

## 220. Contains Duplicate III

March 25, 2016 | 35.1K views

[Previous](#) [Next](#)

★ ★ ★ ★ ★

Average Rating: 4.25 (57 votes)

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

Example 1:

Input: nums = [1,2,3,1], k = 3, t = 0  
Output: true

Example 2:

Input: nums = [1,0,1,1], k = 1, t = 2  
Output: true

Example 3:

Input: nums = [1,5,9,1,5,9], k = 2, t = 3  
Output: false

## Summary

This article is for intermediate readers. It introduces the following ideas: Binary Search Tree, HashMap, and Buckets.

## Solutions

### Approach #1 (Naive Linear Search) [Time Limit Exceeded]

#### Intuition

Compare each element with the previous  $k$  elements and see if their difference is at most  $t$ .

#### Algorithm

This problem requires us to find  $i$  and  $j$  such that the following conditions are satisfied:

- $|i - j| \leq k$
- $|\text{nums}[i] - \text{nums}[j]| \leq t$

The naive approach is the same as [Approach #1 in Contains Duplicate II solution](#), which keeps a virtual sliding window that holds the newest  $k$  elements. In this way, [Condition 1](#) above is always satisfied. We then check if [Condition 2](#) is also satisfied by applying linear search.

Java

Copy

```
1 public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {
2     for (int i = 0; i < nums.length; ++i) {
3         for (int j = Math.max(i - k, 0); j < i; ++j) {
4             if (Math.abs(nums[i] - nums[j]) <= t) return true;
5         }
6     }
7     return false;
8 }
9 // Time limit exceeded.
```

Complexity Analysis

- Time complexity :  $O(n \min(k, n))$ . It costs  $O(\min(k, n))$  time for each linear search. Note that we do at most  $n$  comparisons in one search even if  $k$  can be larger than  $n$ .
- Space complexity :  $O(1)$ . We only used constant auxiliary space.

### Approach #2 (Binary Search Tree) [Accepted]

#### Intuition

- If elements in the window are maintained in sorted order, we can apply binary search twice to check if [Condition 2](#) is satisfied.
- By utilizing self-balancing Binary Search Tree, one can keep the window ordered at all times with logarithmic time [insert](#) and [delete](#).

#### Algorithm

The real bottleneck of [Approach #1](#) is due to all elements in the sliding window are being scanned to check if [Condition 2](#) is satisfied. Could we do better?

If elements in the window are in sorted order, we can apply Binary Search twice to search for the two boundaries  $x + t$  and  $x - t$  for each element  $x$ .

Unfortunately, the window is unsorted. A common mistake here is attempting to maintain a sorted array. Added searching in a sorted array costs only logarithmic time, keeping the order of the elements after [insert](#) and [delete](#) operation is not as efficient. Imagine you have a sorted array with  $k$  elements and you are adding a new item  $x$ . Even if you can find the correct position in  $O(\log k)$  time, you still need  $O(k)$  time to insert  $x$  into the sorted array. The reason is that you need to shift all elements after the insert position one step backward. The same reasoning applies to removal as well. After removing an item from position  $i$ , you need to shift all elements after  $i$  one step forward. Thus, we gain nothing in speed compared to the [naive linear search approach](#) above.

To gain an actual speedup, we need a *dynamic* data structure that supports faster [insert](#), [search](#) and [delete](#). Self-balancing Binary Search Tree (BST) is the right data structure. The term *Self-balancing* means the tree automatically keeps its height small after arbitrary [insert](#) and [delete](#) operations. Why does self-balancing matter? That is because most operations on a BST take time directly proportional to the height of the tree. Take a look at the following non-balanced BST which is skewed to the left:

