219. Contains Duplicate II

#### **6 0 0**

March 20, 2016 | 26.9K views



Given an array of integers and an integer k, find out whether there are two distinct indices i and j in the array such that  $\mathbf{nums[i]} = \mathbf{nums[j]}$  and the  $\mathbf{absolute}$  difference between i and j is at most k.

#### Example 1:

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

#### Example 2:

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

```
Input: nums = [1,2,3,1,2,3], k = 2
Output: false
```

#### This article is for beginners. It introduces the following ideas: Linear Search, Binary Search Tree and Hash

Summary

Table.

Solution

Intuition

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

#### Look for duplicate element in the previous k elements.

Algorithm

elements instead of all its previous elements.

### This algorithm is the same as Approach #1 in Contains Duplicate solution, except that it looks at previous k

Another perspective of this algorithm is to keep a virtual sliding window of the previous k elements. We scan for the duplicate in this window.

Java

#### public boolean containsNearbyDuplicate(int[] nums, int k) {

```
for (int i = 0; i < nums.length; ++i) {
    for (int j = Math.max(i - k, 0); j < i; ++j) {
        if (nums[i] == nums[j]) return true;
    }
}
return false;
}
// Time Limit Exceeded.</pre>
Complexity Analysis
```

#### do at most n comparisons in one search even if k can be larger than n.

ullet Space complexity : O(1).

• Time complexity :  $O(n\min(k,n))$ . It costs  $O(\min(k,n))$  time for each linear search. Apparently we

Approach #2 (Binary Search Tree) [Time Limit Exceeded]

#### Keep a sliding window of $\boldsymbol{k}$ elements using self-balancing Binary Search Tree (BST).

#### Algorithm

Intuition

The key to improve upon Approach #1 above is to reduce the search time of the previous k elements. Can

## we use an auxiliary data structure to maintain a sliding window of k elements with more efficient search, delete, and insert operations? Since elements in the sliding window are strictly First-In-First-Out (FIFO),

queue is a natural data structure. A queue using a linked list implementation supports constant time <code>delete</code> and <code>insert</code> operations, however the <code>search</code> costs linear time, which is no better than Approach #1. A better option is to use a self-balancing BST. A BST supports <code>search</code>, <code>delete</code> and <code>insert</code> operations all in  $O(\log k)$  time, where k is the number of elements in the BST. In most interviews you are not required to implement a self-balancing BST, so you may think of it as a black box. Most programming languages provide

implementations of this useful data structure in its standard library. In Java, you may use a TreeSet or a TreeMap. In C++ STL, you may use a std::set or a std::map.
 If you already have such a data structure available, the pseudocode is:
 Loop through the array, for each element do
 Search current element in the BST, return true if found

Put current element in the BST
If the size of the BST is larger than k, remove the oldest item.

for (int i = 0; i < nums.length; ++i) {

set.add(nums[i]);

• Return false

Java

if (set.contains(nums[i])) return true;

- public boolean containsNearbyDuplicate(int[] nums, int k) {
   Set<Integer> set = new TreeSet<>();

## • Space complexity : $O(\min(n, k))$ . Space is the size of the sliding window which should not exceed n or k.

- Note  $\hbox{The algorithm still gets Time Limit Exceeded for large $n$ and $k$.}$
- Approach #3 (Hash Table) [Accepted]

## $\label{eq:continuition} % \end{substitute} % \end$

Algorithm

From the previous approaches, we know that even logarithmic performance in search is not enough. In this

## Loop through the array, for each element do Search current element in the HashTable, return true if found Put current element in the HashTable

If the size of the HashTable is larger than k, remove the oldest item.
 Return false

Java

case, we need a data structure supporting constant time search, delete and insert operations. Hash

Table is the answer. The algorithm and implementation are almost identical to Approach #2.

Set<Integer> set = new HashSet<>();
for (int i = 0; i < nums.length; ++i) {
 if (set.contains(nums[i])) return true;
 set.add(nums[i]);
 if (set.size() > k) {
 set.remove(nums[i - k]);
}

# Complexity Analysis Time complexity: O(n). We do n operations of search, delete and insert, each with constant time complexity. Space complexity: O(min(n, k)). The extra space required depends on the number of items stored in the hash table, which is the size of the sliding window, min(n, k). See Also

# Problem 220 Contains Duplicate III Rate this article: ★★★★★

Problem 217 Contains Duplicate

Preview

}

return false;

}

O Previous

Type comment here... (Markdown is supported)

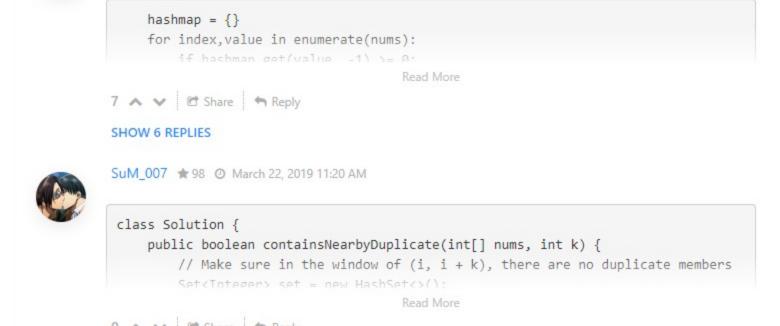
mikqs 🛊 11 🗿 February 22, 2019 12:46 AM

Python using hash map instead of hash set:

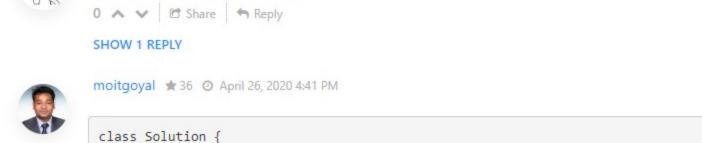
Comments: 7

Next **1** 

Post





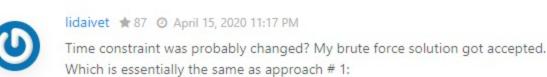


public boolean containsNearbyDuplicate(int[] nums, int k) {
 HashMap<Integer,Integer> map = new HashMap<>();

Read More

what's the difference between the code for Solution2 and Solution3 except the former is a TreeSet, the

mingyangdai1609 \* 10 \* O November 27, 2017 1:37 PM



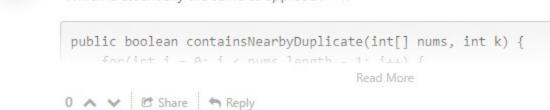
SHOW 1 REPLY

0 ∧ ∨ ☑ Share ¬ Reply

fishercoder # 774 ② January 1, 2020 4:09 AM

SHOW 1 REPLY

the code of Approach2 is wrong



for(int i = 0: i < nums.length: i++){