Dynamic Programming

Dr. Ing. Gianmaria Silvello

Department of Information Engineering
University of Padua
silvello@dei.unipd.it
http://www.dei.unipd.it/~silvello/

Outline

- Elements of Dynamic Programming
- Case 1: Rod Cutting
- Case 2: Longest Common Subsequence
- Case 3: Text Justification

Reference: Chapter 15 of CLRS

Reference: Chapter 10 of Goodrich, Tamassia and Goldwasser

Dynamic Programming

- Break up a problem into a series of *overlapping subproblems*, and build up solutions to larger and larger subproblems
- (fancy name for caching away intermediate results in a table for later reuse)
- Dynamic programming is a way to speed-up certain classes of inefficient recursive algorithms (e.g. divide and conquer does not work well)
  - **Inefficient**: the same recursive call is made many times

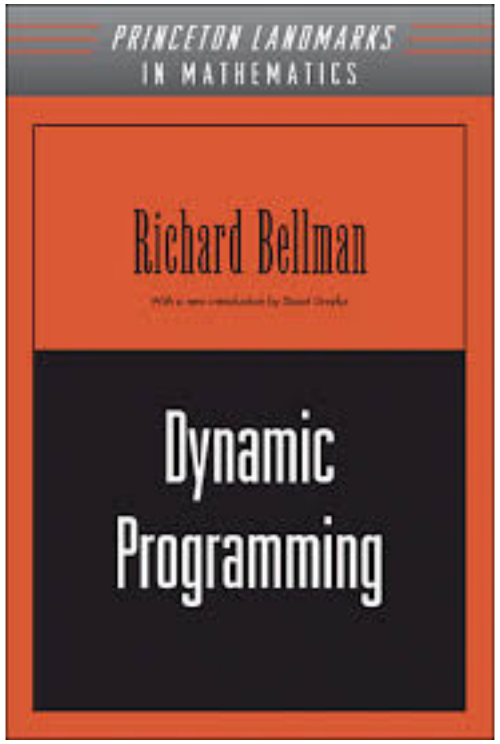
Dynamic Programming

- If the same call is made many times it means that the same problem is being solved many times over
  - Use a (hash) table to store intermediate results
  - Recover the result for an already solved problem
  - Skip several repeated recursive calls

Dynamic Programming (Historical overview)

