LeetCode 第 128 号问题：最长连续序列

> *本文首发于公众号「图解面试算法」，是 图解 LeetCode 系列文章之一。*
>
> *同步博客：https://www.algomooc.com*

题目来源于 LeetCode 上第 128 号问题：最长连续序列。题目难度为 Hard，目前通过率为 48.5% 。

题目描述

给定一个未排序的整数数组，找出最长连续序列的长度。

要求算法的时间复杂度为 O(n)。

示例 1:

```
输入: [100, 4, 200, 1, 3, 2]
输出: 4
解释: 最长连续序列是 [1, 2, 3, 4]。它的长度为 4。
```

题目解析

题目直接明了，给你一个未排序的数组，让你从中找出一些元素，使这些元素能够组成最长 **连续的递增序列**，输出这个序列的长度，元素的先后没有关系，比如：

```
[100, 4, 200, 1, 3, 2]

可以找出 4, 1, 3, 2 组成连续递增序列 1, 2, 3, 4

输出这个序列长度 4
```

很直接的想法是把数组排序一下，然后遍历一遍就可以找到答案，但是这道题目的难点在于它限制时间复杂度为 O(n)，这样一来，排序这条路走不通。

这道题目其实有一个特征，就是这道题目隐含着 **连通性** 这个性质在里面，怎么讲？我们还是拿上面那个例子来举例：

```
[100, 4, 200, 1, 3, 2]

我们从左向右枚举数组里面的元素，你可以认为枚举过的元素是有效的：
.................100     枚举第一个元素，此时有 1 个连通区域
...4............100     枚举第二个元素，第二个元素和前面的元素互不相连，此时有 2 个连通区域
...4............100.............200    枚举第三个元素，三个元素互不相连，此时有 3 个连通区域
1..4............100.............200    枚举第四个元素，四个元素互不相连，此时有 4 个连通区域
1.34............100.............200    枚举第五个元素，这个元素和之前第二个连通区域相连，连通区域维持在 4 个
1234............100.............200    枚举第六个元素，这个元素和两个连通区域相连，连通区域变成 3 个

最后包含元素最多的那个连通区域所包含的元素个数就是我们要的答案
```

知道了这些东西对我们解题有什么帮助呢？关于连通性的问题，首先要想到的一个数据结构就是 **并查集**，这个数据结构的设计初衷就是为了解决连通性的问题，而且它的两个操作，查找以及合并的时间复杂度可以近似看成是 O(1)，因此用来解决这道题目再适合不过了。

如果你能想到并查集，那么这道题目其实就没有更多的难点，但我想说的是，这道题目其实还有一个比较有趣的解法，是利用 HashMap 来记录边界点所涵盖的连通区块长度，还是跟着例子走一遍：

```
[100, 4, 200, 1, 3, 2]

我们还是从左向右枚举数组里面的元素，每次遍历都去看这个元素的左右是否存在，并更新 HashMap：
100   此时 99 以及 101 都没有任何区块，Map {100=1}，表示 100 这个区块大小为 1
4     此时 3 以及 5 都没有任何区块，Map {100=1, 4=1}
200   此时 199 以及 201 都没有任何区块，Map {100=1, 4=1, 200=1}
1     此时 0 以及 2 都没有任何区块，Map {100=1, 4=1, 200=1, 1=1}
3     发现 4 是存在的，4，3 形成一个新的区块，
      这个区块的左边界是 3，右边界是 4，区块大小是 2，
      Map 中更新边界元素所代表的区块大小，Map {100=1, 4=2, 200=1, 1=1, 3=2}
2     发现左右边界同时存在，1，2，3，4 形成一个新的区块
      这个区块的左边界是 1，右边界是 4，区块大小是 4，
      Map 中更新边界元素所代表的区块大小，并记录当前元素避免重复访问，
      Map {100=1, 4=4, 200=1, 1=4, 3=2, 2=4}
```

可以看到，每次记录的时候，我们只需要保证区块的边界元素所表示的区块大小是正确的即可，至于区块中间的元素其实无所谓，因为这些元素并不会被再次访问到

这个方法其实挺巧妙的，通过利用哈希表的元素向左右延伸来确定区块的大小。

代码实现（并查集）

```java
class Solution {
    // roots 用来记录一个连通区域的代表元素
    private Map<Integer, Integer> roots = new HashMap<>();

    // counts 用来记录一个连通区域的元素个数
    private Map<Integer, Integer> counts = new HashMap<>();

    private int find(int a) {
        if (roots.get(a) == a)  {
            return a;
        }

        int root = find(roots.get(a));

        // 路径压缩
        roots.put(a, root);

        return root;
    }

    private void union(int a, int b) {
        int rootA = find(a);
```

```
        int rootB = find(b);

        if (rootA != rootB) {
            roots.put(rootA, rootB);

            // 两个连通区域合并，更新整个区域的元素个数
            counts.put(rootB, counts.get(rootA) + counts.get(rootB));
        }
    }

    public int longestConsecutive(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        for (int i = 0; i < nums.length; ++i) {
            if (roots.containsKey(nums[i])) {
                continue;
            }

            roots.put(nums[i], nums[i]);
            counts.put(nums[i], 1);

            // 查看相邻元素是否存在连通区块
            if (roots.containsKey(nums[i] - 1) && roots.containsKey(nums[i] + 1)) {
                int root = find(roots.get(nums[i] - 1));

                // 左右都存在连通区域，合并这三个区域
                union(nums[i], root);
                union(root, roots.get(nums[i] + 1));
            } else if (roots.containsKey(nums[i] - 1)) {
                int root = find(roots.get(nums[i] - 1));

                // 左边存在连通区域，合并这这两个区域
                union(nums[i], root);
            } else if (roots.containsKey(nums[i] + 1)) {
                int root = find(roots.get(nums[i] + 1));

                // 右边存在连通区域，合并这这两个区域
                union(nums[i], root);
            }
        }

        int result = 1;

        // 遍历所有连通区块，找到包含元素最多的区块
        for (int i : counts.keySet()) {
            result = Math.max(result, counts.get(i));
        }

        return result;
```

```
    }
}
```

动画描述（并查集）

代码实现（哈希表）

```
public int longestConsecutive(int[] nums) {
    if (nums == null || nums.length == 0) {
        return 0;
    }

    Map<Integer, Integer> distances = new HashMap<>();

    int result = 1;

    for (int num : nums) {
        if (distances.containsKey(num)) {
            continue;
        }

        // 查找向左能够延伸的最长距离
        int left = distances.getOrDefault(num - 1, 0);

        // 查找向右能够延伸的最长距离
        int right = distances.getOrDefault(num + 1, 0);

        // 更新此时的左右边界所表示的区块大小
        distances.put(num - left, left + right + 1);
        distances.put(num + right, left + right + 1);

        // 数组中可能存在重复元素，记录当前元素，避免再次访问
        distances.put(num, left + right + 1);

        result = Math.max(result, left + right + 1);
    }

    return result;
}
```

动画描述（哈希表）