1583. Count Unhappy Friends

Medium Array Simulation Leetcode Link

Problem Description

The problem presents a scenario of `n` friends paired with each other, where `n` is an even number. Each friend has a preference order list for all other friends, indicating who they would rather be paired with, in descending order of preference.

The friends have been matched into pairs based on the `pairs` list provided. However, not all friends may be content with their pairing. A friend `x` is considered unhappy if `x` prefers someone else `u` over their current partner `y`, and at the same time `u` prefers `x` over their own partner `v`. The aim here is to calculate the total number of unhappy friends.

Intuition

The intuition behind the solution involves the following key steps:

1. **Convert Preferences to Indexes:** To expedite lookup operations, it is advantageous to convert each friend's preferences list, which stores friends' IDs, into a dictionary where keys are the friend IDs and values are their corresponding preference rankings. This allows us to quickly determine the ranking of one friend over another in anyone's preference list.
2. **Store Pairing Information Efficiently:** Similarly, we use another dictionary to store each friend's current partner from the pairing list. This makes it fast and easy to find out who is paired with whom at any point.
3. **Check for Unhappy Friends:** Iterate over each friend `x`, and within that, iterate over `x`'s preference list up until (but not including) the current partner `y`. For each friend `u` in the preference list who is preferred over `y`, we check if `u` also prefers `x` over their own partner `v`. If such a `u` exists, `x` is unhappy, and we add to our unhappy friends count.
4. **Count**: After processing all friends, the count of unhappy friends gives us the desired answer.

This approach is efficient as it avoids unnecessary comparisons; we only consider friends who are ranked higher than the current partners and hence have the potential to cause unhappiness.

Solution Approach

The core implementation of the solution follows a precise and logical methodology, incorporating specific data structures and algorithms:

1. **Dictionary of Preferences' Ranks:** A list of dictionaries `d` is created where each dictionary corresponds to a friend and holds that friend's preferences with preference rankings as values. This transforms a list of preferred friends for each person into a structure that allows for O(1) time lookup to find out how much `i` prefers `p` over others.

   ```python
   d = [{p: i for i, p in enumerate(v)} for v in preferences]
   ```

   This list comprehension iterates over each friend's preferences list `v`, and for each friend's preferences, it generates a dictionary that maps the friend's ID `p` to its preference rank `i`.

2. **Pairings Dictionary:** Another dictionary `p` is created to hold the current pairings based on the `pairs` list. This dictionary is used to find out directly who is paired with whom.

   ```python
   p = {}
   for x, y in pairs:
       p[x] = y
       p[y] = x
   ```

3. **Determine Unhappiness:** An integer `ans` is set up to keep count of the number of unhappy friends.

   The solution iterates over each friend `x` and checks the list of their preferred friends (up until their current partner `y`) to see if there exists someone they prefer more (`u`) who also prefers them back over their own partner `v`.

   ```python
   ans = 0
   for x in range(n):
       y = p[x]
       ans += any(d[u][x] < d[u][p[u]] for u in preferences[x][: d[x][y]])
   ```

   Here, `d[x][y]` finds the ranking of `y` in `x`'s preferences. The slice `preferences[x][: d[x][y]]` gets all friends preferred more by `x` than `y`. For each such friend `u`, if `d[u][x] < d[u][p[u]]`, it means `u` prefers `x` over their current partner `p[u]`. The `any` function checks if there's any such `u` that satisfies this condition. If yes, `x` is unhappy.

4. **Return Answer:** After iterating over all friends and accumulating the count of unhappy ones, the final value of `ans` is returned.

The algorithm is efficient due to the use of dictionaries for quick lookups and the limited iteration up to the current pair, avoiding unnecessary comparison checks.

```python
return ans
```

Example Walkthrough

Let's go through a small example to illustrate the solution approach. Assume we have 4 friends (with IDs 0, 1, 2, and 3) and their preference listings, as well as their current pairings.

Suppose the friends' preference lists are as follows:

- Friend 0 prefers: [1, 2, 3]
- Friend 1 prefers: [2, 0, 3]
- Friend 2 prefers: [1, 0, 3]
- Friend 3 prefers: [0, 1, 2]

And they are paired like this:

- Pairings: [(0, 1), (2, 3)]

To solve this problem with the proposed solution approach:

1. We first convert the preference lists into dictionaries with rankings for O(1) lookup times.

   For friend 0: `{1: 0, 2: 1, 3: 2}` For friend 1: `{2: 0, 0: 1, 3: 2}` For friend 2: `{1: 0, 0: 1, 3: 2}` For friend 3: `{0: 0, 1: 1, 2: 2}`