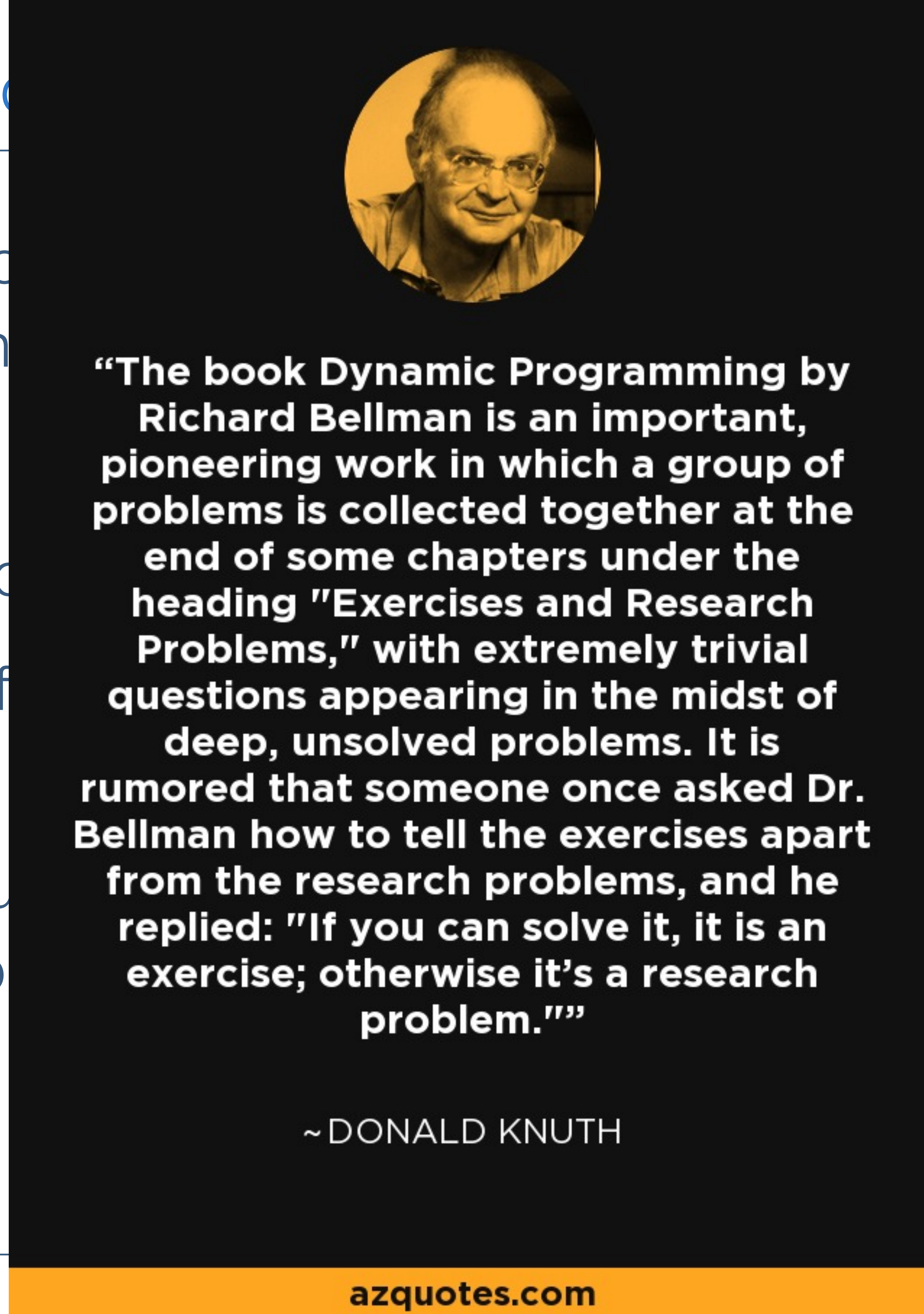
- Bellman. Pioneered the systematic study of dynamic programming in 1950s.

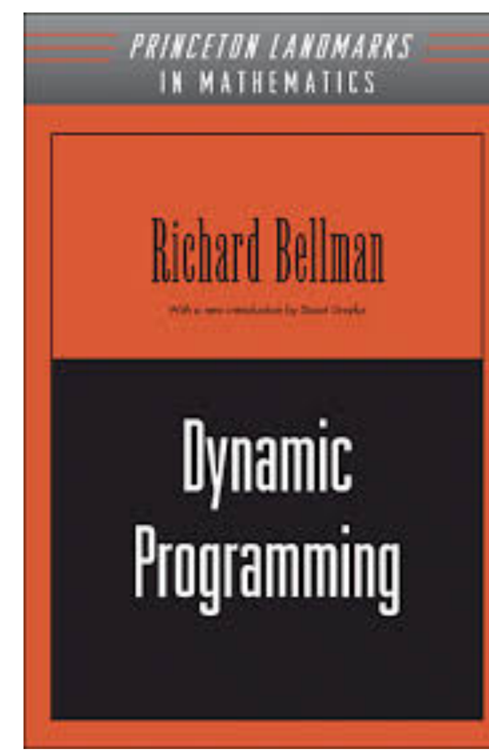
- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an *impressive name* to avoid confrontation.



Dynami… …view)

- Bellman. Pio… …namic programmin…
- Dynamic pro…
- Secretary of… …cal research.
- Bellman sou… confrontatio…





Dynamic Programming

- Some famous dynamic programming algorithms.
  - Unix diff for comparing two files.
  - Viterbi for hidden Markov models.
  - De Boor for evaluating spline curves.
  - Smith–Waterman for genetic sequence alignment.
  - Bellman–Ford for shortest path routing in networks.
  - Cocke–Kasami–Younger for parsing context-free grammars.

