Problem: Sorted Squares

Level: Easy

Given a sorted array in non-decreasing order, return an array of squares of each number, also in non-decreasing order. For example:

[-4,-2,-1,0,3,5] -> [0,1,4,9,16,25]

How can you do it in O(n) time?

Questions to Clarify:

Q. Can there be both negative and positive numbers?
A. Yes

Q. Should the output be a new array or the existing array modified?
A. Either way is ok.

Solution:

Sorting Approach:
Doing this problem in *O(nlog(n))* time is pretty trivial - just square all the numbers and then sort them. In this case though, the interviewer is specifically looking for an O(n) time solution.

There is a pattern in the input array. The largest squares will be at either end of the array. The lowest -ve number and the highest +ve number will be the largest squares. So, if we look at either ends of the array, we can go inwards and find smaller squares.

[**-4**, -2, -1, 0, 3, **5**] -> 25, 16

Go inwards:

[-4, **-2** , -1, 0, **3**, 5] -> 9, 4

This will give us squares in descending order - from largest to smallest.
Keep in mind that we will need to store the output somewhere. We will need to allocate a separate array for that. Unfortunately, we cannot do this in-place (i.e, by rearranging the input array).

We allocate a new array and fill it from the back (since our squares are presented from largest to smallest).

So, the space complexity for this algorithm will be O(n). This is a downgrade from the sorting approach (which used O(1) space), but in this case, we were specifically looking to improve the time complexity.

Pseudocode:

```
(Note: Never write pseudocode in an actual interview. Unless you're writing a
few lines quickly to plan out your solution. Your actual solution should be in
a real language and use good syntax.)

start = 0, end = a.length - 1

int[] result = empty array

while (start <= end)
     // add larger one to the end of result
     if abs(a[start]) > abs(a[end]): // abs(): absolute value
         add square(a[start]) to the end of result
         increase start by 1
     else:
         add square(a[end]) to the end of result
         decrease end by 1

return result
```

<u>Test Cases:</u>
Edge Cases: Empty array, null array
Base Cases: single element (+ve/-ve)
Regular Case: only +ve elements, only -ve elements, both +ve/-ve

<u>Time Complexity: O(n)</u>

<u>Space Complexity: O(n)</u>

```java
public static int[] squares(int[] a) {
    if (a == null)
        return null;

    int start = 0, end = a.length - 1;

    int[] result = new int[a.length];
    int resultIndex = result.length - 1;

    while (start <= end) {
        if (abs(a[start]) > abs(a[end])) {
            result[resultIndex] = square(a[start]);
            start++;
        } else {
            result[resultIndex] = square(a[end]);
            end--;
        }
        resultIndex--;
    }
    return result;
}

/*
 * Helper functions. Ask the interviewer if they want you to implement
 * these.
 */

public static int abs(int number) {
    return number >= 0 ? number : -1 * number;
}

public static int square(int number) {
    return number * number;
}
```

Problem: Shortest Unsorted Subarray to Sort the Array

Level: Medium

Given an array of integers, find the shortest sub array, that if sorted, results in the entire array being sorted.

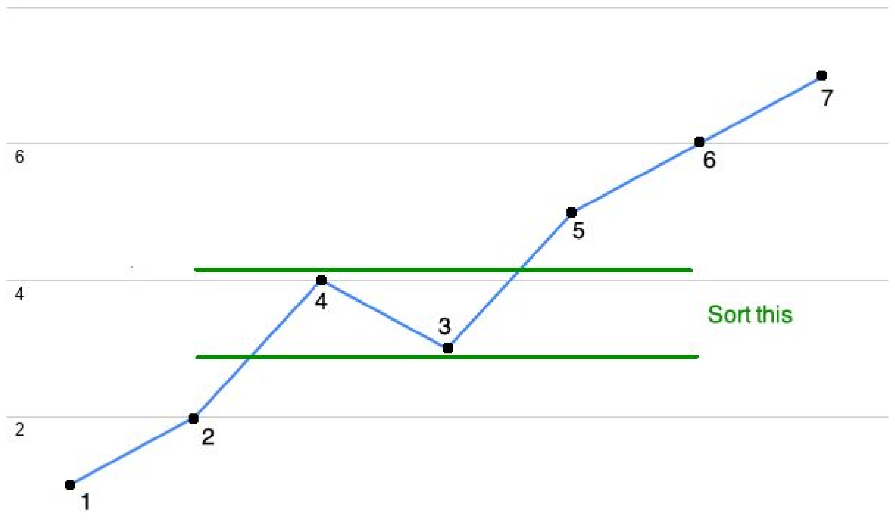
For example:

[1,2,4,5,3,5,6,7] --> [4,5,3] - If you sort from indices 2 to 4, the entire array is sorted.

[1,3,5,2,6,4,7,8,9] --> [3,5,2,6,4] - If you sort from indices 1 to 5, the entire array is sorted.

