1583. Count Unhappy Friends

Medium Array Simulation

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

Leetcode Link

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

Problem Description

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.

partners and hence have the potential to cause unhappiness.

that maps the friend's ID p to its preference rank i.

The intuition behind the solution involves the following key steps:

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)

1. Convert Preferences to Indexes: To expedite lookup operations, it is advantageous to convert each friend's preferences list,

- 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

Solution Approach

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

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

This list comprehension iterates over each friend's preferences list v, and for each friend's preferences, it generates a dictionary

2 for x, y in pairs:

p[x] = y

1 ans = 0

2 for x in range(n):

unnecessary comparison checks.

preference listings, as well as their current pairings.

To solve this problem with the proposed solution approach:

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

and in a manner that is easy to follow and implement.

y = p[x]

structure that allows for O(1) time lookup to find out how much i prefers p over others. 1 d = [{p: i for i, p in enumerate(v)} for v in preferences]

friend's preferences with preference rankings as values. This transforms a list of preferred friends for each person into a

- 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. 1 p = {}

ans += any(d[u][x] < d[u][p[u]] for u in preferences[x][: d[x][y]])

checks if there's any such u that satisfies this condition. If yes, x is unhappy.

- 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.
- 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

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

```
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
```

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]

1 return ans

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

3 is not unhappy.

Python Solution

class Solution:

9

10

13

14

15

16

17

18

20

21

23

24

26

27

33

34

35

36

37

from typing import List

paired_friends = {}

paired_friends[x] = y

paired_friends[y] = x

y = paired_friends[x]

is_unhappy = any(

if is_unhappy:

return unhappy_count

Initialize the count of unhappy friends.

The paired friend of `x`.

unhappy_count += 1

for x, y in pairs:

unhappy_count = 0

for x in range(n):

Friend 3 prefers: [0, 1, 2]

And they are paired like this:

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}

```
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

Check friend 1: Paired with friend 0. Friend 1 prefers friend 2 over 0 and friend 2 also prefers friend 1 over their partner

Check friend 0: Paired with friend 1. Friend 0 prefers friend 2 over 1 (from their preference list), but friend 2, who is paired

By following the outlined solution steps, we efficiently find the total number of unhappy friends without unnecessary comparisons

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

Create a dictionary to store each person's paired friend.

Iterate through each person to determine if they are unhappy.

rankings in the subset of preferences before 'y'.

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

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

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

with friend 3, prefers friend 1 over 0. Therefore, friend 0 is not unhappy.

3. Now we determine if there are any unhappy friends. We proceed as follows:

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]

28 preference_rankings[u][x] < preference_rankings[u][paired_friends[u]]</pre> for u in preferences[x][: preference_rankings[x][y]] 29 30 31 32 # If such a person 'u' exists, increment the unhappy count.

Java Solution

C++ Solution

class Solution {

// Function to calculate the number of unhappy friends

int preferenceRank[n][n];

for (int i = 0; i < n; ++i) {

for (auto& pair : pairs) {

int pairPartner[n];

// Declare a 2D array to store the preference rank

// Array to store the pair partner for each person

// Initialize the preference ranks for each friend

// Assign the pair partners based on the given pairs

int friendOne = pair[0], friendTwo = pair[1];

int unhappyCount = 0; // Counter for unhappy friends

// Iterate through each friend to check unhappiness

for (int friendX = 0; friendX < n; ++friendX) {</pre>

preferenceRank[i][preferences[i][j]] = j;

for (int j = 0; j < n - 1; ++j) {

pairPartner[friendOne] = friendTwo;

pairPartner[friendTwo] = friendOne;

int unhappyFriends(int n, vector<vector<int>>& preferences, vector<vector<int>>& pairs) {

int friendY = pairPartner[friendX]; // Partner of the current friend

// Check whether there is a person that friendX prefers over friendY

for (int i = 0; i < preferenceRank[friendX][friendY]; ++i) {</pre>

int preferredFriend = preferences[friendX][i];

bool isUnhappy = false; // Flag to check if the current friend is unhappy

// If preferredFriend prefers friendX over their current partner, friendX is unhappy

if (preferenceRank[preferredFriend][friendX] < preferenceRank[preferredFriend][pairPartner[preferredFriend]]) {</pre>

2 public:

9

10

11

13

14

15

16

17

19

20

21

22

23

24

25

26

27

28

29

30

31

32

34

35

