1434. Number of Ways to Wear Different Hats to Each Other

Hard · Bit Manipulation · Array · Dynamic Programming · Bitmask · Leetcode Link

Problem Description

In this problem, you are given `n` people and a fixed collection of `40` different types of hats, each type has its unique label ranging from `1` to `40`. A 2D integer array `hats` represents the hat preferences for each person, such that `hats[i]` contains a list of all the hats preferred by the `i-th` person. Your task is to determine the total number of unique ways that these `n` people can wear these hats so that no two people are wearing the same type of hat.

This is a classic combinatorial problem with a restriction that ensures the uniqueness of the hat type for each individual. The problem asks you to return this total number of unique distributions modulo `10^9 + 7`, which is a common technique to manage large output values in computational problems, ensuring the result stays within the limits of standard integer data types.

Intuition

At its core, the solution to this problem is rooted in combinatorics and dynamic programming (DP). The intuition behind using dynamic programming comes from two key observations:

1. The number of people `n` is small enough (maximum `10`) to consider the problem using state space representation where each state represents a set of people who have already been assigned a hat. This allows us to compress the state into a binary number (for instance, a binary representation where each bit represents whether a person has been assigned a hat or not).

2. The solution builds upon subproblems where each subproblem considers one less hat. This helps in constructing the final solution iteratively as DP excels at optimizing problems where the solution can be built from smaller subproblems.

Given the small number of people, we can use bit masks to represent who has a hat already. The dynamic programming array `f[i][j]` signifies the number of ways to assign hats up to the `i-th` hat, considering the people configuration `j`. The bit representation of `j` has '1' in the positions of people who have already been assigned hats and '0' otherwise.

The DP approach then incrementally builds up the solution by considering cases where either the `i-th` hat is not used or it's assigned to one of the people who like it following the rules.

Each state transition thus involves either:

- Keeping the configuration as is (not assigning the new hat), which doesn't change the state, or
- Including a new assignment (giving the `i-th` hat to person `k`), which changes the state by updating the bit mask to reflect that person `k` now has a hat.

This way, the final answer is built up by considering all the possibilities, taking into account which hats can be worn by whom, until all hats up to the maximum value in the given lists are processed.

Solution Approach

The implementation of the solution for this problem utilizes dynamic programming (DP) with bit masking and involves a few steps:

1. **Initialize the DP Table**: A 2-dimensional DP table `f` is created with dimensions `(m + 1) x (1 << n)`, where `m` is the maximum hat number preferred by any person and `n` is the number of people. Each entry `f[i][j]` will store the count of ways to distribute the first `i` types of hats across the people represented by state `j`.

2. **Define the Base Case**: We start with the base case `f[0][0] = 1`, which means there is one way to assign zero hats to zero people.

3. **Map Preferences**: We create a mapping `g` from each hat to a list of people who like that hat. The mapping is needed to quickly find which people can be considered when trying to distribute a particular hat.

4. **Iterate Over Hats**: For each hat type from `1` to `m`, we iterate to assign this hat to different people. For each state `j` that represents which people already have hats, we can have two possibilities:

   - **Hat not assigned**: If the current hat is not assigned to anyone, then the number of ways to distribute hats remains the same as the previous hat, so `f[i][j] = f[i - 1][j]`.

   - **Hat assigned**: If the hat is assigned to one of the potential people who like it, each assignment will result in a new state from `j` to `j ⊕ (1 << k)` where `⊕` denotes the XOR operation and `(1 << k)` denotes a bit mask with only the `k-th` bit set (meaning assigning the `i-th` hat to the `k-th` person). This operation toggles the `k-th` bit of `j`, so we accumulate these new ways into `f[i][j]` i.e., `f[i][j] = (f[i][j] + f[i - 1][j ⊕ (1 << k)]) % mod`.

5. **Modulo Operation**: Since the number of ways can be large, we take every sum modulo `10^9 + 7` (stored in the variable `mod`) to keep the values within the limits of a 32-bit signed integer.