Elements of Dynamic Programming

- Simple subproblems
  - We should be able to break the main problem into smaller subproblems sharing the same structure
- Optimal substructure of the subproblems
  - The optimal solution of a problem contains within the optimal solution of its subproblems
- Overlapping subproblems
  - There exists a place where the same subproblems are solved more than once

Case 1:
Rod Cutting

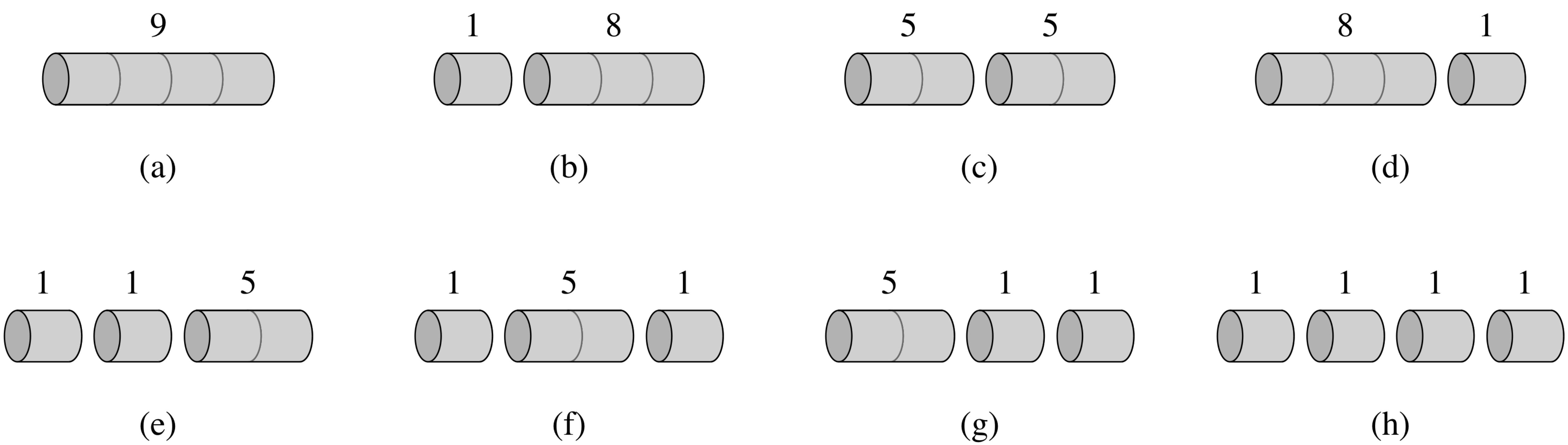
Rod Cutting

- Problem: Given a rod of length *n* inches and a table of prices, determine the maximum revenue obtainable by cutting up the rod and selling the pieces
- Rod cuts are an integral number of inches, cuts are free
- Price table for 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

Rod Cutting

- Eight possible ways to cut a rod of length 4 (prices shown on top)



Rod Cutting

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1) \qquad r_n = \max_{1 \le i \le n}(p_i + r_{n-i})$$

- p_n = no cuts and selling the rod as is
- The other *n-1* arguments to max correspond to the maximum revenue obtained by making an initial cut of the rod into two pieces of size *i* and *n-i*, for each *i=1,2, …, n-1*
  - then optimally cutting up those pieces further, obtaining revenues r_i and r_{n-i} from those two pieces
- Once we make the first cut, we may consider the two pieces as independent instances of the rod-cutting problem.
- The overall optimal solution incorporates optimal solutions to the two related subproblems, maximizing revenue from each of those two pieces.

Recursive Algorithm

CUT-ROD(p, n)

if $n == 0$

  return 0 ← no revenue is possible, and so CUT-ROD returns 0

$q = -\infty$

for $i = 1$ **to** n

  $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$ ← Recursive call on the second piece

return q

Recursive Algorithm

$\text{CUT-ROD}(p, n)$

if $n == 0$

  return 0 ← no revenue is possible, and so CUT-ROD returns 0

$q = -\infty$

for $i = 1$ **to** n

  $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$ ← Recursive call on the second piece