2. Next, we create a dictionary to represent the pairings: `{0: 1, 1: 0, 2: 3, 3: 2}`

3. Now we determine if there are any unhappy friends. We proceed as follows:
   - Check friend 0: Paired with friend 1. Friend 0 prefers friend 2 over 1 (from their preference list), but friend 2, who is paired with friend 3, prefers friend 1 over 0. Therefore, friend 0 is not unhappy.
   - Check friend 1: Paired with friend 0. Friend 1 prefers friend 2 over 0 and friend 2 also prefers friend 1 over their partner friend 3, thus friend 1 is unhappy.
   - Check friend 2: Paired with friend 3. Friend 2 prefers friend 1 over 3, and we already know that friend 1 prefers friend 2 over their current partner (friend 0). Therefore, friend 2 is unhappy.
   - Check friend 3: Paired with friend 2. Friend 3 prefers friend 0 over 2, but friend 0 prefers friend 1 over friend 3, hence friend 3 is not unhappy.

   So from our example, friends 1 and 2 are unhappy.

4. Lastly, we return the count of unhappy friends which is 2 in this case.

By following the outlined solution steps, we efficiently find the total number of unhappy friends without unnecessary comparisons and in a manner that is easy to follow and implement.

Python Solution

```python
from typing import List

class Solution:
    def unhappyFriends(self, n: int, preferences: List[List[int]], pairs: List[List[int]]) -> int:
        # Create a dictionary to store preferences index for quick lookup.
        # `preference_rankings` is a list of dictionaries for each person,
        # mapping their friend to the ranking of that friend in their preference.
        preference_rankings = [{friend: rank for rank, friend in enumerate(prefs)} for prefs in preferences]

        # Create a dictionary to store each person's paired friend.
        paired_friends = {}
        for x, y in pairs:
            paired_friends[x] = y
            paired_friends[y] = x

        # Initialize the count of unhappy friends.
        unhappy_count = 0

        # Iterate through each person to determine if they are unhappy.
        for x in range(n):
            # The paired friend of `x`.
            y = paired_friends[x]

            # Check if there exists a person 'u' who is a better preference for 'x'
            # than 'x's paired friend 'y'. We do this by checking the preference
            # rankings in the subset of preferences before 'y'.
            is_unhappy = any(
                preference_rankings[u][x] < preference_rankings[u][paired_friends[u]]
                for u in preferences[x][: preference_rankings[x][y]]
            )

            # If such a person 'u' exists, increment the unhappy count.
            if is_unhappy:
                unhappy_count += 1

        return unhappy_count

```

Java Solution

```java
class Solution {
    public int unhappyFriends(int n, int[][] preferences, int[][] pairs) {
        // Distance matrix indicating how strongly each friend prefers other friends
        int[][] preferenceDistances = new int[n][n];
        // Fill the preference distance matrix with preference rankings
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n - 1; ++j) {
                preferenceDistances[i][preferences[i][j]] = j;
            }
        }

        // Pairing array where the index is the friend and the value is their pair
        int[] pairings = new int[n];
        // Fill the pairings array based on the given pairs
        for (int[] pair : pairs) {
            int friend1 = pair[0], friend2 = pair[1];
            pairings[friend1] = friend2;
            pairings[friend2] = friend1;
        }

        // Counter for unhappy friends
        int unhappyCount = 0;
        // Iterate over all friends to determine unhappiness
        for (int friendX = 0; friendX < n; ++friendX) {
            int friendY = pairings[friendX];
            boolean isUnhappy = false; // Flag to check if the current friend is unhappy

            // Check if there exists a friend that friendX ranks higher than their current paired friendY
            for (int i = 0; i < preferenceDistances[friendX][friendY]; ++i) {
                int otherFriend = preferences[friendX][i];
                // Check if the other friend (u) prefers friendX over their own pairing
                if (preferenceDistances[otherFriend][friendX] < preferenceDistances[otherFriend][pairings[otherFriend]]) {
                    isUnhappy = true;
                    break;
                }
            }
            // Increment unhappyCount if friendX is found to be unhappy
            if (isUnhappy) {
                unhappyCount++;
            }
        }
        return unhappyCount; // Return the total number of unhappy friends
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to calculate the number of unhappy friends
    int unhappyFriends(int n, vector<vector<int>>& preferences, vector<vector<int>>& pairs) {
        // Declare a 2D array to store the preference rank
        int preferenceRank[n][n];
        // Array to store the pair partner for each person
        int pairPartner[n];

        // Initialize the preference ranks for each friend
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n - 1; ++j) {
                preferenceRank[i][preferences[i][j]] = j;
            }
        }

        // Assign the pair partners based on the given pairs
        for (auto& pair : pairs) {
            int friendOne = pair[0], friendTwo = pair[1];
            pairPartner[friendOne] = friendTwo;
            pairPartner[friendTwo] = friendOne;
        }

        int unhappyCount = 0; // Counter for unhappy friends

        // Iterate through each friend to check unhappiness
        for (int friendX = 0; friendX < n; ++friendX) {
            int friendY = pairPartner[friendX]; // Partner of the current friend
            bool isUnhappy = false; // Flag to check if the current friend is unhappy

            // Check whether there is a person that friendX prefers over friendY
            for (int i = 0; i < preferenceRank[friendX][friendY]; ++i) {
                int preferredFriend = preferences[friendX][i];
                // If preferredFriend prefers friendX over their current partner, friendX is unhappy
                if (preferenceRank[preferredFriend][friendX] < preferenceRank[preferredFriend][pairPartner[preferredFriend]]) {
                    isUnhappy = true;
                    break; // No need to check the rest since friendX is already unhappy
                }
            }

            // If friendX is found to be unhappy, increment the unhappy count
            if (isUnhappy) {
                unhappyCount++;
            }
        }

        // Return the final count of unhappy friends
        return unhappyCount;
    }
};

