CSCI3160 Design and Analysis of Algorithms (2025 Fall)

Dynamic Programming 2: Rod Cutting

Instructor: Xiao Liang[1]

Department of Computer Science and Engineering
The Chinese University of Hong Kong

[1]These slides are primarily based on materials prepared by Prof. Yufei Tao (please refer to Prof. Tao's version from 2024 Fall for the original content). Some modifications have been made to better align with this year's teaching progress, incorporating student feedback, in-class interactions, and my own teaching style and research perspective.

The Rod Cutting Problem

Input:

- a rod of length n
- an array P of length n where
  $P[i]$ is the price for a rod of length i, for each $i \in [1, n]$

Goal: Cut the rod into segments of integer lengths to maximize the revenue.

Example Consider a rod of length $n = 4$. Its price array P is given below

length i	1	2	3	4
price $P[i]$	1	5	8	9

All possible ways to cut a rod of length 4:

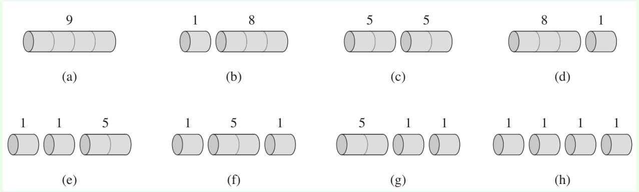

Figure: By courtesy of the textbook [CLRS]

The optimal cutting method: (c), which has a revenue of 10

Subproblems! Subproblems! Subproblems!

The core trick of dynamic programming is to break the problem into subproblems that can be solved in a nice order that later subproblems can reuse the solutions of earlier subproblems.

With that in mind, when solving a problem by dynamic programming, the most crucial question is:

What are the proper subproblems?

Unfortunately, this step requires both experience and creativity. There is no universal method that guarantees you will always find the appropriate subproblems.

For our problem in hand: what are the proper subproblems for rod cutting?

A closer look at the earlier example

Its hardness seems to lie in that

- There are exponentially many (i.e., $O(2^n)$) possible ways to cut a rod, each with a different cost.
- So, it is hopeless to calculate the revenue for each of them to determine the best.

However, there are some crucial **structural** properties of this example:

- Despite of the exponentially many possible ways, the revenue of each way is calculated by add certain items of the price array P; and the size of the array P is linear in n!
- In other words, the "exponentially many ways" is kind of misleading, in the sense that they are from different combinations of linearly many elements (i.e., P's items).
- Also, certain ways of cutting are "perfectly symmetric," and thus repetitive (e.g., (b)-(d) and (e)-(f)-(g)).

Ideas Exploiting the Structural Properties

The previous discussion of the structural properties of this problem inspires the following idea:

- We design subproblems as rod-cutting of smaller size
- It is possible that once we calculate the solution for a length i rod, we can store it so that it will contribute when we cut a length j rod with $j > i$

It turns out that this idea lead to a successful dynamic programming algorithm (subsequent slides).

Caveat: this step is indeed somewhat magical. This is the step that requires experience and creativity.

Intuition on how to exploit our observations so far I

No matter how to cut, we have to **make the first cut**!

This is a hard decision: different first-cut position leads to different revenue. We don't know which one is optimal.

Attempt: just exhaust all the possibilities! Namely, we

- try every possible ways for the first cut (Think: convince yourself that there are n different choices of first-cut position.
- calculate the revenue for each choice of the first cut.
- take the maximum of them.

Intuition on how to exploit our observations so far II

Issues:

- Once the first cut is made, how to calculate the optimal revenue for this choice of first cut? It seems to require us to cut the remaining rod in an optimal way, which we don't know how to do yet.
- If we calculate the revenue for all possible ways of the first cut, would that require exponential amount of work, especially given the issue mentioned in the first bullet?

No Problem!

We've observed that the small segments in Figure 1 are "symmetric." So, all subproblems **of the same size** are actually the same subproblem (i.e., they don't distinguishe with each other.) This will save us from falling into the trap of "exponential time."

Formalizing the Recursive Relation

Define $opt(n)$ as the optimal revenue from cutting up a rod of length n.
Clearly, $opt(0) = 0$.

Consider now $n \geq 1$. Let us practice the above intuition:

Conditioned on that we choose to generate a length-i segment as our first piece, the optimal revenue for (any ways of cutting that start from) such a first cut must be:

$$P[i] + opt(n - i).$$

Think: convince yourself this is true!

Next, recall that there are exactly n choices for i (i.e., n positions to make the first cut). Therefore:

$$opt(n) \quad = \quad \max_{i=1}^{n}(P[i] + opt(n - i))$$

How to Implement the Recursion Efficiently?

Given

$$opt(n) = \max_{i=1}^{n}(P[i] + opt(n-i))$$

we can compute $opt(n)$ in $O(n^2)$ time using dynamic programming (this is the problem solved in the last lecture).

Wait! We need to **generate** a cutting method to achieve revenue $opt(n)$.

This can be done by recording which subproblem yields $opt(n)$.

See the next slide.

Piggyback I

Given

$$opt(n) \quad = \quad \max_{i=1}^{n}(P[i] + opt(n-i))$$

define *bestSub*$(n) = k$ if:

- When cutting a length-n rod, we obtain the optimal revenue if we make the first cut at position k (i.e., first segment having length k).

Example

length i	1	2	3	4
price $P[i]$	1	5	8	9
$opt(i)$	1	5	8	10
$bestSub(i)$	1	2	3	2

Piggyback II

If we have computed $bestSub(i)$ for every $i \in [1, n]$, then the best method for cutting up a rod of length n can be obtained in $O(n)$ time. (Think: how?)

You can use another array to store the values of *bestSub* in the pseudo code shown in the last lecture. (Think: how?)

For each $i \in [1, n]$, computing $bestSub(i)$ is no more expensive than computing $opt(i)$.

We conclude that the rod cutting problem can be solved in $O(n^2)$ time.

The method of using the *bestSub* function to generate an optimal cutting is known as the **piggyback** technique.

Closing Remarks

Recall our train of thoughts when tackling the rod cutting prodlem:

- Distill the problem in a clean mathematically form.
- Take initial experiments/attempts with small-size toy examples (e.g., $n = 4$), trying to understand the problem better.
- Distill some structural properties of the problem from our toy examples.
  - We could try to see if the properties can be generalize. They may not always. This is a trial-error step.
- These structural properties allow us to design a dynamic programming algorithm:
  - They inspire a good definition for subproblems.
  - Subproblems can be solved in a **nice order**, leading to the solution of the original rod cutting problem.

This pattern is typical in the design of dynamic programming algorithms. In fact, it is characteristic of the entire art of algorithm design!

Our philosophy on the design and exposition of algorithms is nicely illustrated by the following analogy with an aspect of Michelangelo's art. A major part of his effort involved looking for interesting pieces of stone in the quarry and staring at them for long hours to determine the form they naturally wanted to take. The chisel work exposed, in a minimalistic manner, this form. By analogy, we would like to start with a clean, simply stated problem (perhaps a simplified version of the problem we actually want to solve in practice). Most of the algorithm design effort actually goes into understanding the algorithmically relevant combinatorial structure of the problem. The algorithm exploits this structure in a minimalistic manner. . . .

— cited from the preface of the book *Aproximation Algorithms* by Vijay V. Vazirani

Figure: Michelangelo's Moses (ca. 1513–15), for the Tomb of Julius II.