2032. Two Out of Three

Easy Array Hash Table

Leetcode Link

Problem Description

The problem gives us three integer arrays named `nums1`, `nums2`, and `nums3`. Our task is to find a distinct array that contains all the unique values which are present in at least two of these arrays. This means if a value is present in either array `nums1` & `nums2`, `nums1` & `nums3`, or `nums2` & `nums3`, it should be included in the final answer. Also, the result should not have any duplicates, and the order of values in the resulting array does not matter.

Intuition

To solve this problem efficiently, we can use the properties of set data structures in Python, which store only unique elements. First, we convert each input array (`nums1`, `nums2`, `nums3`) to a set to eliminate any duplicates within the individual arrays. This step helps to simplify our upcoming operations since sets do not allow duplicate values, and membership checks (to see if a value is present in the set) are done in constant time.

The next step is to iterate through a range of possible values that could be present in the arrays. Since the problem does not specify an upper limit on the values, we can assume that the numbers are within a reasonable domain. In this solution, it is assumed that the values are within the inclusive range of 1 to 100.

For each number in this range, we check if it is present in at least two of the three sets. To do this, we find the sum of the boolean results of the membership tests for the three sets (i.e., `(i in s1) + (i in s2) + (i in s3)`). If a number is present in at least two sets, this sum will be greater than 1. We use a list comprehension to build the final list of numbers that meet this criterion.

This approach is concise and leverages the strengths of Python's data structures to arrive at a solution that is not only efficient but also easy to understand.

Solution Approach

The implementation of the solution is straightforward yet efficient, leveraging the power of Python's set data structures and list comprehensions. Here's a step-by-step breakdown of the algorithm:

1. **Convert Lists to Sets**: Since we are only interested in the presence of values rather than their frequency, the first step is to convert the given lists `nums1`, `nums2`, and `nums3` into sets. This is done using the Python built-in set constructor. By doing so, we remove any duplicates within each of the individual lists, and it also allows us to perform the next steps more efficiently.

```
s1, s2, s3 = set(nums1), set(nums2), set(nums3)
```

2. **Iterate Over a Range of Values**: Given the problem does not provide a specific range for the integer values, the solution assumes a reasonable range from 1 to 100, inclusive. Since there's no information that any value outside this range will appear in the arrays, it is safe to iterate over these 100 numbers.

```
range(1, 101)
```

3. **Check for Presence in Two Sets**: For each value `i` in the range, we check if `i` is present in at least two out of the three sets (`s1`, `s2`, `s3`). This is done using a simple sum of boolean expressions:

```
(i in s1) + (i in s2) + (i in s3) > 1
```

Each expression (`i in s1`), (`i in s2`), or (`i in s3`) returns a Boolean value—`True` if `i` is a member of the set and `False` otherwise. When these are added together, `True` counts as `1` and `False` as `0`. If the sum is greater than 1, it indicates that `i` is present in at least two sets.

4. **List Comprehension to Create the Result**: A list comprehension is used to iterate through the range and apply the check. Only values that pass the check (present in at least two sets) are included in the final list. The beauty of using a list comprehension here is that it returns a new list directly, without having to manually initialize an empty list and append qualifying values one by one.

```
[i for i in range(1, 101) if (i in s1) + (i in s2) + (i in s3) > 1]
```

By incorporating these steps into the provided `twoOutOfThree` method of the `Solution` class, we get a compact yet elegant solution that meets the problem's requirements effectively. This solution approach demonstrates the strength of using sets for membership tests, the efficiency of list comprehensions for building lists, and some basic mathematical operations (like summing boolean values) for conditional checks.

