2442. Count Number of Distinct Integers After Reverse Operations

Medium Array Hash Table Math

Leetcode Link

Problem Description

In this problem, you are provided with an array called `nums` that contains positive integers. Your task is to perform a specific operation on each integer in the array. This operation involves reversing the digits of the integer and then appending the reversed number to the end of the array. It's important to note that the operation is applied only to the original integers and not to any new numbers that you add to the array as a result of the operation.

Once all the original integers have been processed in this way, your goal is to count how many distinct integers are present in the array. The term "distinct" means that if the same integer appears more than once in the array, it only counts as one unique integer. The final result should be the total number of these unique integers.

For example, if the array is `[123, 456]`, after reversing the digits, you'll add `321` and `654` to the array, resulting in `[123, 456, 321, 654]`. The number of distinct integers in this final array is `4`.

Intuition

To arrive at the solution for this problem, we can follow a straightforward approach. We know that we need to count distinct integers which means duplicates should not be counted more than once. A common data structure that helps maintain a collection of unique items is a set. In Python, a set is an unordered collection of distinct hashable objects.

The intuition behind the solution is to:

1. Start by creating a set and adding all the original integers from the `nums` array to it. This step ensures that we have a collection of distinct integers from the original array.
2. Iterate through the original integers in the `nums` array. For each integer:
   - Reverse its digits. In Python, this can be easily done by converting the integer to a string, using slicing to reverse it, and then converting it back to an integer.
   - Add the reversed integer to the set. Since sets automatically ensure that only distinct items are stored, any duplicates formed by reversing the digits will not increase the size of the set.
3. After the loop completes, the set will contain all unique integers, including both the original ones and their reversed counterparts.
4. The last step is simply to return the size of the set, which represents the total number of distinct integers after the operation.

By using a set to eliminate duplicate entries, we ensure that the counting of distinct integers is efficient, with the iteration handling the logic for digit reversal and combination.

Solution Approach

The implementation of the solution is based primarily on the use of a set data structure and string manipulation. Here's a step-by-step explanation of how the solution is implemented in the provided Python code:

1. A set named `s` is created and initialized with the elements of `nums`. This is done using `s = set(nums)`. The property of a set to store only unique elements aids in keeping track of which numbers have already been seen, thus avoiding counting duplicates.
2. A `for` loop is used to iterate through each number `x` in the original `nums` array. This ensures that we process each integer exactly once.
3. For each integer `x` encountered in the for loop, we reverse its digits. This is done by first converting `x` to a string with `str(x)`, then reversing the string with slicing `[::-1]`, and finally converting it back to an integer with `int()`.
4. The resulting reversed integer `y` is added to the set `s` using `s.add(y)`. If `y` is already in the set, this operation has no effect due to the properties of a set. If `y` is not in the set, it gets added as a new element.
5. After the for loop ends, `s` contains all the original numbers and their reversed counterparts, without any duplicates.
6. The final step is to return the number of elements in the set `s` with `return len(s)`. This count represents the number of distinct integers in the final array after performing the reversal process on all original integers.

With this approach, the complexity of the solution is primarily O(n), with n being the number of elements in `nums`. This is because the loop runs for each element exactly once, and set operations like addition and checking for existence are generally O(1).

In summary, this solution efficiently counts the number of distinct integers after reversing the digits of each integer in the initial array and adding them back to the array. The use of a set is crucial as it efficiently manages the uniqueness of the elements without requiring additional logic to handle duplicates.

Example Walkthrough

Let's walk through an example to illustrate the solution approach. Suppose we have an array called `nums` with the following integers: `[12, 21, 3]`. According to the problem statement, we need to reverse each number and then append this reversed number to the array. After that, we will count how many distinct integers are present.

Step 1: Create a set called `s` and add all the unique integers from the `nums` array to it. Initially, `s = {12, 21, 3}`.

Step 2: Iterate through the original array `nums`. On the first iteration, we take the number `12`.

Step 3: Reverse the digits of `12` to get `21`. However, `21` is already present in the set `s`, so adding it will not change the set.

Step 4: Move to the second integer in `nums`, which is `21`. Reverse `21` to get `12`, which is already in the set `s`, so the set remains unchanged.

Step 5: Next, take the third integer, `3`. Reverse `3` to obtain `3` again since it is a single-digit number. Add it to the set `s` even though it is already there, and the set still doesn't change.

After the for loop ends, the set `s` contains `{12, 21, 3}`. These are the original numbers along with their reversed counterparts, but since reversing doesn't produce any new distinct numbers, there are no changes to our set.

