Previous Next

376. Wiggle Subsequence

July 30, 2016 | 39.5K views

Average Rating: 4.66 (77 votes)

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.

Example 1:

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

Example 2:

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

Example 3:

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

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

Summary

We need to find the length of the longest wiggle subsequence. A wiggle subsequence consists of a subsequence with numbers which appears in alternating ascending / descending order.

Solution

Approach #1 Brute Force [Time Limit Exceeded]

Here, we can find the length of every possible wiggle subsequence and find the maximum length out of them. To implement this, we use a recursive function, $\text{calculate}(\text{nums}, \text{index}, \text{isUp})$ which takes the array nums, the index from which we need to find the length of the longest wiggle subsequence, boolean variable isUp to tell whether we need to find an increasing wiggle or decreasing wiggle respectively. If the function calculate is called after an increasing wiggle, we need to find the next decreasing wiggle with the same function. If the function calculate is called after a decreasing wiggle, we need to find the next increasing wiggle with the same function.

Java Copy

```java
public class Solution {
    private int calculate(int[] nums, int index, boolean isUp) {
        int maxcount = 0;
        for (int i = index + 1; i < nums.length; i++) {
            if ((isUp && nums[i] > nums[index]) || (!isUp && nums[i] < nums[index]))
                maxcount = Math.max(maxcount, 1 + calculate(nums, i, !isUp));
        }
        return maxcount;
    }

    public int wiggleMaxLength(int[] nums) {
        if (nums.length < 2)
            return nums.length;
        return 1 + Math.max(calculate(nums, 0, true), calculate(nums, 0, false));
    }
}
```

Complexity Analysis

- Time complexity : $O(n!)$. $\text{calculate}()$ will be called maximum $n!$ times.
- Space complexity : $O(n)$. Recursion of depth n is used.

Approach #2 Dynamic Programming [Accepted]

Algorithm

To understand this approach, take two arrays for dp named up and $down$.

Whenever we pick up any element of the array to be a part of the wiggle subsequence, that element could be a part of a rising wiggle or a falling wiggle depending upon which element we have taken prior to it.

$up[i]$ refers to the length of the longest wiggle subsequence obtained so far considering i^{th} element as the last element of the wiggle subsequence and ending with a rising wiggle.

Similarly, $down[i]$ refers to the length of the longest wiggle subsequence obtained so far considering i^{th} element as the last element of the wiggle subsequence and ending with a falling wiggle.

$up[i]$ will be updated every time we find a rising wiggle ending with the i^{th} element. Now, to find $up[i]$, we need to consider the maximum out of all the previous wiggle subsequences ending with a falling wiggle i.e. $down[j]$, for every $j < i$ and $nums[i] > nums[j]$. Similarly, $down[i]$ will be updated.

Java Copy

```java
public class Solution {
    public int wiggleMaxLength(int[] nums) {
        if (nums.length < 2)
            return nums.length;
        int[] up = new int[nums.length];
        int[] down = new int[nums.length];
        for (int i = 1; i < nums.length; i++) {
            for(int j = 0; j < i; j++) {
                if (nums[i] > nums[j]) {
                    up[i] = Math.max(up[i],down[j] + 1);
                } else if (nums[i] < nums[j]) {
                    down[i] = Math.max(down[i],up[j] + 1);
                }
            }
        }
        return 1 + Math.max(down[nums.length - 1], up[nums.length - 1]);
    }
}
```

Complexity Analysis

- Time complexity : $O(n^2)$. Loop inside a loop.
- Space complexity : $O(n)$. Two arrays of the same length are used for dp.