```

Typescript Solution

```typescript
// Define a type for the preference array that's used multiple times
type Preferences = number[][];

// Function to initialize the preference ranks for each friend
function initializePreferenceRanks(n: number, preferences: Preferences): number[][] {
    const preferenceRanks: number[][] = Array.from({ length: n }, () => Array(n).fill(-1));
    for (let i = 0; i < n; ++i) {
        for (let j = 0; j < n - 1; ++j) {
            preferenceRanks[i][preferences[i][j]] = j;
        }
    }
    return preferenceRanks;
}

// Function to assign pair partners based on the given pairs
function assignPairPartners(pairs: Preferences): number[] {
    const pairPartners: number[] = Array(pairs.length * 2);
    for (const pair of pairs) {
        const [friendOne, friendTwo] = pair;
        pairPartners[friendOne] = friendTwo;
        pairPartners[friendTwo] = friendOne;
    }
    return pairPartners;
}

// Function to calculate the number of unhappy friends
function unhappyFriends(n: number, preferences: Preferences, pairs: Preferences): number {
    const preferenceRanks = initializePreferenceRanks(n, preferences);
    const pairPartners = assignPairPartners(pairs);
    let unhappyCount = 0; // Counter for unhappy friends

    for (let friendX = 0; friendX < n; ++friendX) {
        const friendY = pairPartners[friendX]; // Partner of the current friend
        let isUnhappy = false; // Flag to check if the current friend is unhappy

        // Check whether there is a person that friendX prefers over friendY
        for (let i = 0; i < preferenceRanks[friendX][friendY]; ++i) {
            const preferredFriend = preferences[friendX][i];
            // If preferredFriend prefers friendX over their current partner, friendX is unhappy
            if (preferenceRanks[preferredFriend][friendX] < preferenceRanks[preferredFriend][pairPartners[preferredFriend]]) {
                isUnhappy = true;
                break; // No need to check the rest since friendX is already unhappy
            }
        }

        // Increment the unhappy count if friendX is found to be unhappy
        if (isUnhappy) {
            unhappyCount++;
        }
    }

    return unhappyCount; // Return the final count of unhappy friends
}

```

Time and Space Complexity

Time Complexity

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

1. The comprehension used to create the dictionary `d` takes `O(n^2)` time, where `n` is the number of friends, because the preferences list for each friend is of length `n-1`, and there are `n` such lists.
2. The loop setting up the `p` dictionary runs for `O(n/2)` pairs, which simplifies to `O(n)` because each pair is processed exactly once.
3. The outer loop to calculate `ans` traverses `n` friends, which gives us `O(n)`.
4. Inside this outer loop, there's an `any` function call on a generator expression. In the worst case, it scans through `n-2` elements (since one element is the friend themselves, and another is the paired friend). Therefore, this gives us `O(n-2)` for each friend.

When you put these factors together, the worst-case time complexity is `O(n^2)` due to the initial comprehension for the dictionary `d`. All the other operations, although they depend on `n`, don't involve nested loops over `n`, so they don't contribute a higher order term than `n^2`.

Thus, the final time complexity is `O(n^2)`.

Space Complexity

Now let's analyze the space complexity:

1. The `d` dictionary stores preferences for each of the `n` friends using a dictionary, so it takes `O(n^2)` space.
2. The `p` dictionary holds the paired friends' information, with `n` entries (two entries for each pair). Thus, it consumes `O(n)` space.
3. The space for `ans` and loop variables is constant, `O(1)`.

Adding these together, the dominant term is the `O(n^2)` space required for the `d` dictionary. Thus, the total space complexity is also `O(n^2)`.