1230. Toss Strange Coins

Medium | Array | Dynamic Programming | Probability and Statistics

Problem Description

Imagine you have a collection of coins, each with its own probability of landing heads up when flipped. You flip each coin exactly once. The goal is to find out the likelihood that you will get a specific number of heads—let's call this number `target`. This is not just a simple coin flip where the outcome is a 50/50 chance; instead, each coin has a unique probability that it will land heads up when tossed. So to solve this problem, we need a way to compute the combined probability of achieving exactly `target` heads in one toss of all the coins.

Intuition

To find the solution, we use a common technique in probability and computer science problems called Dynamic Programming (DP). Dynamic Programming is particularly useful when we need to keep track of previous outcomes to calculate the next ones. Here, we want to calculate the probabilities step by step for each coin and number of heads.

The intuition behind the DP approach is based on the fact that for every coin, there are two possible outcomes: it lands heads or tails. The probability of getting heads up to the current coin depends on whether we get heads or tails on this coin flip.

We keep track of probabilities in an array where each element at index `j` represents the probability of having `j` heads after flipping a certain number of coins. Initially, there is a 100% chance of having 0 heads (since we haven't flipped any coins), so we start with the array `[1, 0, 0, ..., 0]`.

When we flip a coin, we update our array. For every number of heads `j` that we might have had before flipping this coin, we now have two new probabilities to consider:

- The coin could land tails (`1 - p`) and our count of heads stays the same.
- The coin could land heads (`p`) and our count of heads increases by 1.

As we are interested in cumulative probabilities, we update the number of ways to have `j` heads by adding the probability of landing heads times the number of ways to have `j-1` heads before this flip.

By iterating through each coin and updating probabilities for each possible number of heads, we build up to the total probability of getting exactly `target` heads across all coins.

The provided solution cleverly reduces space complexity by using a single-dimensional array instead of a two-dimensional one, as we only need information from the previous step to calculate the probabilities for the current step.

Solution Approach

The solution uses Dynamic Programming (DP), which is an algorithmic technique for solving a complex problem by breaking it down into simpler subproblems and solving each of these subproblems just once, and storing their solutions.

Data Structures:

- A one-dimensional array `f` is used to store the probabilities of achieving a certain number of heads up to the current coin. The length of `f` is `target + 1` because the index ranges from 0 (no heads) to the target number of heads inclusively.

Algorithm:

1. Initialize the array `f` with zeros and set `f[0]` to 1, because initially, there is a 100% chance of having zero heads before any coins are flipped.
2. Iterate through each coin's probability `p` in the given list `prob`:
    - For each possible number of heads `j` from `target` to 0 (counting down so the previous probabilities is not overwritten prematurely):
        - Multiply the current value of `f[j]` by `1 - p`, since if the coin lands tails, the probability of having `j` heads remains the same but needs to be adjusted by the probability of getting tails.
        - If `j` is not zero, increment `f[j]` by `p * f[j - 1]`, which accounts for the case where the coin lands heads, thus the probability of having `j` heads from the previous iteration contributes to the current one.
3. After finishing the iteration for all coins, the value at `f[target]` will be the desired probability.

Patterns:

- **Memory optimization**: Instead of maintaining a two-dimensional array which can be space expensive (`O(n*target)` space), the array is compressed to a one-dimensional array (`O(target)` space), since each state only depends on the immediately preceding state.
- **Overlapping Subproblems**: By storing the probabilities in the array `f` and using them for subsequent calculations, we avoid recalculating the same probabilities over and over again, which is a common pattern in DP to reduce the time complexity.
- **Bottom-Up DP**: The problem is solved iteratively starting from the smallest subproblem (zero coins flipped) and building up the solution to the larger problem (all coins flipped).

Using this approach, the solution builds up probabilities for all subtargets (0 through `target`) for all coins, ensuring that by the end of the iteration, the value `f[target]` is the correct probability for getting exactly `target` heads.

The complexity of the algorithm is `O(n * target)`, where `n` is the number of coins, due to the nested loops over the coins and the subtargets.

