3002. Maximum Size of a Set After Removals

Medium Topics Array Hash Table

Problem Description

In this problem, you are given two arrays `nums1` and `nums2`, each having an even length `n`. Your task is to remove half of the elements from each array (`n / 2`) from `nums1` and `n / 2` from `nums2`), and then combine the remaining elements into a set `s`. The goal is to maximize the size of the set `s`. Since sets do not allow duplicate values, any repeated values from `nums1` and `nums2` will only appear once in the set `s`. We're looking for the strategy that yields the largest possible set size after these removals and insertions are performed.

Intuition

The key to solving this problem is understanding that if a number is unique to one array and not present in the other, it has to be included in the final set in order to maximize its size. Conversely, common elements shared between both arrays are already ensuring a unique entry in the set `s`, whether they are removed or not. Thus, the focus should be on the exclusive elements in each array first.

To achieve the maximum set size, you should:

1. Convert each array to a set to remove duplicate values within the same array and to have efficient operations for following steps.
2. Calculate the exclusive elements in each array using set difference operations (`s1 - s2` and `s2 - s1`), and find out how many of these exclusive elements can be included in the set without exceeding the limit of `n / 2` removals.
3. The intersection of `s1` and `s2` gives the shared elements, which should also be included in the final set to maximize its size.
4. The maximum possible size of the final set `s` will be the sum of the number of unique elements we can keep from `s1`, the number of unique elements we can keep from `s2`, and the number of shared elements between `nums1` and `nums2`. This number, however, cannot exceed `n` because you are only allowed to keep `n` elements in total from both arrays combined after `n / 2` removals from each.

The provided solution is direct and works efficiently by utilizing set operations in Python to find unique and shared elements and calculating the result accordingly.

Solution Approach

The solution approaches the problem by leveraging Python sets and basic mathematical computations to maximize the size of the combined set. The algorithm follows these steps:

1. **Convert Arrays to Sets**: The first step is to convert both `nums1` and `nums2` arrays into sets `s1` and `s2`. This operation removes any duplicates within individual arrays and allows for set operations to be performed in the next steps.

```
s1 = set(nums1)
s2 = set(nums2)
```

2. **Find Unique Elements**: The solution calculates the number of unique elements in each set that are not present in the other set:

```
unique_to_s1 = s1 - s2
unique_to_s2 = s2 - s1
```

   This is done using the set difference operation (`-`). The length of these unique elements is then limited to `n // 2` because that's the maximum number of elements that can be removed from each array.

```
a = min(len(unique_to_s1), n // 2)
b = min(len(unique_to_s2), n // 2)
```

3. **Compute Shared Elements**: The solution determines the number of shared elements by computing the intersection of `s1` and `s2`.

```
shared_elements = s1 & s2
```

   These shared elements contribute to the set's size because they would be unique in the combined set.

4. **Calculate Maximum Set Size**: Finally, the maximum possible size of the set `s` is the sum of the unique elements that can be included from both `s1` and `s2`, plus the number of shared elements. However, this cannot exceed `n`, which is the total number of elements that can exist in the set after removals.

```
return min(a + b + len(shared_elements), n)
```

The algorithm effectively uses set theory concepts to arrive at an efficient solution. Since operations on sets such as difference and intersection are typically much faster compared to list operations, and considering the limits on the number of elements that can be included after removals, this approach ensures that the resulting set is as large as possible according to the problem's constraints.

Example Walkthrough

Let's illustrate the solution approach with a small example using two arrays `nums1` and `nums2`.

Assume `nums1 = [1, 2, 2, 3]` and `nums2 = [2, 3, 4, 5]`, and they both have a length of 4. That means, we'll need to remove half of the elements from each array (2 elements from each), and then combine the remaining elements to maximize the size of the set `s`.

