1621. Number of Sets of K Non-Overlapping Line Segments

Medium · Math · Dynamic Programming · Leetcode Link

Problem Description

This problem involves a set of points on a 1-D plane. Each point is designated by a unique integer position from 0 to `n-1`, representing their placement on the line. We are interested in finding the number of ways to create `k` line segments, where each line segment covers at least two points and has endpoints at integral coordinates. It's important to note that these line segments must be non-overlapping, though they can share endpoints. The goal is to calculate the number of distinct configurations you can draw with these requirements.

A crucial detail is that not all points need to be covered by the line segments; thus, some points can remain unused. Also, since the number of ways can be enormous, the solution should be given modulo `10^9 + 7`, a common practice in combinatorics-related problems to keep the numbers manageable.

Intuition

To approach this problem, consider dynamic programming, which is a method used to solve complex problems by breaking them down into simpler subproblems. The idea behind this approach is to create a table, with `f` and `g` to record the number of ways to satisfy the condition up to the `i`-th point with `j` segments used so far.

The 2D arrays `f` and `g` are critical in our dynamic programming approach. `f[i][j]` represents the number of ways we can draw `j` line segments up to the `i`-th point, where the `i`-th point is not shared with the next segment (an endpoint). In contrast, `g[i][j]` counts the number of configurations where the `i`-th point is shared (not the end of the last segment), allowing a segment to extend to include more points.

The key to solving the problem is to correctly update the values in `f` and `g` through iteration. Initially, we set `f[1][0]` to 1, as there is only one way to have 0 segments with 1 point. Then we iterate through all points and all possible numbers of segments, updating `f` based on previous values and `g` based on both its previous value and the value of `f`. If we're allowed to add a new segment (`j` is greater than 0), we can look back at previous points and either start a new segment (`f[i - 1][j - 1]`) or extend an existing one (`g[i - 1][j - 1]`).

In short, `f` and `g` are used to accumulate the number of valid configurations as we progress through the points and segments, with the updates accounting for starting, ending, and extending segments at each step. The final answer consists of the sum of `f[n][k]` and `g[n][k]`, which are the number of configurations after considering all points and exactly `k` segments, modulo `10^9 + 7`.

By setting up these relations and carefully calculating the iterations, we consider all possible segment configurations and meet the constraints of non-overlapping with the option of shared endpoints, efficiently arriving at the correct count.

Solution Approach

The solution involves dynamic programming, which iteratively updates our two tables, `f` and `g`, to account for the various combinations of segments up to each point `i` for `0` to `n-1` and each possible number of segments `j` from `0` to `k`.

We start with the initialization:

```
mod = 10**9 + 7
f = [[0] * (k + 1) for _ in range(n + 1)]
g = [[0] * (k + 1) for _ in range(n + 1)]
f[1][0] = 1
```

Here, `mod` is the number we'll use to perform modulus operations to keep our solution within the required range to avoid integer overflow. The tables `f` and `g` are initialized with zeros, since we start with no configurations, and set `1` for `f[1][0]` since there's exactly `1` way to have `0` segments among `1` point.

Following the initialization, the main iteration begins:

```
for i in range(2, n + 1):
    for j in range(k + 1):
        f[i][j] = (f[i - 1][j] + g[i - 1][j]) % mod
        g[i][j] = g[i - 1][j]
```

For each point `i` and each number of segments `j`, we update `f[i][j]` by the sum of the number of ways we can draw `j` segments up to the previous point without it being an endpoint (`f[i - 1][j]`) and the number of ways with it being a shared point (`g[i - 1][j]`). Then, we carry over the value of `g[i - 1][j]` to `g[i][j]` because we can always extend a line segment to point `i` without adding a new segment.

Additionally, if it's allowed to add a new segment (`j > 0`), we have the following update rules:

```
if j:
    g[i][j] += f[i - 1][j - 1]
    g[i][j] %= mod
    g[i][j] += g[i - 1][j - 1]
    g[i][j] %= mod
```

We increment `g[i][j]` by the number of ways to draw `j-1` segments up to the previous point, and add either a new segment end (`f[i - 1][j - 1]`) or extend an existing segment (`g[i - 1][j - 1]`). This accounts for all the possibilities at point `i` with `j` segments.