6. **Return the Result**: After processing all hats, the result will be in `f[m][2^n - 1]`, which represents all `n` people having different hats.

This DP approach exploits the concept of state transitions while considering all permutations of hat assignments, respecting the individual preferences and ensuring unique ownership of hat types across all the people. The use of bit masking is a clever trick to represent a set of states succinctly when the universe is small, which is typically harder to do when dealing with large sets or where relationships between elements are complex.

The algorithm complexity primarily depends on the number of hats (which is at most 40) and the number of people (the size of the bitmask, which is `2^n` states). Although it looks like a large number, the small limitation of `n` makes this algorithm feasible.

Example Walkthrough

Let's illustrate the solution approach with a small example.

Suppose we have `n = 2` people and `m = 3` different types of hats. The hats preferred by the people are represented by the 2D array `hats` such that `hats[0] = [1, 2]` and `hats[1] = [2, 3]`. Our goal is to find the number of unique ways to distribute hats so that both people wear different hats from their preference list.

Step by Step Implementation:

1. **Initialize the DP Table**: We create a 2-dimensional DP table `f` with dimensions `(3 + 1) x (1 << 2)` (since there are at most 3 types of hats and 2 people). The table is initialized with zeros, except for the base case `f[0][0] = 1`.

2. **Define the Base Case**: As defined, `f[0][0] = 1` means one way to assign zero hats to zero people.

3. **Map Preferences**: We create a mapping `g` from each hat to the list of people who like that hat. From the preference list, we get `g[1] = [0], g[2] = [0, 1], g[3] = [1]`.

4. **Iterate Over Hats**:

   - For hat 1: Only person 0 likes this hat. We update `f[1][1]` to 1, which indicates that there's one way to assign hat 1 to person 0.

   - For hat 2: Both person 0 and person 1 like this hat. We can assign this hat to person 0 if person 0 hasn't been given a hat already, which means updating `f[2][1]` to `f[1][0]`. Similarly, we can assign hat 2 to person 1 if person 1 hasn't been given a hat already, updating `f[2][2]`.

   - For hat 3: Only person 1 likes this hat. We update `f[3][2]` to `f[2][0]` because we can give hat 3 to person 1 if they have no hat yet.

5. **Modulo Operation**: After each assignment, we perform `f[i][j] = (f[i][j] + f[i - 1][j ⊕ (1 << k)]) % mod` to keep numbers within the 32-bit signed integer limit.

6. **Return the Result**: The result is `f[3][3]`, which represents both people having different hats. In this example, `f[3][3]` should give us 2, indicating there are two ways to give hats to the people according to their preferences: Person 0 can wear hat 1 and Person 1 can wear hat 3 or Person 0 can wear hat 2 and Person 1 can wear hat 3.

Thus, the example helps us see how the dynamic programming table builds up solutions with different combinations using bit masks to represent states. The table `f` is updated by considering each hat and each combination of which people already have hats (states). This results in a final count of the number of unique ways to distribute hats so that no two people are wearing the same type of hat.

Python Solution

```python
from collections import defaultdict

class Solution:

    def numberWays(self, hats: List[List[int]]) -> int:
        # Create a mapping of which hat numbers are preferred by each person
        preference_map = defaultdict(list)
        for person_id, preferred_hats in enumerate(hats):
            for hat in preferred_hats:
                preference_map[hat].append(person_id)

        # Define a modulo value for the answer
        mod = 10**9 + 7

        # Determine the number of people
        num_people = len(hats)

        # Find the maximum hat number to define the range of hats
        max_hat_number = max(max(hat_list) for hat_list in hats)

        # Initialize a DP array where f[i][j] is the number of ways where
        # i is the current hat number, and j is the bitmask representing
        # the assignment status of hats to people (1 means assigned)
        dp = [[0] * (1 << num_people) for _ in range(max_hat_number + 1)]

        # Base case: 0 ways with 0 hats assigned
        dp[0][0] = 1

        # Iterate through all hat numbers
        for hat in range(1, max_hat_number + 1):
            # Iterate through all possible combinations of people
            for mask in range(1 << num_people):
                # The number of ways to assign hats without the current hat
                dp[hat][mask] = dp[hat - 1][mask]
                # Try to assign the current hat to each person who prefers this hat
                for person in preference_map[hat]:
                    # Check if the current person has not already been assigned a hat
                    if mask & (1 << person):
                        # Add the number of ways to assign hats with
                        # the current person assigned the current hat
                        dp[hat][mask] = (dp[hat][mask] + dp[hat - 1][mask ^ (1 << person)]) % mod

        # Return the total number of ways to assign all hats to all people
        return dp[max_hat_number][(1 << num_people) - 1]
```