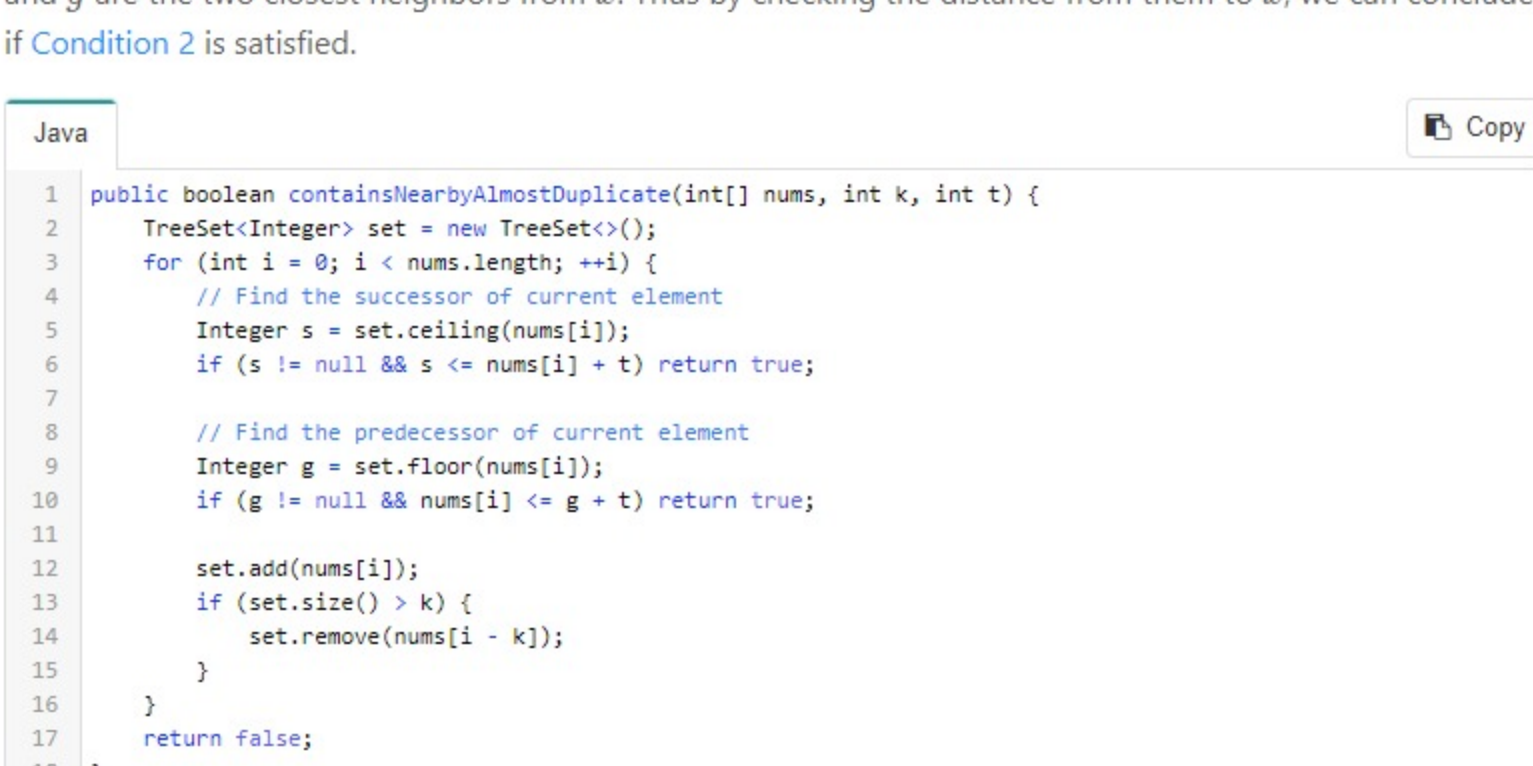


Figure 1. A non-balanced BST that is skewed to the left.

