CSE331 Introduction to Algorithms
Lecture 18
The Rod-Cutting Problem

Antoine Vigneron
antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

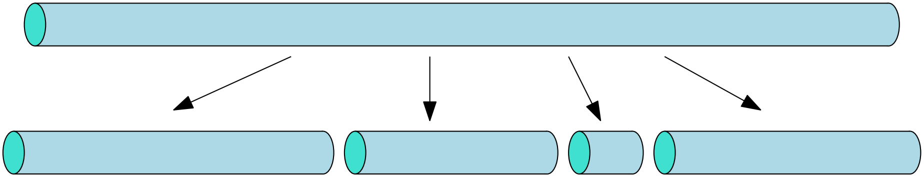
July 23, 2021

Introduction

- This is the second lecture on dynamic programming.
- We use DP to solve the *rod-cutting problem*.
- **Reference**: Section 15.1 of the textbook (p. 360–369) Introduction to Algorithms by Cormen, Leiserson, Rivest and Stein.

Problem Statement

- Serling Enterprises buys long steel rods and cuts them into shorter rods, which it then sells.



Problem Statement

- The price of a rod depends on its length, which is supposed to be an integer.

Example

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 goal is to maximize the total price of the rods obtained from a rod of length n, given the values p_i for $i = 1, 2, \dots n$.

Example

$n = 8$, using prices p_i from the table above.

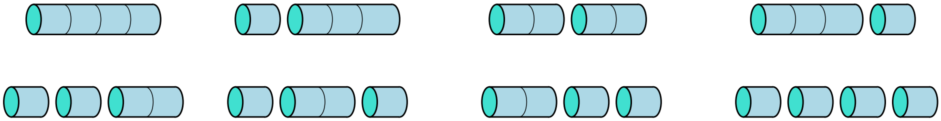
- If we cut into two rods of length 4, the total price is 9+9=18.
- If we cut into two rods of length 3 and one of length 2, the total price is 8+8+5=21.
- So the second solution is better.

Difficulty

- How many different ways are there of cutting a rod of length n?
  - 2^{n-1}, because we can cut at $n-1$ different locations.

Example

If $n = 4$, there are $2^3 = 8$ different ways:



- So brute-force search runs in $\Omega(2^n)$ time.
- This is *exponential* time. We need a faster algorithm.

Structure of the Solution

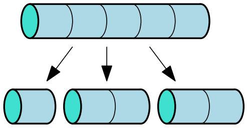
- Suppose that the prices p_i are fixed, for $i = 1, \dots, n$. Let r_i denote the value of the optimal solution of the subproblem where we cut a rod of size $i \leqslant n$.

Example

Using the price table below,

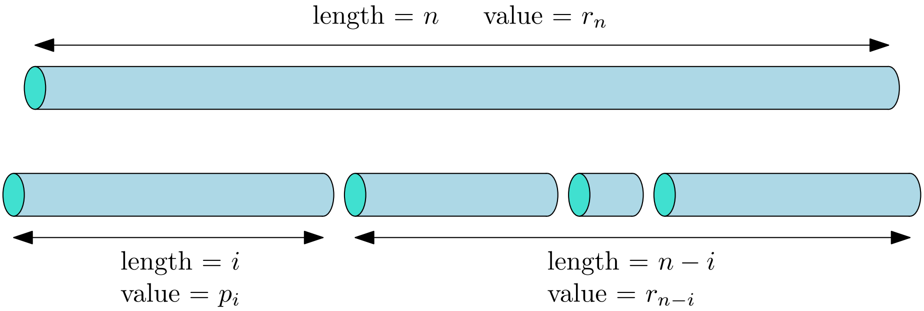
length i	1	2	3	4	5	6	7	8	9	10
price p_i	2	5	6	7	8	9	11	11	13	15

$r_5 = 2 + 5 + 5 = 12$, where we cut into 3 pieces of lengths 1,2 and 2.



$$r_5 = 2 + 5 + 5 = 12$$

Structure of the Solution



Observation

An optimal solution consists of a piece of length i, for some i, followed by a rod of length $n - i$ cut optimally.

Recurrence Relation

- So we either have $r_n = p_i + r_{n-i}$ for some i, or $r_n = p_n$.
- Then if we write $r_0 = 0$, it means:

$$r_n = \max_{1 \leqslant i \leqslant n} \left(p_i + r_{n-i}\right). \tag{1}$$

Structure of the Solution

Proof of Equation (1).