Finally, the return statement sums the number of configurations for `n` points and `k` segments from both tables:

```
return (f[-1][-1] + g[-1][-1]) % mod
```

We use `-1` as an index for convenience to refer to the last element of the list, which represents all points and exactly `k` segments. The modulus operation ensures the result complies with the required number range.

Through this iterative and layered approach, the algorithm efficiently calculates the required configurations, respecting the rules of non-overlapping segments which can optionally share endpoints. This method leverages dynamic programming to handle numerous potential combinations, significantly reducing the time complexity compared to naively enumerating all possible segment configurations.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Consider a set with `n = 4` points and we want to find the number of ways to create `k = 1` line segment. The points on the 1-D plane can be represented as [0, 1, 2, 3].

Using dynamic programming, we initialize our tables `f` and `g` like this:

```
mod = 10**9 + 7
n = 4
k = 1
f = [[0] * (k + 1) for _ in range(n + 1)]
g = [[0] * (k + 1) for _ in range(n + 1)]
f[1][0] = 1
```

After initialization, `f` and `g` look like this:

```
f = [[0, 0],
     [1, 0],
     [0, 0],
     [0, 0],
     [0, 0]]

g = [[0, 0],
     [0, 0],
     [0, 0],
     [0, 0],
     [0, 0]]
```

Now we start iterating over each point and potential number of segments:

1. For `i = 2`, `j = 1`:
   - There's no way to have a segment with just point [0] (`f[1][1] = 0`), so `f[2][1]` remains `0`.
   - Since `j > 0`, we add ways from `f[i-1][j-1]` to `g[i][j]`, resulting in `g[2][1]` becoming `1` (one way to start a new segment at [1,2]).

At this point, our tables are:

```
f = [[0, 0],
     [1, 0],
     [0, 0],
     [0, 0],
     [0, 0]]

g = [[0, 0],
     [0, 0],
     [0, 1],
     [0, 0],
     [0, 0]]
```

2. For `i = 3`, `j = 1`:
   - Now `f[3][1]` becomes the sum of `f[2][1]` and `g[2][1]`, which is `1`.
   - We carry over `g[2][1]` to `g[3][1]`. Since `j > 0`, `g[3][1]` will also include `f[2][0]`, but since `f[2][0]` is `0`, it doesn't change.

Tables now:

```
f = [[0, 0],
     [1, 0],
     [0, 1],
     [0, 1],
     [0, 0]]

g = [[0, 0],
     [0, 0],
     [0, 1],
     [0, 1],
     [0, 0]]
```

3. For `i = 4`, `j = 1`:
   - Again, `f[4][1]` becomes the sum of `f[3][1]` and `g[3][1]`, totaling to `2`.
   - `g[4][1]` carries over `g[3][1]` and adds `f[3][0]`.

Tables now:

```
f = [[0, 0],
     [1, 0],
     [0, 1],
     [0, 1],
     [0, 2]]

g = [[0, 0],
     [0, 0],
     [0, 1],
     [0, 1],
     [0, 1]]
```

Finally, we return the sum of `f[n][k]` and `g[n][k]`, which is `f[4][1] + g[4][1] = 2 + 1 = 3` modulo `10^9 + 7`, so the result is `3`.

So, for `n = 4` and `k = 1`, there are `3` distinct ways we can draw `1` line segment that covers at least two points on the 1-D plane, without overlap, and with the possibility of shared endpoints.

