217. Contains Duplicate

Easy | Array | Hash Table | Sorting

Problem Description

The problem presents us with a task to determine whether an array of integers, `nums`, contains any duplicate elements. The goal is to return `true` if any specific value appears at least twice in the array, indicating the presence of duplicates. If every element in the array is unique, we should return `false`.

Intuition

The intuition behind the solution approach is to utilize a data structure that can handle uniqueness efficiently. A set in Python is a perfect choice for handling unique elements. When we convert the list of numbers into a set, all duplicates are automatically removed because sets cannot contain duplicate elements. Comparing the length of the set with the original list gives us a direct indication of whether there were any duplicates:

1. If there are duplicates in the original list, the set's length will be less because it would have removed the duplicates.
2. If there are no duplicates, both the set and the list will have the same length.

Thus, by checking whether the length of the set is smaller than the length of the original list, we can tell if there are any duplicate numbers in the array. This is a clean and efficient solution as it does not require any additional looping or complex logic.

Solution Approach

The implementation of the solution is straightforward and relies on the properties of sets in Python. Here's a step-by-step breakdown of how the solution works:

- We take the input list `nums` and efficiently convert it into a set using the `set()` constructor. This process removes any duplicate elements that might be present in the `nums` list since sets by definition only allow unique elements.
- The operation `set(nums)` creates a new set containing only unique elements from the original list.
- We then compare the length of the created set with the length of the original `nums` list using the `len()` function. The comparison is done by the expression `len(set(nums)) < len(nums)`.
- If the lengths are different, this means the set is shorter because it consolidated duplicate elements from the `nums` list into single occurrences. Therefore, the original `nums` list contained duplicates, and we should return `true`.
- If the lengths are equal, then there were no duplicates to remove when creating the set. This indicates that all elements in the `nums` list were distinct, so the function should return `false`.

The given solution is efficient because:

- It uses the property of a set to store unique elements, thus eliminating the need for explicit loops or additional data structures to check for duplicates.
- Conversion from a list to a set and length comparison are operations with a time complexity of O(n), making the entire solution O(n).
- The space complexity of the approach is also O(n) in the worst case, where `n` is the number of elements in the input list since we might have to store all elements in the set if they are unique.

This one-line solution elegantly solves the problem:

```
def containsDuplicate(self, nums: List[int]) -> bool:
    return len(set(nums)) < len(nums)
```

Example Walkthrough

Let's assume we are given the following array of integers:

```
nums = [1, 3, 4, 2, 3]
```

We are tasked with finding out if any number appears more than once in the array `nums`.

Following the solution approach:

1. **Convert nums to a set**: We first convert the list into a set.

   ```
   unique_nums = set(nums)
   ```

   The unique_nums set will be: `{1, 2, 3, 4}`.

   After this conversion, the set `unique_nums` contains the numbers `{1, 2, 3, 4}`. Notice that the duplicate `3` from the original `nums` list does not appear twice in the set because a set stores only unique values.

2. **Compare lengths**: We then compare the length of the set `unique_nums` with the original list `nums`.

   ```
   Length of nums list: 5
   Length of unique_nums set: 4
   ```

   The length comparison tells us that the set is shorter than the list, indicating duplicates were removed during set conversion.

3. **Return true if duplicates exist**: Since the lengths aren't equal (the length of `unique_nums` is less than `nums`), we determine that duplicates exist in the original list.

   The function using our above method would return `true`.

For illustration purposes, let's consider another example where `nums = [1, 2, 3, 4, 5]`.

1. **Convert nums to a set**: Converting `nums` to a set gives us `{1, 2, 3, 4, 5}`.
2. **Compare lengths**: Both the set and the list have a length of 5.
3. **Return false if no duplicates**: Since the lengths are equal, we can conclude that there are no duplicate values, and our function would return `false`.

In essence, the function `containsDuplicate` is effectively checking:

```
len(set([1. 3. 4. 2. 3])) < len([1. 3. 4. 2. 3]) # Returns true. duplicates are present
len(set([1, 2, 3, 4, 5])) < len([1, 2, 3, 4, 5]) # Returns false, no duplicates
```

Solution Implementation

Python

```
# We define the Solution class as per the LeetCode standard.
class Solution:
    # Define a function `contains duplicate` that checks for duplicates in a list of integers.
    def containsDuplicate(self, nums: List[int]) -> bool:
        # Convert the list `nums` into a set, which removes duplicates, and compare its length to the original list's length.
        # If the set's length is less than the list's length, there are duplicates in the list.
        return len(set(nums)) < len(nums)
```

Java

```
import java.util.HashSet;
import java.util.Set;

public class Solution {
    /**
     * Checks if the array contains any duplicates.
     *
     * @param numbers The array of integers to check for duplicates.
     * @return true if any value appears at least twice in the array, and false if every element is distinct.
     */
    public boolean containsDuplicate(int[] numbers) {
        // Initialize a HashSet to store unique elements.
        Set<Integer> uniqueNumbers = new HashSet<>();

        // Iterate through all the elements in the array.
        for (int number : numbers) {
            // Attempt to add the current element to the set.
            // If the add method returns false, it means the element is already present in the set.
            if (!uniqueNumbers.add(number)) {
                // Duplicate found, so return true.
                return true;
            }
        }

        // No duplicates were found, return false.
        return false;
    }
}
```

C++

```
#include <vector>       // Include necessary header for vector
#include <unordered_set> // Include necessary header for unordered_set

class Solution {
public:
    // Function to determine if any value appears at least twice in the array
    bool containsDuplicate(vector<int>& nums) {
        // Initializing an unordered set with the elements from the nums vector.
        unordered_set<int> numSet(nums.begin(), nums.end());

        // If the size of the set is smaller than the size of the original vector,
        // it means there were duplicates which were removed in the set creation process.
        // Hence, return true if duplicates were found, false otherwise.
        return numSet.size() < nums.size();
    }
};
```

TypeScript

```
// This function checks if there are any duplicate numbers in the array.
// @param nums - An array of numbers to check for duplicates.
// @returns boolean - Returns true if duplicates are found, otherwise false.
function containsDuplicate(nums: number[]): boolean {
    // Create a Set from the array to eliminate any duplicates.
    const uniqueNums = new Set<number>(nums);

    // Compare the size of the Set with the array length.
    // If they are different, it means there were duplicates (as the Set would be smaller).
    return uniqueNums.size !== nums.length;
}

# We define the Solution class as per the LeetCode standard.
class Solution:
    # Define a function `contains duplicate` that checks for duplicates in a list of integers.
    def containsDuplicate(self, nums: List[int]) -> bool:
        # Convert the list `nums` into a set, which removes duplicates, and compare its length to the original list's length.
        # If the set's length is less than the list's length, there are duplicates in the list.
        return len(set(nums)) < len(nums)
```

Time and Space Complexity

The given Python code checks for duplicates in a list by converting it into a set, which stores only unique elements, and then comparing the length of this set to the length of the original list. If the set has fewer elements, it means that there were duplicates in the original list.

Time Complexity:

The main operation here is the conversion of the list into a set, which typically has a time complexity of `O(n)` where `n` is the number of elements in the list. This is because the operation needs to iterate through all elements once, adding them into the set and checking for uniqueness.

Space Complexity:

The space complexity is also `O(n)` in the worst case because in a situation where all elements are unique, the set would need to store all `n` elements separately from the original list.