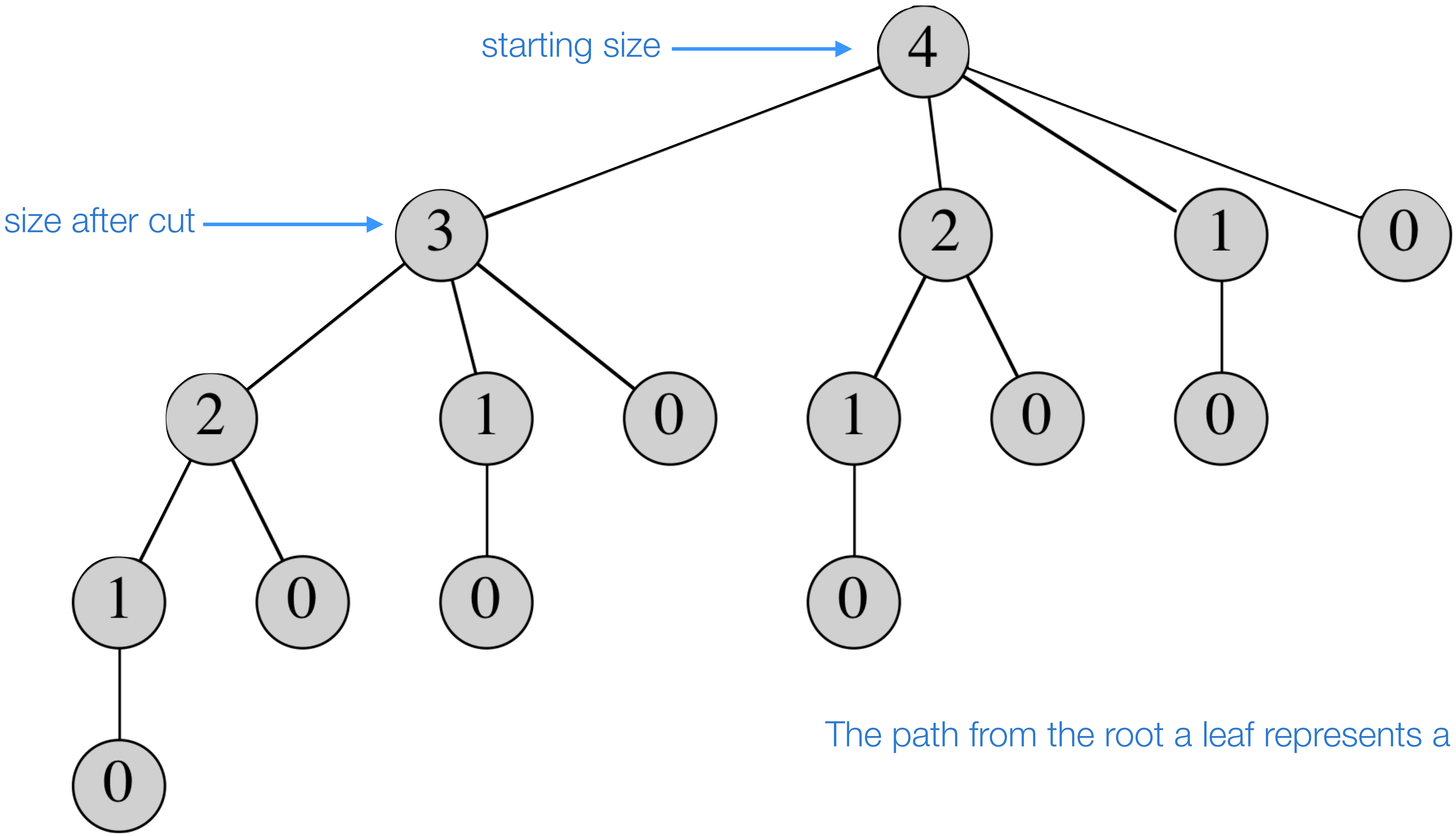
return q

CUT-ROD calls itself recursively many times with the same parameter values.

It solves the **same subproblems** repeatedly.