Approach #3 Linear Dynamic Programming [Accepted]

Algorithm

Any element in the array could correspond to only one of the three possible states:

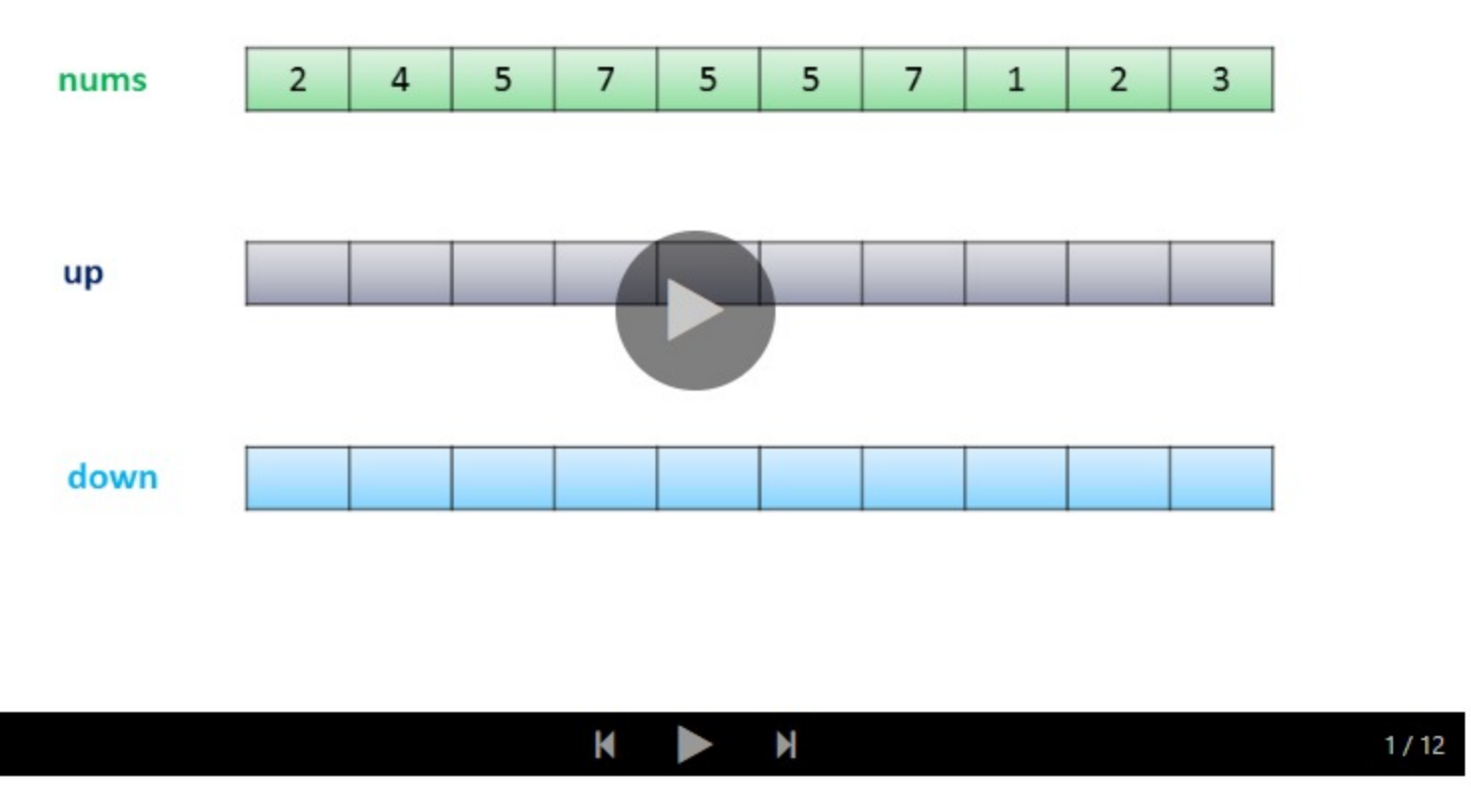
1. up position, it means $nums[i] > nums[i-1]$
2. down position, it means $nums[i] < nums[i-1]$
3. equals to position, $nums[i] == nums[i-1]$

The updates are done as:

If $nums[i] > nums[i-1]$, that means it wiggles up. The element before it must be a down position. So $up[i] = down[i-1] + 1$, $down[i]$ remains the same as $down[i-1]$. If $nums[i] < nums[i-1]$, that means it wiggles down. The element before it must be a up position. So $down[i] = up[i-1] + 1$, $up[i]$ remains the same as $up[i-1]$. If $nums[i] == nums[i-1]$, that means it will not change anything becaue it didn't wiggle at all. So both $down[i]$ and $up[i]$ remain the same as $down[i-1]$ and $up[i-1]$.

