1366. Rank Teams by Votes

Medium Array Hash Table String Counting Sorting Leetcode Link

Problem Description

In this problem, we have a voting system for ranking teams based on preferences from different voters. Each voter ranks all teams from their most preferred (highest rank) to least preferred (lowest rank). We need to compile the votes in a way that reflects the overall ranking of teams.

The ranking of the teams is determined by the following criteria:

1. Primary Criterion: A team is ranked higher if it has received more first-place votes compared to other teams.
2. Tiebreaker: If teams tie for first-place votes, then second-place votes are compared, and so forth, until the tie is broken.
3. Alphabetical Order: If a tie cannot be broken with all provided ranks, teams are then ranked alphabetically.

We are given an array of strings `votes`, where each string contains a voter's ranking of all the teams. The task is to return a string with all teams sorted based on the above ranking system.

Intuition

To solve this problem, we have to interpret the voting data and apply the ranking criteria to establish the correct order of all teams.

Here's the approach to derive the solution:

1. Counting the Votes: We need to keep track of how many times each team gets each rank from all voters. Therefore, for each team, we create an array of counters for each rank position. For instance, if there are `n` teams, each team will have an array of size `n` to hold the counts of being voted for each rank from 1 to `n`.

2. Sorting the Teams: Once we have the counts for each team per rank, we have to sort the teams. The sorting decision is based on:

   - The count arrays: A team is considered higher-ranking if its count array represents more first-place votes, followed by more second-place votes if there is a tie, and so forth.
   - Alphabetical order as a tiebreaker: When count arrays are identical for two teams (indicating a tie across all ranks), the team that comes first in alphabetical order is ranked higher.
3. Constructing the Final String: After sorting, we simply concatenate the teams in the order determined by sorting, which gives us the final ranking as a string.

The Python code provided uses a `defaultdict` to create the count arrays for each team. It then sorts the team based on their count arrays (with first-place votes having the highest importance), and uses the negative ASCII value of the team character as a secondary sort key to handle alphabetical ties (since `reverse=True` flips the usual ordering). Finally, it joins the sorted teams to form the output string reflecting the overall ranking.

Solution Approach

In the provided Python solution, we implement an efficient approach to rank the teams according to the voting criteria. Here's a step-by-step explanation of the implementation using algorithms, data structures, and patterns:

1. **Data Structure - Default Dictionary with Arrays**: We use Python's `defaultdict` with a lambda function to initialize an array of zeros for each team. This data structure is key to tracking the rank counts for each team. The lambda function `lambda: [0] * n` ensures that any new key (representing a team) in the dictionary automatically starts with an array of zeros, where `n` is the number of teams.

2. **Vote Counting - Enumerate Pattern**: Using the `enumerate` function, we iterate over each vote and its index within the vote string. The index represents the rank given by a voter, and by incrementing `cnt[c][i]` where `c` is the team, we accumulate the rank votes for every team.

3. **Sorting Algorithm**: We then sort the teams using the sorted function with a custom sorting key. The sorting key is a tuple consisting of the count array `cnt[x]` and the negative ASCII value `-ord(x)`, where `x` represents a team. Python sorts tuples based on a lexicographical order, meaning it compares the first item, then the second if needed, and so forth. The first element of the tuple ensures that teams are sorted based on the entire ranking information, while `-ord(x)` is used as a tiebreaker because `sorted` is a stable sort, meaning that if two elements have the exact same count array, they will be ordered based on their ASCII character value.

4. **Custom Key Function for Sorting**: By using the `key=lambda x: (cnt[x], -ord(x))`, we tell the sorted function to prioritize the count arrays first. Since we want the teams with the most first-place votes to be at the front, we use `reverse=True` to sort the list in descending order. The negative ASCII value sorts alphabetically in the reverse (because 'A' has a smaller ASCII value than 'Z'), but reversing the list corrects this and yields the correct order.

5. **Building the Final String**: Finally, we join the sorted list of teams into a string, which represents the teams sorted according to the rank system.

Overall, the algorithms and data structures used in the solution are chosen to efficiently handle the rank aggregation and sorting criteria as specified in the problem, resulting in a concise and performant implementation.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose there are three teams: A, B, and C, and we have the following votes from four voters:

```
votes = ["BCA", "CAB", "CBA", "ABC"]
```

In this example, each letter represents a team, and the order of the letters in each vote string represents the ranking from the most preferred team to the least preferred team by a voter. Here, we will walk through the algorithm to determine the final ranking.