Rod Cutting: Recursive Calls

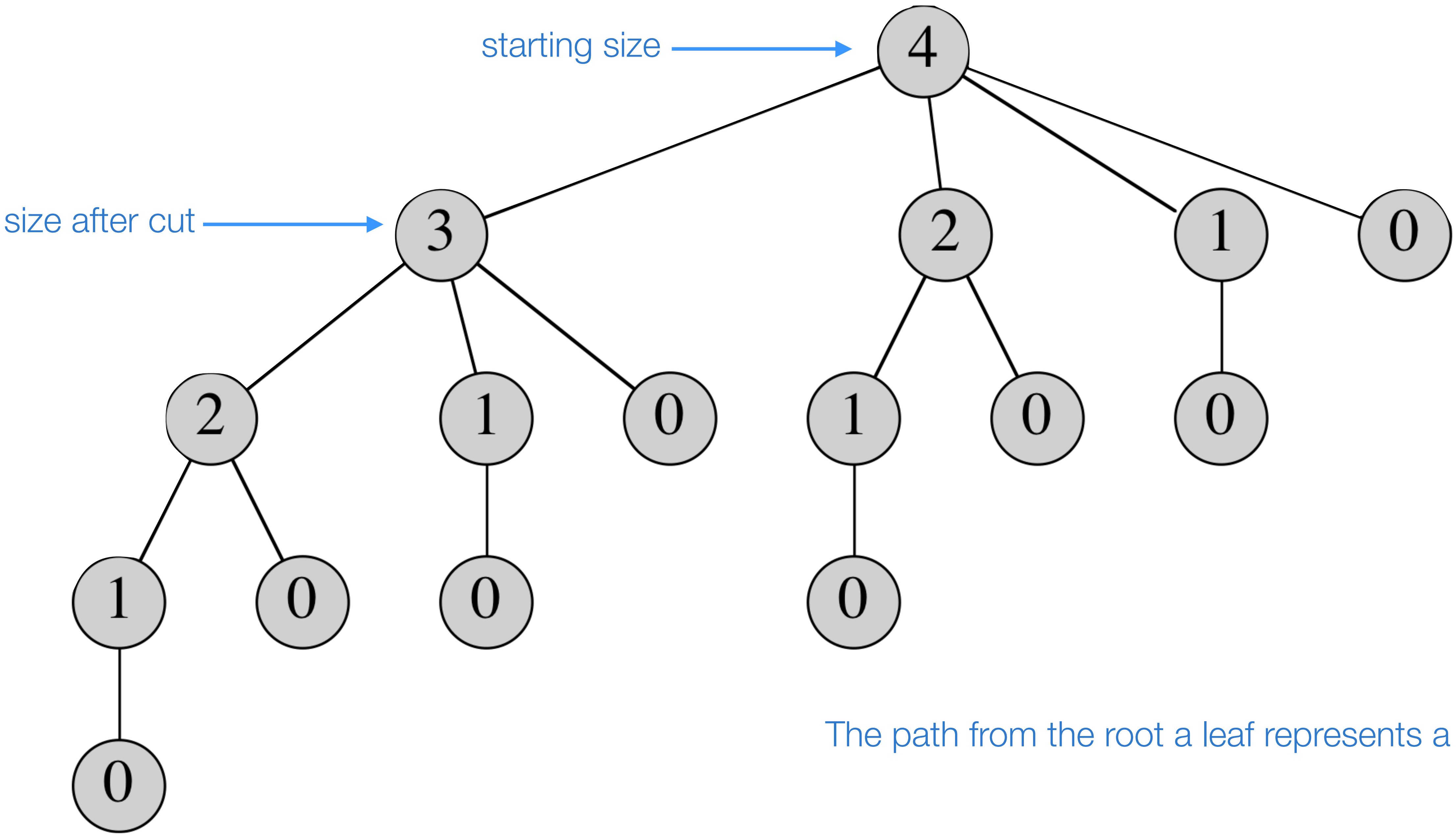
Recursive calls for CUT-ROD(p,4)



The path from the root a leaf represents a solution

Rod Cutting: Recursive Calls

Recursive calls for CUT-ROD(p,4)

