634. Find the Derangement of An Array

Medium Math Dynamic Programming Leetcode Link

Problem Description

In the classical branch of mathematics known as combinatorics, a derangement is a specific type of permutation. In a derangement, the array elements are permuted in such a way that no element stays in its original position. The given problem asks us to find out how many derangements are possible for a set of `n` unique elements initially in ascending order from `1` to `n`. The result should be modulo `10^9 + 7` to avoid handling very large numbers. This means we must do the arithmetic operations and return the final result of the derangements count modulo `10^9 + 7`.

Intuition

Calculating the number of derangements, which are also known as !n (or subfactorials), for a given `n`, can be approached using dynamic programming. The mathematical property governing derangements is usually expressed by the recursive formula:

!n = (n - 1) * (!(n - 1) + !(n - 2))

This formula tells us that the number of derangements for `n` elements can be found by multiplying `n - 1` with the sum of derangements for `n - 1` elements and `n - 2` elements.

To solve this using dynamic programming, we can start with a base case and build up to the solution for `n`. For n=1, there are no derangements since the single element will always be in its place. For n=2, there is exactly 1 derangement (2,1). Using these starting points, we can calculate the number of derangements for each successive value of `n` until we reach our desired number.

In the provided solution, the iterative method is used. Two variables a and b are used. They represent the count of derangements for `i-2` and `i-1` respectively. Initially, `a` is set to 1 and `b` to 0, representing the base cases for n=1 (no derangements) and n=2 (one derangement).

As we iterate from 2 up to `n`:

- The new value of `b` is the count of derangements for the current `i`, which uses the recursion relation.
- The current value of `a` is updated to the previous `b`, preparing for the next iteration.

Finally, `b` gives us the count of derangements for `n` modulo `10^9 + 7`.

Solution Approach

The solution uses dynamic programming to efficiently compute the number of derangements for a given number `n`. Here is a breakdown of the implementation step by step:

1. We define the modulo `mod = 10**9 + 7` to ensure all operations are performed under this modulo to prevent integer overflow.

2. We start with two variables `a` and `b` which will hold the counts of derangements for `i-2` and `i-1` respectively, as per the recursive formula. Initially `a` equals 1 (for `i=0` since there are no derangements) and `b` equals 0 (for `i=1` which would be 1 derangement, but the array is 0-indexed, so `b` is initialized as 0 to correct for this offset).

3. A for-loop is started, ranging from `i = 2` to `i = n + 1`. This loop will iterate through each number `i` and calculate the number of derangements based on the previously calculated values.

4. Inside the loop, first, the current value for `a` (which is the derangement count for `i-2`) is assigned to `b`. This is because after this iteration, `b` (the derangement count for `i-1`) needs to be updated for the next iteration.

5. The new value of `b` is then calculated using the derangement formula: `(i - 1) * (a + b) % mod`. This represents `!i = (i - 1) * (![i - 1] + ![i - 2])`, where `!i` is the number of derangements for `i`.

6. The modulo operation is applied to ensure the result stays within the bounds of integer values defined by `10^9 + 7`.

7. The loop continues until it calculates the derangement for `i = n`.

8. Finally, the function returns the value of `b`, which, at the end of the loop, holds the number of derangements for `n`, modulo `10^9 + 7`.

No additional data structures are needed for this computation other than the two variables `a` and `b`, because the state of the current calculation only depends on the previous two states. This solution is space-efficient, utilizing `O(1)` space, and time-efficient with a complexity of `O(n)` as it requires a single pass from `2` to `n`.

Example Walkthrough

Let's consider a small example to illustrate the solution approach using `n=4`. We want to find out how many derangements are possible for an array of `[1, 2, 3, 4]`.

1. Initialize `mod` as `10^9 + 7` for modulo operations.

2. Start with `a = 1` and `b = 0`. This is because we're using 0-based indexing (for `i=2`, there is 1 derangement `(2,1)`, hence `a = 1`; and for `i=1`, there are no derangements, hence `b = 0`).

3. Begin iterating from `i = 2` to `i = 4`.
   - For `i = 2`, we set the previous `b` to `a`, so `a = 0` (from the previous `b` value).
   - Now, calculate the new `b` as `(i - 1) * (a + b) % mod`, which is `(2 - 1) * (0 + 0) % mod = 1`.
4. Move to `i = 3`.
   - Set `a = 1` (old value of `b`).
   - Now, calculate `b` as `(i - 1) * (a + b) % mod`, which is `(3 - 1) * (1 + 1) % mod = 4`. There are four derangements for `[1, 2, 3]`: `(2, 3, 1)`, `(3, 1, 2)`, `(2, 1, 3)`, and `(3, 2, 1)`.
5. For `i = 4`, the process is similar.
   - Set `a = 4` (old value of `b`).
   - Calculate `b` as `(i - 1) * (a + b) % mod`, which is `(4 - 1) * (4 + 1) % mod = 15`.
6. After the loop ends, `b` holds the number of derangements for `n = 4`, which is 15, modulo `10^9 + 7`.

