956. Tallest Billboard

Hard | Array | Dynamic Programming | Leetcode Link

Problem Description

In this problem, you are given a collection of rods of different lengths, and your goal is to use these rods to create two steel supports for a billboard. The challenge is to make the two supports of equal height. As the supports must be of equal height, you have the flexibility to weld multiple rods together to achieve that. However, you can't cut rods into pieces; you must use the whole rods.

You need to calculate the maximum height that both supports can reach while ensuring they are of the same height. If it's not possible to create two supports of equal height with the given rods, you should return `0`.

For instance, if you are provided with rods of lengths `1, 2, 3`, one of the ways to create two equal supports is to use the rods `1` and `2` to make one support with a height of `3`, and the `3` rod as the other support, also with a height of `3`.

Intuition

The intuition behind the solution to this problem lies in dynamic programming, which is a strategy used to solve optimization problems such as this by breaking it down into simpler subproblems. The key to dynamic programming is caching the results of the subproblems to avoid redundant calculations that would otherwise occur in a naive recursive approach.

In this case, we want to keep track of ways to combine rods in such a fashion that the difference in height between the two supports (left and right) can be maintained, or potentially reduced to zero. The approach involves considering each rod and deciding what to do with it for each possible difference in supports' heights: either add it to the taller support, add it to the shorter support, or do not use it at all.

To this end, we create a table `f` with the first dimension representing the rods considered so far and the second dimension representing the possible differences in height between the two supports. The value in `f[i][j]` is the tallest height of the shorter support where `i` rods are considered and the difference in height between the two supports is `j`.

At every rod `i`, and for each possible height difference `j`, we consider three scenarios:

1. Not using the rod at all and keeping the current height difference.
2. Adding the rod to the shorter support, which increases its height and decreases the difference `j`.
3. Adding the rod to the taller support, which increases the height difference `j`.

We aim to maximize the height of the shorter support since the two supports must be of equal height.

Iteration continues, building upon previous calculated states, until all rods are considered. The result will be the maximum height of the shorter support with a height difference of `0`, meaning both supports are of equal height.

Solution Approach

The solution uses dynamic programming to iteratively build up a table that represents the maximum height of the shorter support for all combinations of rods considered up to that point and all possible differences in height between the two supports.

Here's how the implementation breaks down:

- **Initialization**: The solution creates a 2D list `f` where `f[i][j] = -inf` for all `j`. This list will be filled during the iterations. `f[i][j]` will hold the maximum height of the shorter tower when considering the first `i` rods, and the difference in height between two towers is `j`. We initialize `f[0][0] = 0` because with zero rods considered, the maximum height of the shorter tower with zero height difference is zero.

- **Outer Loop**: The algorithm iterates over each rod in `rods`. The variable `t` is used to keep track of the total length of all rods considered so far, which determines the range of `j` (the difference in height) we need to consider in the inner loop.

- **Inner Loop**: The inner loop iterates through all possible differences in height `j` from `0` to `t`. It computes the maximum height `f[i][j]` of the shorter support for this difference using the following logic:

  1. **Case 1**: Do not use the current rod. The maximum height for the current rod and difference `j` will be the same as the previous rod for the same difference `j`, which is `f[i - 1][j]`.

  2. **Case 2**: Use the current rod on the shorter support. If the current difference `j` is greater than or equal to the current rod length `x`, we compare the current value with `f[i - 1][j - x] + x` and take the maximum. This is because we can add the current rod to the shorter support, increasing its height by `x`, and thus reducing the difference by `x`.

  3. **Case 3**: Use the current rod on the taller support. If the sum of `j` and the current rod length `x` does not exceed `t`, we compare the current value with `f[i - 1][j + x] + x` and take the maximum. This effectively adds the rod length to the difference `j`.

  4. **Case 4**: If `j` is less than the current rod length `x`, we can still add the current rod to the shorter support. In this case, the shorter support would become the taller one, and the difference in height would be `x - j`. So we compare the current value with `f[i - 1][x - j] + x - j` and take the maximum, since in this case we would transfer the excess length `(x - j)` to the previously taller support.

- **Finalization**: After considering all rods and all potential height differences, the maximum height of the billboard will be `f[n][0]`, where `n` is the number of rods. This cell represents the maximum height of the shorter tower with zero height difference between the two towers, which means they are of equal height.

Example Walkthrough

Let's consider a small example to illustrate the solution approach using rods of lengths `1, 2, 4`. We need to create two supports of equal height using these rods.

- **Initialization**:
  We initialize an empty table `f` such that `f[i][j] = -inf` for all `j`.
  `f[0][0]` is set to `0` since no rods means no height, and also no height difference.