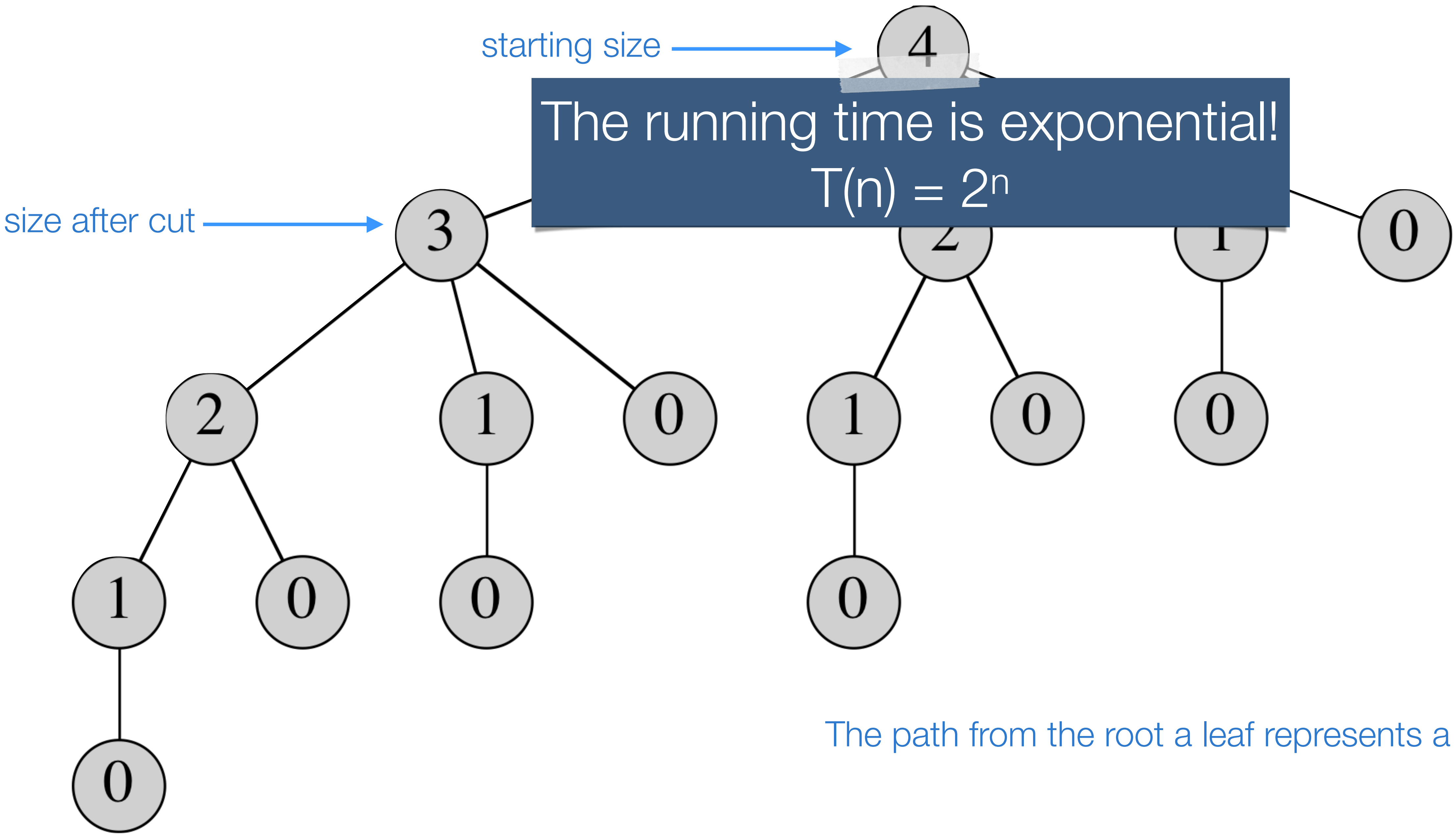


The path from the root a leaf represents a solution

The tree has 2^n nodes and 2^{n-1} leaves
There are 2^{n-1} solutions

Rod Cutting: Recursive Calls

Recursive calls for CUT-ROD(p,4)



The running time is exponential!
$T(n) = 2^n$

The path from the root a leaf represents a solution

The tree has 2^n nodes and 2^{n-1} leaves
There are 2^{n-1} solutions

Using Dynamic Programming

- The idea is to organise the algorithm in order to solve the sub-problems only once
- Store the solution in an appropriate data structure
- Retrieve the solution every time we encounter an already solved subproblem
  - **Time-Memory Trade Off**
- *Dynamic programming runs in polynomial time* when the size of the input is polynomial, the number of *distinct* sub-problems is polynomial and each sub-problem can be solved in polynomial time!

