220. Contains Duplicate III

Hard Array Bucket Sort Ordered Set Sorting Sliding Window

Leetcode Link

Problem Description

This problem requires us to determine if there exist two indices `i` and `j` in an integer array `nums` such that:

- The indices `i` and `j` are different (`i != j`).
- The absolute difference between the indices `i` and `j` does not exceed `indexDiff` (`abs(i - j) <= indexDiff`).
- The absolute difference between the values at indices `i` and `j` does not exceed `valueDiff` (`abs(nums[i] - nums[j]) <= valueDiff`).

We must return `true` if such a pair of indices exists and `false` otherwise.

The challenge lies in doing this efficiently, as the brute force approach of checking all pairs would take too much time for large arrays.

Intuition

The key to solving this problem efficiently is to maintain a set of elements from `nums` that have recently been processed and fall within the `indexDiff` range of the current index. Since we need to check the `valueDiff` condition efficiently, a sorted set is used.

Here's the intuition process to arrive at the solution:

1. We initialize a sorted set `s` which helps to access elements in sorted order and provides efficient operations to find if an element exists within a certain range.
2. We then iterate through each element `v` in the array `nums`.
3. For each element, we find the smallest element in the sorted set that would satisfy the `valueDiff` condition. This is done by searching for the left boundary (`v - valueDiff`) using `bisect_left`.
4. Once we have that element, if there is one within the valid range `v + valueDiff`, then we have found indices `i` and `j` that satisfy both conditions for `indexDiff` and `valueDiff`.
5. If we haven't returned `true` yet, we add the current element `v` to the sorted set. This is because it might be a part of a valid pair with a later element.
6. We then check the size of the sorted set relative to the `indexDiff`. If the set size indicates that an element is too old and cannot satisfy the `indexDiff` based on the current index `i`, we remove the element from the sorted set that corresponds to the index `i - indexDiff`.
7. If we complete the iteration without finding a valid pair, we return `false`, indicating no such pair exists.

By using a sorted set and keeping track of the indices, we ensure that we are always looking at a window of `indexDiff` for potential pairs, and checking for `valueDiff` within this window is efficient, thanks to the sorted nature of the set.

Solution Approach

The solution provided uses the `SortedSet` data structure from the `sortedcontainers` Python module. This data structure maintains its elements in ascending order and supports fast insertion, deletion, and queries, which are essential to our approach.

Let's break down the algorithm step-by-step:

1. We begin by creating an empty `SortedSet` called `s`, which will hold the candidates that could potentially form a valid pair with the current element in terms of `valueDiff`.
2. We then enumerate over our input array `nums` using a for loop, giving us both the index `i` and the value `v` at each iteration.
3. For the current value `v`, we want to find if there is a value in our sorted set `s` that does not differ from `v` by more than `valueDiff`. To achieve this, we perform a binary search in the sorted set for the left boundary `v - valueDiff` using the `bisect_left` function. This returns the index `j` of the first element in `s` that is not less than `v - valueDiff`.
4. After finding this index `j`, we check if it is within the bounds of the sorted set and if the element at this index `s[j]` does not exceed `v + valueDiff`. If these conditions are met, we have found a pair that satisfies the `valueDiff` condition, and we return `true`.
5. If no valid pair is found yet, we add the current value `v` to the sorted set `s` using `add()` method because it may become a part of a valid pair with a later element in the array.
6. To maintain the `indexDiff` condition, we need to remove elements from `s` that are too far from the current index `i`. Specifically, if `i >= indexDiff`, we remove the element that corresponds to the index `i - indexDiff` using the `remove()` method.
7. Finally, if the loop finishes without returning `true`, we conclude that no valid pair exists and return `false`.

This algorithm works effectively because the `SortedSet` maintains the order of elements at all times, so checking for the `valueDiff` condition is very efficient, and by using the index conditions, we keep updating the set so that it only contains relevant elements that could form a valid pair considering the `indexDiff`.

Overall, the use of `SortedSet` optimizes the brute force approach which would be clear when matching face to face with a potentially large `nums` array, where a less efficient process would result in a time complexity that is too high.

Example Walkthrough

Let's use a small example to illustrate the solution approach:

Given `nums = [1, 2, 3, 1]`, `indexDiff = 3`, and `valueDiff = 1`.

1. We start with an empty `SortedSet`, which we denote as `s`.
2. We iterate over each value `v` in `nums` along with its index `i`.
3. At index `i = 0`, value `v = 1`.
   - There are no elements in `s` yet, so we add `v` to `s`. Set `s` now contains `[1]`.
4. At index `i = 1`, value `v = 2`.
   - We use binary search to check for `v - valueDiff = 1` in `s`.
   - The smallest element greater than or equal to `1` is `1`.
   - It is within `valueDiff` from `2`. However, since `s` has only one element, which is from the current index, we don't consider it.
   - We add `v` to `s`. Set `s` now contains `[1, 2]`.
5. At index `i = 2`, value `v = 3`.
   - We use binary search to check for `v - valueDiff = 2` in `s`.
   - The smallest element greater than or equal to `2` is `2`.
   - It is within `valueDiff` from `3`.
   - We return `true` since we found `2` which is within `valueDiff` from `3` and whose index `i` is within `indexDiff` from the current index `2`.

Therefore, the conditions are satisfied and the function would return `true`.

