1692. Count Ways to Distribute Candies

Hard Dynamic Programming Leetcode Link

Problem Description

In this problem, we are given `n` unique candies, each labeled distinctly from `1` to `n`, and `k` bags. We need to distribute all the candies into the bags so that each bag has at least one candy. What makes the problem interesting is the way we consider two distributions to be different: if there is at least one candy that ends up in different bags in the two distributions, they are counted as distinct. However, the order of candies within a bag or the order of the bags themselves does not affect the distinctions between distributions.

The task is to count the number of different ways this distribution can occur, with the added constraint that the final number can be quite large and so we are asked to return the result modulo `10^9 + 7`.

Here's an example for clarity: If we have `3` candies and `2` bags, one way to distribute the candies could be `(1), (2,3)`. Another distinct distribution would be `(2), (1,3)`, but `(3,2), (1)` would not be considered different from `(1), (2,3)` since the same groups of candies are together, just in a different order.

Intuition

To solve this problem, we use Dynamic Programming (DP), which is an algorithmic technique to solve problems by simplifying them into smaller subproblems.

The intuition behind the solution lies in thinking about the problem in terms of stages: adding one candy at a time and one bag at a time, and calculating how many ways we can distribute the candies.

We define a 2D DP array `f`, with `f[i][j]` representing the number of different ways to distribute `i` candies into `j` bags. To build up this table, we consider two actions for the `i-th` candy:

1. **Place it into an existing bag**: We can put the `i-th` candy into any of the `j` existing bags that already have at least one candy. This action does not change the number of bags, so it's just `f[i - 1][j] * j`.
2. **Use a new bag for it**: If we decide to put the `i-th` candy in a new bag, there is exactly 1 way to do that since the bag is empty. This action increases the number of bags by 1, so we take the count from `f[i - 1][j - 1]`.

For each `i` and `j` greater than 1, we sum these two possibilities (and take the result modulo `10^9 + 7` to handle large numbers).

We must also note that we cannot have fewer bags than candies, nor can we have more bags than candies (since each bag must contain at least one candy). Therefore, `f[i][j]` is only defined if `1 <= j <= i`.

The base case of the DP table is `f[0][0] = 1`, which means that there's one way to distribute zero candies into zero bags (doing nothing).

Finally, the answer to the problem will be the value in `f[n][k]` after we've filled out the table according to our rules for adding candies to bags.

Solution Approach

The implementation of the solution employs a Dynamic Programming (DP) approach, where we use a 2D array `f` as our DP table. Each cell `f[i][j]` of this table represents the number of ways we can distribute `i` candies among `j` bags.

Data Structures Used:

- **2D DP Table**: A list of lists (2D array), where the outer list has `n + 1` elements, and each inner list has `k + 1` elements. This is initialized so that each cell, to begin with, has 0 ways (`f[i][j] = 0` for all `i` and `j`), except for `f[0][0]`, which is set to 1 as our base case.

Initialization:

- The DP table is initialized to all zeros except for the base case `f[0][0]`:
  - `f[0][0] = 1` signifies there's exactly one way to distribute zero candies, which is to do nothing.

Building the DP Table:

- We use two nested loops to iterate through our DP table:
  - The outer loop runs over the candies `i` starting from 1 up to `n` (`i` refers to the first `i` candies we're distributing).
  - The inner loop runs over the bags `j` starting from 1 up to `k` (`j` refers to the number of bags we can use).

DP Formula:

- Within the inner loop, we compute `f[i][j]` using the following formula:

  ```
  f[i][j] = (f[i - 1][j] * j + f[i - 1][j - 1]) % mod
  ```

  This formula reflects the two possible actions for candy `i`:

  - Either we place it in one of the `j` existing bags (`f[i - 1][j] * j`),
  - Or we place it in a new bag (`f[i - 1][j - 1]`).
- The modulo operation ensures that our numbers stay within the bounds of the given constraint `10^9 + 7`, which is necessary for large `n` and `k`.

Final Result:

- The final result, which is the answer to our problem, is then given by the value in `f[n][k]` after the DP table has been fully populated.

By following the above approach and using a DP table to store intermediary results, the algorithm efficiently computes the number of different ways to distribute the candies into the bags, ensuring that no recalculation for the same subproblems occurs.

Example Walkthrough

Let's go through an example to illustrate the solution approach.

Suppose we have `4` candies and `2` bags. We want to find out how many distinct ways we can distribute the 4 candies into these 2 bags.

Remember, by distinct, we mean that at least one candy must be in a different bag to count as a different distribution. The order of the candies within each bag and the order in which the bags are considered does not matter.

