1223. Dice Roll Simulation

Dynamic Programming

Problem Description

Array

Hard

each number i cannot be rolled more than rollMax[i] consecutive times, where rollMax is an array given as input and is defined for each number from 1 to 6 (1-indexed). The task is to calculate the number of distinct sequences that can be obtained with exactly n rolls under this constraint, where n

In this problem, we are given a virtual die that can roll numbers from 1 to 6. However, there is an additional constraint on the die:

is a given integer. Since the number of sequences could be very large, we are required to return the result modulo (10^9 + 7). A sequence is considered distinct from another if at least one element in the sequence is different. This means the order and the number rolled matters for the distinctness of sequences.

Intuition

The solution to this problem uses dynamic programming and depth-first search (DFS) to explore all possible combinations of rolls

The intuition behind using DFS is that, for each roll, we have 6 choices (numbers from 1 to 6), but we need to respect the rollMax

under the given constraints.

constraints for consecutive rolls. We can define a function that takes the current position in the sequence (i), the last number rolled (j), and the count of how many times that number has been rolled consecutively (x).

We recursively call this function until we reach n rolls. While exploring each possibility, we conditionally add to our total count

based on two criteria: 1. If the next number we are trying to roll is different from the last (k != j), then we can reset the consecutive count and continue from the next position.

- 2. If the next number is the same and we have not exceeded the maximum allowed consecutive rolls for this number (x < rollMax[j 1]), then we increment the consecutive count and move to the next position. By using a @cache decorator (assuming from Python's functools), we cache the results of subproblems and avoid recomputation,
- thus saving time and optimizing performance. We repeatedly take the modulus of the result to keep the number within the required limit (10^9 + 7) at each step, as the final number can be quite large.

With this approach, we will be able to cover all paths of sequences without violating the given constraints, and we'll incrementally build up the total number of distinct sequences modulo (10⁹ + 7).

The implementation uses a DFS approach combined with memoization to efficiently explore all valid sequences. The key components of the solution are:

• A recursive function dfs that takes three parameters: i, which represents the current position in the sequence (or the current roll number); j,

• The base case for the dfs function occurs when i equals n, which means that we've successfully generated a sequence of n rolls without

the last number rolled; and x, the current streak of the last number rolled (how many times it has been rolled consecutively).

violating the constraints. Whenever this happens, the function returns 1 as it represents a distinct sequence. • To utilize memoization, the @cache decorator is used. This stores the results of the dfs function calls with particular arguments, so when the

Solution Approach

same state is encountered again, the function can return the cached result instead of recalculating it. • The dfs function iterates over the possible numbers to be rolled next (from 1 to 6), and for each choice, it checks whether the choice is

- different from the last rolled number (k != j). If it is, the function resets the consecutive count (x) to 1 and calls dfs for the next position (i + 1).
- If the next number equals the last rolled number, and the consecutive roll count x has not yet breached the maximum allowed by rollMax, it increments the roll count (x + 1) and recurses to i + 1. • To handle the modulus operation, the sum is taken modulo (10^9 + 7) after each increment.
- no last number rolled (0 in this context is not a valid die number and acts as a placeholder) and no consecutive count. The function then branches out into all possible sequences while adhering to the constraints. The memoization ensures that

Given this recursive structure, the algorithm starts by calling dfs(0, 0, 0), meaning it starts with the first roll (position 0), with

previously encountered states contribute to the solution without further computational overhead. Finally, the answer is a sum of

Let's walk through a smaller example to illustrate the solution approach. Suppose we are given n = 2 rolls, and the rollMax array

is [1, 1, 2, 2, 2], which means we can roll the number 1 and 2 only once consecutively, but we can roll 3, 4, 5, and 6 up to

We start with the call dfs(0, 0, 0). Here i equals 0 because we've made no rolls yet, j equals 0 because there is no last

Example Walkthrough

1. If we roll a 1, we call dfs(1, 1, 1) (one roll made, last number rolled is 1, consecutive count for the number 1 is 1). • On the next roll, we cannot roll a 1 again since rollMax[0] is 1, so we explore other numbers: • dfs(2, 2, 1): since we rolled a 2, a valid sequence [1, 2] is formed. This call returns 1 as it represents a distinct sequence. ■ Calls for numbers 3 to 6 follow the same logic as rolling a 2, each return 1 for their respective distinct sequences: [1, 3], [1, 4],

2. If we roll a 2, similarly to rolling a 1, we follow the same process and find we can't roll a 2 again, but we can roll numbers 1, 3, 4, 5, and 6,

∘ We could roll a 3 again (since rollMax[2] is 2), which is dfs(2, 3, 2), resulting in sequence [3, 3] contributing to 1 sequence.

• We can also roll any other number, which would be similar to the previous cases and yield 5 distinct sequences for each initial 3: [3, 1],

For numbers 3 to 6, since we can roll these numbers up to twice consecutively, we will have a few more possibilities:

which gives us another 5 distinct sequences.

3. If we roll a 3, we call dfs(1, 3, 1):

sequences.

Python

Solution Implementation

from typing import List

[3, 2], [3, 4], [3, 5], [3, 6].

