629. K Inverse Pairs Array

Hard · Dynamic Programming

Problem Description

The problem deals with counting the number of different arrays composed of numbers from `1` to `n` in which there are exactly `k` inverse pairs. An inverse pair in an array is a pair of indices `[i, j]` such that `i < j` and `nums[i] > nums[j]`. You are asked to return this number modulo `10^9 + 7` to keep the number manageable and address potential integer overflow issues.

Intuition

The core idea behind the solution is dynamic programming. The challenge is to find a way to efficiently calculate the number of arrays with exactly `k` inverse pairs for any `n`. We maintain an array `f`, where `f[j]` represents the number of arrays that consist of numbers from `1` to the current `i` and have exactly `j` inverse pairs.

To build up the solution, we start from the simplest array `i = 1` (which can only have zero inverse pairs, as it is a single element array) and iteratively calculate up to the size `n` while keeping track of the count of inverse pairs.

For a given `i` (current array size), the number of ways to form `j` inverse pairs is incrementally built upon the number of ways to form fewer inverse pairs with a smaller array size (`i-1`). We achieve this by considering the placement of the largest element `i`. It can be inserted in `i` different positions, each giving a different number of additional inverse pairs.

The secondary array `s`, which is a prefix sum array of `f`, helps in calculating the running sum in order to get the number of ways to form `j` inverse pairs quickly without iterating through each possibility, which would be inefficient.

In summary, we use dynamic programming to build the answer iteratively, utilizing prefix sums to efficiently calculate the dynamic programming states for each array size and inverse pair count.

Solution Approach

The implementation of the solution utilizes dynamic programming, where we define `f[j]` as the number of arrays with elements from `1` to `i` that have `j` inverse pairs.

To calculate `f[j]` for bigger arrays, we consider each possible position to insert the largest element `i` in our array (which has an impact on the number of inverse pairs). If we insert `i` at the end, we do not create any new inverse pairs; if we insert it just before the last element, we create one new inverse pair, and so on. If we insert it at the start, we create `i-1` new inverse pairs.

1. The solution starts with initializing `f` with `1` at index `0` and `0` elsewhere, since there's only one way to arrange an array of one element (which cannot create inverse pairs).
2. To simplify and optimize the computation of the cumulative number of ways to create a certain number of inverse pairs when we increase the size of the array, we create a prefix sum array `s`. Prefix sum arrays provide a way to calculate the sum of a range of elements in constant time, after an initial linear time preprocessing.

The key step is the iteration:

- For each size `i` of the array, we fill `f[j]` for `j = 1` to `k`. For a given number of inverse pairs `j`, the number of arrays is the sum of arrays that can be obtained by inserting the new element `i` in all possible positions. This is computed as `(s[j + 1] - s[max(0, j - (i - 1))]) % mod`. The term `s[j + 1] - s[max(0, j - (i - 1))]` gives us the number of arrays, considering all insertion positions of the element `i` that would keep the number of inverse pairs at exactly `j`.
- We update the prefix sum array `s` in terms of `f`, as `s[j] = (s[j - 1] + f[j - 1]) % mod` for `j = 1` to `k + 1`. By keeping the prefix sums up to date, the calculation of subsequent `f[j]` remains efficient.

3. Finally, we return `f[k]` as the answer which represents the number of arrays of size `n` with exactly `k` inverse pairs.

By using dynamic programming and prefix sums, we efficiently calculate the number of arrays with exactly `k` inverse pairs in a way that's computationally feasible for large values of `k` and `n`.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Consider `n = 3` (arrays composed of numbers 1 to 3) and `k = 1` (we want exactly one inverse pair).

We will maintain an array `f` where `f[j]` represents the number of arrays with elements from 1 to the current `i` that have `j` inverse pairs. We also use a prefix sum array `s`.

1. Initialize `f` as `[1, 0, 0, ...]`, because there's only one array `[1]` with zero inverse pairs.
2. For `i = 2`, we need to account for arrays of two elements (numbers `1` and `2`). There are two possible arrays: `[1, 2]` and `[2, 1]`. The first array has 0 inverse pairs, and the second one has 1 inverse pair. So we update `f` to be `[1, 1]` for `f[0]` and `f[1]` respectively. The prefix sums `s` would be `[0, 1, 2]`.
3. For `i = 3`, we need to account for arrays of three elements (numbers `1`, `2`, and `3`). The largest number `3` can be placed in:
   - Position 3 (no new inverse pairs). The prior arrays of two numbers that can be extended are `[1, 2]` and `[2, 1]`.
   - Position 2 (one new inverse pair). The prior arrays of two numbers are `[1, 2]`.
   - Position 1 (two new inverse pairs). However, we want exactly `k = 1` inverse pair, so we don't consider this case for `f[1]`.

