1452. People Whose List of Favorite Companies Is Not a Subset of Another List

Medium Array Hash Table String Leetcode Link

Problem Description

In this problem, we are given an array `favoriteCompanies` where `favoriteCompanies[i]` represents the list of favorite companies for the `ith` person, with indexing starting from 0. The task is to find out which individuals have a list of favorite companies that is unique and not simply a subset of any other person's list of favorite companies. In other words, we need to identify people with favorite company lists that contain at least one company not listed in any other person's favorite list. The output should be the list of indices of such people, sorted in increasing order.

Intuition

The intuition behind the solution involves understanding the concept of subsets. If a list of favorite companies for a person is a subset of another person's list, it means all companies liked by the first person are also liked by the second person. We want to find those people whose lists are not entirely contained within another's list.

To achieve this, we can use a set data structure to efficiently test if one set is a subset of another. The basic approach to solve this problem goes as follows:

1. Create a hash map (dictionary in Python) to hold a unique index for each company. This is a way of converting the company names from strings to integers, thus enabling us to perform set operations more efficiently.
2. Convert each person's favorite company list into a set of integers using the hash map created in the previous step. In the process, we map each company to its unique integer index and construct a set for each person. This step is crucial because it allows us to use fast set operations like union, intersection, and difference.
3. Iterate over each person's set and compare it with every other person's set to check whether it's not a subset of any other. During iteration, if we find that person `i`'s set is not a subset of any other person `j`'s set (`i != j`), we consider it as a unique list.
4. Whenever we find such a unique list, we record the index of that person. After processing all people, we should have the indices of those with unique favorite company lists.
5. Return the list of indices that we have recorded.

This algorithm shifts the focus from working with strings to working with integers and sets, which is typically more efficient and simpler. Additionally, by early exiting the inner loop when we find a subset, we save time that would otherwise be wasted on unnecessary comparisons.

Solution Approach

The Reference Solution Approach provided above offers a clear pathway for implementing a solution to the problem. Here's a step-by-step explanation of the solution:

1. **Hash Map Creation**: The algorithm begins by creating a hash map named `d` which maps every company to a unique integer. This allows for efficient set operations later on. The hash map is populated as follows:
   - Iterate over every person's favorite company list.
   - For every company not already in the hash map, add it with an incrementing index (`idx`).
2. **Conversion to Integer Sets**: For every person, their favorite companies list is then converted into a set of integers, `t`. The conversion process involves:
   - For each company in the person's list, find its corresponding index in the hash map `d`.
   - Construct a set of these indices.
3. **Unique Lists Identification**: With the integer sets prepared, the next step is to determine which sets are unique, i.e., not subsets of any other sets. This is done by:
   - Iterating over every set (`nums1`) in `t`.
   - Checking `nums1` against every other set (`nums2`) in `t`.
   - Using set difference (`nums1 - nums2`), determine if `nums1` is a subset of `nums2`. If `nums1` is not a subset of any `nums2` (ignoring when `i == j`, which is self-comparison), it is considered unique.
4. **Result Construction**: If a set is found to be unique, the index of that set (representing the person) is added to the answer list, `ans`. This is the list of integers to be returned.
5. **Returning the Output**: Return the answer list, `ans`, which contains the indices of persons with unique favorite companies lists in increasing order.

The Python code provided uses the built-in set data structure, which is highly optimized for operations like union, intersection, and difference. This approach is efficient both in terms of time and space complexity. The critical optimization here is the usage of the set difference operation to quickly check if one set is a subset of another, thereby eliminating the need for a more cumbersome element-by-element comparison.

Example Walkthrough

Let's consider a small example to illustrate the solution approach with the following array of favorite companies for 3 different people:

```
favoriteCompanies = [
    ["Apple", "Google"],  // Person 0
    ["Google", "Amazon"], // Person 1
    ["Apple", "Netflix"]  // Person 2
]
```

Here, we want to find out which person has a unique list of favorite companies that is not simply a subset of any other person's list of favorite companies. Now let's apply the provided solution approach to this example:

1. **Hash Map Creation**:
   - We loop through each person's list and create a hash map of companies to unique integers:
     ```
     d = {"Apple": 0, "Google": 1, "Amazon": 2, "Netflix": 3}
     ```
   - The index `idx` is incremented every time we add a new company to the hash map.
2. **Conversion to Integer Sets**:
   - Convert the favorite companies lists into sets of integers using the hash map `d`:
     ```
     t = [
       {0, 1},  // Person 0's companies converted to integers
       {1, 2},  // Person 1's companies converted to integers
       {0, 3}   // Person 2's companies converted to integers
     ]
     ```
3. **Unique Lists Identification**:
   - Now we check if any set is a subset of another set.
   - For Person 0, `{0, 1}` is not a subset of `{1, 2}` or `{0, 3}`.
   - For Person 1, `{1, 2}` is not a subset of `{0, 1}` or `{0, 3}`.
   - For Person 2, `{0, 3}` is not a subset of `{0, 1}` or `{1, 2}`.
   - Therefore, each person has at least one company that is not included in any other person's list, indicating all sets are unique.
4. **Result Construction**:
   - Since all people have unique sets of favorite companies, we add all their indices to the list `ans`:
     ```
     ans = [0, 1, 2]
     ```
5. **Returning the Output**:
   - The final output `ans` is already sorted in this case, so we can directly return it as the list of people with unique favorite companies:
     ```
     return [0, 1, 2]
     ```