We assume that the rod of length n is cut optimally.

- If there is at least one cut, then the first cut occurs after a length $1 \leqslant i < n$. Then the remaining part should be cut optimally, so its value is r_{n-i}, and the total value of the pieces is $p_i + r_{n-i}$.
- Otherwise, it means that the rod is not cut, and thus the optimal value is $r_n = p_n + r_0 = p_n$.

It follows that the value of the optimal solution satisfies

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

□

Recursive Top-Down Implementation

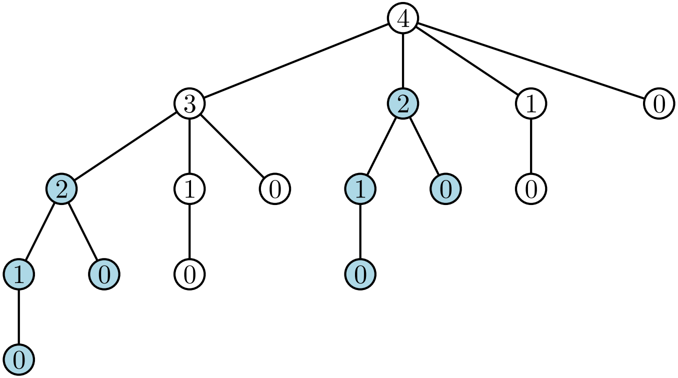
- We can obtain r_n from Equation (1) using the following recursive procedure:

Naive recursive implementation

```
procedure CUTROD($p, n$)
    if $n = 0$ then
        return 0                                   ▷ base case
    $q \leftarrow -\infty$
    for $i \leftarrow 1, n$ do
        $q \leftarrow \max(q, p[i] + \textsc{CutRod}(p, n - i))$
    return $q$
```

- This algorithm turns out to be very slow. Why?
- Similarly to the naive algorithm from the previous lecture, it repeats the same calculations over and over again. (See recursion tree on next slide.)

Recursive Top-Down Implementation



- Entire subtrees are recomputed.
- The running time is $\Omega(2^n)$. Why?
    - It tries the 2^{n-1} possible ways of partitioning the rod.

Top-Down Memoization

- In order to speed it up, we will use dynamic programming.
- We first use a particular form of dynamic programming called *top-down memoization*.

Top-down memoization

The problem is still solved recursively, but the results for all subproblems are saved in a table.
When considering a subproblem, the procedure first checks whether the result can be found in the table, and if so, it return its value immediately.
Otherwise, its value is computed recursively and it is saved in the table.
We say that the procedure has been *memoized*.

Top-Down Memoization

Top-down memoization of CutRod

```
1: procedure MemoizedCutRod($p, n$)
2:     $r[0 \dots n] \leftarrow$ new array filled with $-\infty$
3:     return Auxiliary($p, n, r$)
4: procedure Auxiliary($p, n, r$)
5:     if $r[n] \neq -\infty$ then                  ▷ checks if the result is in the table
6:         return $r[n]$                             ▷ if so, return it immediately
7:     if $n = 0$ then                                ▷ base case
8:         $q \leftarrow 0$
9:     else                                           ▷ recursive computation of $r[n]$
10:        $q \leftarrow -\infty$
11:        for $i = 1, n$ do
12:            $q \leftarrow \max(q, p[i] + $ Auxiliary$(p, n-i, r))$
13:    $r[n] \leftarrow q$
14:    return $q$
```

Analysis

Theorem

The memoized version of $\textsc{CutRod}$ *runs in* $\Theta(n^2)$ *time.*

Proof.

- Line 2–3: $O(1)$
- Line 5–6: $O(k)$ where k is the number of calls to $\textsc{Auxiliary}$
- Line 7–14: $O(\ell n)$ where ℓ is the number of times we reach Line 7
- $\ell = n + 1$ because the value in each cell of the array $r[0 \dots n]$ is computed only once
- $k = O(n^2)$ because for each time we reach Line 7, we execute Line 12 $O(n)$ times

This shows that the running time is $O(n^2)$. Conversely, the loop 11–12 is executed once for each value of n, as the algorithm computes all values of $r[i]$, so the running time is at least $\Omega(\sum_{i=1}^{n} i) = \Omega(n^2)$. □

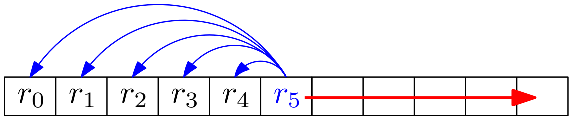
Analysis

- Similarly as for the computation of binomial coefficients, dynamic programming brings the running time down from exponential to polynomial.