The number of arrays with exactly one inverse pair for `i = 3` can be formed by inserting `3` in the first two possible positions for both `[1, 2]` and `[2, 1]`. For `[1, 2]`, we gain one inverse pair if we insert `3` in the second position, and for `[2, 1]`, we don't gain any because we insert `3` at the end.

The updated `f[1]` for `i=3` is the sum of ways we can insert `3` in the array `[1, 2]` with 0 inverse pairs to get 1 inverse pair (1 way), plus the number of ways we can insert it in `[2, 1]` and remain with 1 inverse pair (1 way). So `f[1]` becomes `2`.

We would now generate the new prefix sums `s` for `i = 3` based on the updated `f`.

In the end, you look at `f[1]` for the total number of arrays of size 3 with exactly one inverse pair. The answer for this example is `f[1] = 2`, representing the arrays: `[1, 3, 2]` and `[2, 3, 1]`.

This is how the solution approach can be applied to keep track of the number of arrays of increasing sizes with exactly `k` inverse pairs. For each `i`, we consider how the newest element can be inserted to maintain or reach the desired `k` inverse pairs and update `f` and `s` accordingly.

Solution Implementation

Python

```python
class Solution:
    def kInversePairs(self, n: int, k: int) -> int:
        mod = 10**9 + 7  # Define the modulus value to keep numbers within integer bounds

        # dp table representing the count of inverse pairs for the current number of integers
        dp = [1] + [0] * k

        # Prefix sum array for optimization of the inner loop. The size is k+2 for 1-indexed and ease of access
        prefix_sum = [0] * (k + 2)

        # Iterate through integers from 1 to n
        for current_number in range(1, n + 1):
            # Going through all possible counts of inverse pairs from 1 to k
            for inverse_count in range(1, k + 1):
                # Update the dp table by taking the count from the prefix_sum within the range
                # The range corresponds to the valid inverse pair counts that can be formed with the current number
                dp[inverse_count] = (prefix_sum[inverse_count + 1] -
                                     prefix_sum[max(0, inverse_count - (current_number - 1))]) % mod

            # Updating prefix_sum based on the updated dp table
            for index in range(1, k + 2):
                prefix_sum[index] = (prefix_sum[index - 1] + dp[index - 1]) % mod

        # Returning the number of ways to form k inverse pairs with n integers
        return dp[k]
```

Java

```java
class Solution {

    public int kInversePairs(int n, int k) {
        final int MOD = 1000000007; // Define the modulus value for the operations to prevent overflow

        // Array 'dp' will store the count of arrays that have exactly j inverse pairs
        int[] dp = new int[k + 1];

        // Array 'prefixSum' will be utilized to calculate the total count efficiently
        int[] prefixSum = new int[k + 2];

        dp[0] = 1; // Base case: one way to arrange with 0 inverse pairs (no pairs)
        prefixSum[1] = 1; // Initialize the prefix sum for the base case

        // Iterate from 1 to n to build the answer iteratively
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= k; j++) {
                // Compute the number of arrays that have exactly j inverse pairs
                // by finding the difference of prefix sums, and then taking the modulus
                int val = (prefixSum[j + 1] - prefixSum[Math.max(0, j - i)] + MOD) % MOD;
                dp[j] = val;
            }

            // Update prefix sums for the next iteration
            for (int j = 1; j <= k + 1; j++) {
                prefixSum[j] = (prefixSum[j - 1] + dp[j - 1]) % MOD;
            }
        }

        // Return the count of arrays that have exactly k inverse pairs
        return dp[k];
    }
}
```

