Contains Duplicate III

Given an array of integers, find out whether there are two distinct indices *i* and *j* in the array such that the difference between **nums[i]** and **nums[j]** is at most *t* and the difference between *i* and *j* is at most *k*.

Solution 1

As a followup question, it naturally also requires maintaining a window of size k. When t == 0, it reduces to the previous question so we just reuse the solution.

Since there is now a constraint on the range of the values of the elements to be considered duplicates, it reminds us of doing a range check which is implemented in tree data structure and would take O(LogN) if a balanced tree structure is used, or doing a bucket check which is constant time. We shall just discuss the idea using bucket here.

Bucketing means we map a range of values to the a bucket. For example, if the bucket size is 3, we consider 0, 1, 2 all map to the same bucket. However, if t == 3, (0, 3) is a considered duplicates but does not map to the same bucket. This is fine since we are checking the buckets immediately before and after as well. So, as a rule of thumb, just make sure the size of the bucket is reasonable such that elements having the same bucket is immediately considered duplicates or duplicates must lie within adjacent buckets. So this actually gives us a range of possible bucket size, i.e. t and t + 1. We just choose it to be t and a bucket mapping to be *num / t*.

Another complication is that negative ints are allowed. A simple*num / t* just shrinks everything towards 0. Therefore, we can just reposition every element to start from Integer.MIN_VALUE.

```
public class Solution {
    public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {
        if (k < 1 || t < 0) return false;
        Map<Long, Long> map = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            long remappedNum = (long) nums[i] - Integer.MIN_VALUE;
            long bucket = remappedNum / ((long) t + 1);
            if (map.containsKey(bucket)
                    || (map.containsKey(bucket - 1) && remappedNum - map.get(buck
et - 1) <= t)
                        || (map.containsKey(bucket + 1) && map.get(bucket + 1) -
remappedNum <= t))
                            return true;
            if (map.entrySet().size() >= k) {
                long lastBucket = ((long) nums[i - k] - Integer.MIN_VALUE) / ((lo
ng) t + 1);
                map.remove(lastBucket);
            }
            map.put(bucket, remappedNum);
        }
        return false;
    }
}
```

Edits:

Actually, we can use t + 1 as the bucket size to get rid of the case when t == 0. It simplifies the code. The above code is therefore the updated version.

written by lx223 original link here

Solution 2

This problem requires to maintain a window of size k of the previous values that can be queried for value ranges. The best data structure to do that is Binary Search Tree. As a result maintaining the tree of size k will result in time complexity O(N lg K). In order to check if there exists any value of range abs(nums[i] - nums[j]) to simple queries can be executed both of time complexity O(lg K)

Here is the whole solution using TreeMap.

```
public class Solution {
    public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {
        if (nums == null || nums.length == 0 || k <= 0) {
            return false;
        }

        final TreeSet<Integer> values = new TreeSet<>();
        for (int ind = 0; ind < nums.length; ind++) {

            final Integer floor = values.floor(nums[ind] + t);
            final Integer ceil = values.ceiling(nums[ind] - t);
            if ((floor != null && floor >= nums[ind])
                    || (ceil != null && ceil <= nums[ind])) {
                return true;
            }

            values.add(nums[ind]);
            if (ind >= k) {
                values.remove(nums[ind - k]);
            }
        }

        return false;
    }
}
```

written by jmnarloch original link here

Solution 3

```
 bool containsNearbyAlmostDuplicate(vector<int>& nums, int k, int t) {
    set<int> window; // set is ordered automatically
    for (int i = 0; i < nums.size(); i++) {
        if (i > k) window.erase(nums[i-k-1]); // keep the set contains nums i j a
t most k
        // |x - nums[i]| <= t  ==> -t <= x - nums[i] <= t;
        auto pos = window.lower_bound(nums[i] - t); // x-nums[i] >= -t ==> x >= n
ums[i]-t
        // x - nums[i] <= t ==> |x - nums[i]| <= t
        if (pos != window.end() && *pos - nums[i] <= t) return true;
        window.insert(nums[i]);
    }
    return false;
}
```

written by lchen77 original link here