By summing up all the possible distinct sequences:

all branches modulo (10⁹ + 7).

twice consecutively.

Following the same logic for initial rolls of 4, 5, and 6, we end up with 5 distinct sequences for each of their alternative rolls (as

number rolled, and \mathbf{x} equals 0 because we do not have a consecutive count yet.

[1, 5], [1, 6]. Therefore, rolling a 1 first contributes to 5 distinct sequences.

From this starting point, we try all numbers from 1 to 6:

- they can only be followed by 5 other distinct numbers due to the rollMax constraint).
- When starting with a 1 or 2, we get 5 sequences each. • When starting with 3, 4, 5, or 6, we obtain 6 sequences each because we can roll the same number twice or roll a different number (5 other possibilities).

The total number of sequences for n = 2 would be the sum of all these results, which is (2 * 5 + 4 * 6 = 10 + 24 = 34) distinct

This example clearly illustrates how the dfs function explores all possible combinations, while @cache stores the intermediate results, preventing recalculations and improving performance. The final step would involve taking this total (34 for this example) and returning it modulo (10⁹ + 7).

from functools import lru_cache class Solution:

Adding memoization to avoid repeated calculations @lru cache(None) def roll dice(roll count, last roll, consec_roll_count):

Base case: all rolls are done

for die face in range(1, 7):

if die face != last roll:

return num_sequences % (10**9 + 7)

public int dieSimulator(int n, int[] rollMaxArray) {

num ways = solution.dieSimulator(n=2, roll max=[1, 1, 2, 2, 2, 3])

if roll count >= n:

return roll_dice(0, 0, 0)

def dieSimulator(self, n: int, roll max: List[int]) -> int:

Loop through each possible die face (1 through 6)

If the current die face is not the same as the previous roll

num sequences += roll dice(roll count + 1, die face, 1)

Start the recursion with roll_count=0, last_roll=0, and consec_roll_count=0

private Integer[][][] memoization; // A 3D array for memoization to store results of sub-problems

for (int nextNumber = 1: nextNumber <= 6: ++nextNumber) { // Try all dice faces (1 to 6)</pre>

count %= 1000000007; // Modulo to prevent overflow as per problem statement

return (int) count; // Casting long to int before returning as per method signature

// Invoke the dfs function starting with count 0, lastNumber 0 (dummy), and consecCount 0

const MOD: number = 1e9 + 7; // Define the modulo value for operations

// Initialize a dynamic programming (dp) table with a specific structure

// A recursive depth-first search function to explore the solution space

if (rollCount >= n) { // Base case: all dice have been rolled

Array.from({length: 7}, () =>

return dfs(0, 0, 0, rollMax, n);

from typing import List

class Solution:

from functools import lru_cache

@lru cache(None)

};

Array(16).fill(undefined)));

def dieSimulator(self, n: int, roll max: List[int]) -> int:

Adding memoization to avoid repeated calculations

Initialize the number of sequences to 0

Base case: all rolls are done

if roll count >= n:

return 1

return roll_dice(0, 0, 0)

Time and Space Complexity

def roll dice(roll count, last roll, consec_roll_count):

for (let i = 0; $i \ll 15$; i++) { // The 'n' in this context is assumed to be $\ll 15$

// Store the result in the memoization array before returning

memoization[rollCount][lastNumber][currentStreak] = (int) count;

if (nextNumber != lastNumber) { // If the face number is not equal to the last rolled number

count += dfs(rollCount + 1, lastNumber, currentStreak + 1); // Continue the streak

count += dfs(rollCount + 1, nextNumber, 1); // Reset the streak and increment rollCount

} else if (currentStreak < rollMaxArray[lastNumber - 1]) { // If not exceeding max roll constraint</pre>

private int[] rollMaxArray; // This will store the maximum roll constraints for each face

// Main method to simulate the dice roll and return the total number of distinct sequences

return 1 # Initialize the number of sequences to 0 num_sequences = 0

If the current die face is the same and consecutive roll count is less than allowed max elif consec roll count < roll max[last roll - 1]:</pre> # Roll the die without changing the last roll and increment consecutive roll count num_sequences += roll_dice(roll_count + 1, last_roll, consec_roll_count + 1)

Return the result modulo $10^9 + 7$ to keep the number within integer range for large results

Roll the die changing the last roll to the current and reset the consecutive roll count to 1

```
# print(num_ways) # Output depends on the parameters
Java
```

C++

};

TypeScript

dp[i] = [];

return 1;