Searching in the above BST degrades to *linear* time, which is like searching in a linked list. Now compare to the BST below which is balanced:

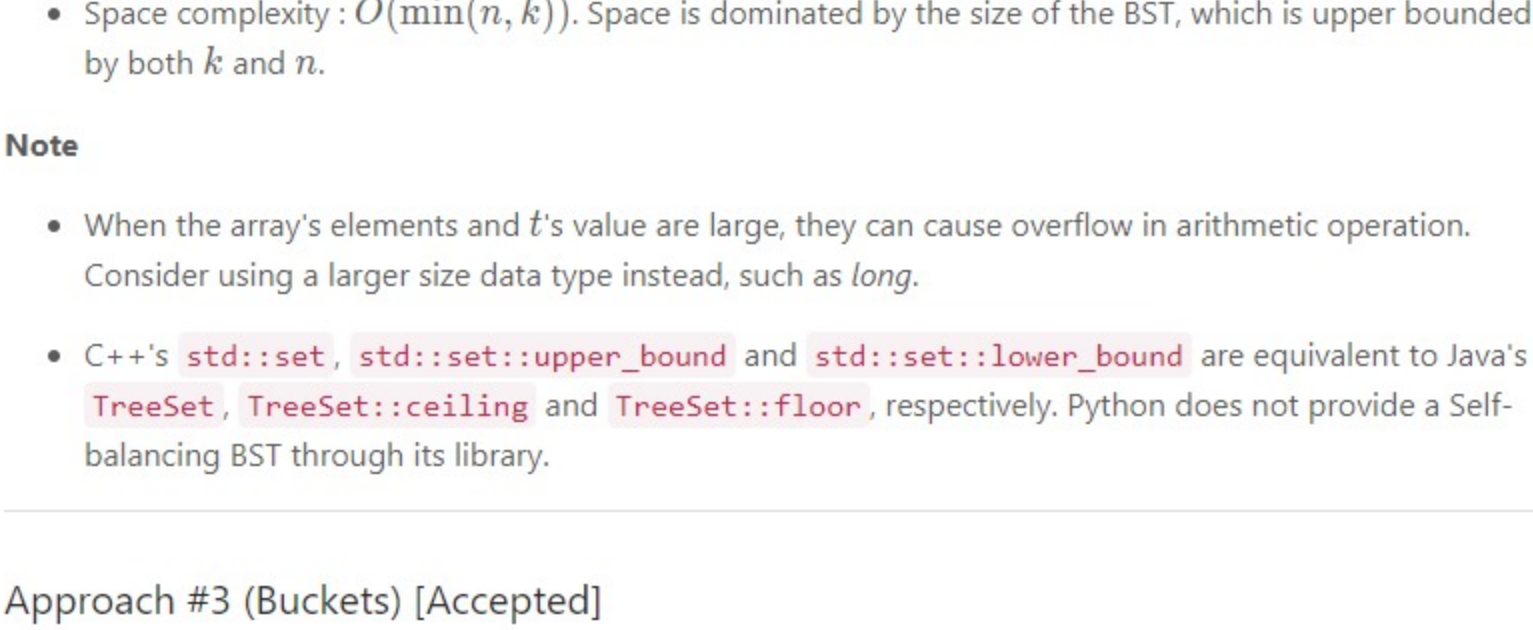


Figure 2. A balanced BST.

Assume that  $n$  is the total number of nodes in the tree, a balanced binary tree maintains its height in the order of  $h = \log n$ . Thus it supports  $O(h) = O(\log n)$  time for each of [insert](#), [search](#) and [delete](#) operations.

Here is the entire algorithm in pseudocode:

- Initialize an empty BST [set](#).
- Loop through the array, for each element  $x$ 
  - Find the *smallest* element  $s$  in [set](#) that is *greater* than or equal to  $x$ , return true if  $s - x \leq t$
  - Find the *greatest* element  $g$  in [set](#) that is *smaller* than or equal to  $x$ , return true if  $x - g \leq t$
  - Put  $x$  in [set](#).
  - If the size of the set is larger than  $k$ , remove the oldest item.
- Return false

One may consider the smallest element  $s$  that is greater or equal to  $x$  as the *successor* of  $x$  in the BST, as in: "What is the next greater value of  $x$ ". Similarly, we consider the greatest element  $g$  that is smaller or equal to  $x$  as the *predecessor* of  $x$  in the BST, as in: "What is the previous smaller value of  $x$ ". These two values  $s$  and  $g$  are the two closest neighbors from  $x$ . Thus by checking the distance from them to  $x$ , we can conclude if [Condition 2](#) is satisfied.

