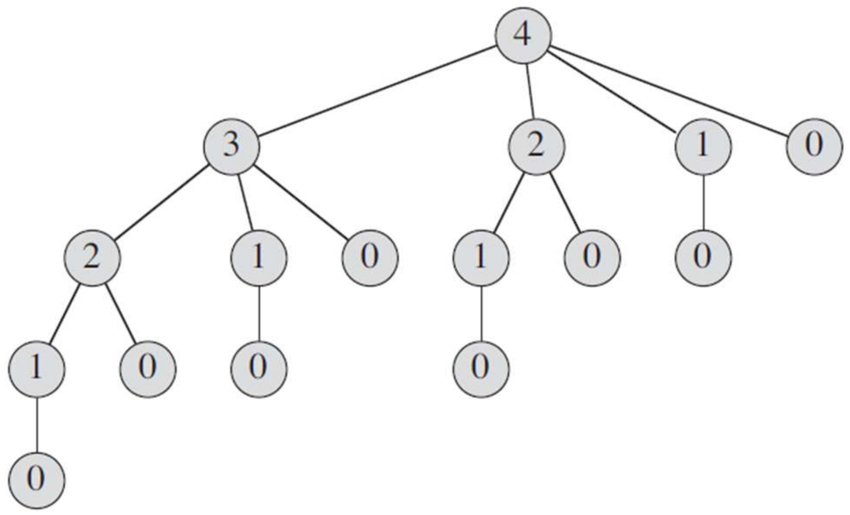
We can cut up a rod of length n in 2^{n-1} different ways.

Rod Cutting

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

- For n = 40, the program takes at least several minutes, and most likely more than an hour.
- In fact, each time you increase n by 1, your program's running time would approximately double.
- this recursion tree has 2^n nodes and 2^{n-1} leaves.

```
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] + $ CUT-ROD$(p, n - i))$
6  return $q$
```

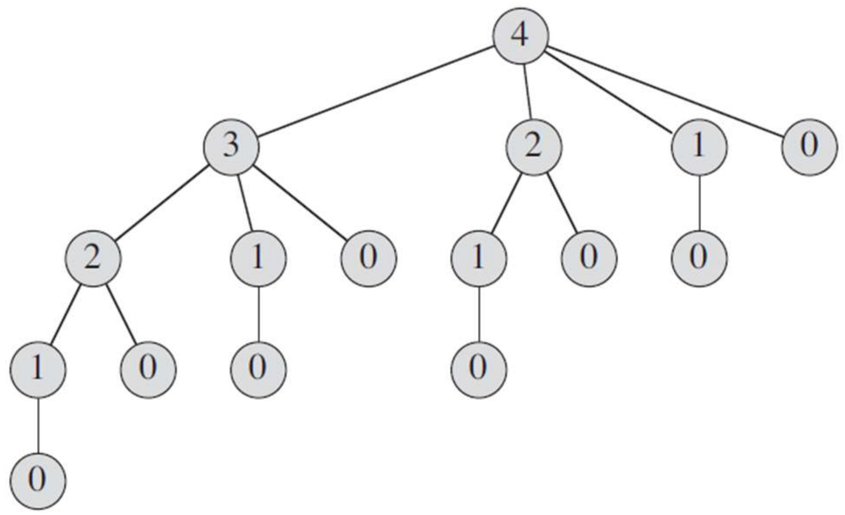


We can cut up a rod of length n in 2^{n-1} different ways.

Rod Cutting – using DP

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

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

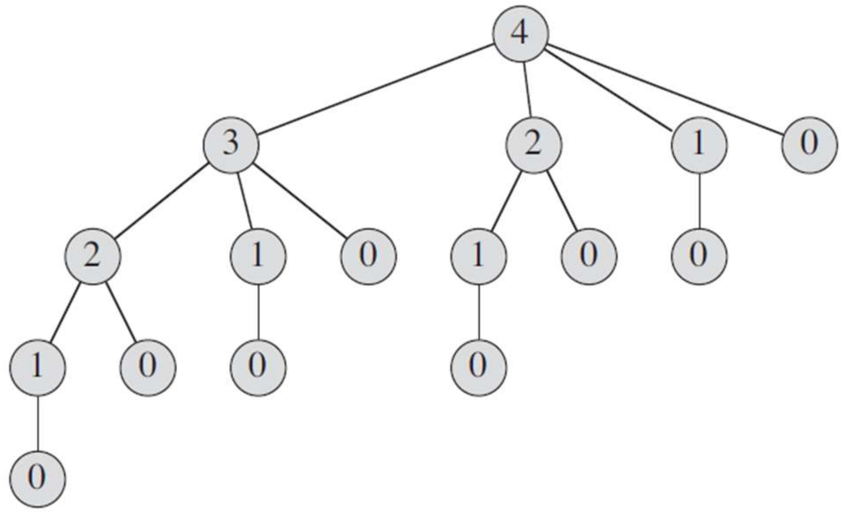


- The dynamic-programming method works as follows. Having observed that a naive recursive solution is inefficient because it solves the same subproblems repeatedly
- we arrange for each subproblem to be solved only *once*, saving its solution.
- If we need to refer to this subproblem's solution again later, we can just look it up, rather than compute it.

Rod Cutting – using DP

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

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



- Dynamic programming thus uses additional memory to save computation time; it serves an example of a **time-memory trade-off.**

- The savings may be dramatic: an exponential-time solution may be transformed into a polynomial-time solution.

A dynamic-programming approach runs in **polynomial time** when the **number of distinct subproblems involved is polynomial** in the input size and we can solve each such subproblem in polynomial time.