Following the above steps with our example data, we determine that each person has a unique set of favorite companies. The corresponding Python code would identify all individuals in `favoriteCompanies` as having a unique list and return their indices `[0, 1, 2]`.

Python Solution

```python
from typing import List

class Solution:
    def peopleIndexes(self, favorite_companies: List[List[str]]) -> List[int]:
        # Create a dictionary to map company names to unique index numbers.
        company_to_index = {}
        index_counter = 0

        # Create a list to hold sets of unique company indices for each person.
        people_company_indices = []

        # Convert each person's list of favorite companies to a set of unique index numbers.
        for companies in favorite_companies:
            for company in companies:
                # If company is not in the dictionary, add it with a new index.
                if company not in company_to_index:
                    company_to_index[company] = index_counter
                    index_counter += 1
            # Add the set of indices for the current person's favorite companies to the list.
            people_company_indices.append({company_to_index[company] for company in companies})

        # Initialize a list to hold the indexes of people with unique company lists.
        unique_people_indexes = []

        # Check each person's favorite companies against all others' to determine uniqueness.
        for i, person_companies_indices in enumerate(people_company_indices):
            unique = True
            for j, other_person_companies_indices in enumerate(people_company_indices):
                # Skip comparison with itself.
                if i == j:
                    continue
                # If the current person's companies are a subset of another's, set unique to false.
                if not (person_companies_indices - other_person_companies_indices):
                    unique = False
                    break
            # If unique is still true, add the current person's index to the result list.
            if unique:
                unique_people_indexes.append(i)

        # Return the list of indexes of people with unique lists of favorite companies.
        return unique_people_indexes

```

Java Solution

```java
class Solution {
    public List<Integer> peopleIndexes(List<List<String>> favoriteCompanies) {
        // Dictionary to map company names to unique indices
        Map<String, Integer> companyIndexMap = new HashMap<>();
        int index = 0; // Running index for assigning to companies
        int peopleCount = favoriteCompanies.size(); // Total number of people
        Set<Integer>[] companySets = new Set[peopleCount]; // Array of sets to hold the indices of companies

        // Step 1: Assign unique index to each company and create sets of company indices per person
        for (int i = 0; i < peopleCount; ++i) {
            List<String> companies = favoriteCompanies.get(i); // Get the favorite companies of person i
            // Map each company to a unique index if not already done
            for (String company : companies) {
                if (!companyIndexMap.containsKey(company)) {
                    companyIndexMap.put(company, index++);
                }
            }
            // Create a set of indices for the person's favorite companies
            Set<Integer> companySet = new HashSet<>();
            for (String company : companies) {
                companySet.add(companyIndexMap.get(company));
            }
            companySets[i] = companySet;
        }

        // Step 2: Find people whose list of favorite companies is not a subset of any other list
        List<Integer> result = new ArrayList<>();
        for (int i = 0; i < peopleCount; ++i) {
            boolean isUnique = true;
            // Check if there is any other person j whose favorite companies include all of i's favorite companies
            for (int j = 0; j < peopleCount; ++j) {
                if (i != j && companySets[j].containsAll(companySets[i])) {
                    isUnique = false; // Person i's list is not unique, as it is a subset of person j's list
                    break;
                }
            }
            // If person i's list is unique, add their index to the result list
            if (isUnique) {
                result.add(i);
            }
        }

        return result;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <unordered_map>
#include <unordered_set>

class Solution {
public:
    // Function to find the indexes of people whose list of favorite companies is not a
    // subset of any other list of favorite companies.
    vector<int> peopleIndexes(vector<vector<string>>& favoriteCompanies) {
        unordered_map<string, int> companyToIndex;
        int index = 0;
        int numberOfPeople = favoriteCompanies.size();
        vector<unordered_set<int>> transformedCompanies(numberOfPeople);

        // Assign an index to each company and transform the favorite companies into a set of unique indexes.
        for (int i = 0; i < numberOfPeople; ++i) {
            for (auto& company : favoriteCompanies[i]) {
                if (!companyToIndex.count(company)) {
                    companyToIndex[company] = index++;
                }
            }
            unordered_set<int> companyIndexes;
            for (auto& company : favoriteCompanies[i]) {
                companyIndexes.insert(companyToIndex[company]);
            }
            transformedCompanies[i] = companyIndexes;
        }

        vector<int> result;
        // Check each person's list of companies to ensure it is not a subset of another list.
        for (int i = 0; i < numberOfPeople; ++i) {
            bool isSubset = false;
            for (int j = 0; j < numberOfPeople; ++j) {
                if (i == j) continue; // Skip comparison with the same person.
                if (isSubsetCheck(transformedCompanies[i], transformedCompanies[j])) {
                    isSubset = true;
                    break; // The current list is a subset of another list, no need to continue checking.
                }
            }
            if (!isSubset) {
                result.push_back(i); // If the list is not a subset, include the person's index in the result.
            }
        }
        return result;
    }

private:
    // Helper function to check if set nums1 is a subset of set nums2.
    bool isSubsetCheck(const unordered_set<int>& nums1, const unordered_set<int>& nums2) {
        for (int value : nums1) {
            if (!nums2.count(value)) {
                return false; // nums1 is not a subset of nums2 since an element is missing in nums2.
            }
        }
        return true; // All elements in nums1 are found in nums2; nums1 is a subset of nums2.
    }
};