Java

Copy

```
1 public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {
2     TreeSet<Integer> set = new TreeSet<>();
3     for (int i = 0; i < nums.length; ++i) {
4         // Find the successor of current element
5         Integer s = set.ceiling(nums[i]);
6         if (s != null && s <= nums[i] + t) return true;
7
8         // Find the predecessor of current element
9         Integer g = set.floor(nums[i]);
10        if (g != null && nums[i] <= g + t) return true;
11
12        set.add(nums[i]);
13        if (set.size() > k) {
14            set.remove(nums[i - k]);
15        }
16    }
17    return false;
18 }
```

Complexity Analysis

- Time complexity :  $O(n \log(\min(n, k)))$ . We iterate through the array of size  $n$ . For each iteration, it costs  $O(\log \min(k, n))$  time ([search](#), [insert](#) or [delete](#)) in the BST, since the size of the BST is upper bounded by both  $k$  and  $n$ .
- Space complexity :  $O(\min(n, k))$ . Space is dominated by the size of the BST, which is upper bounded by both  $k$  and  $n$ .

Note

- When the array's elements and  $t$ 's value are large, they can cause overflow in arithmetic operation. Consider using a larger size data type instead, such as *long*.
- C++'s `std::set`, `std::set::upper_bound` and `std::set::lower_bound` are equivalent to Java's `TreeSet`, `TreeSet::ceiling` and `TreeSet::floor`, respectively. Python does not provide a Self-balancing BST through its library.

### Approach #3 (Buckets) [Accepted]

#### Intuition

Inspired by [bucket sort](#), we can achieve linear time complexity in our problem using *buckets* as window.

#### Algorithm

Bucket sort is a sorting algorithm that works by distributing the elements of an array into a number of buckets. Each bucket is then sorted individually, using a different sorting algorithm. Here is an illustration of buckets.

