1866. Number of Ways to Rearrange Sticks With K Sticks Visible

Hard Math Dynamic Programming Combinatorics Leetcode Link

Problem Description

In this problem, we are given an array of uniquely-sized sticks, with lengths as integers from `1` to `n` inclusive. We need to determine the total number of ways we can arrange these sticks so that exactly `k` sticks are visible from the left. A stick is considered visible from the left if there are no longer sticks to the left of it.

For instance, let's consider `n = 5` and `k = 3`. If the sticks are arranged as `[1, 3, 2, 5, 4]`, the sticks with lengths `1`, `3`, and `5` are visible from the left because every stick to the left is shorter.

Our task is to calculate the number of such possible arrangements. Since the answer can be very large, we are asked to return it modulo `10^9 + 7`.

Intuition

Let's develop the intuition behind the solution. The core of the problem is combinational, determining the number of ways we can arrange the `n` sticks such that only `k` sticks are visible from the left. It helps to approach this task by thinking about dynamic programming - that is, solving smaller subproblems and using their solutions to build up to the solution of our larger problem.

We can define a state `f(i, j)` which represents the number of arrangements of `i` sticks such that `j` sticks are visible from the left. We can initialize `f(0, 0) = 1` as the base case, representing the number of ways to arrange zero sticks with zero visible sticks.

Now consider incrementally adding the `(i+1)`th stick. The new stick can be placed in any position from `1` to `i+1` without affecting the visibility of the already visible sticks.

1. If the `(i+1)`th stick is the tallest among the ones placed so far, then it must be visible, and hence it can only be placed in one position (at the beginning), contributing to `f(i+1, j+1)` arrangements based on `f(i, j)`.
2. On the other hand, if the `(i+1)`th stick is not the tallest (hence, not visible from the left), it has `i` positions to be placed in, and it contributes to `f(i+1, j)` arrangements based on `f(i, j)` arrangements with `i` times the possibility.

The solution involves iterating through all `n` sticks and `k` visible sticks, updating our dynamic array `f` in a bottom-up manner. Finally, the answer to the original problem will be the value of `f(n, k)`.

The code provided initializes a list `f` with `k+1` elements, where `f[j]` represents the current state for exactly `j` sticks visible from the left. Each iteration of the two nested loops updates `f` based on the two cases above, using modulo `10^9 + 7` to keep track of the answer in the bounds of the acceptable result.

This bottom-up tabulation approach makes sure that we build up the solution to our problem without redundant calculations, effectively covering the solution space in `O(n*k)` time complexity.

Solution Approach

The implementation of the solution applies a dynamic programming approach utilizing a one-dimensional array `f`, where `f[j]` keeps track of the number of ways to arrange `i` sticks with exactly `j` of them visible from the left, incrementally built up for each stick added. The approach utilizes the concept of tabulation (bottom-up dynamic programming).

Let's breakdown the algorithm:

1. **Initialization**: The array `f` is initialized with the size `k+1`, where `f[0] = 1` and the rest of the elements are 0. This represents the fact that we have one way to arrange zero sticks with zero visible sticks which is the base case.
2. **Outer Loop - Sticks Iteration**: The first loop iterates through `i` which represents the stick number being considered, ranging from `1` to `n` inclusive.
3. **Inner Loop - Visibility Iteration**: For each stick `i`, the second loop iterates backwards through `j` from `k` down to `1`, which represents the visibility count (the number of visible sticks).
4. **DP State Transition**:
   - The DP state transition is based on two cases: a) When a new stick is placed as the leftmost stick. In this scenario, it is guaranteed to be visible (since it would be the tallest so far), so we move from state `f[j-1]` to `f[j]`. b) When a new stick is placed in any of the other positions, it is not visible, so we transition between states without changing the visibility count, simply adding `f[j]` multiplied by `(i - 1)` – that is, the previous number of arrangements times the number of positions the new stick can be placed without being visible.
5. **Modulo Operation**: After considering both cases, we perform a modulo operation to ensure the result remains within the range specified by the problem (`10^9 + 7`).
6. **Setting `f[0]`**: `f[0]` is reset to `0` at each iteration since there's no way to arrange more than zero sticks without having at least one visible.