Memoization

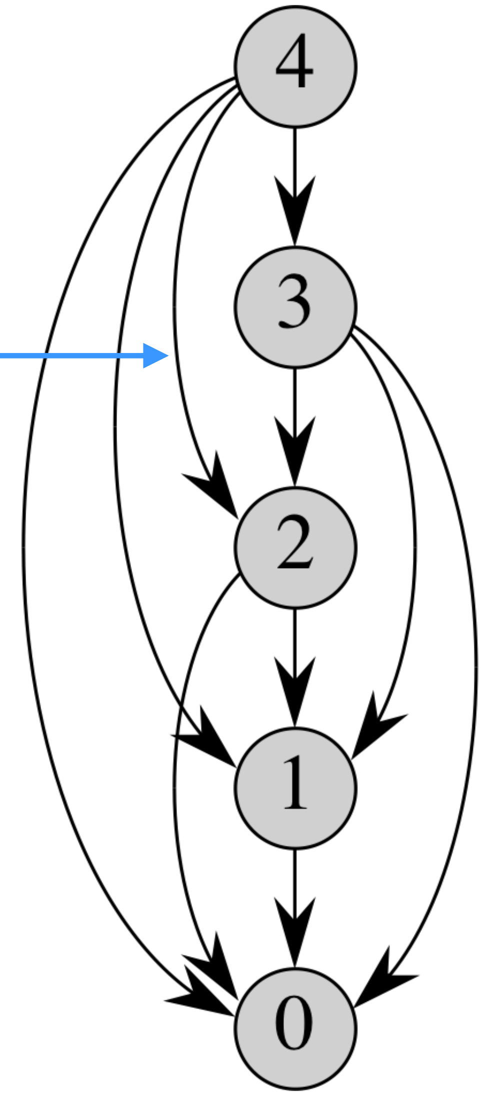
- There are two approaches:
  - Top-down memoization
    - We solve the problem in the usual way but storing the solutions of the sub-problems on the way
  - Bottom-up memoization
    - Defines the sub-problems, order them by size in increasing order, solve them and “go on” solving the rest by using the memorised solutions
- They produce algorithms of equivalent running time
  - The bottom-up strategy generally has lower constant factors
  - Memoization ≠ Memorization

Top-Down Memoization

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)

We need the solution of “3” when solving “4”

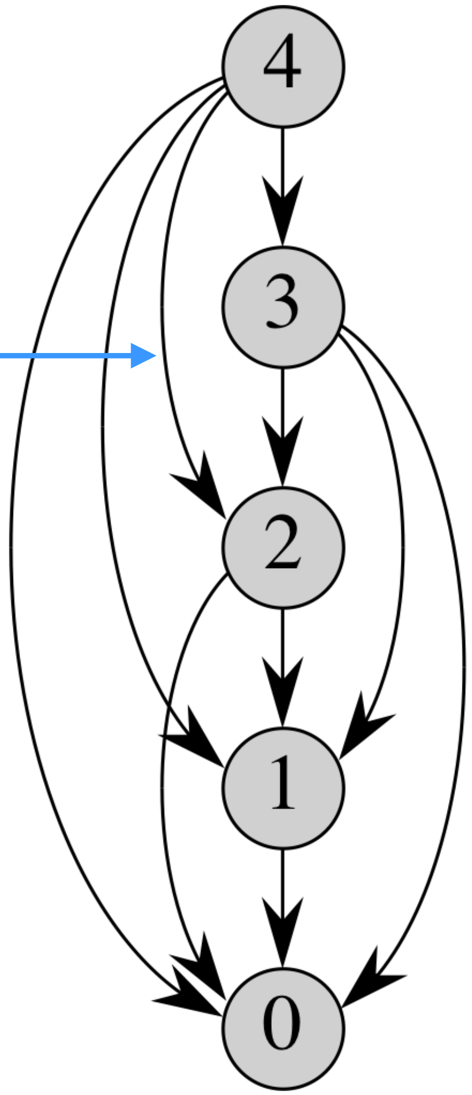


Top-Down Memoization

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)