- **Outer Loop (Iteration over rods)**:
  - First rod: 1
    - Total length `t` so far: 1
    - Possible differences `j`: `0` to 1
    - For `j = 0`:
      1. Do not use the rod: f[1][0] remains `0` (Case 1)
      2. Use the rod on the shorter support: f[1][1] is updated to 1 (Case 2)
    - For `j = 1`:
      (This represents the case where there's already a difference of 1 which isn't possible with only one rod.)
  - Second rod: 2
    - Total length `t` so far: 3
    - Possible differences `j`: `0` to 3
    - For `j = 0`:
      1. Do not use the rod: f[2][0] remains `0` (Case 1)
      2. Use the rod on the shorter support: f[2][2] is updated to 2 since now both supports can have a height of 1 (Case 4)
    - For `j = 1`:
      (This would represent the scenario where one support is taller by 1.)
    - For `j = 2`:
      1. Use the rod on the taller support: f[2][2] becomes 2, as we can have two supports of height 1 (Case 3)
    - For `j = 3`:
      (This scenario isn't viable with the rods we have so far.)
  - Third rod: 4
    - Total length `t` so far: 7
    - Possible differences `j`: `0` to 7
    - For `j = 0`:
      (We already have f[2][2] = 2, suggesting two supports at 1)
      1. Do not use the rod: f[3][0] remains `0` (Case 1)
      2. Use the rod on the shorter support: f[3][4] is updated to 4 (Case 2)
      3. We would not use this rod on the taller support, as it would make the supports unequal.
    - For `j = 2`:
      (The current best at f[2][2] = 2)
      1. Use the current rod on the shorter support: f[3][2] becomes 4 (Case 4)
    - For `j = 4`:
      1. Use the rod on the taller support: f[3][4] is already 4, but this scenario is also valid for creating two supports of equal height of 2.

- **Finalization**:
  After iterating over all the rods, we look at `f[3][0]` to find the maximum height of the shorter support with zero height difference. In our case, `f[3][0]` is `0`, but looking at the populated table reveals that we can make two supports of height 2 using the rods 2 and 4, hence the maximum height for the billboard supports is 2.

Python Solution

```python
from typing import List

class Solution:
    def tallestBillboard(self, rods: List[int]) -> int:
        # Number of rods
        num_rods = len(rods)
        # Sum of all rod lengths
        total_length = sum(rods)
        # Initialize a DP table filled with negative infinity to track the highest score
        # Shape: (num_rods + 1) x (total_length + 1)
        dp_table = [[float('-inf')] * (total_length + 1) for _ in range(num_rods + 1)]
        # Base case: When there are no rods, the height difference of 0 is achievable with height 0
        dp_table[0][0] = 0

        # Running sum of rod lengths
        current_sum = 0
        # Iterate over rods
        for i, length in enumerate(rods, 1):
            current_sum += length
            # Try possible heights between 0 and the running sum of rod lengths
            for j in range(current_sum + 1):
                # Case 1: Do not add the current rod
                dp_table[i][j] = dp_table[i - 1][j]
                # Case 2: Add the current rod to one side
                if j >= length:
                    dp_table[i][j] = max(dp_table[i][j], dp_table[i - 1][j - length])
                # Case 3: Add the current rod to the other side
                if j + length <= current_sum:
                    dp_table[i][j] = max(dp_table[i][j], dp_table[i - 1][j + length] + length)
                # Case 4: Add the current rod to the taller side to make sides more equal
                if j < length:
                    dp_table[i][j] = max(dp_table[i][j], dp_table[i - 1][length - j] + length - j)

        # The goal is to achieve the maximum equal height with height difference 0
        return dp_table[num_rods][0]
```