DP – Two variants

top-down with memoization

- we write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in an array or hash table).
- The procedure now first checks to see whether it has previously solved this subproblem.
  - If so, it returns the saved value, saving further computation at this level;
  - if not, the procedure computes the value in the usual manner.
- We say that the recursive procedure has been **memoized**;
- it "remembers" what results it has computed previously.

bottom-up method.

- This approach typically depends on some natural notion of the "size" of a subproblem, such that solving any particular subproblem depends only on solving "**smaller**" subproblems.
- We sort the subproblems by size and solve them in size order, smallest first.
- When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon, and we have saved their solutions.
- We solve each subproblem only once, and when we first see it, we have already solved all of its prerequisite subproblems.

DP – Two variants

- These two approaches yield algorithms with the same asymptotic running time, except in unusual circumstances where the top-down approach does not actually recurse to examine all possible subproblems.
- The bottom-up approach often has much better constant factors, since it has less overhead for procedure calls.

Rod Cutting

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

top-down with memoization

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = −∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
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
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = −∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

- The procedure MEMOIZED-CUT-ROD-AUX is just the memoized version of our previous procedure, CUT-ROD

Rod Cutting

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

top-down with memoization

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = −∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

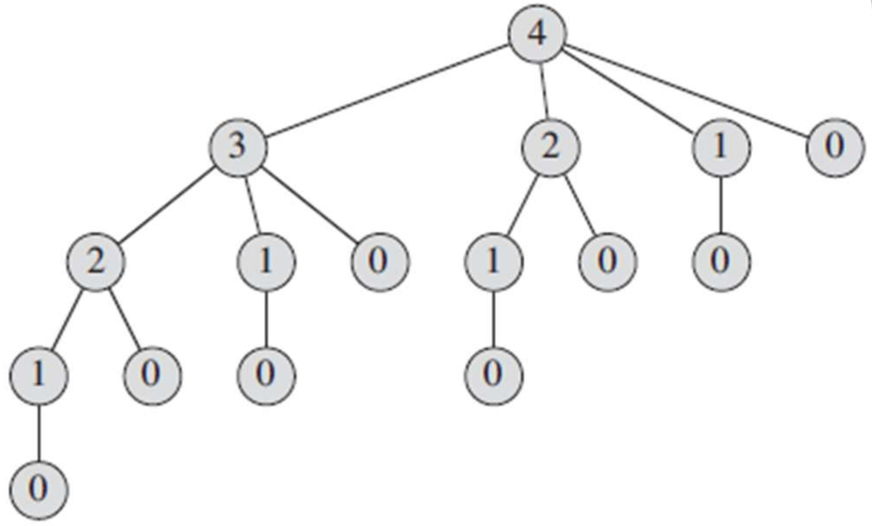
```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = −∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

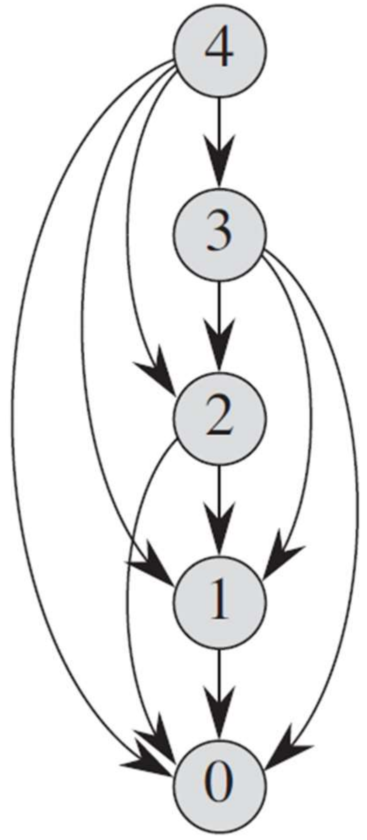
```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = −∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j − i])
7      r[j] = q
8  return r[n]
```

Rod Cutting

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

```
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      $q = -\infty$
5      for $i = 1$ to $j$
6          $q = \max(q, p[i] + r[j-i])$
7      $r[j] = q$
8  return $r[n]$
```





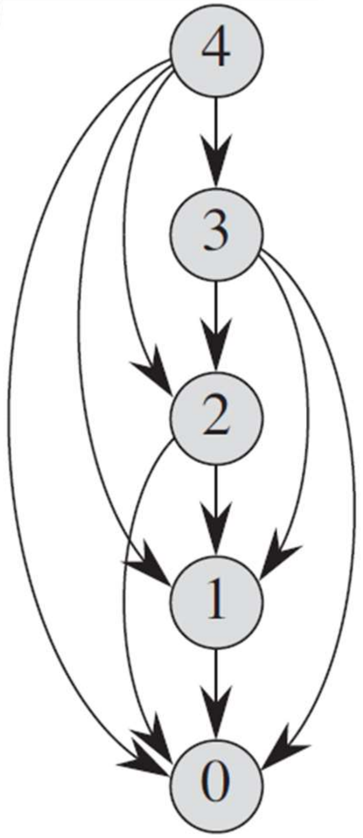
A directed edge (x, y) indicates that we need a solution to subproblem y when solving subproblem x.

This graph is a reduced version of the first tree, in which all nodes with the same label are collapsed into a single vertex and all edges go from parent to child.

Rod Cutting

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

```
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]
```



- 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.

With this information, we can readily print an optimal solution.

Rod Cutting

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

```
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      $q = -\infty$
5      for $i = 1$ to $j$
6          $q = \max(q, p[i] + r[j-i])$
7      $r[j] = q$
8  return $r[n]$
```

```
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      $q = -\infty$
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      $n = n - s[n]$
```

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