599. Minimum Index Sum of Two Lists

Easy Array Hash Table String

Leetcode Link

Problem Description

The problem involves two lists of strings, list1 and list2. Our goal is to identify strings that appear in both lists (common strings) and among those, we want to find the ones that have the smallest index sum. The index sum for a common string is defined as the sum of its indexes in list1 and list2, i.e., if a string is at position `i` in list1 and at position `j` in list2, then its index sum is `i + j`. If there are multiple common strings with the same minimum index sum, we need to return all of them in any order.

This is a practical problem that could be applied to real-world scenarios where we are interested in finding common elements between two sets but prioritizing those that rank higher in both sets. For instance, in recommendation systems where the two lists represent preferences of two different users, and we want to find a common ground that considers the most preferred options by both.

Intuition

The solution is based on the idea of mapping one list into a dictionary (in this case, list2) to allow efficient lookups while iterating over the other list (list1). By maintaining this dictionary, we can check if an element from list1 appears in list2 in constant time rather than linear time, which would be the case if we were to scan list2 repeatedly.

As we want to find the least index sum, we start by assuming a high index sum (e.g., 2000) as the initial minimum `mi`. While iterating through list1, we check for common strings using the previously mentioned dictionary. When we find a common string, we calculate the index sum `t` and compare it to our current minimum index sum `mi`:

- If `t` is less than `mi`, it means we've found a new common string with a smaller index sum. We then update `mi` to this smaller value and start our answer list `ans` afresh with this string.
- If `t` is equal to the current minimum `mi`, it signifies that we've found another common string with the same lowest index sum. Therefore, we add this string to our `ans` list without resetting it.

This way, at the end of the iteration, the `ans` list contains all the common strings with the minimum index sum, and we return this list as the result.

Solution Approach

The solution provided is a practical combination of a hash map (referred to as a dictionary in Python) and a simple linear scan of the two lists. Here's a step-by-step breakdown of the implemented algorithm using the provided code:

1. **Create a dictionary for efficient lookups**: A dictionary `mp` is constructed to map the elements of `list2` to their indexes. This allows for constant-time checks to see if an element from `list1` also exists in `list2`, and if it does, we can instantly obtain its index in `list2`.

```
1 mp = {v: i for i, v in enumerate(list2)}
```

2. **Initialize variables**: A variable `mi` is initialized to a large number (2000 in this case, assuming that the sum of indexes will not exceed this number) to represent the smallest index sum we have found so far. An empty list `ans` is also initialized to store the answer strings.

3. **Iterate through the first list (`list1`)**: We use a loop to go through each item `v` in `list1` and its associated index `i`.

```
1 for i, v in enumerate(list1):
2     ...
```

4. **Check for common elements and calculate index sums**: Within the loop, we check if element `v` is in the dictionary `mp`. If it is a common element, we calculate the total index sum `t` for that element.

```
1 if v in mp:
2     t = i + mp[v]
```

5. **Compare the index sum with the minimum**: For each common element found, we compare the index sum `t` against the current minimum `mi`.

   - If `t` is smaller than `mi`, this means we've found a new common element with a smaller index sum. We update `mi` to the new smaller index sum and reset the `ans` array to only include this element.

```
1 if t < mi:
2     mi = t
3     ans = [v]
```

   - If `t` is equal to the current minimum (`mi`), this means we have found another common element with the same least index sum. We simply add this element to the `ans` array.

```
1 elif t == mi:
2     ans.append(v)
```

6. **Return the result**: Once the loop is done, `ans` contains all the common elements with the least index sum. We return this list as the final answer.

The core algorithm can be seen as a "brute-force" approach improved with a hash map to avoid an unneeded second loop through `list2`. By only needing to iterate through each list once, the algorithm achieves a time complexity of O(n + m) where `n` is the length of `list1` and `m` is the length of `list2`. If we used nested loops without a hash map, the time complexity would have been O(n * m), significantly slower for larger lists.