First, let's initialize our 2D DP table `f` with `n + 1` rows and `k + 1` columns, where `n` is the number of candies and `k` is the number of bags. So, we will have `f[5][3]` with all elements initialized to `0`, apart from `f[0][0]` which is set to `1`.

Our base case is:

- `f[0][0] = 1` because there's one way to distribute 0 candies into 0 bags: doing nothing.

Now for each candy `i` (from 1 to `n`) and each possible number of bags `j` (from 1 to `k`), we will use our DP formula to update the DP table.

Iterating through the candies and the bags

For `i = 1` (1 candy) and `j` from 1 to 2:

- `f[1][1] = (f[0][1] * 1 + f[0][0]) % mod = (0 * 1 + 1) % mod = 1`
- For j = 2, the condition `j <= i` is not met, so we leave `f[1][2]` at 0.

For `i = 2` (2 candies) and `j` from 1 to 2:

- `f[2][1] = (f[1][1] * 1 + f[1][0]) % mod = (1 * 1 + 0) % mod = 1`
- `f[2][2] = (f[1][2] * 2 + f[1][1]) % mod = (0 * 2 + 1) % mod = 1`

For `i = 3` (3 candies) and `j` from 1 to 2:

- `f[3][1] = (f[2][1] * 1 + f[2][0]) % mod = (1 * 1 + 0) % mod = 1`
- `f[3][2] = (f[2][2] * 2 + f[2][1]) % mod = (1 * 2 + 1) % mod = 3`

For `i = 4` (4 candies) and `j` from 1 to 2:

- `f[4][1] = (f[3][1] * 1 + f[3][0]) % mod = (1 * 1 + 0) % mod = 1`
- `f[4][2] = (f[3][2] * 2 + f[3][1]) % mod = (3 * 2 + 1) % mod = 7`

After populating our table according to the formula, we can see that `f[4][2]` contains the value `7`, which means there are 7 distinct ways to distribute 4 candies into 2 bags.

Conclusion

Hence, for our given example of `4` candies and `2` bags, there are `7` distinct distributions possible. This is what the final DP table looks like (non-relevant cells are not filled in):

	0	1	2
0	1	0	0
1	0	1	0
2	0	1	1
3	0	1	3
4	0	1	7

The `7` distinct distributions are (where each tuple represents a bag and order within the tuples doesn't matter):

- `(1), (2, 3, 4)`
- `(2), (1, 3, 4)`
- `(3), (1, 2, 4)`
- `(4), (1, 2, 3)`
- `(1, 2), (3, 4)`
- `(1, 3), (2, 4)`
- `(1, 4), (2, 3)`

This approach allows us to calculate the number of distribution methods efficiently without recalculating for the same scenarios, thanks to the Dynamic Programming method.

Python Solution

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

        # Initialize a 2D list (dp table) to store the number of ways to distribute
        # 'i' candies to 'j' bags. Each element is initialized to 0.
        dp = [[0] * (k + 1) for _ in range(n + 1)]

        # There is 1 way to distribute 0 candies to 0 bags.
        dp[0][0] = 1

        # Iterate through all the candies from 1 to n.
        for i in range(1, n + 1):
            # Iterate through all possible bags from 1 to k.
            for j in range(1, k + 1):
                # The number of ways to distribute 'i' candies to 'j' bags is composed of:
                # (1) The previous number of ways the 'i-1' candies were distributed to 'j' bags
                # multiplied by 'j' (placing the new candy into any existing bag)
                # (2) Plus the number of ways the 'i-1' candies were distributed to 'j-1' bags
                # (placing the new candy into a new bag).
                # We take the sum modulo 'MODULO' to keep the numbers within the integer range.
                dp[i][j] = (dp[i - 1][j] * j + dp[i - 1][j - 1]) % MODULO

        # Return the number of ways to distribute 'n' candies to 'k' bags.
        return dp[n][k]

```

Java Solution

```java
class Solution {

    /**
     * Calculate the number of ways to distribute n items across k bags.
     * @param n The number of items to distribute.
     * @param k The number of bags.
     * @return The number of different ways to distribute the items, modulo 10^9 + 7.
     */
    public int waysToDistribute(int n, int k) {
        // Define the modulus constant for the large number arithmetic to avoid overflow
        final int MOD = (int) 1e9 + 7;

        // Create a two-dimensional array to store intermediate results
        int[][] dp = new int[n + 1][k + 1];

        // Base case: There's 1 way to distribute 0 items into 0 bags
        dp[0][0] = 1;