```
1 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;
9
10
11
12
           // Pairing array where the index is the friend and the value is their pair
13
           int[] pairings = new int[n];
           // Fill the pairings array based on the given pairs
14
15
            for (int[] pair : pairs) {
                int friend1 = pair[0], friend2 = pair[1];
16
                pairings[friend1] = friend2;
                pairings[friend2] = friend1;
18
19
20
21
           // Counter for unhappy friends
22
           int unhappyCount = 0;
           // Iterate over all friends to determine unhappiness
            for (int friendX = 0; friendX < n; ++friendX) {</pre>
24
25
                int friendY = pairings[friendX];
                boolean isUnhappy = false; // Flag to check if the current friend is unhappy
26
27
28
               // Check if there exists a friend that friendX ranks higher than their current paired friendY
                for (int i = 0; i < preferenceDistances[friendX][friendY]; ++i) {</pre>
                    int otherFriend = preferences[friendX][i];
30
                    // Check if the other friend (u) prefers friendX over their own pairing
31
32
                    if (preferenceDistances[otherFriend][friendX] < preferenceDistances[otherFriend][pairings[otherFriend]]) {</pre>
33
                        isUnhappy = true;
34
                        break;
35
36
               // Increment unhappyCount if friendX is found to be unhappy
37
               if (isUnhappy) {
38
39
                    unhappyCount++;
40
41
           return unhappyCount; // Return the total number of unhappy friends
43
45
```

48 49 50 }; 51

```
36
                       isUnhappy = true;
                       break; // No need to check the rest since friendX is already unhappy
37
38
39
40
               // If friendX is found to be unhappy, increment the unhappy count
               if (isUnhappy) {
                   unhappyCount++;
44
45
46
           // Return the final count of unhappy friends
           return unhappyCount;
Typescript Solution
    // 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;
 10
 11
 12
         return preferenceRanks;
 13
 14
 15 // Function to assign pair partners based on the given pairs
    function assignPairPartners(pairs: Preferences): number[] {
         const pairPartners: number[] = Array(pairs.length * 2);
 17
 18
         for (const pair of pairs) {
 19
             const [friendOne, friendTwo] = pair;
 20
             pairPartners[friendOne] = friendTwo;
 21
             pairPartners[friendTwo] = friendOne;
 22
 23
         return pairPartners;
 24 }
 25
    // Function to calculate the number of unhappy friends
     function unhappyFriends(n: number, preferences: Preferences, pairs: Preferences): number {
 28
         const preferenceRanks = initializePreferenceRanks(n, preferences);
         const pairPartners = assignPairPartners(pairs);
 29
         let unhappyCount = 0; // Counter for unhappy friends
 30
 31
 32
         for (let friendX = 0; friendX < n; ++friendX) {</pre>
 33
             const friendY = pairPartners[friendX]; // Partner of the current friend
 34
             let isUnhappy = false; // Flag to check if the current friend is unhappy
 35
 36
             // Check whether there is a person that friendX prefers over friendY
 37
             for (let i = 0; i < preferenceRanks[friendX][friendY]; ++i) {</pre>
```

48 unhappyCount++; 49 50 51 52 return unhappyCount; // Return the final count of unhappy friends

if (isUnhappy) {

Time and Space Complexity

38

39

40

41

42

43

44

45

46

47

53 }

54

1. The comprehension used to create the dictionary d takes 0(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.

Space Complexity

O(n^2).

Now let's analyze the space complexity:

Time Complexity

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

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

const preferredFriend = preferences[friendX][i];

// Increment the unhappy count if friendX is found to be unhappy

isUnhappy = true;

// If preferredFriend prefers friendX over their current partner, friendX is unhappy

break; // No need to check the rest since friendX is already unhappy

if (preferenceRanks[preferredFriend][friendX] < preferenceRanks[preferredFriend][pairPartners[preferredFriend]]) {</pre>

(since one element is the friend themselves, and another is the paired friend). Therefore, this gives us 0(n-2) for each friend. When you put these factors together, the worst-case time complexity is 0(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

2. The loop setting up the p dictionary runs for 0(n/2) pairs, which simplifies to 0(n) because each pair is processed exactly once.

- than n^2. Thus, the final time complexity is $O(n^2)$.
 - 1. The d dictionary stores preferences for each of the n friends using a dictionary, so it takes 0(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, 0(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