Java Solution

```java
class Solution {
    public int numberWays(List<List<Integer>> hats) {
        // Number of friends
        int numFriends = hats.size();
        // Maximum hat number across all friends
        int maxHatNumber = 0;

        // Determine the highest numbered hat
        for (List<Integer> friendHats : hats) {
            for (int hat : friendHats) {
                maxHatNumber = Math.max(maxHatNumber, hat);
            }
        }

        // Create an array to associate each hat with a list of friends who like it
        List<Integer>[] hatToFriends = new List[maxHatNumber + 1];
        Arrays.setAll(hatToFriends, k -> new ArrayList<>());

        // Populate hatToFriends lists with the indices of friends
        for (int i = 0; i < numFriends; ++i) {
            for (int hat : hats.get(i)) {
                hatToFriends[hat].add(i);
            }
        }

        // A modulus value for the result
        final int MOD = (int) 1e9 + 7;

        // Dynamic programming table where 'f[i][j]' represents the number of ways to assign
        // hats to the first 'i' hats such that 'j' encodes which friends have received hats
        int[][] dpTable = new int[maxHatNumber + 1][1 << numFriends];

        // Base case: there's 1 way to assign 0 hats (none to anyone)
        dpTable[0][0] = 1;

        // Build the table from the base case
        for (int i = 1; i <= maxHatNumber; ++i) {
            for (int j = 0; j < 1 << numFriends; ++j) {
                // Start with the number of ways without the current hat
                dpTable[i][j] = dpTable[i - 1][j];

                // Iterate through all friends who like the current hat
                for (int friendIndex : hatToFriends[i]) {
                    // Check if the friend hasn't been given a hat yet in combination 'j'
                    if ((j >> friendIndex & 1) == 1) {
                        // Add ways to assign hats from previous combination with one less hat,
                        // ensuring that friend 'friendIndex' now has a hat
                        dpTable[i][j] = (dpTable[i][j] + dpTable[i - 1][j ^ (1 << friendIndex)]) % MOD;
                    }
                }
            }
        }

        // Return the number of ways for the last hat and all friends (the full set bit mask)
        return dpTable[maxHatNumber][(1 << numFriends) - 1];
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int numberWays(vector<vector<int>>& hats) {
        int numPeople = hats.size(); // Number of people
        int maxHatID = 0;           // Maximum hat ID
        // Find the maximum hat ID to know the range of hat IDs available
        for (auto& personHats : hats) {
            maxHatID = max(maxHatID, *max_element(personHats.begin(), personHats.end()));
        }
        // Create a graph where each hat ID points to a list of people who like that hat
        vector<vector<int>> hatToPeople(maxHatID + 1);
        for (int i = 0; i < numPeople; ++i) {
            for (int hat : hats[i]) {
                hatToPeople[hat].push_back(i);
            }
        }

        const int MOD = 1e9 + 7; // Modulo value for avoiding integer overflow
        // f[i][j] will be the number of ways to assign hats considering first i hats
        // where j is a bitmask representing which people have already been assigned a hat
        int dp[maxHatID + 1][1 << numPeople];
        memset(dp, 0, sizeof(dp));
        dp[0][0] = 1; // Base case: no hats assigned to anyone

        // Iterate over all hats
        for (int i = 1; i <= maxHatID; ++i) {
            // Iterate over all possible assignments of hats to people
            for (int mask = 0; mask < (1 << numPeople); ++mask) {
                // The number of ways without assigning the current hat remains the same
                dp[i][mask] = dp[i - 1][mask];
                // Iterate over all the people who like the current hat
                for (int person : hatToPeople[i]) {
                    // If this person has not yet been assigned a hat
                    if (mask & (1 << person)) {
                        // Add the number of ways by assigning the current hat to this person and update it modulo MOD
                        dp[i][mask] = (dp[i][mask] + dp[i - 1][mask ^ (1 << person)]) % MOD;
                    }
                }
            }
        }
        // Return the number of ways to assign all hats to all people
        return dp[maxHatID][(1 << numPeople) - 1];
    }
};
```