At the end, we can find the larger out of $up[length-1]$ and $down[length-1]$ to find the max. wiggle subsequence length, where $length$ refers to the number of elements in the given array.

The process can be illustrated with the following example:



Java Copy

```java
public class Solution {
    public int wiggleMaxLength(int[] nums) {
        if (nums.length < 2)
            return nums.length;
        int[] up = new int[nums.length];
        int[] down = new int[nums.length];
        up[0] = down[0] = 1;
        for (int i = 1; i < nums.length; i++) {
            if (nums[i] > nums[i - 1]) {
                up[i] = down[i - 1] + 1;
                down[i] = down[i - 1];
            } else if (nums[i] < nums[i - 1]) {
                down[i] = up[i - 1] + 1;
                up[i] = up[i - 1];
            } else {
                down[i] = down[i - 1];
                up[i] = up[i - 1];
            }
        }
        return Math.max(down[nums.length - 1], up[nums.length - 1]);
    }
}
```

Complexity Analysis

- Time complexity : $O(n)$. Only one pass over the array length.
- Space complexity : $O(n)$. Two arrays of the same length are used for dp.

Approach #4 Space-Optimized Dynamic Programming [Accepted]

Algorithm

This approach relies on the same concept as Approach #3. But we can observe that in the DP approach, for updating elements $up[i]$ and $down[i]$, we need only the elements $up[i-1]$ and $down[i-1]$. Thus, we can save space by not using the whole array, but only the last elements.

Java Copy

```java
public class Solution {
    public int wiggleMaxLength(int[] nums) {
        if (nums.length < 2)
            return nums.length;
        int down = 1, up = 1;
        for (int i = 1; i < nums.length; i++) {
            if (nums[i] > nums[i - 1])
                up = down + 1;
            else if (nums[i] < nums[i - 1])
                down = up + 1;
        }
        return Math.max(down, up);
    }
}
```

Complexity Analysis

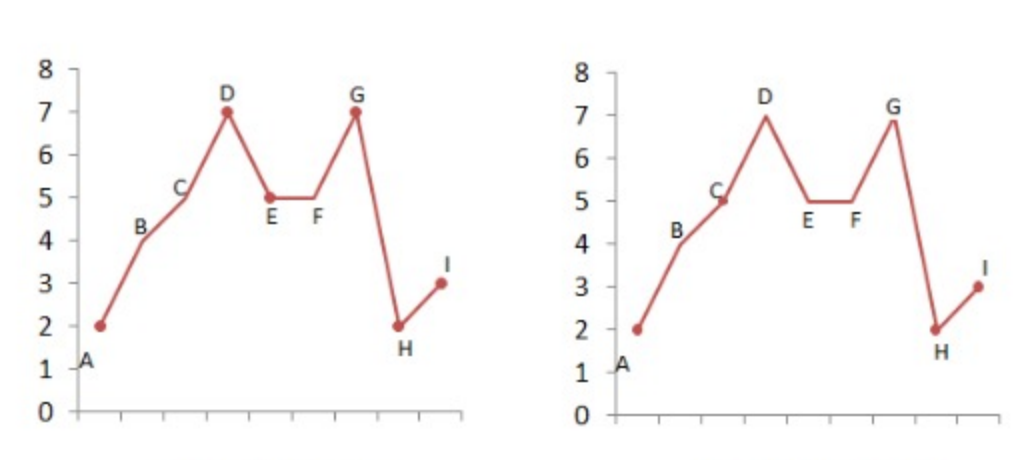
- Time complexity : $O(n)$. Only one pass over the array length.
- Space complexity : $O(1)$. Constant space is used.

Approach #5 Greedy Approach [Accepted]

Algorithm

We need not necessarily need dp to solve this problem. This problem is equivalent to finding the number of alternating max. and min. peaks in the array. Since, if we choose any other intermediate number to be a part of the current wiggle subsequence, the maximum length of that wiggle subsequence will always be less than or equal to the one obtained by choosing only the consecutive max. and min. elements.

