Dynamic Programming

Origin of Name

- Coined by Richard Bellman in the 1950s
- Time when computer programming was an esoteric activity practiced by so few people as to not even merit a name
- Back then programming meant “planning,” and “dynamic programming” was conceived to optimally plan multistage processes

Origin of Name

- This technique represent multi–stage processing

Introduction

- Like divide–and–conquer method, solves problems by combining the solutions to subproblems
- "Programming" – refer to tabular method, not to writing computer code
- Divide–and–conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem

Introduction

- In contrast, dynamic programming applies when the subproblems overlap—that is, when subproblems share subsubproblems
- a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems.

Introduction

- A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem
- We typically apply dynamic programming to optimization problems
- Such problems can have many possible solutions

Introduction

- 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

Steps of Developing a Dynamic–Programming Algorithm

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

Steps of Developing a Dynamic–Programming Algorithm

- Steps 1 – 3 form the basis of a dynamic–programming solution to a problem
- If we need only 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

Nth term in a Fibanocci Series

```
                           fib(5)
                      /                \
               fib(4)                 fib(3)
              /        \               /       \
         fib(3)       fib(2)         fib(2)   fib(1)
         /    \       /    \         /      \
  fib(2)     fib(1)  fib(1) fib(0) fib(1) fib(0)
  /      \
fib(1) fib(0)
```

Rod cutting

- Serling Enterprises buys long steel rods and cuts them into shorter rods, which it then sells
- Each cut is free
- Management of Serling Enterprises wants to know the best way to cut up the rods
- We know, for $i = 1, 2, \ldots$, the price p_i in dollars that Serling Enterprises charges for a rod of length i inches

Rod cutting

- Rod lengths are always an integral number of inches

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

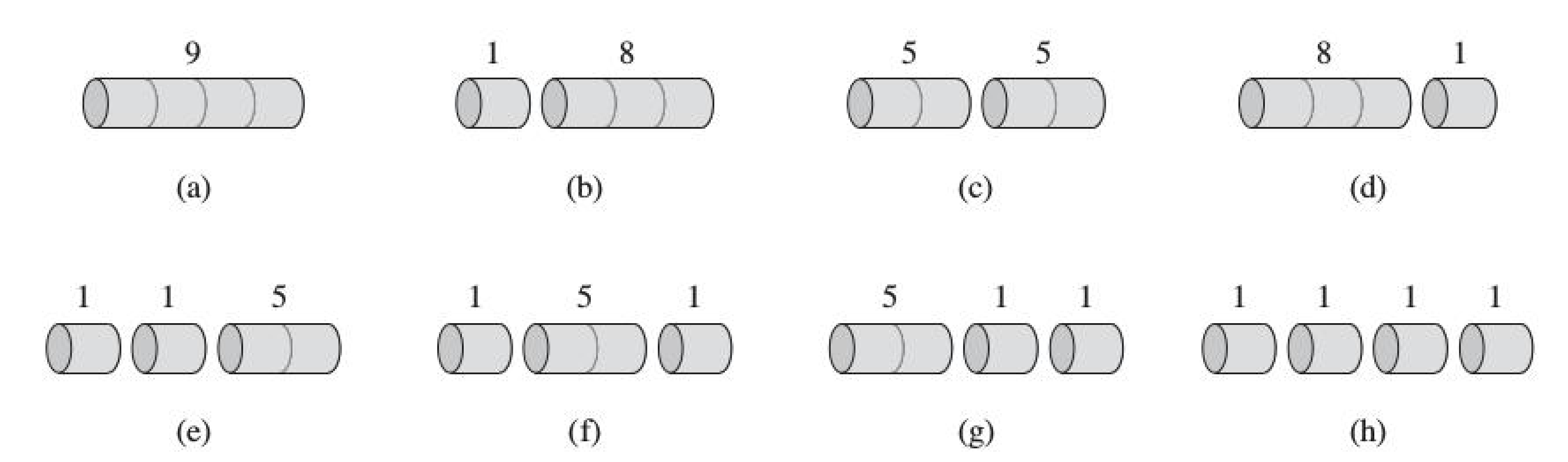
- A sample price table for rods. Each rod of length i inches earns the company p_i dollars of revenue.

Rod cutting Problem Statement

- Input
  - rod of length 'n' inches
  - a table of prices p_i for $i = 1,2,\ldots,n$,
- Expected Output
  - 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.

Rod cutting

- Consider the case when n = 4
- all the ways to cut up a rod of 4 inches in length, including the way with no cuts at all



(a) (b) (c) (d)
(e) (f) (g) (h)

- Cutting a 4–inch rod into two 2–inch pieces produces revenue $p_2 + p_2 = 5 + 5 = 10$, which is optimal.

