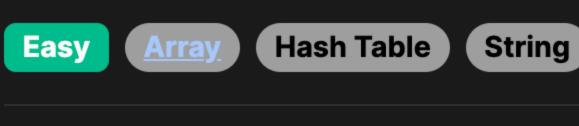
599. Minimum Index Sum of Two Lists



Problem Description

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

The problem involves two lists of strings, list1 and list2. Our goal is to identify strings that appear in both lists (common strings)

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.

over the other list (list1). By maintaining this dictionary, we can check if an element from list1 appears in list2 in constant time

Intuition

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:

rather than linear time, which would be the case if we were to scan list2 repeatedly.

The solution is based on the idea of mapping one list into a dictionary (in this case, list2) to allow efficient lookups while iterating

• 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.

- 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:

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

Initialize variables: A variable mi is initialized to a large number (2000 in this case, assuming that the sum of indexes will not

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

Compare the index sum with the minimum: For each common element found, we compare the index sum t against the

If t is equal to the current minimum (mi), this means we have found another common element with the same least index

index in list2.

if t < mi:</pre>

ans = [v]

elif t == mi:

Example Walkthrough

ans.append(v)

been O(n * m), significantly slower for larger lists.

list1 = ["pineapple", "apple", "banana", "cherry"]

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

list2 = ["berry", "apple", "cherry", "banana"]

for i, v in enumerate(list1):

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

- 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. Iterate through the first list (list1): We use a loop to go through each item v in list1 and its associated index i.
- if v in mp: t = i + mp[v]

```
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.
```

common element, we calculate the total index sum t for that element.

sum. We simply add this element to the ans array.

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

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

Return the result: Once the loop is done, ans contains all the common elements with the least index sum. We return this list

Our goal is to find the common strings with the smallest index sum. Following our solution approach: Create a dictionary for efficient lookups: We start by mapping list2 to a dictionary mp where each value is associated with its index.

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

Check for common elements and calculate index sums: When we reach "apple", which is at index 1 in list1, we find it in mp

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

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,

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

Return the result: At the end of the loop, ans contains ["apple"], which is the common element with the smallest index sum.

for i, v in enumerate(list1):

Iterate through list1:

if v in mp: # v is "apple"

update mi or ans.

Solution Implementation

common_favorites = []

min_sum = len(list1) + len(list2)

if restaurant in index_map:

for index1, restaurant in enumerate(list1):

if current_sum < min_sum:</pre>

min_sum = current_sum

Python

Java

class Solution {

using namespace std;

class Solution:

t = i + mp[v] # t is 1 + 1 = 2

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

which is greater than mi, so we don't update mi or ans.

(it's at index 1 in list2 as well).

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

```
contains "apple".
if t < mi:</pre>
    mi = t \# mi is now 2
    ans = [v] # ans is now ["apple"]
```

def findRestaurant(self, list1: List[str], list2: List[str]) -> List[str]:

Initialize an empty list to store the common favorite restaurant(s) with least index sum

Iterate through the first list to find common restaurants and calculate index sum

If the current index sum is less than the smallest found min_sum

Update the common_favorites list to only include the current restaurant

Calculate the index sum for the current common restaurant

Return the list of common favorite restaurant(s) with the least index sum

// Create a map to store the restaurants from list2 with their indices.

current_sum = index1 + index_map[restaurant]

common_favorites = [restaurant]

public String[] findRestaurant(String[] list1, String[] list2) {

Map<String, Integer> restaurantIndexMap = new HashMap<>();

// Populate the map with the restaurants from list2.

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.

Create a dictionary to map each restaurant in the second list to its index index_map = {restaurant: index for index, restaurant in enumerate(list2)} # Initialize variable to store the minimum index sum; value set higher than possible index sum

```
# If the current index sum is the same as the smallest found min_sum
elif current_sum == min_sum:
    # Add the current restaurant to the common_favorites list
    common_favorites.append(restaurant)
```