```

Typescript Solution

```typescript
type CompanyIndexMap = { [key: string]: number };
type CompanySet = Set<number>;

// Maps a company to a unique index.
let companyToIndex: CompanyIndexMap = {};
// Transformed favorite companies represented as sets of unique indexes.
let transformedCompanies: CompanySet[] = [];

// Function to find the indexes of people whose list of favorite companies is not a
// subset of any other list of favorite companies.
function peopleIndexes(favoriteCompanies: string[][]): number[] {
    let index = 0;
    let numberOfPeople = favoriteCompanies.length;
    transformedCompanies = new Array(numberOfPeople);

    // Assign an index to each company and transform the favorite companies into a set of unique indexes.
    for (let i = 0; i < numberOfPeople; ++i) {
        for (const company of favoriteCompanies[i]) {
            if (companyToIndex[company] === undefined) {
                companyToIndex[company] = index++;
            }
        }
        let companyIndexes = new Set<number>();
        for (const company of favoriteCompanies[i]) {
            companyIndexes.add(companyToIndex[company]);
        }
        transformedCompanies[i] = companyIndexes;
    }

    let result: number[] = [];
    // Check each person's list of companies to ensure it is not a subset of another list.
    for (let i = 0; i < numberOfPeople; ++i) {
        let isSubset = false;
        for (let j = 0; j < numberOfPeople; ++j) {
            if (i === j) continue; // Skip comparison with the same person.
            if (isSubsetCheck(transformedCompanies[i], transformedCompanies[j])) {
                isSubset = true;
                break; // The current list is a subset of another list, no need to continue checking.
            }
        }
        if (!isSubset) {
            result.push(i); // If the list is not a subset, include the person's index in the result.
        }
    }
    return result;
}

// Helper function to check if set A is a subset of set B.
function isSubsetCheck(setA: CompanySet, setB: CompanySet): boolean {
    for (let value of setA) {
        if (!setB.has(value)) {
            return false; // Set A is not a subset of set B since an element is missing in set B.
        }
    }
    return true; // All elements in set A are found in set B; set A is a subset of set B.
}

```

Time and Space Complexity

Time Complexity

The provided code involves several nested loops and set operations, which contribute to its overall time complexity. Here's a breakdown of the main components:

1. Building the dictionary `d`:
   - The outer loop goes through each list of companies, and the inner loop goes through each company within a list.
   - Worst case time complexity for building `d` is `O(M*N)`, where `N` is the number of lists in `favoriteCompanies`, and `M` is the average number of companies in each list.
2. Constructing the set `t`:
   - This part also involves an outer loop through each list and an inner loop through each company.
   - The construction of sets is `O(M)` per list, leading to `O(M*N)` time complexity for this step.
3. The comparison loops to determine if one set is a subset of another:
   - There are two nested loops, each going through the `N` sets.
   - For each pair of sets, the subset check operation involves a difference operation which can take up to `O(M)` in the worst case.
   - Therefore, this part has `O(N^2 * M)` time complexity.

Combining all these, the overall time complexity is dominated by the subset check, giving `O(N^2 * M)`.

Space Complexity

The space complexity consists of the storage for the dictionary `d`, the list of sets `t`, and the answer list `ans`:

1. The dictionary `d` has a space complexity of `O(U)`, where `U` is the total number of unique companies across all lists.
2. The list of sets `t` stores each company as an integer index, so each set takes `O(M)` space, and for `N` lists, this is `O(M*N)` space.
3. The answers list `ans` can contain at most `N` elements, which gives `O(N)`.

Given that `U` may be less than `M*N` (since some companies could be repeated), the overall space complexity is `O(M*N)` due to storing sets corresponding to the list of lists `favoriteCompanies`.