We need the solution of “3” when solving “4”



MEMOIZED-CUT-ROD-AUX(p, n, r)

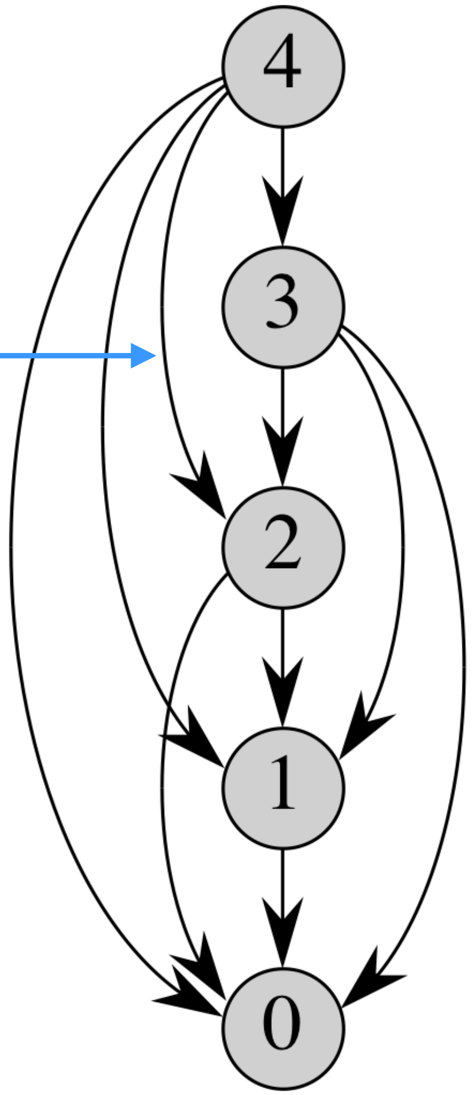
if $r[n] \geq 0$
    return $r[n]$
if $n == 0$
    $q = 0$
else $q = -\infty$
    for $i = 1$ **to** n
        $q = \max(q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n-i, r))$
$r[n] = q$
return q

Top-Down Memoization

MEMOIZED-CUT-ROD(p, n)

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

We need the solution of “3” when solving “4”



MEMOIZED-CUT-ROD-AUX(p, n, r)

```
if r[n] ≥ 0
    return r[n]          ← check for the stored solution
if n == 0
    q = 0
else q = -∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
r[n] = q                 ← store the solution for n
return q
```

Bottom-Up Memoization

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

No recursive call, but explicit call to the stored solution that we already know it exists

- It uses the natural ordering of the subproblems: a problem of size *i* is "smaller" than a subproblem of size *j* if $i < j$.
- Thus, the procedure solves subproblems of sizes $j = 0, 1, \ldots, n$, in that order.

Bottom-Up Memoization

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

No recursive call, but explicit call to the stored solution that we already know it exists

- It uses the natural ordering of the subproblems: a problem of size i is "smaller" than a subproblem of size j if $i < j$.
- Thus, the procedure solves subproblems of sizes $j = 0, 1, \ldots, n$, in that order.

The running time of top-down and bottom-up approaches is $\Theta(n^2)$

Each subproblem is solved just once, so the running time is the sum of the times needed to solve each subproblem.

Print the optimal solution

```
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
      q = -∞
      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
```

stores the optimal size *i* of the first piece to cut off when solving a subproblem of size *j*.

Print the optimal solution

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
    q = -∞
    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
```

stores the optimal size *i* of the first piece to cut off when solving a subproblem of size *j*.

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

Print the optimal solution

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
    q = -∞
    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
```

stores the optimal size *i* of the first piece to cut off when solving a subproblem of size *j*.

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]
```

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