Split Array into Consecutive Subsequences

You are given an integer array sorted in ascending order (may contain duplicates), you need to split them into several subsequences, where each subsequences consist of at least 3 consecutive integers. Return whether you can make such a split.

Example 1:

```
Input: [1,2,3,3,4,5]
Output: True
Explanation:
You can split them into two consecutive subsequences :
1, 2, 3
3, 4, 5
```

Example 2:

```
Input: [1,2,3,3,4,4,5,5]
Output: True
Explanation:
You can split them into two consecutive subsequences :
1, 2, 3, 4, 5
3, 4, 5
```

Example 3:

```
Input: [1,2,3,4,4,5]
Output: False
```

Note:

1. The length of the input is in range of [1, 10000]

Solution 1

1. We iterate through the array once to get the frequency of all the elements in the array
2. We iterate through the array once more and for each element we either see if it can be appended to a previously constructed consecutive sequence or if it can be the start of a new consecutive sequence. If neither are true, then we return false.

```
public boolean isPossible(int[] nums) {
    Map<Integer, Integer> freq = new HashMap<>(), appendfreq = new HashMap<>();
    for (int i : nums) freq.put(i, freq.getOrDefault(i,0) + 1);
    for (int i : nums) {
        if (freq.get(i) == 0) continue;
        else if (appendfreq.getOrDefault(i,0) > 0) {
            appendfreq.put(i, appendfreq.get(i) - 1);
            appendfreq.put(i+1, appendfreq.getOrDefault(i+1,0) + 1);
        }
        else if (freq.getOrDefault(i+1,0) > 0 && freq.getOrDefault(i+2,0) > 0) {
            freq.put(i+1, freq.get(i+1) - 1);
            freq.put(i+2, freq.get(i+2) - 1);
            appendfreq.put(i+3, appendfreq.getOrDefault(i+3,0) + 1);
        }
        else return false;
        freq.put(i, freq.get(i) - 1);
    }
    return true;
}
```

written by compton_scatter original link here

Solution 2

The basic idea is that, for each distinct element `ele` in the input array, we only need to maintain three pieces of information: the number of consecutive subsequences ending at `ele` with length of `1`, length of `2` and length `>= 3`.

The input array will be scanned linearly from left to right. Let `cur` be the element currently being examined and `cnt` as its number of appearance. `pre` is the element processed immediately before `cur`. The number of consecutive subsequences ending at `pre` with length of `1`, length of `2` and length `>= 3` are denoted as `p1`, `p2` and `p3`, respectively. There are two cases in general:

1. `cur != pre + 1`: for this case, `cur` cannot be added to any consecutive subsequences ending at `pre`, therefore, we must have `p1 == 0 && p2 == 0`; otherwise the input array cannot be split into consecutive subsequences of length `>= 3`. Now let `c1, c2, c3` be the number of consecutive subsequences ending at `cur` with length of `1`, length of `2` and length `>= 3`, respectively, we will have `c1 = cnt, c2 = 0, c3 = 0`, which means we only have consecutive subsequence ending at `cur` with length of `1` and its number given by `cnt`.
2. `cur == pre + 1`: for this case, `cur` can be added to consecutive subsequences ending at `pre` and thus extend those subsequences. But priorities should be given to those with length of `1` first, then length of `2` and lastly length `>= 3`. Also we must have `cnt >= p1 + p2`; otherwise the input array cannot be split into consecutive subsequences of length `>= 3`. Again let `c1, c2, c3` be the number of consecutive subsequences ending at `cur` with length of `1`, length of `2` and length `>= 3`, respectively, we will have: `c2 = p1, c3 = p2 + min(p3, cnt - (p1 + p2)), c1 = max(cnt - (p1 + p2 + p3), 0)`. The meaning is as follows: first adding `cur` to the end of subsequences of length `1` will make them subsequences of length `2`, and we have `p1` such subsequences, therefore `c2 = p1`. Then adding `cur` to the end of subsequences of length `2` will make them subsequences of length `3`, and we have `p2` such subsequences, therefore `c3` is at least `p2`. If `cnt > p1 + p2`, we can add the remaining `cur` to the end of subsequences of length `>= 3` to make them even longer subsequences. The number of such subsequences is the smaller one of `p3` and `cnt - (p1 + p2)`. In total, `c3 = p2 + min(p3, cnt - (p1 + p2))`. If `cnt > p1 + p2 + p3`, then we still have remaining `cur` that cannot be added to any subsequences. These residue `cur` will form subsequences of length `1`, hence `c1 = max(cnt - (p1 + p2 + p3), 0)`.

