2215. Find the Difference of Two Arrays

Easy Array Hash Table

Problem Description

The problem provides us with two integer arrays, `nums1` and `nums2`, which are both 0-indexed. The goal is to create a list called `answer` that consists of two sublists:

- `answer[0]` should include all unique integers that are found in `nums1` but not in `nums2`.
- `answer[1]` should include all unique integers that are found in `nums2` but not in `nums1`.

These lists of unique integers should exclude any duplicates and can be returned in any order.

Intuition

To solve this problem, we start by understanding the requirement for unique elements that are present in one array but not the other. This naturally leads us to think of set operations because sets automatically handle uniqueness and provide efficient operations for difference and intersection.

The intuitive approach is to:

- Convert both `nums1` and `nums2` into sets to eliminate any repeated numbers within each array.
- Use set subtraction (`-`) to find elements that are in one set but not the other.

Set subtraction `s1 - s2` yields a set of elements that are in `s1` but not in `s2`. Here are the steps:

1. Convert `nums1` to a set `s1` to obtain unique elements from `nums1`.
2. Convert `nums2` to a set `s2` to get unique elements from `nums2`.
3. Perform `s1 - s2` to find all elements that are in `nums1` but not in `nums2`.
4. Do the reverse and compute `s2 - s1` to find all elements in `nums2` but not in `nums1`.
5. Convert these results back into lists and return them as the `answer` list with two sublists as described above.

Solution Approach

The solution consists of a class `Solution` with a method `findDifference` which takes two lists, `nums1` and `nums2`, and returns a list containing two lists representing the differences between the two original lists. Here is a step-by-step explanation of the implementation:

- First, we need to create two sets: `s1` is created from `nums1` and `s2` is created from `nums2`. Using sets is a critical choice here because:
  - Sets automatically remove duplicate values.
  - They provide efficient operations for set difference (`-`), which is directly needed for this problem.
- `s1` is our set representing all unique elements from `nums1`, and `s2` for `nums2`. Now, we use set subtraction to get the elements that are unique to each set:
  - `s1 - s2` will give us a new set of elements that are present in `s1` (unique to `nums1`) but not in `s2`.
  - `s2 - s1` will give us a new set of elements that are present in `s2` (unique to `nums2`) but not in `s1`.
- Finally, these sets are converted back to lists. This is necessary because the problem asks us to return a list of lists; sets wouldn't be an acceptable return type according to the problem constraints.
- The two lists are then packed into one list and returned. The order in which the lists are returned is as per the problem requirements: the first sublist represents numbers in `nums1` not present in `nums2`, and the second sublist for the opposite.

The algorithm's complexity is beneficial because set operations like the difference are generally O(n), where n is the number of elements in the set. This efficiency comes from the fact that sets are usually implemented as hash tables, allowing for constant-time average performance for add, remove, and check-for-existence operations, which is much faster than trying to perform these operations on an unsorted list.

The code is thus:

```
class Solution:
    def findDifference(self, nums1: List[int], nums2: List[int]) -> List[List[int]]:
        s1, s2 = set(nums1), set(nums2)
        return [list(s1 - s2), list(s2 - s1)]
```

Using sets not only makes the code cleaner and more readable but also ensures the operations are done as efficiently as possible for the problem's requirements.

Example Walkthrough

Let's consider the arrays `nums1 = [1, 2, 3, 3, 4]` and `nums2 = [3, 4, 4, 5, 6]`. We want to use the solution approach to find the differences between these arrays.

Here is the step-by-step walkthrough of the solution:

1. Convert the `nums1` array to a set `s1` to remove duplicates and gain unique elements. The conversion results in `s1 = {1, 2, 3, 4}`.
2. Similarly, convert `nums2` array to a set `s2` yielding `s2 = {3, 4, 5, 6}`.
3. Subtract the set `s2` from `s1` to find elements that are unique to `nums1`. Performing `s1 - s2` gives us `{1, 2}` since numbers 3 and 4 are present in both sets and are hence not part of the set difference.
4. Do the reverse subtraction, `s2 - s1`, to find elements unique to `nums2`. We get `{5, 6}` because again, 3 and 4 are present in both sets.
5. These set differences are converted back to lists: `list(s1 - s2)` becomes `[1, 2]` and `list(s2 - s1)` becomes `[5, 6]`. Note that the order of elements in these lists doesn't matter.
6. Finally, return these lists as sublists in a single list: `[[1, 2], [5, 6]]`.

Using this approach ensures the returned lists have no duplicates and only contain elements that are unique to each of the original lists, `nums1` and `nums2`. The final answer for our example input is therefore `[[1, 2], [5, 6]]`.

Solution Implementation

Python

```
class Solution:
    def findDifference(self, nums1: List[int], nums2: List[int]) -> List[List[int]]:
        # Convert the lists into sets to eliminate any duplicates and allow set operations.
        set_nums1, set_nums2 = set(nums1), set(nums2)

        # Calculate the difference between the two sets.
        # The difference operation (s1 - s2) returns a set with elements in s1 but not in s2.
        differences1 = list(set_nums1 - set_nums2)
        differences2 = list(set_nums2 - set_nums1)

        # Return the differences as a list of lists.
        # The first inner list contains elements unique to nums1.
        # The second inner list contains elements unique to nums2.
        return [differences1, differences2]
```