1. **Initialize the Default Dictionary with Arrays**: We create a default dictionary called `cnt` that will map each team to an array of zeros. Since there are three teams, each array will have a length of three.

```
cnt = defaultdict(lambda: [0, 0, 0])
```

After observing all votes the `cnt` dictionary would be (not in order):

```
cnt = {
  'A': [1, 1, 2],
  'B': [0, 2, 2],
  'C': [3, 1, 0]
}
```

'A' has been voted first once, second once, and third twice; 'B' has been voted second and third twice, but never first; 'C' has been voted first three times, second once.

2. **Counting Votes**: We iterate over each vote in `votes` and each team within the vote string:

```
for vote in votes:
  for i, c in enumerate(vote):
    cnt[c][i] += 1
```

Using enumeration, we increment the count of rank i for each team c.

3. **Sorting Teams**: We now sort the teams based on their count arrays, and in case of a tie, by their alphabetical order:

```
teams = sorted(cnt, key=lambda x: (cnt[x], -ord(x)), reverse=True)
```

Here, `teams` will end up being `['C', 'B', 'A']` because 'C' has the most first-place votes, followed by 'A' and 'B'. Since 'A' and 'B' never received a first-place vote, they are sorted by the number of second-place votes they received, in which 'B' beats 'A'. In a situation where 'B' and 'A' also had the same number of second-place votes, they would be sorted alphabetically, and 'A' would come before 'B'.

4. **Building the Final Ranking String**: The final step is to concatenate the sorted teams into one string, which would be our final ranking based on the votes:

```
result = ''.join(teams)
```

The result here would be `'CBA'`, which completes our example walkthrough. Team C is ranked first, team B is ranked second, and team A is ranked third based on the given votes.

Python Solution

```python
from collections import defaultdict

class Solution:
    def rankTeams(self, votes: List[str]) -> str:
        # Get the length of a vote, which is the number of teams
        num_teams = len(votes[0])

        # Create a default dictionary to store the vote count for each position for each team
        # The dictionary is initialized with lists of zeros, with the same length as the number of teams
        vote_counts = defaultdict(lambda: [0] * num_teams)

        # Count the votes for each position for each team
        for vote in votes:
            for rank, team in enumerate(vote):
                vote_counts[team][rank] += 1

        # Sort the teams based on their vote counts
        # If vote counts are the same, fallback to lexicographical order (ASCII values)
        # 'reverse=True' ensures a descending order since we want the highest voted team first
        ranked_teams = sorted(votes[0], key=lambda team: (vote_counts[team], -ord(team)), reverse=True)

        # Join the sorted teams into a string that represents their ranking
        return "".join(ranked_teams)

```

Java Solution

```java
import java.util.Arrays;

class Solution {
    public String rankTeams(String[] votes) {
        // The number of teams is determined by the length of a single vote string
        int numTeams = votes[0].length();

        // Create a 2D array to count the position based votes for each team (A-Z mapped to 0-25)
        int[][] count = new int[26][numTeams];
        for (String vote : votes) {
            for (int i = 0; i < numTeams; ++i) {
                // Increment the vote count for the team at the current position
                count[vote.charAt(i) - 'A'][i]++;
            }
        }

        // Create an array of Characters representing each team in the initial vote order
        Character[] teams = new Character[numTeams];
        for (int i = 0; i < numTeams; ++i) {
            teams[i] = votes[0].charAt(i);
        }

        // Sort the array of teams based on the vote counts and then by alphabetical order
        Arrays.sort(teams, (a, b) -> {
            int indexA = a - 'A', indexB = b - 'A';
            for (int k = 0; k < numTeams; ++k) {
                // Compare the vote count for the current position
                int difference = count[indexA][k] - count[indexB][k];
                if (difference != 0) {
                    // If there's a difference, return the comparison result
                    return difference > 0 ? -1 : 1;
                }
            }
            // If all vote counts are equal, sort by alphabetical order
            return a - b;
        });

        // Build the final ranking string based on the sorted array of teams
        StringBuilder result = new StringBuilder();
        for (char team : teams) {
            result.append(team);
        }
        return result.toString();
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <algorithm>
#include <cstring>

class Solution {
public:
    // This method will process a vector of strings representing votes and
    // return a string representing the rank of teams.
    string rankTeams(vector<string>& votes) {
        // First, get the number of teams from the first vote
        int num_teams = votes[0].size();

        // Initialize a 2D array to keep track of the count of ranks for each team
        int rank_count[26][num_teams];
        memset(rank_count, 0, sizeof rank_count);

        // Iterate over each vote and increment the count for team's rank position
        for (auto& vote : votes) {
            for (int i = 0; i < num_teams; ++i) {
                rank_count[vote[i] - 'A'][i]++;
            }
        }

        // Start with an initial ordering of teams as in the first vote
        string ranking = votes[0];

        // Sort the teams according to their rank counts
        sort(ranking.begin(), ranking.end(), [&](auto& a, auto& b) {
            // Retrieve the rank array indices corresponding to the teams
            int team1_idx = a - 'A', team2_idx = b - 'A';

            // Iterate over each rank position
            for (int rank = 0; rank < num_teams; ++rank) {
                // If count for a rank position is different, determine order by the higher count
                if (rank_count[team1_idx][rank] != rank_count[team2_idx][rank]) {
                    return rank_count[team1_idx][rank] > rank_count[team2_idx][rank];
                }
            }

            // If the teams are tied in all rank positions, order alphabetically
            return a < b;
        });

        // Return the final ranking string
        return ranking;
    }
};