Java Solution

```java
import java.util.Arrays; // Import Arrays utility class

class Solution {
    public int tallestBillboard(int[] rods) {
        int numRods = rods.length;
        int sumRods = 0;
        // Calculate the sum of all elements in the array rods
        for (int rod : rods) {
            sumRods += rod;
        }

        // Initialize the dp (Dynamic Programming) array with a very small negative value
        int[][] dp = new int[numRods + 1][sumRods + 1];
        for (int[] row : dp) {
            Arrays.fill(row, Integer.MIN_VALUE / 2);
        }

        // The base case - a pair of empty billboards has a height difference of 0
        dp[0][0] = 0;

        // Iterate over the rods
        for (int i = 1, totalHeight = 0; i <= numRods; ++i) {
            int currentRod = rods[i - 1];
            totalHeight += currentRod;

            // Update dp array for all possible height differences
            for (int heightDiff = 0; heightDiff <= totalHeight; ++heightDiff) {
                // Case 1: Do not use the current rod
                dp[i][heightDiff] = dp[i - 1][heightDiff];

                // Case 2: Use the current rod in the taller billboard
                if (heightDiff >= currentRod) {
                    dp[i][heightDiff] = Math.max(dp[i][heightDiff], dp[i - 1][heightDiff - currentRod]);
                }

                // Case 3: Use the current rod in the shorter billboard
                if (heightDiff + currentRod <= totalHeight) {
                    dp[i][heightDiff] = Math.max(dp[i][heightDiff], dp[i - 1][heightDiff + currentRod] + currentRod);
                }

                // Case 4: Current rod makes up the difference in billboard heights
                if (heightDiff < currentRod) {
                    dp[i][heightDiff] = Math.max(dp[i][heightDiff], dp[i - 1][currentRod - heightDiff] + currentRod - heightDiff);
                }
            }
        }

        // The maximum height of 2 billboards with the same height (height difference of 0)
        return dp[numRods][0];
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <numeric>
#include <cstring>
#include <algorithm>

class Solution {
public:
    int tallestBillboard(std::vector<int>& rods) {
        const int numRods = rods.size();
        // Find the sum of all rod lengths to define the dimensions of dp array
        const int sumRods = std::accumulate(rods.begin(), rods.end(), 0);

        // dp[i][j] will store the maximum height of the taller tower
        // of the two we are trying to balance when considering the first i rods
        // where the difference in height between the towers is j
        std::vector<std::vector<int>> dp(numRods + 1, std::vector<int>(sumRods + 1));

        // Initialize dp array with very negative numbers to represent unattainable states
        for (auto &row : dp) {
            std::fill(row.begin(), row.end(), INT_MIN/2);
        }

        // Base case: the first 0 rods, with 0 height difference has 0 height
        dp[0][0] = 0;

        for (int i = 1, totalRodLength = 0; i <= numRods; ++i) {
            int rodLength = rods[i - 1];
            totalRodLength += rodLength;
            for (int j = 0; j <= totalRodLength; ++j) {
                // Don't use the current rod, inherit the value from the previous decision
                dp[i][j] = dp[i - 1][j];

                // If possible, add current rod to the shorter tower to try and balance the towers
                if (j >= rodLength) {
                    dp[i][j] = std::max(dp[i][j], dp[i - 1][j - rodLength]);
                }
                // If possible, add current rod to the taller tower
                if (j + rodLength <= totalRodLength) {
                    dp[i][j] = std::max(dp[i][j], dp[i - 1][j + rodLength] + rodLength);
                }
                // If current rod is longer than the difference j, then add the
                // difference to the shorter tower to balance the towers
                if (rodLength >= j) {
                    dp[i][j] = std::max(dp[i][j], dp[i - 1][rodLength - j] + j);
                }
            }
        }

        // The final state for balanced towers (difference = 0) is the answer
        return dp[numRods][0];
    }
};
```

Typescript Solution

```typescript
function tallestBillboard(rods: number[]): number {
    // Sum of all rod lengths
    const totalLength = rods.reduce((acc, rod) => acc + rod, 0);
    // The number of rods present
    const numRods = rods.length;
    // Initialize a DP table with default values of -1
    const dpTable = new Array(numRods).fill(0).map(() => new Array(totalLength + 1).fill(-1));

    // Define the depth-first search function for DP computation
    const depthFirstSearch = (currentIndex: number, currentDifference: number): number => {
        // Base case: if we have considered all rods, return 0 if no difference, else return negative infinity
        if (currentIndex >= numRods) {
            return currentDifference === 0 ? 0 : Number.MIN_SAFE_INTEGER;
        }
        // Return the cached result if already calculated for this state
        if (dpTable[currentIndex][currentDifference] !== -1) {
            return dpTable[currentIndex][currentDifference];
        }

        // Compute max height by ignoring the current rod
        let maxHeight = Math.max(depthFirstSearch(currentIndex + 1, currentDifference),
                                 // Including the current rod in one of the sides
                                 depthFirstSearch(currentIndex + 1, currentDifference + rods[currentIndex]));
        // Including the current rod in the shorter side if it makes the billboard taller
        maxHeight = Math.max(maxHeight,
                             depthFirstSearch(currentIndex + 1, Math.abs(currentDifference - rods[currentIndex]))
                             + Math.min(currentDifference, rods[currentIndex]));
        // Cache the result in DP table
        return (dpTable[currentIndex][currentDifference] = maxHeight);
    };

    // Call the dfs function to compute the maximum height of balanced billboard
    return depthFirstSearch(0, 0);
}
```

Time and Space Complexity

The time complexity of the function `tallestBillboard` is `O(n * s^2)`, where `n` is the number of elements in `rods` and `s` is the sum of all elements in `rods`. This is because there is a nested loop structure within the function: one loop iterating over the `rods` with length `n` and two nested loops iterating up to `s`, resulting in a cubic time complexity with respect to `s`.

The space complexity of the function is `O(n * s)`. This is due to the two-dimensional array `f` which has dimensions `[n + 1]` by `[s + 1]`, resulting in space usage proportional to the product of `n` and `s`.