Rod cutting

- We can cut up a rod of length n in 2^{n-1} different ways, since we have an independent option of cutting, or not cutting, at distance 'i' inches from the left end
- If we required the pieces to be cut in order of nondecreasing size, there would be fewer ways to consider
- For n = 4, we would consider only 5 such ways: parts (a), (b), (c), (e), and (h)

Rod cutting

- The number of ways is called the partition function
- it is approximately equal to $e^{\pi\sqrt{2n/3}}/4n\sqrt{3}$.
- This quantity is less than 2^{n-1}
- but still much greater than any polynomial in n

Rod cutting

- We denote a decomposition into pieces using ordinary additive notation, so that $7 = 2 + 2 + 3$ indicates that a rod of length 7 is cut into three pieces—two of length 2 and one of length 3
- If an optimal solution cuts the rod into k pieces, for some $1 \leq k \leq n$, then an optimal decomposition
- $n = i_1 + i_2 + \ldots + i_k$

Rod cutting

- For our sample problem, we can determine the optimal revenue figures r_i , for $i = 1,2,\ldots,10$, by inspection, with the corresponding optimal decompositions
- $r_1 = 1$ from solution $1 = 1$ (no cu

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

- $r_2 = 5$ from solution $2 = 2$ (no cuts)
- $r_3 = 8$ from solution $3 = 3$ (no cuts)
- $r_4 = 10$ from solution $4 = 2 + 2$

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

- $r_5 = 13$ from solution $5 = 2 + 3$
- $r_6 = 17$ from solution $6 = 6$ (no cuts)
- $r_7 = 18$ from solution $7 = 1 + 6$
- $r_8 = 22$ from solution $8 = 2 + 6$
- $r_9 = 25$ from solution $9 = 3 + 6$
- $r_{10} = 30$ from solution $10 = 10$ (no cuts)

Rod cutting

- More generally, we can frame the values r_n for $n \geqslant 1$ in terms of optimal revenues from shorter rods:
- $r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$
- First argument, p_n, corresponds to making no cuts at all and selling the rod of length n as it is
- 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 $i = 1, 2, \ldots, n - 1$

Rod cutting

- and then optimally cutting up those pieces further, obtaining revenues r_i and r_{n-i} from those two pieces
- Since we don't know ahead of time which value of i optimizes revenue, we have to consider all possible values for i and pick the one that maximizes revenue
- We also have the option of picking no 'i' at all if we can obtain more revenue by selling the rod uncut

Rod cutting

- to solve the original problem of size n, we solve smaller problems of same type, but of smaller sizes
- Once we make the first cut, we may consider the two pieces as independent instances of the rod–cutting problem.
- Overall optimal solution incorporates optimal solutions to the two related subproblems, maximizing revenue from each of those two pieces

Rod cutting

- We say that the rod–cutting problem exhibits optimal substructure: optimal solutions to a problem incorporate
- Optimal solutions to related subproblems, which we may solve independently
- Slightly simpler, view a decomposition as consisting of a first piece of length i
- Cut off the left–hand end, and then a right–hand remainder of length n–i

Rod cutting

- Only remainder, and not the first piece, may be further divided
- Every decomposition of a length–n rod is viewed in this way:
- as a first piece followed by some decomposition of the remainder
- Solution with no cuts at all
    - First piece has size $i = n$ and revenue p_n
    - Remainder has size 0 with corresponding revenue $r_0 = 0$

Rod cutting

- thus obtain the following simpler version of equation (15.1):

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

- In this formulation, an optimal solution embodies the solution to only one related subproblem—the remainder—rather than two