C++

```cpp
class Solution {
public:
    int kInversePairs(int n, int k) {
        vector<int> dp(k + 1, 0); // Use vector instead of C array for dynamic array, initialized with 0s.
        vector<int> prefixSums(k + 2, 0); // Prefix sums array for dynamic programming optimization.
        dp[0] = 1; // Base case - zero inverse pairs
        prefixSums[1] = 1; // Base case for prefix sums - one way to have 0 inverse pairs.
        const int MOD = 1e9 + 7; // Define the modulo constant.

        // Iterate over all numbers from 1 to n.
        for (int i = 1; i <= n; ++i) {
            // Compute the number of ways to have j inverse pairs.
            for (int j = 1; j <= k; ++j) {
                // The number of ways to arrange i numbers with j inverse pairs, we update dp for the current number.
                // We use prefix sums to calculate the range sum, which optimizes the computation from O(k) to O(1) time.
                dp[j] = (prefixSums[j + 1] - prefixSums[max(0, j - i)] + MOD) % MOD;
            }
            // Update prefix sums after processing each number i.
            for (int j = 1; j <= k + 1; ++j) {
                prefixSums[j] = (prefixSums[j - 1] + dp[j - 1]) % MOD;
            }
        }
        // Return the number of ways to arrange n numbers with exactly k inverse pairs.
        return dp[k];
    }
};
```

TypeScript

```typescript
function kInversePairs(n: number, k: number): number {
    // Initialize an array to store the number of ways to form arrays
    const numWays: number[] = new Array(k + 1).fill(0);
    numWays[0] = 1; // Base case: 0 inverse pairs

    // Initialize the prefix sums array of numWays for efficient range sum queries
    const prefixSums: number[] = new Array(k + 2).fill(1);
    prefixSums[0] = 0; // Base case

    // Define the modulus value to prevent integer overflow in calculations
    const mod: number = 1e9 + 7;

    // Iterate over the integers from 1 to n
    for (let i = 1; i <= n; i++) {
        // Iterate over the possible number of inverse pairs from 1 to k
        for (let j = 1; j <= k; j++) {
            // Calculate the number of ways to form j inverse pairs with i numbers.
            // This is done by calculating the range sum from the prefix sum array and adjusting for the modulus.
            numWays[j] = (prefixSums[j + 1] - prefixSums[Math.max(0, j - (i - 1))] + mod) % mod;
        }

        // Update the prefix sums array using the new values from numWays
        for (let j = 1; j <= k + 1; j++) {
            prefixSums[j] = (prefixSums[j - 1] + numWays[j - 1]) % mod;
        }
    }

    // Return the total number of ways to form k inverse pairs with n numbers
    return numWays[k];
}
```


```python
class Solution:
    def kInversePairs(self, n: int, k: int) -> int:
        mod = 10**9 + 7  # Define the modulus value to keep numbers within integer bounds

        # dp table representing the count of inverse pairs for the current number of integers
        dp = [1] + [0] * k

        # Prefix sum array for optimization of the inner loop. The size is k+2 for 1-indexed and ease of access
        prefix_sum = [0] * (k + 2)

        # Iterate through integers from 1 to n
        for current_number in range(1, n + 1):
            # Going through all possible counts of inverse pairs from 1 to k
            for inverse_count in range(1, k + 1):
                # Update the dp table by taking the count from the prefix_sum within the range
                # The range corresponds to the valid inverse pair counts that can be formed with the current number
                dp[inverse_count] = (prefix_sum[inverse_count + 1] -
                                     prefix_sum[max(0, inverse_count - (current_number - 1))]) % mod

            # Updating prefix_sum based on the updated dp table
            for index in range(1, k + 2):
                prefix_sum[index] = (prefix_sum[index - 1] + dp[index - 1]) % mod

        # Returning the number of ways to form k inverse pairs with n integers
        return dp[k]
```


Time and Space Complexity

Time Complexity

The time complexity of the code is primarily determined by two nested loops. The outer loop runs for `n` iterations, where `n` represents the number of elements. The inner loop runs up to `k` iterations for every outer loop iteration, where `k` is the number of inverse pairs we want to find. This suggests that the overall time complexity is `O(nk)`, as for each of the `n` elements, we potentially examine every `k`.

Space Complexity

The space complexity of the algorithm is determined by the storage used for the array `f` and the prefix sum array `s`. The array `f` has a size of `k + 1` and the array `s` has a size of `k + 2`. As `k + 2` is the larger of the two, we can consider it for the space complexity estimation. The space complexity is, therefore, `O(k)` because the amount of storage required increases linearly with `k`.

Note that `mod` and the loop counters such as `i` and `j` only use a constant amount of space and don't contribute significantly to the space complexity.