#include <vector>

#include <cstring> // For memset

return dfs(0, 0, 0);

let dp: number[][][] = [];

dp[i][j] = [];

for (let i = 0; j <= 6; j++) {

#include <functional> // Include for std::function

Example usage:

class Solution {

solution = Solution()

```
this.memoization = new Integer[n][7][16]; // Initialize memoization array with nulls
    this.rollMaxArray = rollMaxArray; // Store the max roll constraints
    return dfs(0, 0, 0); // Start the Depth-First Search process
// Helper method to perform DFS recursively and calculate the count
private int dfs(int rollCount, int lastNumber, int currentStreak) {
    if (rollCount >= memoization.length) { // Base case: If we've made all the rolls
        return 1; // return 1, as this forms one valid sequence
    if (memoization[rollCount][lastNumber][currentStreak] != null) { // If already computed
        return memoization[rollCount][lastNumber][currentStreak]; // return the stored result
    long count = 0; // Initialize the count for the current roll
```

```
class Solution {
public:
   int dieSimulator(int n, std::vector<int>& rollMax) {
       // The state of the dynamic programming (dp) table
       // dp[i][i][x] represents the number of sequences where:
       // i is the total rolls so far,
       // j is the last number rolled (1-6),
       // x is the consecutive times the last number j has been rolled.
       int dp[n][7][16]:
       memset(dp, 0, sizeof dp); // Initialize the dp table with 0
       const int MOD = 1e9 + 7; // Define the modulo value
       // The recursive depth-first search function to explore the solution space
       std::function<int(int, int, int)> dfs = [&](int rollCount, int lastNumber, int consecCount) -> int {
            if (rollCount >= n) { // Base case: all dice have been rolled
                return 1;
           if (dp[rollCount][lastNumber][consecCount]) { // Return memoized result
                return dp[rollCount][lastNumber][consecCount];
            long long totalWays = 0; // Use long long to prevent overflow before taking mod
            for (int face = 1; face <= 6; ++face) {</pre>
                if (face != lastNumber) { // If the current face is different from the last number rolled
                    totalWays += dfs(rollCount + 1, face, 1); // Start count new number with 1
               \} else if (consecCount < rollMax[lastNumber - 1]) { // If it's the same and under the rollMax limit
                    totalWays += dfs(rollCount + 1, lastNumber, consecCount + 1); // Continue sequence
            totalWays %= MOD; // Take modulo to prevent overflow
            return dp[rollCount][lastNumber][consecCount] = totalWays; // Memoize and return
       };
```

if (dp[rollCount][lastNumber][consecutiveCount] !== undefined) { // Return memoized result return dp[rollCount][lastNumber][consecutiveCount]; let totalWays: number = 0; // Variable to keep track of total ways for (let face = 1; face <= 6; ++face) {</pre> if (face !== lastNumber) { // If the current face is different from the last number rolled totalWays = (totalWays + dfs(rollCount + 1, face, 1, rollMax, n)) % MOD; // Start count new number with 1 $\}$ else if (consecutiveCount < rollMax[lastNumber - 1]) { // If it's the same and under the rollMax limit totalWays = (totalWays + dfs(rollCount + 1, lastNumber, consecutiveCount + 1, rollMax, n)) % MOD; // Continue sequence return dp[rollCount][lastNumber][consecutiveCount] = totalWays; // Memoize and return the result **}**; // This is the main simulation function that initiates the dice roll simulation const dieSimulator = (n: number, rollMax: number[]): number => { // Clear existing dp table dp = Array.from({length: n}, () =>

const dfs = (rollCount: number, lastNumber: number, consecutiveCount: number, rollMax: number[], n: number): number => {

```
num sequences = 0
# Loop through each possible die face (1 through 6)
for die face in range(1, 7):
   # If the current die face is not the same as the previous roll
   if die face != last roll:
       # Roll the die changing the last roll to the current and reset the consecutive roll count to 1
       num sequences += roll dice(roll count + 1, die face, 1)
   # If the current die face is the same and consecutive roll count is less than allowed max
```

Start the recursion with roll_count=0, last_roll=0, and consec_roll_count=0

elif consec roll count < roll max[last roll - 1]:</pre>

// Invoke the dfs function starting with count 0, lastNumber 0 (dummy), and consecutiveCount 0

The given code defines a function dieSimulator which uses depth-first search (DFS) with memoization to count the number of distinct die sequences that can be rolled.

print(num_ways) # Output depends on the parameters

return num_sequences % (10**9 + 7)

num ways = solution.dieSimulator(n=2, roll max=[1, 1, 2, 2, 2, 3])

The time complexity of the algorithm is O(n * 6 * max(rollMax)). n is the number of dice rolls.

Roll the die without changing the last roll and increment consecutive roll count

num_sequences += roll_dice(roll_count + 1, last_roll, consec_roll_count + 1)

Return the result modulo 10^9 + 7 to keep the number within integer range for large results

• max(rollMax) represents the maximum constraint for consecutive rolls of the same face. For each state in our DFS, we have at most 6 choices of the die face to consider. Since we also consider the number of

Time Complexity

Example usage:

solution = Solution()

consecutive rolls of the same face (bounded by max(rollMax)), the time complexity includes this factor. DFS will run for each roll, so we multiply by n. Memoization ensures each state is calculated once, thus reducing the time complexity.

• 6 represents each possible die face value.

- **Space Complexity** The space complexity of the algorithm is O(n * max(rollMax) * 6).
- The cache potentially stores results for every combination of: • The number of dice left to roll (n). The current face being rolled (6 faces).

 The number of times the current face has been rolled consecutively, which is at most max(rollMax). Each state requires a constant amount of space, and since the space is used to store the combination of states mentioned above, the space complexity is a product of these factors.