Example Walkthrough

Let's consider a small example where we have 3 coins with probabilities of landing heads up as follows: `prob = [0.5, 0.6, 0.7]`, and we are trying to compute the probability of getting exactly `target = 2` heads.

Step-by-step:

1. We create our array `f` with a length of `target + 1`, which in this case is 3, and initialize it: `f = [1, 0, 0]`.
2. Now, we iterate through each coin's probability.
    - **First Coin (0.5 probability of heads)**
        - For `j = 2 (target)`: `f[2] = f[2] * (1 - 0.5) + f[1] * 0.5`.
        - For `j = 1`: `f[1] = f[1] * (1 - 0.5) + f[0] * 0.5`.
        - Since `f[0]` always stays as 1 (100% chance of 0 heads), we don't need to update it.
        - After the first coin, `f` becomes `[1, 0.5, 0]`.
    - **Second Coin (0.6 probability of heads)**
        - For `j = 2`: `f[2] = f[2] * (1 - 0.6) + f[1] * 0.6 = 0 * (1 - 0.6) + 0.5 * 0.6`.
        - For `j = 1`: `f[1] = f[1] * (1 - 0.6) + f[0] * 0.6 = 0.5 * (1 - 0.6) + 1 * 0.6`.
        - After the second coin, `f` becomes `[1, 0.8, 0.3]`.
    - **Third Coin (0.7 probability of heads)**
        - For `j = 2`: `f[2] = f[2] * (1 - 0.7) + f[1] * 0.7 = 0.3 * (1 - 0.7) + 0.8 * 0.7`.
        - For `j = 1`: `f[1] = f[1] * (1 - 0.7) + f[0] * 0.7 = 0.8 * (1 - 0.7) + 1 * 0.7`.
        - After the third coin, `f` becomes `[1, 0.86, 0.59]`.
3. After processing all the coins, we look at the value `f[target]` to get our answer. In this case, `f[2]` is 0.59, so the probability of getting exactly 2 heads is 59%.

Explanation:

- After flipping the first coin, we have a 50% chance of one head.
- After flipping the second coin, the probabilities get updated. The chances to get 1 head are now the sum of (0.5 chances to remain with 1 head from the previous stage, and not getting a head now) + (the chance we had 0 heads previously, which is 1, and getting a head now with the second coin). The same logic applies to getting 2 heads.
- The third coin updates the probabilities once more following the same formula. We sum the chance of staying with the same number of heads after not getting head with this coin and the chance of having one less head before and getting a head with this coin.

This DP algorithm efficiently keeps a running computation of probabilities as coins are considered, utilizing the previous computations to inform the next, leading to our final probability in `f[target]`.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def probabilityOfHeads(self, probabilities: List[float], target: int) -> float:
        # `dp` is a list where dp[j] represents the probability to get `j` heads so far
        dp = [0.0] * (target + 1)

        # The probability to get 0 heads (all tails) is initially 1
        dp[0] = 1.0

        # Iterate through each coin's probability
        for p in probabilities:
            # Iterate backwards through the number of heads we're looking for
            # This is to prevent overwriting values that we still need to use
            for j in range(target, -1, -1):
                # Update dp[j] for the probability of not flipping a head with this coin
                dp[j] *= (1 - p)

                # If we have at least one head,
                # add the probability of the previous number of heads
                # times the probability of this coin being a head
                if j > 0:
                    dp[j] += p * dp[j - 1]

        # Return the probability to get exactly `target` heads
        return dp[target]
```

Java

```java
class Solution {

    // This function calculates the probability of getting exactly 'target' number of heads
    // when coins with given probabilities are tossed
    public double probabilityOfHeads(double[] coinProbabilities, int target) {

        // Initialize an array to store the probabilities of getting exactly 'i' heads after tossing 'j' coins.
        // f[i] represents the probability of getting exactly 'i' heads.
        double[] probabilities = new double[target + 1];

        // Base case: the probability of getting 0 heads (all tails) is initially 1.
        probabilities[0] = 1;

        // Iterate over each coin probability.
        for (double coinProbability : coinProbabilities) {

            // Iterate backwards over the possible numbers of heads.
            // This is to prevent that the updating of f[j] affects the updating of f[j+1].
            for (int j = target; j >= 0; --j) {