Example Walkthrough

Let's examine the solution approach with a small example. Assume the input arrays are as follows:

```
nums1 = [1, 2, 3]
nums2 = [2, 4]
nums3 = [3, 4, 5]
```

Applying the solution approach step-by-step:

1. **Convert Lists to Sets**: We convert `nums1`, `nums2`, and `nums3` to sets to remove any duplicates and to simplify the upcoming membership checks:

```
s1 = set(nums1) # s1 = {1, 2, 3}
s2 = set(nums2) # s2 = {2, 4}
s3 = set(nums3) # s3 = {3, 4, 5}
```

2. **Iterate Over a Range of Values**: We loop through the range from 1 to 100, as we are assuming a reasonable domain for the numbers:

```
range(1, 101) # Will iterate from 1 to 100 inclusive
```

3. **Check for Presence in Two Sets**: For each number `i` from 1 to 100, we perform the membership tests to see if `i` is present in at least two of the sets `s1`, `s2`, and `s3`. For example:

```
(2 in s1) + (2 in s2) + (2 in s3) # This equals 2, since 2 is in s1 and s2
(4 in s1) + (4 in s2) + (4 in s3) # This equals 2, since 4 is in s2 and s3
```

Only the numbers meeting the condition `> 1` will signify they are present in at least two arrays.

4. **List Comprehension to Create the Result**: We use a list comprehension to go through the range and collect numbers that meet our condition:

```
[i for i in range(1, 101) if (i in s1) + (i in s2) + (i in s3) > 1]
```

Given our example arrays, the final processing would look like this:

```
s1, s2, s3 = set(nums1), set(nums2), set(nums3)
result = [i for i in range(1, 101) if (i in s1) + (i in s2) + (i in s3) > 1]
```

The numbers that are present in at least two of the sets are 2, 3, and 4. Therefore, the `result` would be:

```
result = [2, 3, 4]
```

This small example illustrates how the four steps of the solution approach come together to solve the problem efficiently. Using this method, we quickly find the numbers that are present in at least two of the input arrays and return them as part of the distinct result array.

Python Solution

```
class Solution:
    def twoOutOfThree(self, nums1: List[int], nums2: List[int], nums3: List[int]) -> List[int]:
        # Convert each list into a set to remove duplicates and perform efficient lookups
        set_nums1, set_nums2, set_nums3 = set(nums1), set(nums2), set(nums3)

        # Create a list comprehension to find numbers present in at least two out of the three sets
        # The range 1 to 101 is used based on the problem description, which implies that numbers
        # between 1 and 100 (inclusive) should be considered
        common_numbers = [
            number for number in range(1, 101)
            if (number in set_nums1) + (number in set_nums2) + (number in set_nums3) > 1
        ]

        # Return the list of numbers found in at least two of the three sets
        return common_numbers
```

Java Solution

```
class Solution {
    // Main method that finds the common elements present in at least two out of the three arrays
    public List<Integer> twoOutOfThree(int[] nums1, int[] nums2, int[] nums3) {
        // Get the frequency array for each input array
        int[] frequencyArray1 = getFrequencyArray(nums1);
        int[] frequencyArray2 = getFrequencyArray(nums2);
        int[] frequencyArray3 = getFrequencyArray(nums3);

        // Initialize the list to store the result
        List<Integer> result = new ArrayList<>();

        // Traverse the frequency arrays and check if a number is present in at least two arrays
        for (int i = 1; i <= 100; ++i) {
            // If the sum of frequencies at this index is greater than 1, it's present in at least two arrays
            if (frequencyArray1[i] + frequencyArray2[i] + frequencyArray3[i] > 1) {
                result.add(i); // Add to the result list
            }
        }

        return result; // Return the list of numbers present in at least two out of the three arrays
    }

    // Helper method to create a frequency array for a given input array
    private int[] getFrequencyArray(int[] nums) {
        // Array of size 101, since the range is from 1 to 100 inclusive
        int[] frequency = new int[101];

        // Populate the frequency array; mark 1 for each number that appears in the array
        for (int num : nums) {
            frequency[num] = 1; // Mark the occurrence of 'num' by setting the corresponding index to 1
        }

        return frequency; // Return the frequency array
    }
}
```

