128. Longest Consecutive Sequence

Approach 1: Brute Force

Intuition

Because a sequence could start at any number in `nums`, we can exhaust the entire search space by building as long a sequence as possible from every number.

Algorithm

The brute force algorithm does not do anything clever - it just considers each number in `nums`, attempting to count as high as possible from that number using only numbers in `nums`. After it counts too high (i.e. `currentNum` refers to a number that `nums` does not contain), it records the length of the sequence if it is larger than the current best. The algorithm is necessarily optimal because it explores every possibility.

Java Python Copy

Complexity Analysis

- Time complexity : $O(n^3)$.

  The outer loop runs exactly n times, and because `currentNum` increments by 1 during each iteration of the `while` loop, it runs in $O(n)$ time. Then, on each iteration of the `while` loop, an $O(n)$ lookup in the array is performed. Therefore, this brute force algorithm is really three nested $O(n)$ loops, which compound multiplicatively to a cubic runtime.

- Space complexity : $O(1)$.

  The brute force algorithm only allocates a handful of integers, so it uses constant additional space.

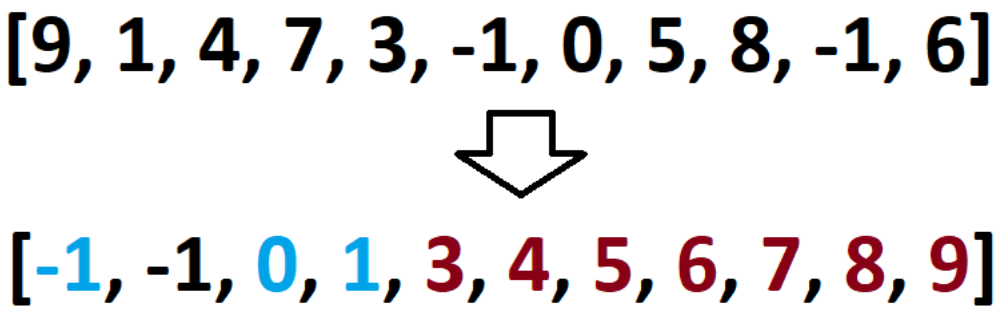
Approach 2: Sorting

Intuition

If we can iterate over the numbers in ascending order, then it will be easy to find sequences of consecutive numbers. To do so, we can sort the array.

Algorithm

Before we do anything, we check for the base case input of the empty array. 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. If they are unequal, then we must check whether the current number extends the sequence (i.e. `nums[i] == nums[i-1] + 1`). If it does, then we add to our current count and continue. Otherwise, the sequence is broken, so we record our current sequence and reset it to 1 (to include the number that broke the sequence). It is possible that the last element of `nums` is part of the longest sequence, so we return the maximum of the current sequence and the longest one.



Here, an example array is sorted before the linear scan identifies all consecutive sequences. The longest sequence is colored in red.

Java Python Copy

Complexity Analysis

- Time complexity : $O(nlgn)$.

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 $O(nlgn)$ 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.

Approach 3: HashSet and Intelligent 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.

Algorithm

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.

Java | Python — Copy

```
class Solution {
    public int longestConsecutive(int[] nums) {
        Set<Integer> num_set = new HashSet<Integer>();
        for (int num : nums) {
            num_set.add(num);
        }

        int longestStreak = 0;

        for (int num : num_set) {
            if (!num_set.contains(num-1)) {
                int currentNum = num;
                int currentStreak = 1;

                while (num_set.contains(currentNum+1)) {
                    currentNum += 1;
                    currentStreak += 1;
                }

                longestStreak = Math.max(longestStreak, currentStreak);
            }
        }

        return longestStreak;
    }
}
```

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. This means that despite looking like $O(n \cdot n)$ complexity, the nested loops actually run in $O(n + n) = O(n)$ time. 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 hash table to store the $O(n)$ numbers in `nums`. Other than that, the space complexity is identical to that of the brute force solution.

Comments: 16

Sort By

Type comment here... (Markdown is supported)

Preview Post

(/ryndvs96)

ryndvs96 (/ryndvs96) ★ 1 September 6, 2018 11:10 PM

I actually thought a nice solution is a quick union/find algorithm. Implement union by size and find with path compression and you got yourself a near linear algorithm. Not quite linear, but it'll still get accepted. :)

1 Share Reply

SHOW 2 REPLIES

(/anishsgm)

AnishSGM (/anishsgm) ★ 14 May 22, 2018 10:07 AM

If the list is very long, then the O(n) hash-set solution is a drain on memory so, not an elegant solution. If the list is small, then O(nlogn) shouldn't make much of a difference either. It's a stupid question.

12 Share Reply

SHOW 2 REPLIES

(/_shadow)

_shadow (/_shadow) ★ 0 March 2, 2018 11:43 AM

Amazing solutions! I don't understand how people come up with this lol Seems like it takes only genius lol my approach was to use a hashmap which stores the current number and maps it to a boolean and doublylinkedlist (i.e HashMap<Integer, <Boolean, DoublyLinkedList>>. For every number, add it to the list and mark num - 1 and num + 1 as false. every time you check for a number in the map and it's present, update the value by extending the list. Finally sweep through the map and return the longest.

Read More

0 Share Reply

SHOW 2 REPLIES

(/wcarvalho)

wcarvalho (/wcarvalho) ★ 5 January 8, 2018 3:49 PM

wouldn't [100,99, 98, ..., 1] (i.e. an array in reverse order) make this run in $$O(n^2)$$ because for every num, it have to go through the entire set so far?

5 Share Reply

SHOW 4 REPLIES

(/leetcode_deleted_user)

leetcode_deleted_user (/leetcode_deleted_user) ★ 72 January 1, 2018 6:34 PM

@shlok.gandhi, the `in` operator is overloaded, so it functions different depending on what container is used. For a `list`, the complexity is O(n), but we are using a `set` here (which has O(1) complexity).

0 Share Reply

(/shlokgandhi)

shlok.gandhi (/shlokgandhi) ★ 0 January 1, 2018 4:51 PM

@emptyset
I think the complexity is O(n*n) because the 'in' operator in python is of complexity O(n)*
'in' operator inside of for loop would give complexity to be O(nn)

ref: https://wiki.python.org/moin/TimeComplexity (https://wiki.python.org/moin/TimeComplexity)

Read More

0 Share Reply