1359. Count All Valid Pickup and Delivery Options

Hard · Math · Dynamic Programming · Combinatorics · Leetcode Link

Problem Description

The given problem asks for the number of valid sequences of pickup and delivery operations for `n` orders. Each order consists of two operations: a pickup (`P`) and a delivery (`D`). For each order, the delivery must always occur after the corresponding pickup. The goal is to determine how many different sequences there could be for the `n` orders, considering this constraint. Finally, because the answer could be very large, the problem instructs us to return the result modulo `10^9 + 7`, which is a common practice to keep the numbers within the limits of integer storage in most programming languages.

To illustrate, let's consider there are 2 orders. The valid sequences we can arrange them in, while keeping pickups before deliveries, are:

- `P1 P2 D1 D2`
- `P1 P2 D2 D1`
- `P1 D1 P2 D2`
- `P2 P1 D1 D2`
- `P2 P1 D2 D1`
- `P2 D2 P1 D1`

Notice that as n increases, the number of sequences grows very fast and calculating them one by one might not be feasible, thus a more mathematical or algorithmic approach is required to solve the problem efficiently.

Intuition

The intuition behind the solution comes from considering the possibilities of inserting the `i-th` pickup and delivery operations into a sequence that has already been arranged for `i-1` orders.

At any stage, let's say we have arranged `i-1` orders. Now, we intend to insert the `i-th` order's pickup and delivery operations into this sequence. For the `i-th` pickup, there are `2 * (i - 1) + 1` different places to put the pickup operation. This is because for each of the previous `i-1` orders, there are two operations (a pickup and a delivery), plus one additional place for the start of the sequence. After we've put in the pickup, we need to insert the delivery. Since delivery must come after pickup, there are only `i` spots where it could go—after each of the delivery and pickup operations that have already been placed.

Mathematically, this means that for the `i-th` order, there are `i` spaces for the delivery and `2 * (i - 1) + 1` spaces for the pickup, giving a total of `i * (2 * (i - 1) + 1)` positions to insert both P and D.

So, the total number of arrangements for all `n` orders can be calculated by multiplying these numbers for all `i` from 1 to `n`. The provided solution uses a for loop to carry out this computation, while also applying the modulo operator at each step to prevent the number from getting too large.

Solution Approach

The solution provided is a direct application of the mathematical intuition discussed earlier. The solution uses no special data structures—the only structure used is a simple loop, and the approach is entirely iterative and mathematical. Here's the breakdown:

The function `countOrders` calculates the number of sequences dynamically by using a single integer `f` to keep track of the result, which is initialized to 1 (since there's only one way to arrange one order).

Within the for loop, for each `i` from 2 to `n` inclusive, we calculate the number of possible ways to insert the `i-th` order into the sequences created for `i-1` orders. As explained before, for pickup there are `2 * (i - 1) + 1` options, and for delivery, there are `i` options afterwards. We then multiply `f` by these numbers to update the total possible arrangements:

```
f = (f * i * (2 * i - 1)) % mod
```

The `mod` variable is set to `10**9 + 7`, which is a large prime number, and is used here to perform modulo operation to ensure the result stays within integer limits and to avoid overflow.

After the loop ends, `f` will contain the number of valid sequences modulo `10**9 + 7`, and this is exactly what the function returns.

The entire implementation is quite efficient due to its simplicity and directness; it runs in `O(n)` time complexity because it processes each of the `n` orders exactly once. Since it only uses a few simple integer variables, its space complexity is `O(1)`.

```python
class Solution:
    def countOrders(self, n: int) -> int:
        mod = 10**9 + 7
        f = 1
        for i in range(2, n + 1):
            f = (f * i * (2 * i - 1)) % mod
        return f
```

Each iteration computes the possibilities for insertion without building the sequences explicitly, resulting in an elegant and highly efficient solution that scales well with the value of `n`.

Example Walkthrough

Let's walk through an example using the solution approach for `n = 3`, meaning we have three orders to sequence.

From the initial explanation, we know we have these constraints:

- For each order, the pickup must occur before the delivery.
- We need to consider all valid combinations of pick up and delivery operations for the given orders.

To begin with the first order (when `n = 1`), we have no choice other than sequencing it as `P1 D1`. This is our base case and, intuitively, there's only 1 way to arrange one order.

Now, for the second order (when `n = 2`), we have to insert `P2` and `D2` into the existing sequence `P1 D1`.

- We have three possible spots to insert `P2`: before `P1`, between `P1` and `D1`, or after `D1`.
- Once `P2` is inserted, we can insert `D2` in two places: anywhere after `P2`.

Thus, we have `3 (spots for P2) * 2 (spots for D2) = 6` different sequences for two orders.

Now we go to three orders (our example case). Suppose we have already arranged the first two orders (2 pickups and 2 deliveries), making a sequence with four operations. Now we consider inserting `P3` and `D3`.

- We have five spots to insert `P3`: before the first operation, between any of the two existing operations, or after the last operation in our sequence.
- After placing `P3`, we have three spots to insert `D3`, all of them coming after `P3`.

The total possibilities for `n = 3` now are `5 (spots for P3) * 3 (spots for D3) = 15` which we would multiply with the previous possibilities for `n = 2` (which were 6).