Example Walkthrough

Let's go through a small example to illustrate the solution approach. Assume our inputs are as follows:

```
1 list1 = ["pineapple", "apple", "banana", "cherry"]
2 list2 = ["berry", "apple", "cherry", "banana"]
```

Our goal is to find the common strings with the smallest index sum. Following our solution approach:

1. **Create a dictionary for efficient lookups**: We start by mapping `list2` to a dictionary `mp` where each value is associated with its index.

```
1 mp = {"berry":0, "apple":1, "cherry":2, "banana":3}
```

2. **Initialize variables**: Set `mi` to a large number, let's say `mi = 2000`, and `ans = []`.

3. **Iterate through list1**:

```
1 for i, v in enumerate(list1):
```

   At this step, the loop will go through "pineapple", "apple", "banana", "cherry" at indices 0, 1, 2, 3 respectively.

4. **Check for common elements and calculate index sums**: When we reach "apple", which is at index 1 in `list1`, we find it in `mp` (it's at index 1 in `list2` as well).

```
1 if v in mp:  # v is "apple"
2     t = i + mp[v]  # t is 1 + 1 = 2
```

   Since "apple" is common, we calculate index sum `t = 2`.

5. **Compare the index sum with the minimum**: Since `t` (2) is less than `mi` (2000), we update `mi` to 2, and our answer list now contains "apple".

```
1 if t < mi:
2     mi = t  # mi is now 2
3     ans = [v]  # ans is now ["apple"]
```

   Continuing the iteration, we eventually come across "banana" at index 2 in `list1` and at index 3 in `list2`. The index sum `t` is 5, which is greater than `mi`, so we don't update `mi` or `ans`.

   Next, for "cherry" at index 3 in `list1` and index 2 in `list2`, the index sum `t` is 5 again, still greater than `mi`, so we still don't update `mi` or `ans`.

6. **Return the result**: At the end of the loop, `ans` contains ["apple"], which is the common element with the smallest index sum. This is our final answer.

And that is a direct application of the described algorithm, demonstrating with our small example that "apple" is common to both lists and has the smallest index sum of 2.

Python Solution

```
1  class Solution:
2      def findRestaurant(self, list1: List[str], list2: List[str]) -> List[str]:
3          # Initialize an empty list to store the common favorite restaurant(s) with least index sum
4          common_favorites = []
5
6          # Create a dictionary to map each restaurant in the second list to its index
7          index_map = {restaurant: index for index, restaurant in enumerate(list2)}
8
9          # Initialize variable to store the minimum index sum; value set higher than possible index sum
10         min_sum = len(list1) + len(list2)
11
12         # Iterate through the first list to find common restaurants and calculate index sum
13         for index1, restaurant in enumerate(list1):
14             if restaurant in index_map:
15                 # Calculate the index sum for the current common restaurant
16                 current_sum = index1 + index_map[restaurant]
17
18                 # If the current index sum is less than the smallest found min_sum
19                 if current_sum < min_sum:
20                     min_sum = current_sum
21                     # Update the common_favorites list to only include the current restaurant
22                     common_favorites = [restaurant]
23
24                 # If the current index sum is the same as the smallest found min_sum
25                 elif current_sum == min_sum:
26                     # Add the current restaurant to the common_favorites list
27                     common_favorites.append(restaurant)
28
29         # Return the list of common favorite restaurant(s) with the least index sum
30         return common_favorites
31
```

Java Solution

```
1  class Solution {
2      public String[] findRestaurant(String[] list1, String[] list2) {
3          // Create a map to store the restaurants from list2 with their indices.
4          Map<String, Integer> restaurantIndexMap = new HashMap<>();
5          // Populate the map with the restaurants from list2.
6          for (int i = 0; i < list2.length; ++i) {
7              restaurantIndexMap.put(list2[i], i);
8          }
9
10         // Create a list to store the answer.
11         List<String> commonRestaurants = new ArrayList<>();
12         // Initialize minimum index sum with a large number.
13         int minIndexSum = 2000;
14
15         // Iterate through the list1 to find common restaurants with minimum index sum.
16         for (int i = 0; i < list1.length; ++i) {
17             // If the current restaurant is in list2,
18             if (restaurantIndexMap.containsKey(list1[i])) {
19                 // Calculate the index sum.
20                 int currentIndexSum = i + restaurantIndexMap.get(list1[i]);
21                 // If the index sum is smaller than the minimum found so far,
22                 if (currentIndexSum < minIndexSum) {
23                     // Start a new list as we found a restaurant with a smaller index sum.
24                     commonRestaurants = new ArrayList<>();
25                     // Add this restaurant to the list.
26                     commonRestaurants.add(list1[i]);
27                     // Update the minimum index sum.
28                     minIndexSum = currentIndexSum;
29                 } else if (currentIndexSum == minIndexSum) {
30                     // If the index sum is equal to the current minimum, add the restaurant to the list.
31                     commonRestaurants.add(list1[i]);
32                 }
33             }
34         }
35
36         // Return the list as an array.
37         return commonRestaurants.toArray(new String[0]);
38     }
39 }
40
```