1. **Convert Arrays to Sets**: By converting `nums1` and `nums2` into sets, we eliminate any duplicates within them and can perform set operations efficiently.

```
s1 = set([1, 2, 2, 3])  # s1 becomes set({1, 2, 3})
s2 = set([2, 3, 4, 5])  # s2 becomes set({2, 3, 4, 5})
```

2. **Find Unique Elements**: We find the elements unique to each set by using set difference operations.

```
unique_to_s1 = s1 - s2  # unique_to_s1 becomes set({1}) as 1 is only in s1
unique_to_s2 = s2 - s1  # unique_to_s2 becomes set({4, 5}) as 4 and 5 are only in s2
```

   Then we limit the number of these unique elements to `n // 2`, which is 2 in this case.

```
a = min(len(unique_to_s1), 2)  # a is 1 because there is only 1 unique element in s1
b = min(len(unique_to_s2), 2)  # b is 2 because there are 2 unique elements in s2, within our limit
```

3. **Compute Shared Elements**: We calculate the shared elements which is the intersection of the two sets.

```
shared_elements = s1 & s2   # shared_elements becomes set({2, 3})
```

4. **Calculate Maximum Set Size**: The maximum possible size of the set `s` is computed by summing up the unique and shared elements.

```
max_set_size = min(a + b + len(shared_elements), 4)  # max_set_size is 4, which is the length of `nums1` and `num
```

The resulting `max_set_size` is 4, which means we can keep all the three elements from `s1` (as one of the elements '2' is common and only counted once) and two unique elements from `s2` ('4' and '5') to form the set `{1, 2, 3, 4, 5}`. However, since we need to abide by the rule of removing 2 elements from each array, we would remove '2' and '3' from either `nums1` or `nums2` to respect the problem's constraints. Thus, the final size of set `s` is maximized within the given rules.

Solution Implementation

Python

```
class Solution:
    def maximumSetSize(self, nums1: List[int], nums2: List[int]) -> int:
        # Convert lists into sets for efficient set operations
        set_nums1 = set(nums1)
        set_nums2 = set(nums2)

        # Get the length of the first list
        list_length = len(nums1)

        # Calculate unique elements in nums1 not in nums2
        # Limit the count to half of nums1's length because we aim for balanced numbers
        unique_nums1 = min(len(set_nums1 - set_nums2), list_length // 2)

        # Calculate unique elements in nums2 not in nums1
        # Limit the count to half of nums1's length for the same reason as above
        unique_nums2 = min(len(set_nums2 - set_nums1), list_length // 2)

        # Count elements common to both sets
        common_elements = len(set_nums1 & set_nums2)

        # Maximize set size by combining the unique and common elements
        # Ensure the combined size does not exceed the length of nums1
        max_set_size = min(unique_nums1 + unique_nums2 + common_elements, list_length)

        return max_set_size
```

Java

```
class Solution {
    public int maximumSetSize(int[] nums1, int[] nums2) {
        // Use HashSet to remove duplicates from both arrays
        Set<Integer> setNums1 = new HashSet<>();
        Set<Integer> setNums2 = new HashSet<>();

        // Add all elements from nums1 to setNums1
        for (int num : nums1) {
            setNums1.add(num);
        }

        // Add all elements from nums2 to setNums2
        for (int num : nums2) {
            setNums2.add(num);
        }

        // Length of array nums1, given all arrays are same length
        int length = nums1.length;

        // Initialize counters for unique elements in setNums1 (a),
        // in setNums2 (b), and common elements in both (c)
        int uniqueNum1 = 0, uniqueNum2 = 0, commonElements = 0;

        // Count unique elements in setNums1
        for (int num : setNums1) {
            if (!setNums2.contains(num)) {
                ++uniqueNum1;
            }
        }

        // Count unique elements in setNums2 and common elements
        for (int num : setNums2) {
            if (!setNums1.contains(num)) {
                ++uniqueNum2;
            } else {
                ++commonElements;
            }
        }

        // We can have at most n/2 unique elements from each set
        uniqueNum1 = Math.min(uniqueNum1, length / 2);
        uniqueNum2 = Math.min(uniqueNum2, length / 2);

        // Maximum size of the set we can choose is sum of unique elements from both sets plus common elements.
        // We ensure not to exceed the original length of the array.
        return Math.min(uniqueNum1 + uniqueNum2 + commonElements, length);
    }
}
```