Typescript Solution

```typescript
function numberWays(hats: number[][]): number {
    // Number of people
    const numPeople = hats.length;
    // Maximum hat number
    const maxHatNumber = Math.max(...hats.flat());

    // Graph representing which people can wear which hats
    const hatToPeopleGraph: number[][] = Array.from({ length: maxHatNumber + 1 }, () => []);

    // Populate the graph with the information about which people can wear which hats
    for (let i = 0; i < numPeople; ++i) {
        for (const hat of hats[i]) {
            hatToPeopleGraph[hat].push(i);
        }
    }

    // DP array to store ways to distribute hats
    // dp[hatNumber][mask] will represent the number of ways to distribute
    // the first hatNumber hats among people represented by mask
    const dp: number[][] = Array.from({ length: maxHatNumber + 1 }, () =>
        Array.from({ length: 1 << numPeople }, () => 0),
    );

    // Base case: for 0 hats we have one way to assign - none to anyone
    dp[0][0] = 1;

    // Modulus for the result to prevent integer overflow
    const mod = 1e9 + 7;

    // Iterate over all hats
    for (let hatNumber = 1; hatNumber <= maxHatNumber; ++hatNumber) {
        // Iterate over all combinations of people
        for (let mask = 0; mask < 1 << numPeople; ++mask) {
            // The number of ways to distribute hats without considering the current hat
            dp[hatNumber][mask] = dp[hatNumber - 1][mask];
            // Iterate over all people that can wear the current hat
            for (const person of hatToPeopleGraph[hatNumber]) {
                // Check if the current person is included in the mask
                if (((mask >> person) & 1) === 1) {
                    // Add ways from the previous hat state excluding the current person
                    dp[hatNumber][mask] = (dp[hatNumber][mask] + dp[hatNumber - 1][mask ^ (1 << person)]) % mod;
                }
            }
        }
    }

    // Return the number of ways to assign all hats to all people
    // All people are represented by the mask (1 << numPeople) - 1, which has all bits set
    return dp[maxHatNumber][(1 << numPeople) - 1];
}
```

Time and Space Complexity

The provided code defines a function `numberWays` which calculates the number of ways people can wear hats given certain constraints. The analysis of its time and space complexity is as follows:

Time Complexity

The time complexity of the function is `O(m * 2^n * n)`. Here's a breakdown of why that's the case:

- `m` represents the maximum number of different hats, which can go up to `40` as per the problem constraints.
- `n` is the number of people, limited to `10` in this scenario.
- `2^n` signifies the number of different states or combinations for the assignment of hats to the `n` people. Since each person can either wear a hat or not, there are `2^n` possible states.

For each of the `m` hats, the algorithm iterates over all `2^n` combinations, and for each combination, it can potentially iterate over all `n` people to update the state (`f[i][j]`). As a result, the time complexity amounts to the multiplicative product of these terms.

Space Complexity

The space complexity of the function is `O(m * 2^n)`. This is due to the following reasons:

- An `m + 1` by `2^n` sized 2D list `f` is created to store the states of hat assignments, where each sub-list `f[i]` has a length of `2^n` to represent all combinations of `n` people, and there are `m + 1` such sub-lists (ranging from `0` to `m`).
- The additional data structures use negligible space compared to the size of `f`, so their contribution to space complexity is not considered dominant.

In summary, the algorithm requires a significant amount of space proportional to the number of hat combinations multiplied by the number of different hats.