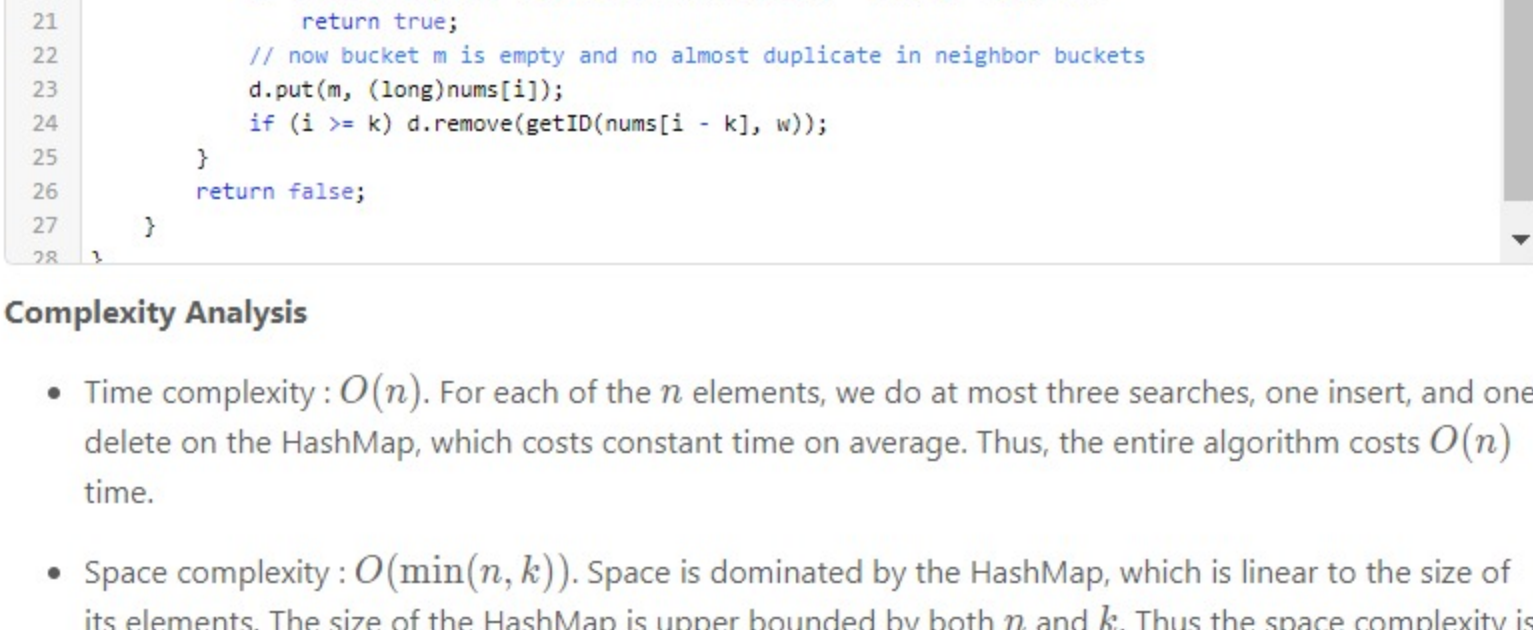


Figure 3. Illustration of buckets.

From the above example, we have 8 unsorted integers. We create 5 buckets covering the inclusive ranges of  $[0, 9]$ ,  $[10, 19]$ ,  $[20, 29]$ ,  $[30, 39]$ ,  $[40, 49]$  individually. Each of the eight elements is in a particular bucket. For element with value  $x$ , its bucket label is  $x/w$  and here we have  $w = 10$ . Sort each bucket using some other sorting algorithm and then collect all of them bucket by bucket.

Back to our problem, the critical issue we are trying to solve is:

1. For a given element  $x$  is there an item in the window that is within the range of  $[x - t, x + t]$ ?

2. Could we do this in constant time?

Let us consider an example where each element is a person's birthday. Your birthday, say some day in *March*, is the new element  $x$ . Suppose that each month has 30 days and you want to know if anyone has a birthday within  $t = 30$  days of yours. Immediately, we can rule out all other months except *February*, *March*, *April*.

The reason we know this is because each birthday belongs to a *bucket* we called *month*! And the range covered by the buckets are the same as distance  $t$  which simplifies things a lot. Any two elements that are not in the same or adjacent buckets must have a distance greater than  $t$ .

We apply the above bucketing principle and design buckets covering the ranges of  $\dots, [0, t]$ ,  $[t + 1, 2t + 1]$ ,  $\dots$ . We keep the window using this buckets. Then, each time, all we need to check is the bucket that  $x$  belongs to and its two adjacent buckets. Thus, we have a constant time algorithm for searching almost duplicate in the window.

One thing worth mentioning is the difference from bucket sort – Each of our buckets contains at most one element at any time, because two elements in a bucket means "almost duplicate" and we can return early from the function. Therefore, a HashMap with an element associated with a bucket label is enough for our purpose.

Java

Copy

```
1 public class Solution {
2     // get the ID of the bucket from element value x and bucket width w
3     // In Java, '-3 / 5 = 0' and but we need '-3 / 5 = -1'.
4     private long getID(long x, long w) {
5         return x < 0 ? (x + 1) / w - 1 : x / w;
6     }
7
8     public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {
9         if (t < 0) return false;
10        Map<Long, Long> d = new HashMap<>();
11        long w = (long)t + 1;
12        for (int i = 0; i < nums.length; ++i) {
13            long m = getID(nums[i], w);
14            // check if bucket m is empty, each bucket may contain at most one element
15            if (d.containsKey(m))
16                return true;
17            // check the neighbor buckets for almost duplicate
18            if (d.containsKey(m - 1) && Math.abs(nums[i] - d.get(m - 1)) < w)
19                return true;
20            if (d.containsKey(m + 1) && Math.abs(nums[i] - d.get(m + 1)) < w)
21                return true;
22            // now bucket m is empty and no almost duplicate in neighbor buckets
23            d.put(m, (long)nums[i]);
24            if (i >= k) d.remove(getID(nums[i - k], w));
25        }
26        return false;
27    }
28 }
```