Step 6: To get the final answer, return the size of the set `s`. Since `s` has three elements, the result is `3`.

Implementation of the described process in Python is very simple:

```python
nums = [12, 21, 3]
s = set(nums)

for x in nums:
    y = int(str(x)[::-1])
    s.add(y)

result = len(s)  # This will be the output, which is 3 in this case.
```

Thus, the final array after considering the reverse of each number would look like `[12, 21, 3, 21, 12, 3]`, and the count of unique integers is `3`, which is the output of our example walkthrough.

Python Solution

```python
class Solution:
    def countDistinctIntegers(self, nums):
        # Initialize an empty set to store unique integers
        unique_integers = set(nums)

        # Iterate through the list of numbers
        for number in nums:
            # Reverse the current number by converting it to a string,
            # reversing it, and then converting it back to an integer
            reversed_number = int(str(number)[::-1])

            # Add the reversed number to the set of unique integers
            unique_integers.add(reversed_number)

        # Return the count of unique integers (original and reversed)
        return len(unique_integers)
```

Java Solution

```java
class Solution {
    public int countDistinctIntegers(int[] nums) {
        // Create a HashSet to ensure all integers are unique.
        Set<Integer> uniqueIntegers = new HashSet<>();

        // Add each number in the array to the HashSet.
        for (int num : nums) {
            uniqueIntegers.add(num);
        }

        // Iterate over the array again to add reversed numbers.
        for (int num : nums) {
            int reversedNum = 0;
            // Reverse the current number.
            while (num > 0) {
                reversedNum = reversedNum * 10 + num % 10; // Append the last digit to reversedNum.
                num /= 10; // Remove the last digit from num.
            }
            // Add the reversed number to the HashSet.
            uniqueIntegers.add(reversedNum);
        }

        // Return the size of the HashSet, which represents the count of distinct integers.
        return uniqueIntegers.size();
    }
}
```

C++ Solution

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

class Solution {
public:
    int countDistinctIntegers(vector<int>& nums) {
        // Initialize an unordered_set to keep track of distinct integers
        unordered_set<int> distinctIntegers(nums.begin(), nums.end());

        // Loop through each number in the input vector
        for (int num : nums) {
            int reversedNum = 0;
            // Reverse the current number
            while (num > 0) {
                reversedNum = reversedNum * 10 + num % 10;
                num /= 10;
            }
            // Add the reversed number to the set of distinct integers
            distinctIntegers.insert(reversedNum);
        }

        // Return the size of the set, which represents the count of distinct integers
        return distinctIntegers.size();
    }
};
```

Typescript Solution

```typescript
function countDistinctIntegers(nums: number[]): number {
    // Get the length of the initial array
    const length = nums.length;

    // Iterate over each number in the array
    for (let i = 0; i < length; i++) {
        // Convert the number at the current index to a string,
        // split it to form an array, reverse the array,
        // join it back to a string, and then convert it back to a number
        const reversedNum = Number(nums[i].toString().split('').reverse().join(''));

        // Append the reversed number to the nums array
        nums.push(reversedNum);
    }

    // Create a Set from the nums array to remove duplicates,
    // and then return the size of the Set, which gives us the count
    // of distinct integers
    return new Set(nums).size;
}

```

Time and Space Complexity

The provided Python code calculates the count of distinct integers in the input list `nums` after including both the original numbers and their reversed forms.

Time Complexity:

The time complexity of this function primarily consists of iterating through the `nums` list once, reversing each number, and adding it to the set `s`.

- Creating the initial set `s` with all elements in `nums` happens in O(n) time, where `n` is the number of elements in `nums`.
- The for loop runs once for each element in `nums`, so it runs `n` times.
- Inside the loop, reversing the string representation of a number `x` and converting it back into an integer `y` is O(m), where `m` is the number of digits in the number. However, since m is much smaller than n and is bounded by a constant (the number of digits will never be more than around 10 for 32-bit integers), this can be considered O(1) operation for each element in the context of the larger input size `n`.
- Adding the reversed number to the set is O(1) on average due to hash-based implementation, but since this is done for each element, it adds O(n) time over the entire input.

Therefore, the time complexity of the function is O(n), where `n` is the length of `nums`.

Space Complexity:

- The space complexity is determined by the size of the set `s`. In the worst case, each element in `nums` and its reversed form is unique, thus the set could hold 2n elements.
- However, space is also required for the string manipulation when reversing the numbers. This is temporary space within the for loop and does not scale with the size of `nums`, hence it's O(1).

Considering that, the overall space complexity is O(n), where `n` is the size of the original input `nums`.