By iteratively updating the array `f` using the transition rules described, the algorithm builds up the number of possible arrangements with the desired number of visible sticks. The complexity of this algorithm is `O(n*k)` because it iterates through `n` sticks and for each one, it calculates `k` visibility counts.

Here's a visualization of the state transition:

```
for i from 1 to n:
    for j from k to 1:
        f[j] = (f[j] * (i - 1) + f[j - 1]) % mod
    f[0] = 0
```

In the end, after `n` iterations, `f[k]` will hold the final answer, which is the number of ways we can arrange `n` sticks so that exactly `k` of them are visible from the left.

Understanding this implementation requires recognizing the dynamic programming pattern where the current state depends on the previous state in a very specific way, as defined by our state transition rules. It uses iteration rather than recursion and memoization, which is often more efficient and is particularly suited for this kind of problem where we have to work through a two-dimensional problem space with dependant states.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider `n = 3` (we have sticks of lengths `1`, `2`, and `3`) and `k = 2` (we want to find the number of ways to arrange these sticks so that exactly `2` of them are visible from the left).

- Initialize the array `f` with `k+1 = 3` elements: `f[0]`, `f[1]`, `f[2]`, with `f[0] = 1` and the rest as `0`. Now `f` looks like this: `[1, 0, 0]`.
- Begin the outer loop (`i` from `1` to `n`):
  - For `i = 1`, there's only one stick, so it's always visible. Update `f[1]` to `1` (since `f[j-1]` where `j` is `1` is `f[0]`, which is `1`). Now `f` is `[0, 1, 0]`.
  - For `i = 2`, we have two sticks (`1`, and `2`).
    - Stick `2` can go in position `1` making `f[2] = f[1]`, which adds `1` way.
    - Stick `2` can also go in the second position, behind stick `1`, keeping it invisible and preserving the previous visibility count (`f[1]` gets multiplied by `1`, the number of sticks before it). Now `f` is `[0, 1, 1]`.
  - For `i = 3`, we have three sticks (`1`, `2`, and `3`).
    - Stick `3` can be the leftmost which makes it visible. This updates `f[2]` to now be `f[2] + f[1]` because `f[1]` is the number of ways with `1` visible which now move to `2` visible. Also `f[2]` gets multiplied by `2` because stick `3` can also be one of the two non-visible positions in any arrangement where two sticks are already visible. After this operation, `f` becomes `[0, 2, 3]`.

After these iterations are done, `f` contains the number of arrangements with `j` visible sticks, where `j` ranges from `0` to `k`.

For our example, when `i = n`, the final array `f` looks like this: `[0, 2, 3]`. Therefore, the number of ways we can arrange `3` sticks so that exactly `2` are visible from the left is `3`, which is the value of `f[2]`.

The dynamic programming approach successfully breaks this problem down into a series of overlapping subproblems that build upon one another to find the final solution in an efficient manner.

Python Solution

```python
class Solution:
    def rearrangeSticks(self, n: int, k: int) -> int:
        # Define the modulus for large number handling according to the problem statement
        MOD = 10**9 + 7

        # dp array to store the intermediate results, dp[j] represents the number of ways
        # to arrange j sticks out of a certain amount such that k sticks are visible
        dp = [1] + [0] * k

        # Iterate over each stick
        for i in range(1, n + 1):
            # Iterate over the number of visible sticks in reverse
            # so that we do not overwrite the data we need to access
            for j in range(k, 0, -1):
                # Update the dp array with the formula:
                # dp[j] = dp[j] * (i - 1) + dp[j - 1]
                # The dp[j] * (i - 1) part counts the placements where the new stick is not visible
                # The dp[j - 1] part counts the placements where the new stick is visible
                dp[j] = (dp[j] * (i - 1) + dp[j - 1]) % MOD

            # The first position in dp should always stay 0 because we cannot have
            # 0 visible sticks if we have at least one stick.
            dp[0] = 0

        # Return the number of ways to arrange n sticks so that exactly k sticks are visible
        return dp[k]