Complexity Analysis

- Time complexity :  $O(n)$ . For each of the  $n$  elements, we do at most three searches, one insert, and one delete on the HashMap, which costs constant time on average. Thus, the entire algorithm costs  $O(n)$  time.
- Space complexity :  $O(\min(n, k))$ . Space is dominated by the HashMap, which is linear to the size of its elements. The size of the HashMap is upper bounded by both  $n$  and  $k$ . Thus the space complexity is  $O(\min(n, k))$ .

## See Also

- [Problem 217 Contains Duplicate](#)
- [Problem 219 Contains Duplicate II](#)

Rate this article: ★ ★ ★ ★ ★

Type comment here... (Markdown is supported)

Preview

Post

- anmingyu11 ★ 430 · March 15, 2019 8:14 PM

Only me TheApproach3 img broken in chrome?

43 · [Like](#) · [Share](#) · [Reply](#)

SHOW 1 REPLY
- Zizhen\_Huang ★ 92 · March 13, 2020 4:10 AM

This problem should be hard. Lol

12 · [Like](#) · [Share](#) · [Reply](#)
- yellowcola ★ 16 · April 6, 2018 9:49 PM

BST solution is incorrect in following case:  
[1, -2147483648]  
1  
-1  
You'll need to convert "t", nums[i] - floor & ceiling - nums[i] to long before comparison.

11 · [Like](#) · [Share](#) · [Reply](#)

SHOW 2 REPLIES
- zizhen1gwu ★ 9 · October 16, 2016 2:58 AM

C++'s std::set, std::set::upper\_bound and std::set::lower\_bound are NOT equivalent to Java's TreeSet.  
TreeSet::ceiling and TreeSet::floor

6 · [Like](#) · [Share](#) · [Reply](#)
- Patchouli ★ 12 · January 24, 2017 1:48 PM

C++'s equivalent of s.ceiling() is s.lower\_bound(), but there is no equivalent of s.floor(). To achieve s.floor() you have to use std::next(supper\_bound(), -1). Note std::set::iterator is a bidirectionaliterator not a randomAccessIterator so you can't use operator-.

5 · [Like](#) · [Share](#) · [Reply](#)
- coder2 ★ 108 · September 14, 2016 3:44 AM

Your first solution is wrong for Input:  
[-1,2147483647]  
1  
2147483647

Output:  
4 · [Like](#) · [Share](#) · [Reply](#)

Read More
- NYZhang ★ 39 · August 21, 2018 10:19 PM

This article is for intermediate readers [doge][doge][doge]

5 · [Like](#) · [Share](#) · [Reply](#)

SHOW 1 REPLY
- liuyang5832 ★ 2 · September 3, 2019 4:20 AM

I think solution 2 BST might have a problem when removing a number from set when set size is larger than k. when in the window there're two identical numbers, removal would be incorrect.

2 · [Like](#) · [Share](#) · [Reply](#)

SHOW 2 REPLIES
- gecho ★ 25 · May 27, 2020 12:36 AM

This solution is passing. should it?

```
class Solution(object):
    def containsNearbyAlmostDuplicate(self, nums, k, t):
        ...
```

1 · [Like](#) · [Share](#) · [Reply](#)

SHOW 1 REPLY
- luiscovar ★ 24 · March 29, 2020 11:04 AM

I'm having a hard time understanding why it's sufficient to only check the successor and predecessor and not everything in between k distance. Anyone care to explain?

1 · [Like](#) · [Share](#) · [Reply](#)

SHOW 2 REPLIES