Pipe Cutting (Beecrowd 1798) Lab Report

Algorithms and Computation

Name: Robert Rice
Date: December 4, 2025
Student ID: 657588340
Language: Java

1. Introduction

This report is for the Beecrowd problem 1798, "Pipe Cutting".

The Problem Statement is about a company that makes long pipes and then cuts those into smaller pieces to sell. Each smaller pipe has the following:

- a length C_i (how much of the large pipe it uses), and
- a value V_i (how much money it brings in).

We start with a pipe of length T. We can then cut as many pieces of each type as we want, as long as the total length is at most T. We are also allowed to leave some remaining pipe that we do not use. Our goal is to get the largest total value.

This is a classic dynamic programming problem. It is basically the same as the **unbounded knapsack** or **rod cutting** problem, where we can use each item many times.

2. Problem and Basic Idea

We are given:

- T: the length of the original tube.
- N: the number of different types of pipes.
- For each i from 1 to N:
  - C_i: length of that pipe type.
  - V_i: value of that pipe type.

We want to choose how many pieces of each type to cut so that:

$$\sum_{i=1}^{N} x_i C_i \le T$$

and the total value;

$$\sum_{i=1}^{N} x_i V_i$$

is as large as possible. Here x_i can be $0, 1, 2, \ldots$ (we can use each type many times).

A simple but bad idea would be to write a recursive function that, for each remaining length, tries every possible piece and calls itself again. So I used dynamic programming with a **1-D array**. This lets me reuse the answers to smaller subproblems and gives a fast solution.

3. Dynamic Programming Solution

3.1 DP Definition

I use a one-dimensional array `dp` where:

$$dp[\ell] = \text{the maximum value we can get using at most length } \ell,$$

for $\ell = 0, 1, 2, \ldots, T$.

- **Base case:** $dp[0] = 0$ (no length, no value).
- Other entries start at 0 (we could cut nothing).

In the end, the answer for a test case is just $dp[T]$.

3.2 Transition (Unbounded Knapsack)

For each pipe type i with length C_i and value V_i, I update the array as follows:

$$\text{for } \ell = C_i \text{ to } T: \quad dp[\ell] = \max(dp[\ell],\, dp[\ell - C_i] + V_i).$$

The idea is as follows.

- At length ℓ, we can either:
  - **does not** use this type at this step, so $dp[\ell]$ stays the same, or
  - use this type once more, and then we add V_i to the best value for the remaining length $\ell - C_i$, which is $dp[\ell - C_i]$.
- We take the maximum of these two choices.

The key detail is the direction of the loop over ℓ.

3.3 Complexity

We have N pipe types and we consider all lengths from 0 to T.

- Time: about $N \times T$ steps, which is fine for $N \leq 1000$ and $T \leq 2000$.
- Space: the `dp` array has size $T + 1$, so the memory is $O(T)$.

4. What Is New Compared to the Previous Lab

In the earlier lab, I did not fully finish the implementation, but the plan was to use a dynamic programming table with two dimensions. The state looked like $dp[i][\ell]$:

This idea works in theory, but it uses $O(N \cdot T)$ memory. It also feels more complicated, because I have to keep track of both the item index i and the length ℓ. Although I did not complete that lab, I still learned how this 2-D table is supposed to work.

5. My Understanding of the 1-D Array Implementation

5.1 Why 1-D DP Is Enough

The 1-D array works because each value $dp[\ell]$ depends only on values with smaller lengths:

$$dp[\ell] \text{ depends on } dp[\ell - C_i].$$

When I build dp from left to right (from 0 to T), I know that $dp[\ell - C_i]$ is already correct when updating $dp[\ell]$. So I do not need a separate dimension for the index of the items i.

I can think of it as this:

- First, I know the best value for length 0.
- Using that, I can get the best value for length 1, then 2, etc.
- Each time I add a type of piece, I "spread" its effect throughout the array.

5.2 Pros and Cons

Pros of 1-D DP:

- Uses much less memory than a 2-D table.
- The code is shorter and easier to read once I understand it.
- Often faster in practice, because it works on one array over and over.

Things to be careful about:

- If I use the wrong loop direction, I get the wrong answer and might not notice right away.

- The 2-D table is sometimes easier to imagine the first time I learn DP, because it looks like a grid of subproblems.

6. Testing and Final Thoughts

To test my program, I tried both small hand-made tests and larger tests that were closer to what Beecrowd uses.

- **Sample of the problem statement.**
  First, I ran the sample input from the Beecrowd problem description (for example, $N = 3$, $T = 10$ with pieces $(6, 3)$, $(2, 1)$, $(5, 2)$). My program printed the expected answer, which was 5.
- **Small custom tests.**
  I created very small test cases where I could do the math by hand, such as:
  - One type of pipe that fits exactly T.
  - One type of pipe that does not divide T evenly, to ensure that leftovers are allowed.
  - Some cases with two or three types where I could manually list all combinations and check the maximum value.

  These helped me to check that the DP logic and the loop directions were correct.
- **Edge cases.**
  I also thought about edge cases:
  - When $T = 0$ (pipe length zero), the best value should be 0.
  - When $N = 0$ (no pipe types), the best value should also be 0 regardless of what T is.

  My program handled these cases as expected.
- **Using an input text file like Beecrowd.**
  For larger tests, I copied a full Beecrowd-style test case into a file called `input.txt`. This file contained N, T, and all the pairs (C_i, V_i), one per line, as the online judge.

  Then I ran my program from the command line and fed the file into it, so the program read from `standard input` just like it would on Beecrowd. For example:
  - In a normal Command Prompt, I could use:

    ```
    java -cp .  Main < input.txt
    ```
  - In PowerShell, I could use:

    ```
    Get-Content input.txt | java -cp .  Main
    ```

This allowed me to test real input blocks and check that the output matched what I expected. It also helped me practice running the program almost the same way the judge does.

Everything behaved as expected, and my solution passed the Beecrowd tests.

If I had to summarize what I want to remember later, it would be this:

- This pipe cutting problem is really an unbounded knapsack problem.
- Dynamic programming turns a slow recursive idea into a fast algorithm by storing answers to smaller subproblems.
- A 1-D DP array is often enough, but I have to be very careful about the direction of the loops.
- For an unbounded knapsack in 1-D: loop the capacity from small to large.
- Using an input file and running the program from the command line is a good way to test code in the same style as an online judge.