However, we've missed an important step: taking the modulo to avoid large numbers and potential integer overflow issues. This is done after each multiplication to keep the result manageable.

In code, this is how the process would look:

```python
mod = 10**9 + 7
f = 1  # Starting with the case n=1, f starts at 1.

# Now we loop from i=2 to i=3, as we're considering the third order.
for i in range(2, 4):
    # Multiply the current number of ways `f` with the possible spots for P and D
    f = (f * i * (2 * i - 1)) % mod

# By the end of the loop, `f` is our desired output for `n = 3`.
```

If we run this code, after the iteration for `i = 2`, `f` becomes 6, and after the iteration for `i = 3`, `f` becomes: `f = (6 * 3 * (2 * 3 - 1)) % mod = (6 * 3 * 5) % mod = 90 % mod = 90`.

Thus, `f` is 90, indicating that there are 90 possible sequences for arranging the 3 orders while considering the modulo `10**9 + 7`. Since 90 is already less than the modulo, taking the modulo does not change the number. The function would then return 90 as the final result.

Python Solution

```python
class Solution:
    def countOrders(self, n: int) -> int:
        # Define the modulus value to perform computations under modulo 10^9 + 7
        # to avoid integer overflow issues
        MODULO = 10**9 + 7

        # Initialize the factorial value with 1 for the base case
        factorial = 1

        # Compute factorial of all numbers from 2 to n taking into account the number
        # of valid sequences for pick up and delivery orders. For each new order i,
        # there are i pick up options and (2*i - 1) delivery options because once a
        # pick up is made, there will be (i-1) remaining pickups and i deliveries.
        for i in range(2, n + 1):
            factorial = (factorial * i * (2 * i - 1)) % MODULO

        # Return the computed factorial which corresponds to the total count of valid
        # sequences for the given number of orders
        return factorial

```

Java Solution

```java
class Solution {
    // Method to count the number of valid combinations of placing n orders, each with a pickup and delivery operation
    public int countOrders(int n) {
        final int MODULO = (int) 1e9 + 7; // Define the modulo to prevent overflow issues
        long factorial = 1; // Initialize factorial to 1 for the case when n = 1

        // Loop through numbers from 2 to n to calculate the number of valid combinations
        for (int i = 2; i <= n; ++i) {
            // Multiply the current factorial value by 'i' (the number of orders) and
            // '(2 * i - 1)' which represents the number of valid positions the next order
            // (and its corresponding delivery) can be placed in without violating any conditions.
            factorial = factorial * i * (2 * i - 1) % MODULO;
        }

        // Return the final factorial value cast to an integer
        return (int) factorial;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int countOrders(int n) {
        const int MOD = 1e9 + 7; // Constant for the modulo operation
        long long factorial = 1; // Initialize factorial to 1 for the initial case when n=1

        // Loop through the numbers from 2 to n to calculate the number of valid combinations
        for (int i = 2; i <= n; ++i) {
            // For each pair of delivery and pickup, there are 'i' options for pickup
            // and '2*i - 1' options for delivery since one spot is already taken by pickup.
            // The result is the product of the current factorial and the number of permutations
            // for the current pair of delivery and pickup. It is then taken modulo MOD
            // to handle large numbers and prevent integer overflow.
            factorial = factorial * i * (2 * i - 1) % MOD;
        }

        // Return the total number of valid combinations modulo MOD
        return factorial;
    }
};

```

Typescript Solution

```typescript
// Initialize the constant for the modulo operation
const MOD = 1e9 + 7;

// Function to count all valid combinations of order delivery and pickup
function countOrders(n: number): number {
    // Initialize factorial as a bigint to handle large numbers
    // bigint is used here because JavaScript can't safely represent integers larger
    // than the MAX_SAFE_INTEGER (2^53 - 1)
    let factorial: bigint = BigInt(1);

    // Loop through the numbers from 2 to n to calculate the number of valid combinations
    for (let i = 2; i <= n; i++) {
        // For each pair of delivery and pickup, there are 'i' options for pickup
        // and '2*i - 1' options for delivery since one spot is already taken by pickup.
        // The result is the product of the current factorial and the number of permutations
        // for the current pair of delivery and pickup. The modulus operator is applied to
        // each multiplication step to handle large numbers and prevent integer overflow.
        factorial = (factorial * BigInt(i) * BigInt(2 * i - 1)) % BigInt(MOD);
    }

    // Return the total number of valid combinations modulo MOD as a number
    // by converting the bigint result to a number (this is safe because the modulus ensures
    // the result is less than MOD, which is within the safe integer range for JavaScript/TypeScript numbers)
    return Number(factorial);
}

// Example usage
// const numberOfOrders = countOrders(3); // Calculating the number of valid combinations for n = 3

```

Time and Space Complexity

Time Complexity: The time complexity of the provided code is `O(n)` because there is a single loop that iterates from `2` to `n`. In each iteration, it performs a constant number of mathematical operations which do not depend on `n`, thus the time complexity is linear with respect to the input size `n`.

Space Complexity: The space complexity of the code is `O(1)` as it uses a fixed amount of additional space. There are only a few variables (`mod`, `f`, and `i`) that do not scale with the input size `n`, hence the space used is constant.