C++

```
#include <vector>
#include <unordered_set>

class Solution {
public:
    int maximumSetSize(std::vector<int>& nums1, std::vector<int>& nums2) {
        // Creating sets from the input vectors to eliminate duplicates and allow for efficient lookups.
        std::unordered_set<int> uniqueNums1(nums1.begin(), nums1.end());
        std::unordered_set<int> uniqueNums2(nums2.begin(), nums2.end());

        // Store the size of the first vector.
        int maxSize = nums1.size();

        // Variables to keep count of unique elements in each set and common elements between the sets.
        int uniqueInNums1 = 0, uniqueInNums2 = 0, commonElements = 0;

        // Count elements unique to nums1's set.
        for (int element : uniqueNums1) {
            if (uniqueNums2.count(element) == 0) {
                ++uniqueInNums1;
            }
        }

        // Count elements unique to nums2's set and those that are common.
        for (int element : uniqueNums2) {
            if (uniqueNums1.count(element) == 0) {
                ++uniqueInNums2;
            } else {
                ++commonElements;
            }
        }

        // Ensure uniqueInNums1 count does not exceed half of maxSize.
        uniqueInNums1 = std::min(uniqueInNums1, maxSize / 2);
        // Ensure uniqueInNums2 count does not exceed half of maxSize.
        uniqueInNums2 = std::min(uniqueInNums2, maxSize / 2);

        // The maximum size is determined by the sum of unique and common elements but cannot exceed maxSize.
        return std::min(uniqueInNums1 + uniqueInNums2 + commonElements, maxSize);
    }
};
```

TypeScript

```
function maximumSetSize(nums1: number[], nums2: number[]): number {
    // Convert the input arrays to sets to eliminate duplicates
    const setNums1: Set<number> = new Set(nums1);
    const setNums2: Set<number> = new Set(nums2);

    // Establish the length constraint for the subsets
    const maxLength: number = nums1.length;

    // Initialize counters for elements unique to each set and shared elements
    let uniqueNums1: number = 0;
    let uniqueNums2: number = 0;
    let commonElements: number = 0;

    // Count elements unique to the first set
    for (const num of setNums1) {
        if (!setNums2.has(num)) {
            uniqueNums1++;
        }
    }

    // Count elements unique to the second set and shared elements
    for (const num of setNums2) {
        if (!setNums1.has(num)) {
            uniqueNums2++;
        } else {
            commonElements++;
        }
    }

    // The number of unique elements we can take from each set is limited by half the length of nums1
    uniqueNums1 = Math.min(uniqueNums1, maxLength >> 1);
    uniqueNums2 = Math.min(uniqueNums2, maxLength >> 1);

    // The maximum size of the resulting set is the sum of unique and common elements,
    // but it can't exceed the length of nums1
    return Math.min(uniqueNums1 + uniqueNums2 + commonElements, maxLength);
}
```

```
class Solution:
    def maximumSetSize(self, nums1: List[int], nums2: List[int]) -> int:
        # Convert lists into sets for efficient set operations
        set_nums1 = set(nums1)
        set_nums2 = set(nums2)

        # Get the length of the first list
        list_length = len(nums1)

        # Calculate unique elements in nums1 not in nums2
        # Limit the count to half of nums1's length because we aim for balanced numbers
        unique_nums1 = min(len(set_nums1 - set_nums2), list_length // 2)

        # Calculate unique elements in nums2 not in nums1
        # Limit the count to half of nums1's length for the same reason as above
        unique_nums2 = min(len(set_nums2 - set_nums1), list_length // 2)

        # Count elements common to both sets
        common_elements = len(set_nums1 & set_nums2)

        # Maximize set size by combining the unique and common elements
        # Ensure the combined size does not exceed the length of nums1
        max_set_size = min(unique_nums1 + unique_nums2 + common_elements, list_length)

        return max_set_size
```

Time and Space Complexity

Time Complexity

The time complexity of the `maximumSetSize` function mainly stems from a few operations:

1. Converting `nums1` to a set `s1` and `nums2` to a set `s2` requires $O(N)$ time each, where N is the number of elements in the respective lists.
2. Finding the difference between sets (`s1 - s2` and `s2 - s1`) and their intersection (`s1 & s2`) have a time complexity of $O(N)$ each, under the assumption that the elements in the sets are hashed properly.
3. The `min` function calls are constant time operations and do not significantly contribute to the overall time complexity.

Hence, the overall time complexity is $O(N)$, considering all N elements would have to be processed to convert them into sets and perform set operations.

Space Complexity

The space complexity is dominated by the additional sets we create:

1. `s1` and `s2` both require $O(N)$ space, where N is the length of `nums1` and `nums2` respectively, as in the worst-case scenario they could be all unique.
2. Temporary space for set operations is also $O(N)$ in the worst case when all elements are unique.

Therefore, the overall space complexity is also $O(N)$, where N is the size of the larger input list because we need space to store unique elements of each input list.