Previous Next

219. Contains Duplicate II

March 20, 2016 | 26.9K views

Average Rating: 4.37 (19 votes)

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 **nums[i] = nums[j]** and the **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
```

Example 3:

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

Summary

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

Solution

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

Intuition

Look for duplicate element in the previous k elements.

Algorithm

This algorithm is the same as Approach #1 in Contains Duplicate solution, except that it looks at previous k elements instead of all its previous elements.

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

```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.
```

Complexity Analysis

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

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

Intuition

Keep a sliding window of k elements using self-balancing Binary Search Tree (BST).

Algorithm

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 `delete` and `insert` operations, however the `search` costs linear time, which is *no better* than Approach #1.

A better option is to use a self-balancing BST. A BST supports `search`, `delete` and `insert` 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.
- Return `false`

Java

```java
public boolean containsNearbyDuplicate(int[] nums, int k) {
    Set<Integer> set = new TreeSet<>();
    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]);
        }
    }
    return false;
}
// Time Limit Exceeded.
```

Complexity Analysis

- Time complexity : $O(n \log(\min(k,n)))$. We do n operations of `search`, `delete` and `insert`. Each operation costs logarithmic time complexity in the sliding window which size is $\min(k, n)$. Note that even if k can be greater than n, the window size can never exceed n.
- Space complexity : $O(\min(n,k))$. Space is the size of the sliding window which should not exceed n or k.

Note

The algorithm still gets Time Limit Exceeded for large n and k.

Approach #3 (Hash Table) [Accepted]

Intuition

Keep a sliding window of k elements using Hash Table.

Algorithm

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

- 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

```java
public boolean containsNearbyDuplicate(int[] nums, int k) {
    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]);
        }
    }
    return false;
}
```

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 217 Contains Duplicate
- Problem 220 Contains Duplicate III

Rate this article:

Previous Next

mikqs ★ 11 February 22, 2019 12:46 AM

Python using hash map instead of hash set:

```
    hashmap = {}
    for index,value in enumerate(nums):
```

Read More

7 Share Reply

SHOW 6 REPLIES

SuM_007 ★ 98 March 22, 2019 11:20 AM

```
class Solution {
    public boolean containsNearbyDuplicate(int[] nums, int k) {
        // Make sure in the window of (i, i + k), there are no duplicate members
```

Read More

0 Share Reply

bernardkjr1990 ★ 13 April 18, 2018 10:02 AM

Solution2 should be set.remove(nums[i - k - 1]

0 Share Reply

SHOW 1 REPLY

mingyangdai1609 ★ 10 November 27, 2017 1:37 PM

the code of Approach2 is wrong

0 Share Reply

SHOW 1 REPLY

moitgoyal ★ 36 April 26, 2020 4:41 PM

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

Read More

0 Share Reply

lidaivet ★ 87 April 15, 2020 11:17 PM

Time constraint was probably changed? My brute force solution got accepted.
Which is essentially the same as approach # 1:

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

Read More

0 Share Reply

fishercoder ★ 774 January 1, 2020 4:09 AM

what's the difference between the code for Solution2 and Solution3 except the former is a TreeSet, the latter is a HashSet?
And actually the code for Solution2 is also accepted on OJ.

Is there any mistake for the actual code of Solution2? since the explanation of Solution2 says it's TLE.

Read More

0 Share Reply

SHOW 1 REPLY