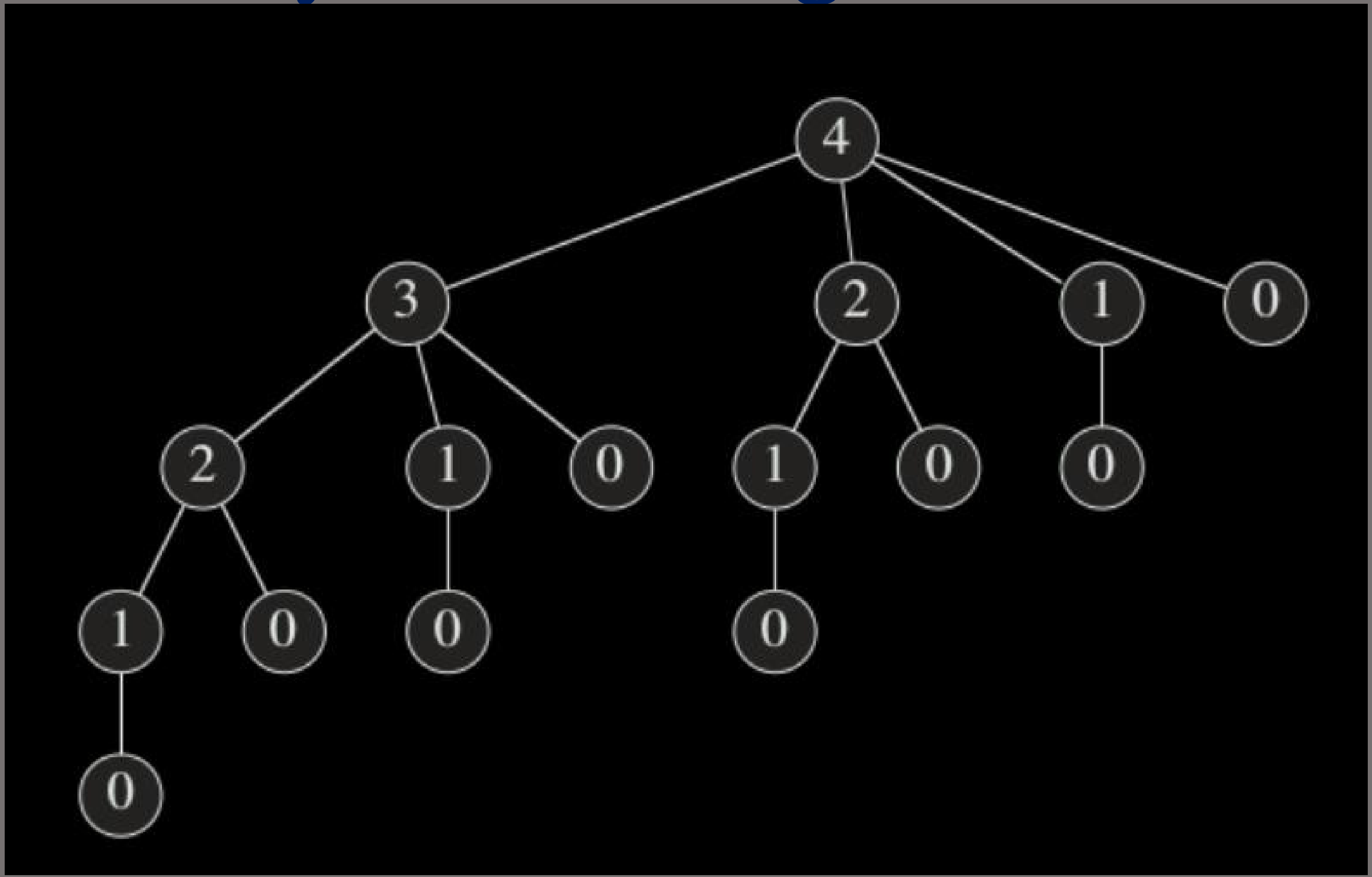
Recursive implementation

```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

Rod cutting

- Once the input size becomes moderately large, your program would take a long time to run
- For n = 40, you would find that your program takes at least several minutes, and most likely more than an hour
- Each time you increase n by 1, your program's running time would approximately double

Why Rod Cutting is inefficient?



- Recursion tree showing recursive calls for $n = 4$
- A path from root to a leaf corresponds to one of the 2^{n-1} ways of cutting up a rod of length n
- In general, this recursion tree has 2^n nodes and 2^{n-1} leaves

Analysis of Running Time of Rod–Cut

- T(n) denote total number of calls made to CUT–ROD when called with its second parameter equal to n
- Equals number of nodes in a subtree whose root is labeled 'n' in the recursion tree
- Count includes initial call at its root
- Thus, $T(0) = 1$

Analysis of Running Time of Rod–Cut

- The initial 1 is for the call at the root, and the term T(j) counts the number of calls (including recursive calls) due to the call CUT–ROD(p, n–i), where $j = n - i$
- $T(n) = 2^n$
- so the running time of CUT–ROD is exponential in n
- CUT–ROD explicitly considers all the 2^{n-1} possible ways of cutting up a rod of length n

Using Dynamic Programming for Optimal Rod Cutting

- Recursive solution is inefficient because it solves the same subproblems repeatedly
- We solve each subproblem only once saving its solution
- Dynamic programming thus uses additional memory to save computation time; it serves an example of a time–memory trade–off

Using Dynamic Programming for Optimal Rod Cutting

- Savings may be dramatic: an exponential–time solution may be transformed into a polynomial–time solution
- two ways to implement a dynamic–programming approach
  - top–down with memoization
  - bottom–up method

Top–down with memoization for Optimal Rod Cutting

- We write procedure recursively in a natural manner, but modified to save the result of each subproblem
- Procedure first checks to see whether it has previously solved
- If so, returns saved value and computes 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 for Optimal Rod Cutting

- Depends on some natural notion of the “size” of a subproblem, such that solving any particular subproblem depends only on solving “smaller” subproblems
- 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

Top Down vs Bottom Up

- 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
- bottom–up approach often has much better constant factors, since it has less overhead for procedure call

Top Down Algorithm

```
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)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
        return r[n]
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
return q
```

Top Down Algorithm

```
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]
```

Top Down Algorithm