1155. Number of Dice Rolls With Target Sum

Medium | Dynamic Programming | Leetcode Link

Problem Description

In this problem, we are working with a scenario involving `n` dice, each having `k` faces numbered from 1 to `k`. We want to figure out how many different ways we can roll these dice so that the sum of the numbers showing on the tops of all dice equals a specific `target` value. To clarify, there are a total of `k^n` different roll combinations since each die can land on any of its `k` faces and we have `n` dice in total. The challenge is to filter out and count only those combinations where the sum equals `target`. Because the number of possible ways can get extremely large, the result should be provided modulo `10^9 + 7` to avoid overflow issues.

Intuition

The core of the solution lies in dynamic programming, a method that involves breaking down a complex problem into smaller, more manageable subproblems and building up the solution from the answers to these subproblems.

The main idea is to keep track of the number of ways to achieve a certain sum using a particular number of dice. To do this, we define a state `f[i][j]`, which represents the count of ways to reach a sum of `j` using `i` dice. We start from the base case where no dice are used, so the only possible sum of zero is to have zero dice (which is one way).

From there, we incrementally build up by considering one more die at each step. For every number of dice `i` from 1 to `n` and for each possible sum `j` up to the `target`, we determine the number of ways to achieve that sum by considering all the possible outcomes of the new die. In simple terms, for every possible value `h` that the new die could land on, we add the number of ways to achieve the sum `j - h` using `i - 1` dice to our current count.

Through these iterations, we build a table of values reflecting the different ways to reach the sum, all the way up to using all `n` dice to reach the `target` sum. The final answer is then the value stored in `f[n][target]`.

The given solution cleverly optimizes the space complexity by using a single-dimensional array `f[]`, where the values of the previous iteration are used to calculate the values of the current iteration. As each value is computed, it is stored back in `f[]` after considering all possible faces of the current die. This is done to ensure that we are neither running out of memory due to a large number of subproblems nor repeating the calculation for the same subproblems multiple times.

Solution Approach

The implementation of the solution involves a two-dimensional problem being solved using a one-dimensional array to save space. The mathematical formula that guides the solution is:

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

This equation is the heart of our dynamic programming approach, where `f[i][j]` is the number of ways to get a sum of `j` using `i` dice. However, instead of using a two-dimensional array, we optimize the solution by reusing a single array `f` where `f[j]` at any point contains the number of ways to reach sum `j` with the current number of dice considered.

The innermost loop runs through all the possible face values (`h`) of the current die (up to `k`, but not more than `j` since we can't have the face value of a die be more than the sum we're trying to achieve). For each face value, it adds to the temporary array `g[j]` the number of ways we can achieve the remaining sum (`j - h`) with one less die, modulo `10^9 + 7` to prevent overflow.

After each die is considered (outer loop), the `f` array is updated to be the temporary array `g` for the next iteration, effectively moving from analyzing `i-1` dice to `i` dice while preserving the required state and avoiding the direct use of a two-dimensional array.

Here is an implementation perspective that breaks down each part of the code:

1. **Initialization:** An array `f` is initialized with size `target + 1`, starting with `f[0] = 1` and the rest `0`. This reflects the base case where a sum of `0` can be achieved in one way with zero dice.
2. **Building up solutions for each die:**
   - A temporary array `g` is initialized to hold the number of ways to reach every possible sum up to `target` with one additional die.
   - We iterate over each sum `j` that could be reached with the current number (`i`) of dice.
   - For each `j`, we calculate `g[j]` based on previous sums computed in `f`.
3. **Updating state:** After considering all sums for the current die, we make `f = g` to update our state without using additional space for a second dimension.
4. **Return the answer:** After considering all `n` dice, `f[target]` gives us the final number of ways to reach the `target` sum.
5. **Modulus Operation:** A `mod` constant is used at each step of the calculation to ensure the number stays within the specified range, `10^9 + 7`.

By iterating through dice and potential sums in this manner, the dynamic programming approach efficiently calculates the number of ways to roll the dice to reach a targeted sum.

Example Walkthrough

Let's illustrate the solution approach using an example where we have `n = 2` dice and each die has `k = 3` faces numbered from 1 to 3. We want to find out how many ways we can roll the dice so that the sum is equal to the `target` value of 4.