C++ Solution

```
#include <vector>

class Solution {
public:
    // Renamed 'get' to a more descriptive function name 'calculateFrequencyCount'.
    // This function takes a vector of integers and returns a frequency count vector
    // where the index corresponds to the integer and the value is 1 if the integer is present.
    vector<int> calculateFrequencyCount(vector<int>& nums) {
        vector<int> count(101, 0); // Initialize frequency count with zeros for up to 100 unique integers.
        for (int num : nums) {
            count[num] = 1; // Mark the presence of an integer with 1.
        }
        return count;
    }

    // This function takes three integer vectors and returns a vector containing
    // integers that appear in at least two of the three input vectors.
    vector<int> twoOutOfThree(vector<int>& nums1, vector<int>& nums2, vector<int>& nums3) {
        // Calculate frequency count for each input vector.
        vector<int> countNums1 = calculateFrequencyCount(nums1);
        vector<int> countNums2 = calculateFrequencyCount(nums2);
        vector<int> countNums3 = calculateFrequencyCount(nums3);

        vector<int> result; // Initialize an empty vector to store the result.

        // Loop through possible integer values (1 to 100).
        for (int i = 1; i <= 100; ++i) {
            // If an integer appears in at least two out of the three vectors,
            // add it to the result vector.
            if (countNums1[i] + countNums2[i] + countNums3[i] > 1) {
                result.emplace_back(i);
            }
        }

        return result; // Return the result vector.
    }
};
```

Typescript Solution

```
function twoOutOfThree(nums1: number[], nums2: number[], nums3: number[]): number[] {
    // Initialize an array 'counts' to store the frequency of each number (0 - 100)
    // assuming the values in the input arrays are within 1 - 100 based on the problem constraints.
    const counts = new Array(101).fill(0);

    // Loop through unique values of nums1 and increment the count
    new Set(nums1).forEach(val => counts[val]++);

    // Loop through unique values of nums2 and increment the count
    new Set(nums2).forEach(val => counts[val]++);

    // Loop through unique values of nums3 and increment the count
    new Set(nums3).forEach(val => counts[val]++);

    // Initialize an array 'result' to store the numbers that appear in at least two arrays
    const result = [];

    // Iterate over 'counts' and push the numbers with a count of 2 or more to 'result'
    counts.forEach((val, idx) => {
        if (val >= 2) {
            result.push(idx);
        }
    });

    // Return the 'result' array containing the numbers that meet the criteria
    return result;
}
```

Time and Space Complexity

The given code aims to find all the numbers present in at least two out of the three input lists, and the solution leverages sets for efficient lookups. Here's the analysis of the time and space complexities:

Time Complexity

1. Converting `nums1`, `nums2`, and `nums3` into sets `s1`, `s2`, and `s3`. This step has a time complexity of `O(n)` for each list, where `n` is the length of the longest list among `nums1`, `nums2`, and `nums3`.
2. Iterating through the range `[1, 101]` and checking the presence of each number in the sets takes `O(100)` time, since it iterates over a fixed range of numbers.
3. The presence checks `i in s1`, `i in s2`, and `i in s3` are `O(1)` operations due to set lookup properties. Since these checks are done for each number in `[1, 101]`, this does not change the overall fixed time iteration.

Considering these steps, the overall time complexity can be approximated to `O(n + 100)`, which simplifies to `O(n)` because, typically, the constant factors are dropped in complexity analysis.

Space Complexity

1. The additional sets `s1`, `s2`, and `s3` each have a maximum space complexity of `O(n)`, assuming all elements are unique in the original lists.
2. The final list comprehension doesn't store more than `100` integers since it's constrained by the range `[1, 101]`, which accounts for `O(1)` space.

Hence, the total space complexity of the code is `O(3n + 1)`, which simplifies to `O(n)` as constant factors and coefficients are omitted in Big O notation.