For either case, we need to update: `pre = cur, p1 = c1, p2 = c2, p3 = c3` after processing the current element. When all the elements are done, we check the values of `p1` and `p2`. The input array can be split into consecutive subsequences of length `>= 3` if and only if `p1 == 0 && p2 == 0`.

Here is the `O(n)` time and `O(1)` space Java solution:

```java
public boolean isPossible(int[] nums) {
    int pre = Integer.MIN_VALUE, p1 = 0, p2 = 0, p3 = 0;
    int cur = 0, cnt = 0, c1 = 0, c2 = 0, c3 = 0;

    for (int i = 0; i < nums.length; pre = cur, p1 = c1, p2 = c2, p3 = c3) {
        for (cur = nums[i], cnt = 0; i < nums.length && cur == nums[i]; cnt++, i++);

        if (cur != pre + 1) {
            if (p1 != 0 || p2 != 0) return false;
            c1 = cnt; c2 = 0; c3 = 0;

        } else {
            if (cnt < p1 + p2) return false;
            c1 = Math.max(0, cnt - (p1 + p2 + p3));
            c2 = p1;
            c3 = p2 + Math.min(p3, cnt - (p1 + p2));
        }
    }

    return p1 == 0 && p2 == 0;
}
```

written by fun4LeetCode original link here

Solution 3

The Algorithm

Maintain a set of consecutive sequences, call this set `s`. `s` begins as an empty set of consecutive sequences.

Now, iterate through each `num` in `nums`. For each iteration, if there exists a consecutive sequence in `s` that ends with element `num-1`, then append `num` to the end of the shortest such sequence; otherwise, create a new sequence that begins with `num`.

The problem has a solution (i.e. the array can be split into consecutive subsequences such that each subsequence consists of at least 3 consecutive integers) if and only if each sequence in `s` has size greater than or equal to 3.

Proof of Algorithm

Why does this algorithm work? It was intuitive to me, but I could not indisputably prove that it was correct. Hopefully, someone else can prove it.

Implementation

We don't need to actually store each sequence. Instead, we just need to know (1) the number of sequences that end at a particular element, and (2) the size of each of those sequences. To implement this, we can have an unordered map `backs` to represent the sequences: `backs[key]` returns a priority queue (smallest value at top) of the sizes of all sequences that end with element `key`. Now that we have (1) and (2), we can implement the algorithm above without knowing each particular sequence.

For each `num` in `nums`, if there exists any sequence that ends with `num-1` (i.e. if `backs[num-1]` is a non-empty priority queue), then find such a sequence with the smallest possible size (get the smallest value from the priority queue at `backs[num-1]`). Now, the sequence will be extended by 1 since we will add `num` to it. So pop the smallest value `count` from the priority queue at `backs[num-1]`, and add a new value `count+1` to the priority queue at `backs[num]`.

If no sequence was found that ends in `num-1` (i.e. `backs[num-1]` is empty), then create a new sequence. In other words, add `1` to the priority queue at `backs[num]`.

The Code

```
class Solution {
public:
 bool isPossible(vector<int>& nums)
 {
  unordered_map<int, priority_queue<int, vector<int>, std::greater<int>>> backs;

  // Keep track of the number of sequences with size < 3
  int need_more = 0;

  for (int num : nums)
  {
   if (! backs[num - 1].empty())
   { // There exists a sequence that ends in num-1
    // Append 'num' to this sequence
    // Remove the existing sequence
    // Add a new sequence ending in 'num' with size incremented by 1
    int count = backs[num - 1].top();
    backs[num - 1].pop();
    backs[num].push(++count);

    if (count == 3)
     need_more--;
   }
   else
   { // There is no sequence that ends in num-1
    // Create a new sequence with size 1 that ends with 'num'
    backs[num].push(1);
    need_more++;
   }
  }
  return need_more == 0;
 }
};
```

Improvements

I know that there are O(n) solutions out there that use different algorithms. Can my algorithm be implemented more efficiently to be O(n) instead of O(nlogn)?

Thoughts?

written by jay520 original link here