1. **Initialization:** We initialize an array `f` with a length of `target + 1`, which means `f` has a size of 5 in this case (0, 1, 2, 3, 4) since `target` is 4. We have `f = [1, 0, 0, 0, 0]`.
2. **Considering the first die:** We create a temporary array `g` which will be used to compute the sums reachable with one die. For `j = 1` to `target` (which is 4 in this example), we calculate the number of ways to reach the sum `j` by adding the face values of the die to `f[j - h]`, where `h` ranges from 1 to `min(j, k)`.
   - For `j = 1`, `g[1] = f[1 - 1] = f[0] = 1` (`f[0]` initialized as 1).
   - For `j = 2`, `g[2] = f[2 - 1] = f[1] = 1` (since we could roll a 2 on the first die).
   - For `j = 3`, `g[3] = f[3 - 1] = f[2] = 1` (rolling a 3 on the first die).
   - For `j = 4`, since the maximum face value (k) is 3, we cannot achieve a sum of 4 with one die, so `g[4] = 0`. After considering the first die, `f` gets updated to `f = g = [1, 1, 1, 1, 0]`.
3. **Considering the second die:** We repeat the process with the second die.
   - For `j = 1`, there's no change as we can't add a positive face value to reach 1 without exceeding it.
   - For `j = 2`, `g[2]` will now include the number of ways to reach a sum of `1` from the first die and then roll a `1` on the second die, which is just reusing previous `f[1] = 1`.
   - For `j = 3`, `g[3]` will include the ways to reach sums of `2` and `1` from the first die and rolling `1` or `2` on the second die, respectively. Thus, `g[3] = f[2] + f[1] = 1 + 1 = 2`.
   - For `j = 4`, we sum the ways to get `3`, `2`, and `1` with the first die and roll `1`, `2`, or `3` on the second die: `g[4] = f[3] + f[2] + f[1] = 1 + 1 + 1 = 3`. Now, `f` is updated to `f = g = [1, 1, 1, 2, 3]`.
4. **Return the answer:** After considering all `n` dice, in this case, two dice, the final computation for each sum is done. The final answer is `f[target]`, which is `f[4]` in this case, so there are 3 different ways to roll the dice to get the sum of 4.

By using this approach, we have efficiently computed the desired sum without needing a two-dimensional array, thus optimizing our space usage. For larger values of `n` and `k`, following the same steps while considering the modulo `10^9 + 7` would ensure we avoid integer overflow issues.

Python Solution

```python
class Solution:
    def numRollsToTarget(self, dice: int, sides: int, target: int) -> int:
        # Initialize the array of ways to achieve each sum, with 1 way to achieve a sum of 0
        ways_to_achieve_sum = [1] + [0] * target
        # Define the modulo according to the problem statement to avoid large numbers
        modulo = 10**9 + 7

        # Iterate through each dice
        for i in range(1, dice + 1):
            # Initialize the temporary array for the current dice's sum calculations
            current_ways = [0] * (target + 1)
            # Calculate the number of ways to achieve each sum with the current number of dice
            for sum_value in range(1, min(i * sides, target) + 1):
                # Calculate the ways to get to this sum_value using the previous number of dice
                for face_value in range(1, min(sum_value, sides) + 1):
                    current_ways[sum_value] = (current_ways[sum_value] + ways_to_achieve_sum[sum_value - face_value]) % modulo
            # Update the array of ways with the current dice's calculation
            ways_to_achieve_sum = current_ways

        # Return the total number of ways to achieve the target sum with all dice
        return ways_to_achieve_sum[target]