Thus, the final answer for the number of derangements when `n=4` is 15, under the modulo `10^9 + 7`. The solution uses only two variables and one loop, making it very space and time efficient.

Python Solution

```python
class Solution:
    def findDerangement(self, n: int) -> int:
        # Initialize a modulo constant as per the problem statement
        mod = 10**9 + 7

        # Initialize variables to store results of subproblems.
        # 'prev_two' holds the (i-2)th derangement number, 'prev_one' for (i-1)th.
        prev_two, prev_one = 1, 0

        # Loop through numbers 2 to n to calculate the derangement of n using the
        # recursive relation: D(i) = (i - 1) * (D(i - 1) + D(i - 2))
        for i in range(2, n + 1):
            # 'current' holds the current derangement number being calculated
            current = ((i - 1) * (prev_two + prev_one)) % mod

            # Update 'prev_two' and 'prev_one' for the next iteration.
            # 'prev_two' becomes 'prev_one', and 'prev_one' becomes 'current'.
            prev_two, prev_one = prev_one, current

        # 'prev_one' holds the derangement of n, return this value.
        return prev_one

```

Java Solution

```java
class Solution {
    public int findDerangement(int n) {
        final int MOD = (int) 1e9 + 7; // Define the modulo value as a constant
        // Initialize the variables to store previous results
        long previous = 1; // this will eventually hold the derangement count for (i-2)
        long current = 0; // this will eventually hold the derangement count for (i-1)

        // Iterate from 2 to n to build up the derangement counts
        for (int i = 2; i <= n; ++i) {
            // Calculate the derangement count for the current value of i using the recurrence relation
            // D(n) = (n-1) * (D(n-1) + D(n-2))
            long next = (i - 1) * (previous + current) % MOD;
            // Update the previous values for the next iteration
            previous = current;
            current = next;
        }

        // Cast the result to int and return it as the final derangement count for n
        return (int) current;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to find the number of derangements (permutations where no element appears in its original position) for a given numbe
    int findDerangement(int n) {
        long long prevPrev = 1; // Represents the derangement count for n-2, initialized for n = 0 base case
        long long prev = 0;     // Represents the derangement count for n-1, initialized for n = 1 base case
        const int MOD = 1e9 + 7; // Modulo value to prevent integer overflow

        // Loop to calculate the derangements for all numbers from 2 to n
        for (int i = 2; i <= n; ++i) {
            // Calculate the current derangement count using the recursive formula:
            // D(n) = (n - 1) * (D(n - 1) + D(n - 2))
            long long current = (i - 1) * (prevPrev + prev) % MOD;

            // Shift derangement counts for next iteration
            prevPrev = prev;
            prev = current;
        }

        // The last computed value is the derangement count for n
        return prev;
    }
};

```

Typescript Solution

```typescript
// Modulo value to prevent integer overflow
const MOD = 1e9 + 7;

// Function to find the number of derangements (permutations where no element appears in its original position) for a given number n
function findDerangement(n: number): number {
    let prevPrev: bigint = BigInt(1); // Represents the derangement count for n-2, initialized for n = 0 base case
    let prev: bigint = BigInt(0);     // Represents the derangement count for n-1, initialized for n = 1 base case

    // Loop to calculate the derangements for all numbers from 2 to n
    for (let i = 2; i <= n; ++i) {
        // Calculate the current derangement count using the recursive formula:
        // D(n) = (n - 1) * (D(n - 1) + D(n - 2))
        let current: bigint = BigInt(i - 1) * (prevPrev + prev) % BigInt(MOD);

        // Shift derangement counts for next iteration
        prevPrev = prev;
        prev = current;
    }

    // The last computed value is the derangement count for n
    // Conversion of bigint to number while ensuring it fits within the safe integer range for JavaScript
    return Number(prev % BigInt(MOD));
}

```

Time and Space Complexity

The code provided calculates the number of derangements of n items, which is a permutation where no element appears in its original position.

Time Complexity

The time complexity of the given function is `O(n)`. This is determined by the for loop, which iterates from `2` to `n+1`, making exactly `n-1` iterations. Each iteration involves a constant amount of work, consisting of basic arithmetic operations and a modulo operation, all of which have a time complexity of `O(1)`. Hence, the overall time complexity is the product of the number of iterations `(n-1)` and the time complexity of the work done per iteration `O(1)`, leading to an `O(n)` time complexity.

Space Complexity

The space complexity of the function is `O(1)` because it uses a fixed amount of extra space. Regardless of the input size `n`, it only maintains a constant number of variables (`a`, `b`, `mod`, and `i`) that do not depend on the size of the input. This implies that the memory used does not scale with `n`, making the space complexity constant.