        // Populate the dp array using a bottom-up dynamic programming approach
        for (int i = 1; i <= n; i++) {         // Iterate over the number of items
            for (int j = 1; j <= k; j++) {     // Iterate over the number of bags
                // The state transition equation calculates the number of ways to distribute items
                // into bags considering two scenarios:
                // 1. All previous i-1 items distributed in j bags, and the ith item goes to any of j bags.
                // 2. All previous i-1 items distributed in j-1 bags, and the ith item goes to a new bag.
                dp[i][j] = (int)((long) dp[i - 1][j] * j % MOD + dp[i - 1][j - 1]) % MOD;
            }
        }

        // Return the computed value representing the number of ways to distribute n items into k bags
        return dp[n][k];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int waysToDistribute(int n, int k) {
        const int MOD = 1000000007; // Define the modulus value for preventing integer overflow.
        int dp[n+1][k+1];           // Initialize a 2D array to use as a dynamic programming table.
        memset(dp, 0, sizeof(dp));  // Set all values in the dp array to 0.

        // Base case: There is 1 way to distribute 0 items into 0 groups.
        dp[0][0] = 1;

        // Populate the dynamic programming table.
        for (int items = 1; items <= n; ++items) {        // Iterate through items.
            for (int groups = 1; groups <= k; ++groups) { // Iterate through possible groups.
                // There are two possible scenarios:
                // 1. Include the current item in one of the existing groups, which is the same as
                //    the number of ways to distribute the rest items into the same number of groups
                //    multiplied by the number of groups (because the item can go into any group).
                long long includeInExisting = (1LL * dp[items - 1][groups] * groups) % MOD;

                // 2. Put the current item into a new group by itself, which is the same as
                //    the number of ways to distribute the rest items into one less group.
                long long createNewGroup = dp[items - 1][groups - 1];

                // The total ways to distribute items into groups is the sum of the above two scenarios.
                // We modulo by MOD to keep the number within integer limits.
                dp[items][groups] = (includeInExisting + createNewGroup) % MOD;
            }
        }

        // Return the result for distributing n items into k groups.
        return dp[n][k];
    }
};

```

Typescript Solution

```typescript
const MOD = 1000000007; // Define the modulus value for preventing integer overflow.

// Define a 2D array to use as a dynamic programming table. The "+1" accounts for the base case with 0 index.
let dp: number[][] = Array.from({ length: n + 1 }, () => Array(k + 1).fill(0));

// Function to calculate the ways to distribute n items into k groups.
function waysToDistribute(n: number, k: number): number {

    // Base case: There is 1 way to distribute 0 items into 0 groups.
    dp[0][0] = 1;

    // Populate the dynamic programming table.
    for (let items = 1; items <= n; ++items) {        // Iterate through the number of items.
        for (let groups = 1; groups <= k; ++groups) { // Iterate through the number of groups.
            // There are two possible scenarios:
            // 1. Include the current item in one of the existing groups, which is the same as
            //    the number of ways to distribute the remaining items into the same number of groups
            //    multiplied by the number of groups (because the item can go into any group).
            let includeInExisting = (dp[items - 1][groups] * groups) % MOD;

            // 2. Put the current item into a new group by itself, which is the same as
            //    the number of ways to distribute the remaining items into one fewer group.
            let createNewGroup = dp[items - 1][groups - 1];

            // The total ways to distribute items into groups is the sum of the two scenarios above.
            // We use the modulus operator to keep the number within integer limits.
            dp[items][groups] = (includeInExisting + createNewGroup) % MOD;
        }
    }

    // Return the result for distributing n items into k groups.
    return dp[n][k];
}

// Example of usage:
// let result = waysToDistribute(5, 3); // Call the function with the required parameters.

```

Time and Space Complexity

The provided Python code implements a dynamic programming approach to count the different ways to distribute `n` items into `k` distinct boxes. To determine the time and space complexity of this program, we will analyze the nested loops and the data structures used.

Time Complexity:

The time complexity of the code is dictated by the double `for` loop structure, where the outer loop runs `n` times (from 1 to `n` inclusive), and the inner loop runs `k` times (from 1 to `k` inclusive). The operation inside the inner loop has constant time complexity. Therefore, the total time complexity is `O(n * k)` because each cell `f[i][j]` is computed exactly once.

Space Complexity:

For space complexity, we have created a two-dimensional list `f` of size `(n+1) * (k+1)`. The size of this list does not change throughout the execution of the algorithm. Hence, the space complexity is `O(n * k)` as well, since we need to store an `n+1` by `k+1` matrix of integers.

Therefore, the overall time complexity is `O(n * k)` and the space complexity is also `O(n * k)`.