1547. Minimum Cost to Cut a Stick

Hard | Array | Dynamic Programming | Sorting

Problem Description

The problem presents the scenario where you have a wooden stick of length n units, marked at each unit along its length. For example, a stick that is 6 units long would be marked at 0, 1, 2, 3, 4, 5, and 6. You are given an integer array `cuts` where `cuts[i]` represents a position along the stick where you need to make a cut.

The primary task is to cut the stick at these marked positions in such a way that the total cost is minimized. The cost of making a cut is equal to the current length of the stick being cut. After a cut, the stick is divided into two smaller sticks. The objective is to determine the minimum total cost to perform all the cuts.

A key point to note is that the order of the cuts can be changed. The goal is not just to perform the cuts in the order they are given, but to find the order that results in the least total cost.

Intuition

To arrive at the solution, we need to realize that this problem is a classic example of *dynamic programming*, specifically the matrix-chain multiplication problem. The main idea is to find the most cost-effective order to perform the cuts, akin to finding the optimal parenthesization of a product of matrices.

Here's the intuition to approach the problem:

- Introduce "virtual" cuts at the start and end of the stick (positions 0 and n). These do not contribute to the cost but serve as the boundaries of the stick.
- Arrange the cuts in a sorted sequence, including the virtual cuts.
- Define a recursive relation to represent the minimum cost of making cuts between two endpoints on the stick.
- Realize that the best way to split the stick involves splitting it at some point i between two endpoints, leading to two new problems of the same nature but on smaller sticks. The cost to make a cut at i would include the cost of cutting the left part, the cost of cutting the right part, and the cost of the cut itself, which is the length of the stick between the two endpoints.
- The *dynamic programming* aspect involves storing solutions to subproblems to avoid redundant calculations. This is done by filling a table `f` with the minimum costs for cutting between every possible pair of cuts.
- The solution `f[0][n]` will be the computed value representing the minimum cost to cut the entire stick between the first and last positions (`cuts[0]` and `cuts[-1]`).

To ensure we don't miss the best split point for a given segment of the stick, every possible split point k needs to be considered, and the minimum cost to cut between the endpoints will be stored in the table `f`.

Solution Approach

The implementation of the solution follows the *dynamic programming* approach, and here is a step-by-step explanation:

1. Include the start and end of the stick as cuts, so we add `0` and `n` to the list of cuts for convenience. After this, the `cuts` array contains all the positions where we need to make the cuts. Inserting the beginning and end of the stick into our cuts array doesn't change the problem but simplifies the implementation.

```
cuts.extend([0, n])
```

2. Sort the `cuts` array. It's important that the `cuts` array is sorted so that we can consider each segment between two cuts in order. This is a common practice for *dynamic programming*.

```
cuts.sort()
```

3. Create a 2D list `f` for storing the minimum costs with all values initially set to zero. The size of the list is the number of cuts including the virtual cuts at the beginning and the end. `f[i][j]` will store the minimum cost to cut the stick from `cuts[i]` to `cuts[j]`.

```
m = len(cuts)
f = [[0] * m for _ in range(m)]
```

4. Use a nested loop to consider all segments between cuts with length `l` ranging from 2 to `m` (inclusive), effectively covering all possible sub-segments of the stick. `l=2` means just one cut in the segment (since we're including the two virtual cuts at the ends), and `l=m` would mean considering the entire stick.

```
for l in range(2, m):
    for i in range(m - l):
        j = i + l
        f[i][j] = inf
        ...
```

5. For each pair of indices `i` and `j`, iterate through all potential cutting points `k` between `i` and `j` to find the minimum cost. We have initialized `f[i][j]` with infinity, and we aim to find the least possible cost.

```
for k in range(i + 1, j):
    f[i][j] = min(f[i][j], f[i][k] + f[k][j] + cuts[j] - cuts[i])
```

6. The above step considers each possible way to split the stick into two parts: the left part from `cuts[i]` to `cuts[k]` and the right part from `cuts[k]` to `cuts[j]`. The minimum cost for the split position `k` is the sum of the minimum cost to cut the left part (`f[i][k]`), the minimum cost to cut the right part (`f[k][j]`), and the cost of making the actual cut at `k` which is `cuts[j] - cuts[i]` (the current length of the stick segment we're cutting).

7. Finally, return the value `f[0][-1]` which now contains the minimum cost to cut the entire stick, taking into the account the order in which cuts must be made to minimize the total cost.

```
return f[0][-1]
```

The implementation uses a *dynamic programming* table to ensure that the solution to each subproblem is only calculated once and then reused, leading to a more efficient algorithm than a naive recursive approach would allow.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Suppose we have a wooden stick with a length of $n = 7$ units, and we have been asked to make cuts at positions given by the array `cuts = [1, 3, 4, 5]`.