```

Java Solution

```java
class Solution {

    public int rearrangeSticks(int n, int k) {
        // Define the modulus value for the large numbers as per the problem statement
        final int MOD = 1000000007;

        // Create an array to keep track of the subproblems
        int[] dp = new int[k + 1];

        // Base case, when there's no stick visible (k=0), there's one way to arrange
        dp[0] = 1;

        // Iterate through all sticks
        for (int i = 1; i <= n; ++i) {
            // Iterate through the number of visible sticks from k to 1
            for (int j = k; j > 0; --j) {
                // Recurrence relation:
                // The number of ways to arrange i sticks with j visible is the sum of:
                // 1. The ways to arrange (i-1) sticks with j visible, since the new stick is not visible when added.
                //    However, every configuration is multiplied by (i-1) as the new stick can be placed
                //    in any of (i-1) positions for each configuration without increasing the number of visible sticks.
                // 2. The ways to arrange (i-1) sticks with (j-1) visible, since the new stick will be the tallest and visible.
                dp[j] = (int) ((dp[j] * (long) (i - 1) + dp[j - 1]) % MOD);
            }
            // When there are more sticks than the visible count, the base case is always 0
            dp[0] = 0;
        }

        // Return the number of ways to arrange n sticks such that k are visible
        return dp[k];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int rearrangeSticks(int n, int k) {
        const int MOD = 1000000007; // Define the modulus as a constant for easy reference
        int dp[k + 1];   // Initialize a dynamic programming array to store the number of ways
        memset(dp, 0, sizeof(dp)); // Set all the elements of dp array to 0 initially
        dp[0] = 1; // Base case: one way to arrange 0 sticks with 0 visible sticks

        // Loop over each stick
        for (int i = 1; i <= n; ++i) {
            // Loop over the number of visible sticks in reverse
            // so that we do not overwrite the values that we still need
            for (int j = k; j > 0; --j) {
                // The recurrence relation:
                // dp[j] represents the number of ways to arrange i sticks with j visible
                // Update dp[j] using the number of ways to arrange i-1 sticks with
                // (j-1) visible (by placing the tallest stick at the end) and i-1 sticks
                // with j visible sticks (by placing any of the other i-1 sticks at the end).
                dp[j] = (dp[j - 1] + dp[j] * static_cast<long long>(i - 1)) % MOD;
            }
            // There is no way to arrange i sticks with 0 visible
            dp[0] = 0;
        }

        // Return the number of ways to arrange n sticks with k visible
        return dp[k];
    }
};

```

Typescript Solution

```typescript
const MOD = 1000000007; // Define the modulus as a constant for easy reference

function rearrangeSticks(n: number, k: number): number {
    let dp: number[] = new Array(k + 1).fill(0); // Initialize a dynamic programming array with k+1 elements set to 0
    dp[0] = 1; // Base case: one way to arrange 0 sticks with 0 visible

    // Loop over each stick
    for (let i = 1; i <= n; ++i) {
        // Loop over the number of visible sticks in reverse
        // so that we do not overwrite the values that we still need
        for (let j = k; j > 0; --j) {
            // The recurrence relation:
            // dp[j] represents the number of ways to arrange i sticks with j visible
            // Update dp[j] using the number of ways to arrange (i-1) sticks with (j-1) visible
            // (by placing the tallest stick at the end) and the number of ways to arrange i-1 sticks
            // with j visible sticks (by placing any of the other i-1 sticks at the end).
            dp[j] = (dp[j - 1] + BigInt(dp[j]) * BigInt(i - 1)) % BigInt(MOD);
        }
        // There is no way to arrange i sticks with 0 visible
        dp[0] = 0;
    }

    // Return the number of ways to arrange n sticks with k visible
    // Convert BigInt back to number for the result, ensuring it fits into the JavaScript number precision
    return Number(dp[k]);
}

// The `rearrangeSticks` function can now be used globally

```

Time and Space Complexity

The algorithm has a nested for loop where the outer loop runs for `n` times corresponding to the number of sticks, and the inner loop runs for `k` times corresponding to the number of visible sticks from the left. Inside the inner loop, there are constant time operations performed, including multiplication, addition, and modulo operation. Therefore, the overall time complexity can be represented as `O(n * k)`.

As for the space complexity, there is a one-dimensional list `f` of size `k + 1` that is used for dynamic programming to store intermediate results. The space complexity is determined by the size of this list, which does not grow with `n`, hence the space complexity is `O(k)`.