Bottom-Up Method

- Another way of doing dynamic programming is the *bottom-up method*.
- It is the method we used for computing binomial coefficients.
- The idea is to consider the subproblems in an order that guarantees that, at any step, all the intermediate results we need have already been computed.
- In the previous lecture, we achieved it by computing $\binom{n}{k}$ by increasing value of n, and decreasing value of k.
- Often it is achieved using a notion of *size* of subproblems, and solving smaller subproblems first.

Bottom-Up Method



Bottom-up version of CutRod

```
1: procedure BottomUpCutRod($(p, n)$)
2:     $r[0 \dots n] \leftarrow$ new array
3:     $r[0] \leftarrow 0$
4:     for $j = 1, n$ do
5:         $q \leftarrow -\infty$
6:         for $i \leftarrow 1, j$ do
7:             $q \leftarrow \max(q, p[i] + r[j-i])$
8:         $r[j] \leftarrow q$
9:     return $r[n]$
```

Bottom-Up Method

- Correctness follows from Equation 1, and the fact that, at line 7, $r[j-i]$ has already been computed because $j-i<j$.
- It runs in time $\Theta(n^2)$ because line 7 is iterated $1+2+\cdots+n=\Theta(n^2)$ times.
- So the bottom-up method is asymptotically as fast as the memoized version of CUTROD, and is simpler.
- It will often be the case, so the bottom-up method is often employed.
- On the other hand, the bottom-up method requires to find a proper ordering of the subproblems, which is not needed for the memoized top-down approach.

Reconstructing a Solution

- The three algorithms we gave return the total *value* of the optimal solution, but it does not return the solution itself.
- We would like our algorithm to return the list of the *sizes* of the pieces in an optimal solution.
- We show on next slide a simple modification of the procedure that prints a solution.
- It is usually the case with dynamic programming that we can reconstruct an optimal solution by slightly modifying the algorithm

Reconstructing a Solution

Extended version of BottomUpCutRod

```
procedure ExtendedBottomUpCutRod($(p, n)$)
    $r[0 \dots n] \leftarrow$ new array
    $s[0 \dots n] \leftarrow$ new array
    $r[0] \leftarrow 0$
    for $j \leftarrow 1, n$ do
        $q \leftarrow -\infty$
        for $i \leftarrow 1, j$ do
            if $q < p[i] + r[j-i]$ then
                $q \leftarrow p[i] + r[j-i]$
                $s[j] \leftarrow i$
        $r[j] \leftarrow q$
    return $r$ and $s$
```

Reconstructing a Solution

- $s[j]$ records the size of the first piece in an optimal solution to the subproblem of size j.
- So the following procedure prints out an optimal solution:

Printing the solution

```
procedure PrintCutRodSolution($p, n$)
    $(r, s) \leftarrow$ ExtendedBottomUpCutRod($p, n$)
    while $n > 0$ do
        print s[n]
        $n \leftarrow n - s[n]$
```

Reconstructing a Solution

- On the example from Slide 5, ExtendedBottomUpCutRod returns:

length i	0	1	2	3	4	5	6	7	8	9	10
price p_i		1	5	8	9	10	17	17	20	24	30
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

- PrintCutRodSolution$(p, 10)$ prints 10.
- PrintCutRodSolution$(p, 9)$ prints 3 and 6.

Conclusion

Dynamic programming in 4 steps

1. Study the structure of the optimal solution (Slide 8).
2. Deduce from it a recurrence relation for the *value* of an optimal solution (Slide 9).
3. Compute the *value* of an optimal solution by remembering the value of each subproblem that was previously solved. Two approaches:
   - Memoization (Slide 13), or
   - Bottom-up method (Slide 17).
4. If you need an optimal solution (not just its value), modify your code so that it reconstructs an optimal solution (Slide 20).

Conclusion

- Steps 1 and 2 are the non-trivial steps and require problem solving skills. Steps 3 and 4 are purely technical.
- Step 3:
  - Memoization works more often because you do not need to determine a proper ordering of the problems.
  - The bottom-up approach often gives simpler code, and is likely to be faster.
- Step 4: Usually only requires minor modifications of the code and does not affect the asymptotic running time.
- Dynamic programming is a general technique that gives polynomial-time algorithms for a variety of optimization problems.
- But it does not always work (in particular, for **NP**-hard problems).