Python Solution

```python
from sortedcontainers import SortedSet
from typing import List

class Solution:
    def containsNearbyAlmostDuplicate(self, nums: List[int], index_diff: int, value_diff: int) -> bool:
        # Create a SortedSet to maintain a sorted list of numbers
        # we have seen so far.
        sorted_set = SortedSet()

        # Loop through each number in the given list along with its index
        for i, num in enumerate(nums):
            # Find the left boundary where number becomes greater than or equal to num-value_diff.
            left_boundary_index = sorted_set.bisect_left(num - value_diff)

            # Check if there exists a value within the range [num-value_diff, num+value_diff]
            # inside our SortedSet.
            if left_boundary_index < len(sorted_set) and sorted_set[left_boundary_index] <= num + value_diff:
                return True  # If found, return true as the condition is satisfied.

            # If not found, add the current number to the SortedSet.
            sorted_set.add(num)

            # If the SortedSet's size exceeds the allowed indexDiff,
            # we remove the oldest element from the SortedSet to maintain the sliding window constraint.
            if i >= index_diff:
                sorted_set.remove(nums[i - index_diff])

        # If we never return True within the loop, there is no such pair which satisfies the condition,
        # hence we return False.
        return False
```

Java Solution

```java
class Solution {
    public boolean containsNearbyAlmostDuplicate(int[] nums, int indexDiff, int valueDiff) {
        // Use TreeSet to maintain a sorted set.
        TreeSet<Long> sortedSet = new TreeSet<>();

        // Iterate through the array of numbers.
        for (int i = 0; i < nums.length; ++i) {
            // Try finding a value in the set within the range of (value - valueDiff) and (value + valueDiff).
            Long floorValue = sortedSet.ceiling((long) nums[i] - (long) valueDiff);
            if (floorValue != null && floorValue <= (long) nums[i] + (long) valueDiff) {
                // If such a value is found, return true.
                return true;
            }

            // Add the current number to the sorted set.
            sortedSet.add((long) nums[i]);

            // If the sorted set size exceeded the allowed index difference, remove the oldest value.
            if (i >= indexDiff) {
                sortedSet.remove((long) nums[i - indexDiff]);
            }
        }

        // Return false if no such pair is found in the set.
        return false;
    }
}
```

C++ Solution

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

class Solution {
public:
    // Function to determine if the array contains nearby almost duplicate elements
    bool containsNearbyAlmostDuplicate(vector<int>& nums, int indexDiff, int valueDiff) {
        // Initialize a set to keep track of values in the window defined by indexDiff
        set<long> windowSet;
        for (int i = 0; i < nums.size(); ++i) {
            // Find the lower bound of the acceptable value difference
            auto lower = windowSet.lower_bound((long) nums[i] - valueDiff);
            // If an element is found within the value range, return true
            if (lower != windowSet.end() && *lower <= (long) nums[i] + valueDiff) {
                return true;
            }
            // Insert the current element into the set
            windowSet.insert((long) nums[i]);
            // If our window exceeds the permitted index difference, remove the oldest value
            if (i >= indexDiff) {
                windowSet.erase((long) nums[i - indexDiff]);
            }
        }
        // If no duplicates are found in the given range, return false
        return false;
    }
};
```

Typescript Solution

```typescript
// Define the interfaces and global variables
interface IComparator<T> {
    (lhs: T, rhs: T): number;
}

interface IRBTreeNode<T> {
    data: T;
    count: number;
    left: IRBTreeNode<T> | null;
    right: IRBTreeNode<T> | null;
    parent: IRBTreeNode<T> | null;
    color: number;
    // Methods like sibling, isOnLeft, and hasRedChild are removed as they should be part of the class
}

const RED = 0;
const BLACK = 1;
let root: IRBTreeNode<any> | null = null; // Global tree root

// Define global methods
function createNode<T>(data: T): IRBTreeNode<T> {
    return {
        data: data,
        count: 1,
        left: null,
        right: null,
        parent: null,
        color: RED  // Newly created nodes are red
    };
}

// Example usage of creating a node
const node = createNode(10);

// Define the rotate functions
function rotateLeft<T>(pt: IRBTreeNode<T>): void {
    if (!pt.right) {
        throw new Error("Cannot rotate left without a right child");
    }
    let right = pt.right;
    pt.right = right.left;
    if (pt.right) pt.right.parent = pt;

    right.parent = pt.parent;
    if (!pt.parent) {
        root = right;
    } // ... Rest of the rotateLeft logic
}

function rotateRight<T>(pt: IRBTreeNode<T>): void {
    // ... Implement rotateRight logic
}

// Define other necessary functions like find, insert, delete, etc...
```

Time and Space Complexity

Time Complexity

The time complexity of the given code primarily depends on the number of iterations over the `nums` array, the operations performed with the `SortedSet`, and maintaining the `indexDiff` constraint. The main operations within the loop are:

1. Checking for a nearby almost duplicate (`bisect_left` and comparison): The `bisect_left` method in a sorted set is typically O(log n), where n is the number of elements in the set.
2. Adding a new element to the sorted set (`s.add(v)`): Inserting an element into a `SortedSet` is also O(log n) as it keeps the set sorted.
3. Removing the oldest element when the `indexDiff` is exceeded (`s.remove(nums[i - indexDiff])`): This is O(log n) to find the element and O(n) to remove it because removing an element from a sorted set can require shifting all elements to the right of the removed element.

Since each of these operations is called once per iteration and the `remove` operation has the higher complexity of O(n), the time complexity per iteration is O(n). However, since the size of the sorted set is capped by the `indexDiff`, let's use k to denote `indexDiff` as the maximum number of elements the sorted set can contain. The complexity now becomes O(k) for insertion and removal, and the complexity of `bisect_left` is O(log k). Therefore, the time complexity is O(n * log k) where n is the length of the input array and k is `indexDiff`.

Space Complexity

The space complexity is determined by the maximum size of the sorted set `s`, which can grow up to the largest `indexDiff`. Therefore, the space complexity is O(k), where k is the maximum number of entries in the `SortedSet`, which is bounded by `indexDiff`.