return common_favorites

```
for (int i = 0; i < list2.length; ++i) {</pre>
            restaurantIndexMap.put(list2[i], i);
       // Create a list to store the answer.
       List<String> commonRestaurants = new ArrayList<>();
       // Initialize minimum index sum with a large number.
        int minIndexSum = 2000;
        // Iterate through the list1 to find common restaurants with minimum index sum.
        for (int i = 0; i < list1.length; ++i) {</pre>
           // If the current restaurant is in list2,
            if (restaurantIndexMap.containsKey(list1[i])) {
                // Calculate the index sum.
                int currentIndexSum = i + restaurantIndexMap.get(list1[i]);
                // If the index sum is smaller than the minimum found so far,
                if (currentIndexSum < minIndexSum) {</pre>
                    // Start a new list as we found a restaurant with a smaller index sum.
                    commonRestaurants = new ArrayList<>();
                    // Add this restaurant to the list.
                    commonRestaurants.add(list1[i]);
                    // Update the minimum index sum.
                    minIndexSum = currentIndexSum;
                } else if (currentIndexSum == minIndexSum) {
                    // If the index sum is equal to the current minimum, add the restaurant to the list.
                    commonRestaurants.add(list1[i]);
       // Return the list as an array.
        return commonRestaurants.toArray(new String[0]);
C++
#include <vector>
#include <string>
#include <unordered_map>
```

```
class Solution {
public:
    vector<string> findRestaurant(vector<string>& list1, vector<string>& list2) {
        // Create a hash map to store restaurant names and their indices from list2
       unordered_map<string, int> index_map;
        for (int i = 0; i < list2.size(); ++i) {</pre>
            index_map[list2[i]] = i;
       // Initialize the minimum index sum to a large value
        int min index sum = 2000;
        // This will hold the answer — the list of restaurants with the minimum index sum
        vector<string> result;
       // Iterate through list1 and look up each restaurant in the hash map
        for (int i = 0; i < list1.size(); ++i) {</pre>
            // Check if current restaurant from list1 exists in list2
            if (index_map.count(list1[i])) {
                // Calculate the total index sum for the current restaurant
                int current_index_sum = i + index_map[list1[i]];
                // If the current index sum is less than the known minimum
                if (current_index_sum < min_index_sum) {</pre>
                    // Clear the result array and start fresh as we found a smaller index sum
                    result.clear();
                    // Add the current restaurant to the result
                    result.push_back(list1[i]);
                    // Update the minimum index sum
                    min_index_sum = current_index_sum;
                } else if (current_index_sum == min_index_sum) {
                    // If the current index sum equals the known minimum, add the restaurant to the result array
                    result.push_back(list1[i]);
        // Return the list of restaurants with the minimum index sum
        return result;
};
TypeScript
function findRestaurant(list1: string[], list2: string[]): string[] {
    // Initialize the minimum index sum found so far to a very large number
```

```
});
```

let minIndexSum = Infinity;

const commonRestaurants = [];

list2.forEach((item, index) => {

if (indexMap.has(item)) {

// Initialize an array to hold the final result

// Iterate over list2 to find common restaurants

// Create a map to store restaurant names and their index from list1

// Check if the current restaurant from list2 exists in list1

const currentIndexSum = index + indexMap.get(item);

const indexMap = new Map<string, number>(list1.map((item, index) => [item, index]));

// Calculate the sum of indices from list1 and list2 for the common restaurant

// Compare the current index sum with the smallest index sum encountered so far

```
if (currentIndexSum <= minIndexSum) {</pre>
                  // If we found a smaller index sum, we reset the result array
                  if (currentIndexSum < minIndexSum) {</pre>
                      minIndexSum = currentIndexSum;
                      commonRestaurants.length = 0;
                  // Add the common restaurant to the result array
                  commonRestaurants.push(item);
      // Return the array of common restaurants with the smallest index sum
      return commonRestaurants;
class Solution:
   def findRestaurant(self, list1: List[str], list2: List[str]) -> List[str]:
       # Initialize an empty list to store the common favorite restaurant(s) with least index sum
        common_favorites = []
       # Create a dictionary to map each restaurant in the second list to its index
        index_map = {restaurant: index for index, restaurant in enumerate(list2)}
       # Initialize variable to store the minimum index sum; value set higher than possible index sum
       min_sum = len(list1) + len(list2)
       # Iterate through the first list to find common restaurants and calculate index sum
        for index1, restaurant in enumerate(list1):
            if restaurant in index_map:
               # Calculate the index sum for the current common restaurant
                current_sum = index1 + index_map[restaurant]
               # If the current index sum is less than the smallest found min_sum
                if current_sum < min_sum:</pre>
                   min_sum = current_sum
                   # Update the common_favorites list to only include the current restaurant
                    common_favorites = [restaurant]
                # If the current index sum is the same as the smallest found min_sum
               elif current_sum == min_sum:
                    # Add the current restaurant to the common_favorites list
                    common_favorites.append(restaurant)
       # Return the list of common favorite restaurant(s) with the least index sum
        return common_favorites
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 0(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.