C++ Solution

```
1  #include <vector>
2  #include <string>
3  #include <unordered_map>
4  using namespace std;
5
6  class Solution {
7  public:
8      vector<string> findRestaurant(vector<string>& list1, vector<string>& list2) {
9          // Create a hash map to store restaurant names and their indices from list2
10         unordered_map<string, int> index_map;
11         for (int i = 0; i < list2.size(); ++i) {
12             index_map[list2[i]] = i;
13         }
14
15         // Initialize the minimum index sum to a large value
16         int min_index_sum = 2000;
17         // This will hold the answer - the list of restaurants with the minimum index sum
18         vector<string> result;
19
20         // Iterate through list1 and look up each restaurant in the hash map
21         for (int i = 0; i < list1.size(); ++i) {
22             // Check if the current restaurant from list1 exists in list2
23             if (index_map.count(list1[i])) {
24                 // Calculate the total index sum for the current restaurant
25                 int current_index_sum = i + index_map[list1[i]];
26                 // If the current index sum is less than the known minimum
27                 if (current_index_sum < min_index_sum) {
28                     // Clear the result array and start fresh as we found a smaller index sum
29                     result.clear();
30                     // Add the current restaurant to the result
31                     result.push_back(list1[i]);
32                     // Update the minimum index sum
33                     min_index_sum = current_index_sum;
34                 } else if (current_index_sum == min_index_sum) {
35                     // If the current index sum equals the known minimum, add the restaurant to the result array
36                     result.push_back(list1[i]);
37                 }
38             }
39         }
40
41         // Return the list of restaurants with the minimum index sum
42         return result;
43     }
44 };
45
```

Typescript Solution

```
1  function findRestaurant(list1: string[], list2: string[]): string[] {
2      // Initialize the minimum index sum found so far to a very large number
3      let minIndexSum = Infinity;
4      // Initialize an array to hold the final result
5      const commonRestaurants = [];
6      // Create a map to store restaurant names and their index from list1
7      const indexMap = new Map<string, number>(list1.map((item, index) => [item, index]));
8
9      // Iterate over list2 to find common restaurants
10     list2.forEach((item, index) => {
11         // Check if the current restaurant from list2 exists in list1
12         if (indexMap.has(item)) {
13             // Calculate the sum of indices from list1 and list2 for the common restaurant
14             const currentIndexSum = index + indexMap.get(item);
15             // Compare the current index sum with the smallest index sum encountered so far
16             if (currentIndexSum <= minIndexSum) {
17                 // If we found a smaller index sum, we reset the result array
18                 if (currentIndexSum < minIndexSum) {
19                     minIndexSum = currentIndexSum;
20                     commonRestaurants.length = 0;
21                 }
22                 // Add the common restaurant to the result array
23                 commonRestaurants.push(item);
24             }
25         }
26     });
27
28     // Return the array of common restaurants with the smallest index sum
29     return commonRestaurants;
30 }
31
```

Time and Space Complexity

The time complexity of the given code is $O(N + M)$, where N is the length of `list1` and M is the length of `list2`. This is because the code iterates through `list2` once to create a hash map of its elements along with their indices and then iterates through `list1` to check if any element exists in `list2` and perform index summation and comparison. Each of these operations (hash map lookups, summation, and comparison) are $O(1)$, so the overall time complexity is determined by the lengths of the lists.

The space complexity of the code is $O(M)$, where M is the length of `list2`. This is due to the creation of a hash map that might contain all elements of `list2`. The `ans` list has a space complexity of $O(min(N, M))$ in the worst case, where we must store a list of restaurants found in both `list1` and `list2`. However, this does not change the overall space complexity, which is dominated by the size of the hash map.