```

Typescript Solution

```typescript
// Define a function that processes an array of strings representing votes
// and returns a string representing the rank of teams.
function rankTeams(votes: string[]): string {
    // Get the number of teams from the size of the first vote
    const numTeams = votes[0].length;

    // Initialize a 2D array to keep track of the count of ranks for each team
    const rankCount: number[][] = Array.from({ length: 26 }, () => new Array(numTeams).fill(0));

    // Iterate over each vote and increment the count for team's rank position
    for (let vote of votes) {
        for (let i = 0; i < numTeams; i++) {
            const teamIndex = vote.charCodeAt(i) - 'A'.charCodeAt(0);
            rankCount[teamIndex][i]++;
        }
    }

    // Start with an initial ordering of teams as in the first vote
    let ranking = votes[0].split('');

    // Sort the teams according to their rank counts
    ranking.sort((a, b) => {
        const team1Index = a.charCodeAt(0) - 'A'.charCodeAt(0);
        const team2Index = b.charCodeAt(0) - 'A'.charCodeAt(0);

        // Compare the teams based on their rank counts
        for (let rank = 0; rank < numTeams; rank++) {
            if (rankCount[team1Index][rank] !== rankCount[team2Index][rank]) {
                return rankCount[team1Index][rank] > rankCount[team2Index][rank] ? -1 : 1;
            }
        }

        // If the teams are tied in all rank positions, order alphabetically by comparing ASCII values
        return a < b ? -1 : 1;
    });

    // Join the sorted array back into a string and return the final ranking string
    return ranking.join('');
}

// Example Usage:
const votes = ["ABC", "ACB", "ABC", "ACB", "ACB"];
const ranking = rankTeams(votes);
console.log(ranking); // Should print the team ranking based on the votes

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is determined by several factors:

1. Counting the rankings for each team involves iterating over all the votes and updating a list of size `n`, which is the number of teams. Each vote takes `O(n)` time to iterate, and this is done for all `m` votes. So, this part of the algorithm takes `O(m * n)` time.

2. Sorting the teams according to their rank counts and in case of a tie, using alphabetic ordering. Python's sort function uses TimSort, which has a worst-case time complexity of `O(n * log(n))`. Since there are `n` teams, sorting them takes `O(n * log(n))` time.

3. Generating the final string involves creating a string from the sorted teams, which takes `O(n)` time.

Combining these factors, the overall time complexity is `O(m * n + n * log(n) + n)`. Since `n * log(n)` is likely to be the dominant term as `n` grows in comparison to `n`, we can approximate the time complexity as `O(m * n + n * log(n))`.

Space Complexity

The space complexity of the code is determined by:

1. The space used by the `cnt` dictionary, which contains a list of counters, of size `n`, for each distinct team. Since there are `n` teams, the total size of the `cnt` dictionary is `O(n^2)`.

2. The space used by the sorting function could be `O(n)` in the worst case for the internal working storage during the sort.

Taking these into account, the overall space complexity of the algorithm is `O(n^2)`.