Wiggle Subsequence

A sequence of numbers is called a **wiggle sequence** if the differences between successive numbers strictly alternate between positive and negative. The first difference (if one exists) may be either positive or negative. A sequence with fewer than two elements is trivially a wiggle sequence.

For example, `[1,7,4,9,2,5]` is a wiggle sequence because the differences (6,-3,5,-7,3) are alternately positive and negative. In contrast, `[1,4,7,2,5]` and `[1,7,4,5,5]` are not wiggle sequences, the first because its first two differences are positive and the second because its last difference is zero.

Given a sequence of integers, return the length of the longest subsequence that is a wiggle sequence. A subsequence is obtained by deleting some number of elements (eventually, also zero) from the original sequence, leaving the remaining elements in their original order.

Examples:

```
Input: [1,7,4,9,2,5]
Output: 6
The entire sequence is a wiggle sequence.

Input: [1,17,5,10,13,15,10,5,16,8]
Output: 7
There are several subsequences that achieve this length. One is [1,17,10,13,10,16,8
].

Input: [1,2,3,4,5,6,7,8,9]
Output: 2
```

Follow up:
Can you do it in O(n) time?

Credits:
Special thanks to @agave and @StefanPochmann for adding this problem and creating all test cases.

Solution 1

In Wiggle Subsequence, think that the solution we need should be in a way that we get alternative higher, lower,higher number.
Eg: 2, 5, 3, 8, 6, 9
In above example, the sequence of numbers is small,big,small,big,small,big numbers (In shape of hill).

Now for explanation, we take example series:
2,1,4,5,6,3,3,4,8,4

First we check if the series is starting as (big, small) or (small, big). So as 2,1 is big, small. So we will start the loop as we need small number first that is 1 as 2 is already there.

```
Step 1: First we check our requirement is to get small number. As 1<2 so the seri
es will be
 2,1
```

```
Step 2: Now we need big number that is  greater than 1. As 4>1 so series  will be
2,1,4
```

```
Step 3: Now we need small number. But 5>4 so 4 will be replaced by 5. So the seri
es will become
2,1,5
```

```
Step 4:  We need small number. But 6>5. Series will be
2,1,6
```

```
Step 5: Require small number. 3<6. Series will be
2,1,6,3
```

```
Step 6: Require big number. 3=3. No change in series
2,1,6,3
```

```
Step 7: Require big number. 4>3. Series will become
2,1,6,3,4
```

```
Step 8:  Require small number. 8>4. 8 will  replace 4 and series will become
2,1,6,3,8
```

```
Step 9: Require small number. 4<8. So final series will  be
2,1,6,3,8,4
```

Answer is 6.

In the code, for constant space O(1) we will modify the same 'num' array to store the (small, big, small) hill shape values. So the code will not only calculate the length of the sequence but if the interviewers asks for the Wiggle series also then we can return the series too. The leetcode Online Judge skipped a test case if the series starts with same set of numbers. Thanks to @ztq63830398, modified the code to consider that test case also.

Code:

```
    public class Solution {
 public int wiggleMaxLength(int[] nums) {
  if (nums.length == 0 || nums.length == 1) {
   return nums.length;
  }
  int k = 0;
  while (k < nums.length - 1 && nums[k] == nums[k + 1]) {  //Skips all the same n
umbers from series beginning eg 5, 5, 5, 1
   k++;
  }
  if (k == nums.length - 1) {
   return 1;
  }
  int result = 2;     // This will track the result of result array
  boolean smallReq = nums[k] < nums[k + 1];       //To check series starting patt
ern
  for (int i = k + 1; i < nums.length - 1; i++) {
   if (smallReq && nums[i + 1] < nums[i]) {
    nums[result] = nums[i + 1];
    result++;
    smallReq = !smallReq;    //Toggle the requirement from small to big number
   } else {
    if (!smallReq && nums[i + 1] > nums[i]) {
     nums[result] = nums[i + 1];
     result++;
     smallReq = !smallReq;    //Toggle the requirement from big to small number
    }
   }
  }
  return result;
 }
}
```

written by keshavk original link here

Solution 2

DP:

```
class Solution {
public:
    int wiggleMaxLength(vector<int>& nums) {
        int size=nums.size();
        if(size==0) return 0;
        vector<int> f(size, 1);
        vector<int> d(size, 1);
        for(int i=1; i<size; ++i){
            for(int j=0; j<i; ++j){
                if(nums[j]<nums[i]){
                    f[i]=max(f[i], d[j]+1);
                }
                else if(nums[j]>nums[i]){
                    d[i]=max(d[i], f[j]+1);
                }
            }
        }
        return max(d.back(), f.back());
    }
};
```

Greedy:

```
class Solution {
public:
    int wiggleMaxLength(vector<int>& nums) {
        int size=nums.size(), f=1, d=1;
        for(int i=1; i<size; ++i){
                 if(nums[i]>nums[i-1]) f=d+1;
            else if(nums[i]<nums[i-1]) d=f+1;
        }
        return min(size, max(f, d));
    }
};
```

written by clubmaster original link here

Solution 3

```
def wiggleMaxLength(self, nums):
    nan = float('nan')
    diffs = [a-b for a, b in zip([nan] + nums, nums + [nan]) if a-b]
    return sum(not d*e >= 0 for d, e in zip(diffs, diffs[1:]))
```

Explanation / Proof:

Imagine the given array contains [..., **10, 10, 10, 10**, ...]. Obviously we can't use more than one of those tens, as that wouldn't be wiggly. So right away we can ignore all consecutive duplicates.

Imagine the given array contains [..., 10, **7, 11, 13, 17, 19, 23** , 20, ...]. So increasing from 7 to 23. What can we do with that? Well we can't use more than two of those increasing numbers, as that wouldn't be wiggly. And if we do use two, we'd better use the 7 and the 23, as that offers the best extensibility (for example, the 19 wouldn't allow to next pick the 20 for the wiggly subsequence). And if we do use only one, it still should be either the 7 or the 23, as the 7 is the best wiggle-low and the 23 is the best wiggle-high of them. So whether we actually use the 7 and the 23 or not, we definitely can and should remove the 11, 13, 17, and 19. So then we have [..., 10, **7, 23**, 20, ...]. Now, notice that the 7 is a local minimum (both the 10 and the 23 are larger) and the 23 is a local maximum. And if we do this with **all** increasing or decreasing streaks, i.e., keep only their first and last number, then all the numbers we have left are local extrema, either smaller than both neighbors or larger than both neighbors. Which means that at that point, we're already fully wiggly. And we only removed as many numbers as we have to. So it's a longest possible wiggly subsequence.

My solution first computes differences of neighbors and throws out zeros (which does get rid of those useless consecutive duplicates). And then it just counts the local extrema (by checking two consecutive differences).

I use `nan` for some convenience, I'll let you figure that part out :-)

Alternative implementations not using the `nan` trick: First remove repetitions, then count the local extrema except the ends, and add the number of ends (because the ends are always local extrema)

```
def wiggleMaxLength(self, nums):
    norep = [num for num, _ in itertools.groupby(nums)]
    triples = zip(norep, norep[1:], norep[2:])
    return sum((b>a) == (b>c) for a, b, c in triples) + len(norep[:2])

def wiggleMaxLength(self, nums):
    norep = [num for num, _ in itertools.groupby(nums)]
    if len(norep) < 2: return len(norep)
    triples = zip(norep, norep[1:], norep[2:])
    return 2 + sum(a<b>c or a>b<c for a, b, c in triples)
```

written by StefanPochmann original link here

From Leetcoder.