**128. Longest Consecutive Sequence**

<https://leetcode.com/problems/longest-consecutive-sequence/>

1. **Listen**

**Problem Statement:**

Given an unsorted array of integers nums, return *the length of the longest consecutive elements sequence.*

You must write an algorithm that runs in O(n) time.

**Input:**

unsorted array of integers **nums**

**Goal:**

find the longest consecutive elements sequence

**Return:**

return *the* ***length*** *(int) of the longest consecutive elements sequence*

1. **Examples**

**Example 1: -----------------------------------------------------------------------------------------------------**

**Input:** nums = [100,4,200,1,3,2]

**Output:** 4

**Explanation:** The longest consecutive elements sequence is [1, 2, 3, 4].

Therefore, its length is 4.

**Example 2: -----------------------------------------------------------------------------------------------------**

**Input:** nums = [0,3,7,2,5,8,4,6,0,1]

**Output:** 9

**Explanation:** The longest consecutive elements sequence is [0, 1, 2, 3, 4, 5, 6, 7, 8].

Therefore, its length is 9.

**Example 3: -----------------------------------------------------------------------------------------------------**

**Input:** nums = [4,75,3,1,74,2,76]

**Output:** 9

**Explanation:** There are two consecutive sequences in this array.

[1,2,3,4] and [74,75,76]

[1,2,3,4] is the longer of the two, so return 4.

**Constraints:**

* 0 <= nums.length <= 105
* -109 <= nums[i] <= 109
* Algorithm must run in O(n) time.
* There can be more than one consecutive sequence of numbers. Return the longest sequence.

**Test Cases:**

* Single consecutive sequence with rest of array random: [100,4,200,1,3,2]
* Single consecutive sequence in entire array: [0,3,7,2,5,8,4,6,0,1]
* Two consecutive sequences: [4,75,3,1,74,2,76]
* There is no consecutive sequence, so return 0: [25, 4, 90, 3, 17]
* Empty array: []

1. **Brute Force**

**Solution 1: Sorting – Time = O(nlogn) – Space = O(1)**

**Intuition:**

We can sort and iterate over the array, counting consecutive sequences of numbers along the way. We can keep a global max variable to keep track of the longest sequence if there are multiple.

Note that we are looking for the longest **consecutive** sequence of integers. Intuitively, once we sort the array, we can use a two-pointer sliding window to keep track of a consecutive number window.

The longest sequence in an empty array is, of course, 0, so we can simply return that. For all other cases, we sort nums and consider each number after the first (because we need to compare each number to its previous number). If the current number and the previous are equal, then our current sequence is neither extended nor broken, so we simply move on to the next number.

**Algorithm:**

Check for empty array.

Sort the array.

Have two variables: a local count variable set to 1 (in case of only one element in the array), and a global max count variable (in case of multiple consecutive sequences).

for all elements in the array:

if current index is equal to previous index plus 1

iterate local count

else

compare local count and global max

local count goes to 1

**Complexity Analysis**

**Time complexity: O(nlogn)**

The main for loop does constant work *n* times, so the algorithm's time complexity is dominated by the invocation of sort, which will run in O(nlogn) time for any sensible implementation.

**Space complexity: O(1) (or O(n)).**

For the implementations provided here, the space complexity is constant because we sort the input array in place. **If we are not allowed to modify the input array**, we must spend linear space to store a sorted copy.

1. **Optimize**

**Solution 2: HashSet and Sequence Building**

**Intuition:**

It turns out that our initial brute force solution was on the right track, but missing a few optimizations necessary to reach O(n) time complexity.

This optimized algorithm contains only two changes from the brute force approach: the numbers are stored in a HashSet (or Set, in Python) to allow O(1) lookups, and we only attempt to build sequences from numbers that are not already part of a longer sequence.

This is accomplished by first ensuring that the number that would immediately precede the current number in a sequence is not present, as that number would necessarily be part of a longer sequence.

**Algorithm**

Load all numbers from input array into hashset

A global max count variable (in case of multiple consecutive sequences).

for all elements in the array:

if hashset does not contain number preceding current element

A local count variable set to 1

A tracker to keep track of current number

while the hashset contains a consecutive (element+1) number

iterate local count

iterate tracker

compare local and global

**Complexity Analysis**

Time complexity: O(n)

Although the time complexity appears to be quadratic due to the while loop nested within the for loop, closer inspection reveals it to be linear.

Because the while loop is reached only when currentNum marks the beginning of a sequence (i.e. currentNum-1 is not present in nums), the while loop can only run for n iterations throughout the entire runtime of the algorithm.

For example, if we had an input array of [6,5,4,3,2,1], only the value 1 would pass the ‘if hashset does not contain number preceding current element’ test and go on to execute the while loop. All other values have their value - 1 in the set.

Therefore, the while loop only runs once, so despite looking like O(n⋅n) complexity, the nested loops actually run in O(n+n)=O(n) time.

Another corner example, 2, 5, 6, 7, 9, 11. All of these numbers are the "entrance" for the logic but the while loop doesn't run much. That is O(n) as well.

All other computations occur in constant time, so the overall runtime is linear.

Space complexity : O(n)

In order to set up O(1)containment lookups, we allocate linear space for a hashset to store the O(n) numbers in nums.

Other than that, the space complexity is identical to that of the brute force solution.

1. **Walkthrough**
2. **Implement**
3. **Test**