★★★★★ Average Rating: 4.61 (105 votes)

Summary

This article is for beginners. It introduces the following ideas: Loop Invariant, Linear Search, Sorting and Hash Table.

Solution

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

Intuition

For an array of n integers, there are $C(n, 2) = \frac{n(n-1)}{2}$ pairs of integers. Thus, we may check all $\frac{n(n-1)}{2}$ pairs and see if there is any pair with duplicates.

Algorithm

To apply this idea, we employ the linear search algorithm which is the simplest search algorithm. Linear search is a method of finding if a particular value is in a list by checking each of its elements, one at a time and in sequence until the desired one is found.

For our problem, we loop through all n integers. For the ith integer `nums[i]`, we search in the previous `i-1` integers for the duplicate of `nums[i]`. If we find one, we return true; if not, we continue. Return false at the end of the program.

To prove the correctness of the algorithm, we define the loop invariant. A loop invariant is a property that holds before (and after) each iteration. Knowing its invariant(s) is essential for understanding the effect of a loop. Here is the *loop invariant*:

> Before the next search, there are no duplicate integers in the searched integers.

The loop invariant holds true before the loop because there is no searched integer. Each time through the loop we look for any any possible duplicate of the current element. If we found a duplicate, the function exits by returning true; If not, the invariant still holds true.

Therefore, if the loop finishes, the invariant tells us that there is no duplicate in all n integers.

Java

```java
public boolean containsDuplicate(int[] nums) {
    for (int i = 0; i < nums.length; ++i) {
        for (int j = 0; j < i; ++j) {
            if (nums[j] == nums[i]) return true;
        }
    }
    return false;
}
// Time Limit Exceeded
```

Complexity Analysis

- Time complexity : $O(n^2)$. In the worst case, there are $\frac{n(n-1)}{2}$ pairs of integers to check. Therefore, the time complexity is $O(n^2)$.
- Space complexity : $O(1)$. We only used constant extra space.

Note

This approach will get Time Limit Exceeded on LeetCode. Usually, if an algorithm is $O(n^2)$, it can handle n up to around 10^4. It gets Time Limit Exceeded when $n \geq 10^5$.

Approach #2 (Sorting) [Accepted]

Intuition

If there are any duplicate integers, they will be consecutive after sorting.

Algorithm

This approach employs sorting algorithm. Since comparison sorting algorithm like *heapsort* is known to provide $O(n \log n)$ worst-case performance, sorting is often a good preprocessing step. After sorting, we can sweep the sorted array to find if there are any two consecutive duplicate elements.

Java

```java
public boolean containsDuplicate(int[] nums) {
    Arrays.sort(nums);
    for (int i = 0; i < nums.length - 1; ++i) {
        if (nums[i] == nums[i + 1]) return true;
    }
    return false;
}
```

Complexity Analysis

- Time complexity : $O(n \log n)$. Sorting is $O(n \log n)$ and the sweeping is $O(n)$. The entire algorithm is dominated by the sorting step, which is $O(n \log n)$.
- Space complexity : $O(1)$. Space depends on the sorting implementation which, usually, costs $O(1)$ auxiliary space if `heapsort` is used.

Note

The implementation here modifies the original array by sorting it. In general, it is not a good practice to modify the input unless it is clear to the caller that the input will be modified. One may make a copy of `nums` and operate on the copy instead.

Approach #3 (Hash Table) [Accepted]

Intuition

Utilize a dynamic data structure that supports fast search and insert operations.

Algorithm

From Approach #1 we know that search operations is $O(n)$ in an unsorted array and we did so repeatedly. Utilizing a data structure with faster search time will speed up the entire algorithm.

There are many data structures commonly used as dynamic sets such as Binary Search Tree and Hash Table. The operations we need to support here are `search()` and `insert()`. For a self-balancing Binary Search Tree (TreeSet or TreeMap in Java), `search()` and `insert()` are both $O(\log n)$ time. For a Hash Table (HashSet or HashMap in Java), `search()` and `insert()` are both $O(1)$ on average. Therefore, by using hash table, we can achieve linear time complexity for finding the duplicate in an unsorted array.

Java

```java
public boolean containsDuplicate(int[] nums) {
    Set<Integer> set = new HashSet<>(nums.length);
    for (int x: nums) {
        if (set.contains(x)) return true;
        set.add(x);
    }
    return false;
}
```

Complexity Analysis

- Time complexity : $O(n)$. We do `search()` and `insert()` for n times and each operation takes constant time.
- Space complexity : $O(n)$. The space used by a hash table is linear with the number of elements in it.

Note

For certain test cases with not very large n, the runtime of this method can be slower than Approach #2. The reason is hash table has some overhead in maintaining its property. One should keep in mind that real world performance can be different from what the Big-O notation says. The Big-O notation only tells us that for *sufficiently* large input, one will be faster than the other. Therefore, when n is not sufficiently large, an $O(n)$ algorithm can be slower than an $O(n \log n)$ algorithm.

See Also

- Problem 219 Contains Duplicate II
- Problem 220 Contains Duplicate III

Report Article Issue

Comments: 156 Best Most Votes Newest to Oldest Oldest to Newest

```python
class Solution:
    def containsDuplicate(self, nums: List[int]) -> bool:

        vals = collections.Counter(nums).values()

        for i in vals :
            if i > 1 :
                return True
        return False
```

Testcase | Run Code Result | Debugger

Accepted Runtime: 57 ms

Your input [1,2,3,1]

Output true

Expected true

Diff

Console

Use Example Testcases

Run Code Submit