1. First, we include the start and end of the stick as cuts according to step 1, so our cuts array becomes `cuts = [1, 3, 4, 5, 0, 7]`.
2. Next, we sort the `cuts` array, resulting in `cuts = [0, 1, 3, 4, 5, 7]`. This allows us to treat this array as a sequence of segments to consider in order.
3. Following step 3, we create a 2D list `f` for storing the minimum costs. Since there are 6 positions to make cuts, including the virtual ones at the beginning and at the end, our matrix will be 6 by 6, filled with zeros.
4. Now we consider all segments between cuts with lengths ranging from 2 to 6 (the size of our `cuts` array).
5. For a segment length of 2 (meaning a single cut between two points), the cost is straightforward. It is the difference between the points since there's no choice in how to split the stick. For example:
   - For segment `[0, 1, 3]`, the cost of cutting the stick at point 1 is the difference, $3 - 0 = 3$.
   - However, for segments longer than that, we must consider potential splitting points. Let's take a segment `[0, 1, 3, 4]`. We can cut at 1 or at 3.
   - If we cut at 1 first, the cost is 4 (length of the segment) + 2 (for cutting the second piece at 3) = 6.
   - If we cut at 3 first, the cost is 4 (length of the segment) + 1 (for cutting the first piece at 1) = 5. So we'll choose to cut at 3 first, and the minimum cost would be stored in `f[0][3]`.
6. We continue this process for larger segments, calculating costs and choosing the minimum until the entire matrix `f` is filled. For each segment, the selection of where to cut depends on the previously calculated minimum costs for smaller segments. This cumulative buildup ensures that we are using optimal substructure to find the minimum cost.
7. After we finish populating our matrix, the value stored at `f[0][5]` (which corresponds to the full length of the stick from `cuts[0]` to `cuts[-1]`) will be the minimum cost to cut the entire stick.

Let's simulate this for our segments. We assume *inf* is a placeholder for infinity.

	0	1	3	4	5	7
0	0	0	3	4	5	8
1		0	0	3	4	5
3			0	0	3	4
4				0	0	3
5					0	0

- The cost of making a cut at the last segment (from 5 to 7) is the length of the stick from 5 to 7, which is 2 units. This is reflected in `f[4][5]` at the end of the matrix.
- As observed during the example, the cost to cut at position `j` when considering from `i` to `i` is less than cutting at position `i`, so `f[0][3]` is populated with `3`.
- The final cost `f[0][5]` indicates that the entire stick can be cut at 1 units, which is the minimum cost to perform the cuts at `[1, 3, 4, 5]`.

By systematically considering each possible segment and sub-segment, storing the costs, and building upwards from those, we have used dynamic programming to avoid redundant calculations and to find the minimum cost to cut the stick at specific points.

Python Solution

```
class Solution:
    def minCost(self, n: int, cuts: List[int]) -> int:
        # Extend the list of cuts to include the start (0) and the end (n)
        cuts.extend([0, n])
        # Sort the cuts so they are in ascending order
        cuts.sort()
        # Count the total number of cuts (including start and end)
        num_cuts = len(cuts)
        # Create a 2D list (matrix) filled with zeros for dynamic programming
        dp = [[0] * num_cuts for _ in range(num_cuts)]

        # Start evaluating the minimum cost in increasing order of lengths
        for length in range(2, num_cuts):
            # Iterate over the cuts for the current length
            for start in range(num_cuts - length):
                end = start + length
                # Initialize current cell with infinity, representing the uncalculated minimum cost
                dp[start][end] = float('inf')
                # Check for the best place to split the current piece being considered
                for k in range(start + 1, end):
                    # Calculate the cost of the current cut and compare it
                    # to the cost already present in this cell
                    dp[start][end] = min(dp[start][end], dp[start][k] + dp[k][end] + cuts[end] - cuts[start])

        # Return the minimum cost to make all cuts for the full length of the rod
        return dp[0][-1]
```

Java Solution

```
class Solution {
    public int minCost(int n, int[] cuts) {
        // Initialize a new list to hold the cuts as well as the two ends of the stick
        List<Integer> cutPoints = new ArrayList<>();

        // Add each cut point from the given cuts array to the list
        for (int cut : cuts) {
            cutPoints.add(cut);
        }

        // Add the start and the end of the stick to the list as cut points
        cutPoints.add(0);
        cutPoints.add(n);

        // Sort the list of cut points (includes the ends of the stick now)
        Collections.sort(cutPoints);

        // Calculate the size of the list post additions
        int size = cutPoints.size();

        // Initialize a 2D array to store the minimum cost for cutting between two points
        int[][] dp = new int[size][size];

        // Start from the second cut point because the first will always be zero
        for (int length = 2; length < size; ++length) {
            // Determine the minimum cost for each range of cut points
            for (int start = 0; start + length < size; ++start) {
                // Mark the end of the range starting from i
                int j = i + length;

                // Initialize the minimum cost for this range as a large number
                dp[i][j] = Integer.MAX_VALUE;

                // Try all possible intermediate cuts to find the minimum cost split
                for (int k = i + 1; k < j; ++k) {
                    // Calculate the cost of making this cut and if it's the new minimum, update dp[i][j]
                    dp[i][j] = Math.min(dp[i][j], dp[i][k] + dp[k][j] + cutPoints.get(j) - cutPoints.get(i));
                }
            }
        }

        // Return the minimum cost to cut the entire stick
        return dp[0][size - 1];
    }
}
```