                // Each time a coin is tossed, the probability of getting 'j' heads is updated:
                // 1. The probability of getting 'j' heads without the current coin (probability of tails
                //    of the current coin is multiplied by the previous 'j' heads probability).
                probabilities[j] *= (1 - coinProbability);

                // 2. If j is more than 0, update the probability by adding:
                //    the probability of getting 'j - 1' heads after the previous tosses and getting heads
                //    this time (current coin's probability of heads is multiplied by the previous 'j-1' heads probability).
                if (j > 0) {
                    probabilities[j] += coinProbability * probabilities[j - 1];
                }
            }
        }

        // The final result is the probability of getting exactly 'target' heads after tossing all coins.
        return probabilities[target];
    }
}
```

C++

```cpp
#include <vector> // Required for using the vector type
#include <cstring> // Required for using the memset function

class Solution {
public:
    // Function to calculate the probability of getting exactly 'target' number of heads.
    double probabilityOfHeads(vector<double>& prob, int target) {
        // Create an array to store the probabilities of getting a certain number of heads
        vector<double> dp(target + 1, 0.0);

        // Initialize the probability of getting 0 heads to 1
        dp[0] = 1.0;

        // Iterate over the probability of heads for each coin toss
        for (double p : prob) {
            // Update the probabilities in reverse order to avoid overwriting values needed for calculations
            for (int j = target; j >= 0; --j) {
                // When a coin turns up tails, probability is (1 - probability of head) * probability of previous state
                dp[j] *= (1 - p);

                // If not the first coin (since we can't have a negative number of coins),
                // add the probability of getting one less head multiplied by the probability of getting head
                if (j > 0) {
                    dp[j] += p * dp[j - 1];
                }
            }
        }
        // Return the probability of getting exactly 'target' number of heads
        return dp[target];
    }
};
```

TypeScript

```typescript
function probabilityOfHeads(probabilityArray: number[], targetHeads: number): number {
    // Initialize dp (Dynamic Programming) array which will hold the probability of getting 'i' heads
    let dp = new Array(targetHeads + 1).fill(0);
    // There is always a 100% chance of getting 0 heads (all tails)
    dp[0] = 1;

    // Iterate through the probabilities of flipping a head for each coin
    for (const probability of probabilityArray) {
        // Iterate backwards from the target number of heads to zero
        // This is needed to ensure we are using results from 'previous' coins
        for (let j = targetHeads; j >= 0; --j) {
            // Update the dp array for 'j' heads with the probability of flipping a tail
            dp[j] *= 1 - probability;
            // If we are not at the first coin, update the dp array to include the case where the current coin is head
            if (j > 0) {
                dp[j] += dp[j - 1] * probability;
            }
        }
    }
    // Return the probability of getting exactly 'targetHeads' heads
    return dp[targetHeads];
}
```

```python
from typing import List

class Solution:
    def probabilityOfHeads(self, probabilities: List[float], target: int) -> float:
        # `dp` is a list where dp[j] represents the probability to get `j` heads so far
        dp = [0.0] * (target + 1)

        # The probability to get 0 heads (all tails) is initially 1
        dp[0] = 1.0

        # Iterate through each coin's probability
        for p in probabilities:
            # Iterate backwards through the number of heads we're looking for
            # This prevents overwriting values that we still need to use
            for j in range(target, -1, -1):
                # Update dp[j] for the probability of not flipping a head with this coin
                dp[j] *= (1 - p)

                # If we have at least one head,
                # add the probability of the previous number of heads
                # times the probability of this coin being a head
                if j > 0:
                    dp[j] += p * dp[j - 1]

        # Return the probability to get exactly `target` heads
        return dp[target]
```

Time and Space Complexity

// The time complexity of the code is `O(n * target)` because there are two nested loops, where `n` is the number of coins (the outer loop) and `target` is the number of successful flips we want (the inner loop runs in reverse from `target` to `0`). Each iteration of the inner loop performs a constant number of operations.

// The space complexity of the code is `O(target)` since a one-dimensional list `f` of size `target + 1` is being used to store intermediate probabilities. No other data structures are used that scale with the size of the input.