Python Solution

```python
class Solution:
    def numberOfSets(self, n: int, k: int) -> int:
        # Define the modulus for large numbers to prevent overflow.
        MOD = 10**9 + 7

        # Create two 2D lists (tables) for dynamic programming.
        # 'dp' tracks the number of sets that can be formed using the first 'i' points with 'j' segments including the ith point.
        # 'cumulative' tracks the cumulative count of sets including previous points.
        dp = [[0] * (k + 1) for _ in range(n + 1)]
        cumulative = [[0] * (k + 1) for _ in range(n + 1)]

        # Initialize the base case: 1 way to form 0 segments using 1 point.
        dp[1][0] = 1

        # Fill the dynamic programming tables.
        for i in range(2, n + 1):  # For each point from 2 to n
            for j in range(k + 1):  # For each number of segments from 0 to k
                # The number of ways to form 'j' segments considering 'i' as the last point
                # is equal to the sum of the previous point without the ith point and the cumulative count.
                dp[i][j] = (dp[i - 1][j] + cumulative[i - 1][j]) % MOD

                # The cumulative count up to 'i' for 'j' segments is the same as the previous cumulative count.
                cumulative[i][j] = cumulative[i - 1][j]

                # If there is at least one segment to place,
                if j:
                    # Add the number of ways to form 'j-1' segments not including the ith point.
                    cumulative[i][j] += dp[i - 1][j - 1]
                    cumulative[i][j] %= MOD
                    # Add the number of ways to form 'j-1' segments including previous points.
                    cumulative[i][j] += cumulative[i - 1][j - 1]
                    cumulative[i][j] %= MOD

        # The final answer is the sum of the ways to form 'k' segments including the nth point and not including the nth point.
        return (dp[-1][-1] + cumulative[-1][-1]) % MOD

```

Java Solution

```java
class Solution {
    private static final int MOD = (int) 1e9 + 7;  // Defining the modulus value for large numbers to prevent overflow

    public int numberOfSets(int n, int k) {
        int[][] dpEnd = new int[n + 1][k + 1];  // dpEnd[i][j] represents the number of ways to form j segments with i points, with
        int[][] dpNotEnd = new int[n + 1][k + 1];  // dpNotEnd[i][j] represents the number of ways to form j segments with i points

        dpEnd[1][0] = 1;  // Base case: There is 1 way to form 0 segments with 1 point

        // Iterate over the number of points
        for (int i = 2; i <= n; ++i) {
            // Iterate over the number of segments
            for (int j = 0; j <= k; ++j) {
                // The number of ways with a segment ending at the current point i is equal to the sum of the number of ways
                // with and without a segment ending at the previous point (i-1), for the same number of segments
                dpEnd[i][j] = (dpEnd[i - 1][j] + dpNotEnd[i - 1][j]) % MOD;

                // For the number of ways without a segment ending at the current point i carry over the number
                // from the previous point i-1 without a segment ending
                dpNotEnd[i][j] = dpNotEnd[i - 1][j];

                // If we can form at least one segment
                if (j > 0) {
                    // There are two ways to extend the number of ways to form j segments without a segment ending at point i:
                    //   1) Use the number of ways to form j-1 segments with a segment ending at point i-1, and
                    //      start a new segment at point i-1 extending to i.
                    //   2) Extend the number of ways to form j segments without a segment ending at point i-1 by adding a point i.
                    dpNotEnd[i][j] += (dpEnd[i - 1][j - 1] + dpNotEnd[i - 1][j - 1]) % MOD;
                    // Modulo operation to prevent overflow
                    dpNotEnd[i][j] %= MOD;
                }
            }
        }

        // The total number of ways to form k segments with n points is the sum of the ways with and without a segment ending at po
        return (dpEnd[n][k] + dpNotEnd[n][k]) % MOD;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int dpSegmentEnd[1010][1010];  // dp array for cases where a segment ends at a point
    int dpNoSegmentEnd[1010][1010]; // dp array for cases with no segment ending at a point
    const int MOD = 1e9 + 7;       // Define the modulo operator value

    // Method that calculates the number of ways to divide a line into k non-overlapping line segments
    int numberOfSets(int n, int k) {
        // Initialize dp arrays
        memset(dpSegmentEnd, 0, sizeof(dpSegmentEnd));
        memset(dpNoSegmentEnd, 0, sizeof(dpNoSegmentEnd));
        dpSegmentEnd[1][0] = 1;  // Base case: only one way when there is one point and no segments

        // Dynamic programming iterations
        for (int i = 2; i <= n; ++i) {  // Iterate over points
            for (int j = 0; j <= k; ++j) {  // Iterate over number of segments
                // Ways where a segment does not end at the current point
                dpSegmentEnd[i][j] = (dpSegmentEnd[i - 1][j] + dpNoSegmentEnd[i - 1][j]) % MOD;

                // Ways where a segment ends at the current point
                dpNoSegmentEnd[i][j] = dpNoSegmentEnd[i - 1][j];

                if (j > 0) {
                    // If it is possible to add a new segment, update the ways count for segments ending at current point
                    dpNoSegmentEnd[i][j] = (dpNoSegmentEnd[i][j] + dpSegmentEnd[i - 1][j - 1]) % MOD;
                    dpNoSegmentEnd[i][j] = (dpNoSegmentEnd[i][j] + dpNoSegmentEnd[i - 1][j - 1]) % MOD;
                }
            }
        }
        // Total number of ways is the sum of ways where last segment ends on point n and where it doesn't
        return (dpSegmentEnd[n][k] + dpNoSegmentEnd[n][k]) % MOD;
    }
};