C++ Solution

```
#include <vector>
#include <algorithm>
using namespace std;

class Solution {
public:
    int minCost(int totalLength, vector<int>& cuts) {
        // Initialize the vector with 0s (cuts start and end of the rod)
        cuts.push_back(totalLength);
        cuts.push_back(totalLength);
        // Sort the cuts for dynamic programming approach
        sort(cuts.begin(), cuts.end());

        int numOfCuts = cuts.size();
        int costMatrix[110][110] = {0}; // Assuming we will not have more than 100 cuts

        // Filling cost matrix diagonally, starting from the smallest subproblems
        for (int len = 2; len < numOfCuts; ++len) {
            for (int i = 0; i + len < numOfCuts; ++i) {
                int j = i + len;
                costMatrix[i][j] = 1 << 30; // Initialize with a large number (infinity)

                // Find the minimum cost to cut this piece further
                for (int k = i + 1; k < j; ++k) {
                    int cost = cuts[j] + costMatrix[i][k] + costMatrix[k][j] - cuts[i];
                    costMatrix[i][j] = min(costMatrix[i][j], cost);
                }
            }
        }

        // Return the minimum cost to cut the whole rod
        return costMatrix[0][numOfCuts - 1];
    }
};
```

Typescript Solution

```
function minCost(n: number, cuts: number[]): number {
    // Add the start and end of the rod to the cuts array
    cuts.push(0);
    cuts.push(n);

    // Sort the cuts in ascending order
    cuts.sort((a, b) => a - b);

    // Determine the number of cuts including the start and end positions
    const totalCuts = cuts.length;

    // Initialize a 2D array (matrix) to store the minimum cost for each segment
    const dpMatrix: number[][] = new Array(totalCuts).fill(0).map(() => new Array(totalCuts).fill(0));

    // Iterate over the lengths of the segments starting from 2 (since we need at least two cuts to define a segment)
    for (let len = 2; len < totalCuts; ++len) {
        // Iterate over all possible segments for the current length
        for (let i = 0; i + len < totalCuts; ++i) {
            // Calculate the end index of the segment
            const j = i + len;

            // Initialize the minimum cost for current segment to a large number (acts like infinity)
            dpMatrix[i][j] = Number.MAX_SAFE_INTEGER;

            // Iterate over all possible positions to make a cut within the current segment
            for (let k = i + 1; k < j; ++k) {
                // Update dpMatrix[i][j] with the minimum cost found so far
                // Options: dpMatrix[i][j] with the minimum cost found so far
                //          dp[i][j] + dp[k][j] + cuts[j] - cuts[i]
                dpMatrix[i][j] = Math.min(dpMatrix[i][j], dpMatrix[i][k] + dpMatrix[k][j] + cuts[j] - cuts[i]);
            }
        }
    }

    // Return the minimum cost to cut the rod into the specified pieces
    return dpMatrix[0][totalCuts - 1];
}
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code can be analyzed by examining the nested loops and the operations within those loops:

- There are two outer loops iterating over the length `l` from 2 to `n` (where `n` is the total number of cuts plus the two endpoints, 0 and `n`), and over the starting index `i`. This results in a time complexity of approximately $O(n^2)$ for these two loops.
- Inside the two outer loops, there is an inner loop that iterates over the possible intermediate cuts `k` between the `i`th cut and the `j`th cut. In the worst case, this inner loop runs $O(n)$ times.
- The operations inside the inner loop involve computing the minimum of the current value and the sum of the costs for the left and right pieces plus the cost of making this cut: `f[i][k] + f[k][j] + cuts[j] - cuts[i]`. This operation runs in constant time, $O(1)$.

Multiplying these together, the overall time complexity is $O(n^2) * O(n) * O(1) = O(n^3)$.

Space Complexity

The space complexity can be determined by analyzing the memory allocations in the code:

- The 2D list `f` of size $n \times n$ is the primary data structure, where n is the number of cuts plus 2 (for the endpoints). This 2D list constitutes a space complexity of $O(n^2)$.
- No other significant data structures are used, and the input list `cuts` is modified in place (not requiring additional space relative to inputs).

Thus, the space complexity is $O(n^2)$.