```

Java Solution

```java
class Solution {
    public int numRollsToTarget(int n, int k, int target) {
        final int MODULO = (int) 1e9 + 7; // Define the modulo to prevent integer overflow
        int[] dp = new int[target + 1]; // dp array to store the number of ways to achieve each sum
        dp[0] = 1; // Base case: there's 1 way to achieve sum 0 - no dice rolled

        // Loop through each dice
        for (int diceCount = 1; diceCount <= n; ++diceCount) {
            int[] tempDp = new int[target + 1]; // Temporary array for the current number of dices

            // Calculate number of ways for each sum value possible with the current number of dices
            for (int currentSum = 1; currentSum <= Math.min(target, diceCount * k); ++currentSum) {

                // Go through each face value of the dice and accumulate ways to achieve 'currentSum'
                for (int faceValue = 1; faceValue <= Math.min(currentSum, k); ++faceValue) {
                    tempDp[currentSum] = (tempDp[currentSum] + dp[currentSum - faceValue]) % MODULO;
                }
            }

            // Update the dp array with the current number of dices' results
            dp = tempDp;
        }

        // Return the number of ways to achieve the 'target' sum with 'n' dices
        return dp[target];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to calculate the number of distinct ways to roll the dice
    // such that the sum of the face-up numbers equals to 'target'
    int numRollsToTarget(int numberOfDice, int faces, int targetSum) {
        const int MODULO = 1e9 + 7; // Modulo for avoiding integer overflow

        // Create a dynamic programming table initialized with zeros,
        // `dp[currentTarget]` will represent the number of ways to get sum `currentTarget`
        vector<int> dp(targetSum + 1, 0);

        // Base case: one way to reach sum 0 - by not choosing any dice
        dp[0] = 1;

        // Iterate over the number of dice
        for (int i = 1; i <= numberOfDice; ++i) {
            // Temporary vector to store ways to reach a certain sum with the current dice
            vector<int> newDp(targetSum + 1, 0);

            // Calculate number of ways to get each sum from 1 to `targetSum` with `i` dice
            for (int currentSum = 1; currentSum <= min(targetSum, i * faces); ++currentSum) {
                // Look back at most `faces` steps to find the number of ways to
                // reach the current sum from previous sums using different face values
                for (int faceValue = 1; faceValue <= min(currentSum, faces); ++faceValue) {
                    newDp[currentSum] = (newDp[currentSum] + dp[currentSum - faceValue]) % MODULO;
                }
            }

            // Move the updated values in `newDp` to `dp` for the next iteration
            dp = move(newDp);
        }

        // Final answer is the number of ways to reach `targetSum` with `numberOfDice` dice
        return dp[targetSum];
    }
};

```

Typescript Solution

```typescript
function numRollsToTarget(diceCount: number, facesPerDie: number, targetSum: number): number {
    // Initialize a dynamic programming array to keep track of ways to reach each sum
    const waysToTarget = Array(targetSum + 1).fill(0);
    waysToTarget[0] = 1; // There is 1 way to reach the sum 0 (rolling no dice)

    // Define a modulus constant to prevent integer overflow
    const mod = 1e9 + 7;

    // Iterate over each die
    for (let currentDie = 1; currentDie <= diceCount; ++currentDie) {
        // Temporary array to calculate the current number of ways to achieve each sum
        const tempWays = Array(targetSum + 1).fill(0);

        // Calculate for all possible sums up to the current target
        // constrained by the number of dice thrown so far and the desired target
        for (let currentTarget = 1; currentTarget <= Math.min(currentDie * facesPerDie, targetSum); ++currentTarget) {
            // Check for all the possible outcomes of a single die roll and update
            for (let faceValue = 1; faceValue <= Math.min(currentTarget, facesPerDie); ++faceValue) {
                // Update the ways to achieve currentTarget sum considering the current die roll
                tempWays[currentTarget] = (tempWays[currentTarget] + waysToTarget[currentTarget - faceValue]) % mod;
            }
        }

        // Update the original waysToTarget array with the computed values for the current die roll
        waysToTarget.splice(0, targetSum + 1, ...tempWays);
    }

    // Return the number of ways to reach the desired target sum using all the dice
    return waysToTarget[targetSum];
}

```

Time and Space Complexity

The given Python code is solving the problem of finding the number of distinct ways to roll a set of dice so that the sum of the faces equals the target sum.

Time Complexity:

The time complexity of the code can be analyzed based on the nested loops present in the function:

- There is an outer loop running `n` iterations where `n` is the number of dice.
- The first inner loop iterates up to `min(i * k, target)`. Since `i` goes from `1` to `n`, in the worst case, this loop runs `target` times.
- The second inner loop iterates up to `min(j, k)`. In the worst case, `j` can be `target`, this loop runs `k` times.

In the worst case, the time complexity of the algorithm will be `O(n * target * k)`.

Space Complexity:

Based on the reference answer and the code provided, we are using a rolling array (`f`) to represent the states, which has a size of `target + 1`.

- `f` and `g` arrays both are of size `target + 1`, but they are reused in every iteration instead of creating a 2D array of size `n * target`.

Hence, the space complexity of the algorithm is `O(target)`, as only a single-dimensional array is used which is proportional to the `target` value.