Java

```
class Solution {

    // This method finds the difference between two integer arrays.
    // It returns a list of lists where the first list contains elements unique to nums1
    // and the second list contains elements unique to nums2.
    public List<List<Integer>> findDifference(int[] nums1, int[] nums2) {
        // Convert both arrays to sets to remove duplicates and allow for O(1) lookups.
        Set<Integer> set1 = convertToSet(nums1);
        Set<Integer> set2 = convertToSet(nums2);

        // Initialize the answer list that will contain two lists.
        List<List<Integer>> answer = new ArrayList<>();
        List<Integer> uniqueToNums1 = new ArrayList<>();
        List<Integer> uniqueToNums2 = new ArrayList<>();

        // Iterate over set1 and add elements not in set2 to uniqueToNums1.
        for (int value : set1) {
            if (!set2.contains(value)) {
                uniqueToNums1.add(value);
            }
        }

        // Iterate over set2 and add elements not in set1 to uniqueToNums2.
        for (int value : set2) {
            if (!set1.contains(value)) {
                uniqueToNums2.add(value);
            }
        }

        // Add both lists to the answer list.
        answer.add(uniqueToNums1);
        answer.add(uniqueToNums2);

        // Return the final list of lists containing unique elements.
        return answer;
    }

    // This method converts an integer array to a set to remove duplicates.
    private Set<Integer> convertToSet(int[] nums) {
        Set<Integer> set = new HashSet<>();
        for (int value : nums) {
            set.add(value);
        }
        return set;
    }
}
```

C++

```
#include <vector>
#include <unordered_set>
using namespace std;

class Solution {
public:
    vector<vector<int>> findDifference(vector<int>& nums1, vector<int>& nums2) {
        // Convert vectors to unordered sets to remove duplicates and for constant time lookups
        unordered_set<int> setNums1(nums1.begin(), nums1.end());
        unordered_set<int> setNums2(nums2.begin(), nums2.end());

        // Initialize a vector of vectors to store the unique elements from each set
        vector<vector<int>> uniqueElements(2);

        // Find the elements unique to setNums1 by checking if they are not in setNums2
        for (int value : setNums1) {
            if (setNum2.count(value) == 0) {
                uniqueElements[0].push_back(value);
            }
        }

        // Find the elements unique to setNums2 by checking if they are not in setNums1
        for (int value : setNums2) {
            if (setNum1.count(value) == 0) {
                uniqueElements[1].push_back(value);
            }
        }

        // Return the vector containing unique elements from both sets
        return uniqueElements;
    }
};
```

TypeScript

```
// Finds the difference between two arrays by giving unique elements in each of them
// that are not present in the other.
// @param {number[]} firstArray - First input array of numbers
// @param {number[]} secondArray - Second input array of numbers
// @return A two-dimensional array where the first subarray contains unique elements
// from `firstArray` not in `secondArray`, and the second subarray contains unique
// elements from `secondArray` not in `firstArray`.
function findDifference(firstArray: number[], secondArray: number[]): number[][] {
    // Filter elements from the first array that are not present in the second array
    // and remove duplicates by converting it to a Set, then spread it back to array.
    const uniqueInFirst = [...new Set<number>(firstArray.filter(value => !secondArray.includes(value)))];

    // Filter elements from the second array that are not present in the first array
    // and remove duplicates by converting it to a Set, then spread it back to array.
    const uniqueInSecond = [...new Set<number>(secondArray.filter(value => !firstArray.includes(value)))];

    // Return the two arrays encapsulated in another array.
    return [uniqueInFirst, uniqueInSecond];
}
```

```
class Solution:
    def findDifference(self, nums1: List[int], nums2: List[int]) -> List[List[int]]:
        # Convert the lists into sets to eliminate any duplicates and allow set operations.
        set_nums1, set_nums2 = set(nums1), set(nums2)

        # Calculate the difference between the two sets.
        # The difference operation (s1 - s2) returns a set with elements in s1 but not in s2.
        differences1 = list(set_nums1 - set_nums2)
        differences2 = list(set_nums2 - set_nums1)

        # Return the differences as a list of lists.
        # The first inner list contains elements unique to nums1.
        # The second inner list contains elements unique to nums2.
        return [differences1, differences2]
```

Time and Space Complexity

The given Python code defines a function `findDifference` that takes two lists of integers, `nums1` and `nums2`, and returns two lists:

- The first list contains all elements that are in `nums1` but not in `nums2`.
- The second list contains all elements that are in `nums2` but not in `nums1`.

To achieve this, the function converts both lists to sets, which are then used to find the difference between them.

Time Complexity

The time complexity of the function is determined by multiple operations:

- Converting `nums1` to a set: $O(n)$ where n is the length of `nums1`.
- Converting `nums2` to a set: $O(m)$ where m is the length of `nums2`.
- Finding the difference `s1 - s2`: This is $O(len(s1))$ because it essentially involves checking each element in `s1` to see if it is not in `s2`.
- Finding the difference `s2 - s1`: Analogously, this is $O(len(s2))$.

Assuming n is the length of `nums1` and m is the length of `nums2`, the overall time complexity is $O(n + m)$ as the set differences are proportional to the size of the sets.

Space Complexity

The space complexity is also determined by multiple factors:

- The space used by `s1`: $O(n)$ where n is the number of unique elements in `nums1`.
- The space used by `s2`: $O(m)$ where m is the number of unique elements in `nums2`.
- The space used by the output lists: This largely depends on how many elements are unique to each list after the set difference operation. However, in the worst case (where all elements are unique), this would again be $O(n + m)$.

The overall space complexity is thus $O(n + m)$ where n is the number of unique elements in `nums1` and m is the number of unique elements in `nums2`.