```

Typescript Solution

```typescript
// A function that calculates the number of ways to divide a line into `k` non-overlapping segments.
function numberOfSets(n: number, k: number): number {
    // Create a 2D array `f` where `f[i][j]` represents the number of ways to draw `j` segments with `i` points,
    // ending with a point.
    const f = Array.from({ length: n + 1 }, () => new Array(k + 1).fill(0));
    // Create a 2D array `g` where `g[i][j]` represents the number of ways to draw `j` segments with `i` points,
    // possibly ending with a segment.
    const g = Array.from({ length: n + 1 }, () => new Array(k + 1).fill(0));
    // Base case: there's one way to draw 0 segments with 1 point.
    f[1][0] = 1;
    // The modulo value for preventing integer overflow.
    const mod = 10 ** 9 + 7;

    // Loop through the points.
    for (let i = 2; i <= n; ++i) {
        // Loop through the number of segments.
        for (let j = 0; j <= k; ++j) {
            // Number of ways to draw `j` segments up to point `i`, ending with a point.
            f[i][j] = (f[i - 1][j] + g[i - 1][j]) % mod;

            // Number of ways to draw `j` segments up to point `i`, possibly ending with a segment (continuing).
            g[i][j] = g[i - 1][j];

            // If we have at least one segment.
            if (j) {
                // Add ways to draw `j-1` segments up to point `i-1`, then add a new segment ending at point `i`.
                g[i][j] += f[i - 1][j - 1];
                g[i][j] %= mod;

                // Add ways to continue the segment from `j-1` segments up to point `i-1`.
                g[i][j] += g[i - 1][j - 1];
                g[i][j] %= mod;
            }
        }
    }

    // Return the number of ways to draw `k` segments with `n` points, which could either end with a point or a segment.
    return (f[n][k] + g[n][k]) % mod;
}

```

Time and Space Complexity

The given Python code defines a class `Solution` with a method `numberOfSets` that calculates the number of ways to draw `k` non-overlapping line segments from `n` distinct points. The dynamic programming approach is used with two 2D lists (or matrices if you prefer), `f` and `g`. Here's an analysis of the time and space complexity of the code:

Time Complexity

The code consists of nested loops: the outer loop runs for `n - 1` iterations (starting from 2 up to `n`), and the inner loop runs for `k + 1` iterations (starting from 0 up to `k`). Since each iteration performs a constant number of operations including assignments and modular additions, the time complexity is a direct product of the number of iterations of these loops.

Therefore, the time complexity is `O(n * k)`.

Space Complexity

Two matrices `f` and `g` are created, each of size `(n + 1) x (k + 1)`. No additional space that grows with `n` or `k` is used in the algorithm, so the space complexity is determined by the size of these matrices.

Thus, the space complexity is `O(n * k)` since it's proportional to the amount of space used to store the matrices.