This can be clarified by looking at the following figure:



Wiggle Subsequence (choosing D as 2nd point): ADEGHI

Wiggle Subsequence (choosing C as the 2nd point): ACHI

From the above figure, we can see that if we choose **C** instead of **D** as the 2nd point in the wiggle subsequence, we can't include the point **E**. Thus, we won't obtain the maximum length wiggle subsequence.

Thus, to solve this problem, we maintain a variable prevdiff, where prevdiff is used to indicate whether the current subsequence of numbers lies in an increasing or decreasing wiggle. If $\text{prevdiff} > 0$, it indicates that we have found the increasing wiggle and are looking for a decreasing wiggle now. Thus, we update the length of the found subsequence when diff $(nums[i] - nums[i-1])$ becomes negative. Similarly, if $\text{prevdiff} < 0$, we will update the count when diff $(nums[i] - nums[i-1])$ becomes positive.

When the complete array has been traversed, we get the required count, which represents the length of the longest wiggle subsequence.

Java Copy

```java
public class Solution {
    public int wiggleMaxLength(int[] nums) {
        if (nums.length < 2)
            return nums.length;
        int prevdiff = nums[1] - nums[0];
        int count = prevdiff != 0 ? 2 : 1;
        for (int i = 2; i < nums.length; i++) {
            int diff = nums[i] - nums[i - 1];
            if ((diff > 0 && prevdiff <= 0) || (diff < 0 && prevdiff >= 0)) {
                count++;
                prevdiff = diff;
            }
        }
        return count;
    }
}
```

Complexity Analysis

- Time complexity : $O(n)$. We traverse the given array once.
- Space complexity : $O(1)$. No extra space is used.

Rate this article:

Previous Next

markov_r ★219 December 3, 2018 5:32 PM

For the greedy approach, in brief:
When you have increasing or decreasing sub-sequence longer than 2 just ignore all middle elements and use the first and the last only (you don't gain anything from the middle ones). Also ignore duplicates.
So at the end you'll have a "clean" wiggle sequence with the required length.

Read More

19 Share Reply

Aeonaxx ★381 September 12, 2016 7:02 AM

Hi, LASkuma. I volunteer to give a proof of the correctness of approach #3. Sorry I didn't notice that I make it so long :)

#3 Approach

Read More

12 Share Reply

SHOW 4 REPLIES

sean67 ★2 August 31, 2018 11:50 AM

I have one question - In the example of approach #3, there is the following array [2,4,5,7,5,5,7,1,2,3].
I don't understand why down[6] could be 3?
The down array is defined as:
down[i] refers to the length of the longest wiggle subsequence obtained so far considering ith element as the last element of the wiggle subsequence and ending with a falling wiggle.

Read More

3 Share Reply

SHOW 5 REPLIES

LASkuma ★8 August 30, 2016 11:13 PM

It would be better if you can prove and elaborate more on approach #3. :)

2 Share Reply

beibeixhb ★8 September 3, 2018 1:08 AM

Why if the current element is a up position, the element before it must be a down position in Approach 3?

2 Share Reply

SHOW 2 REPLIES

lenchen1112 ★1006 January 23, 2020 3:54 PM

Clean Python 3 greedy approach.

```
import operator
class Solution:
    def wiggleMaxLength(self, nums: List[int]) -> int:
```

Read More

1 Share Reply

lenchen1112 ★1006 January 23, 2020 2:42 PM

Clean Python3 O(n)/O(n) DP

```
class Solution:
    def wiggleMaxLength(self, nums: List[int]) -> int:
        diff = [prev - curr for prev, curr in zip(nums, nums[1:]) if prev != curr
```

Read More

1 Share Reply

yangyangjuanjuan ★73 May 8, 2018 7:23 PM

I feel in algorithm 5, the text and the figure are speaking differently. That is confusion

1 Share Reply

SHOW 2 REPLIES

swoveland ★21 August 7, 2017 12:20 PM

Approach #5 in Ruby follows.

```
def longest_wiggle(arr)
  arr.each_cons(2).
    reject { |a,b| a==b }.
```

Read More

1 Share Reply

SHOW 1 REPLY

jasonzzzzz ★168 January 7, 2018 10:01 PM

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

Read More

0 Share Reply

‹ 1 2 3 ›