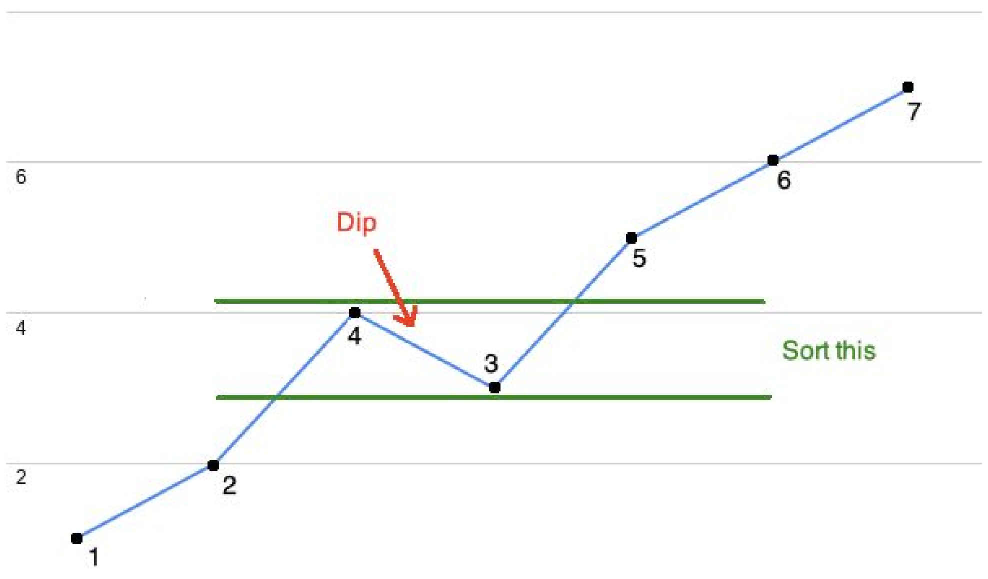
Questions to Clarify:
Q. How to return the result?
A. Return the result as a pair of indices.

Solution:
To develop intuition for this problem, think of your array as a line chart. For e.g,

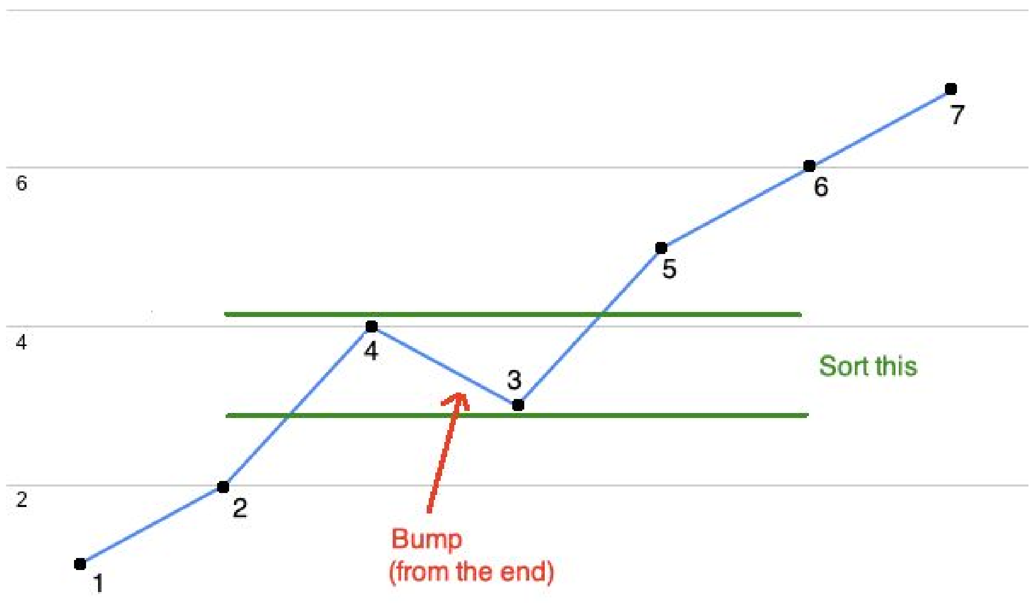


In the chart above, there's only two numbers out of order - 4 and 3. This is the smallest possible subarray that when sorted, leads to the entire array being sorted.

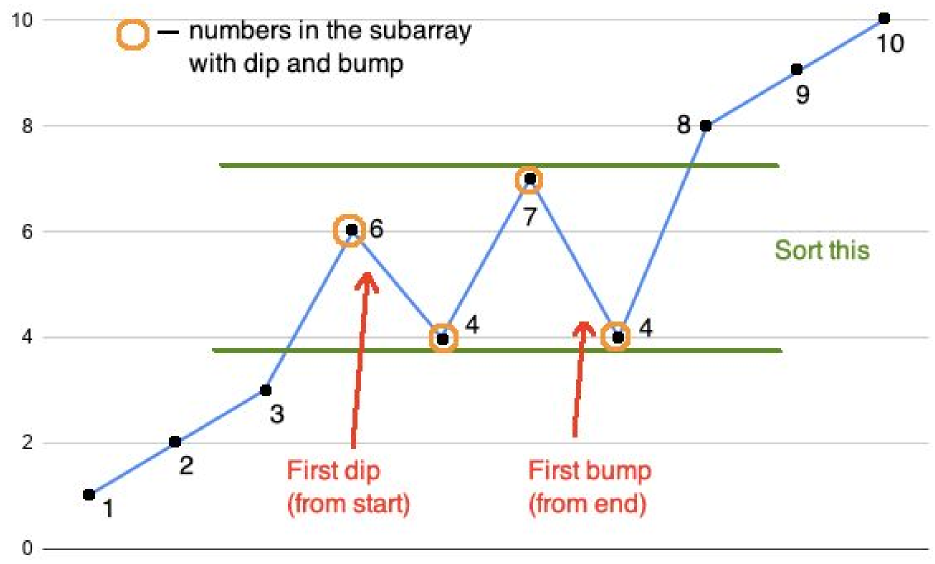
How do we spot such a subarray? A good starting point is to look for a dip in value. If you go through the sorted array one by one, the first dip indicates that the array is unsorted starting from this point.



Similarly, from the end, we can find a bump. When we go from end to start, if we find a bump, that indicates that the array is unsorted from there.

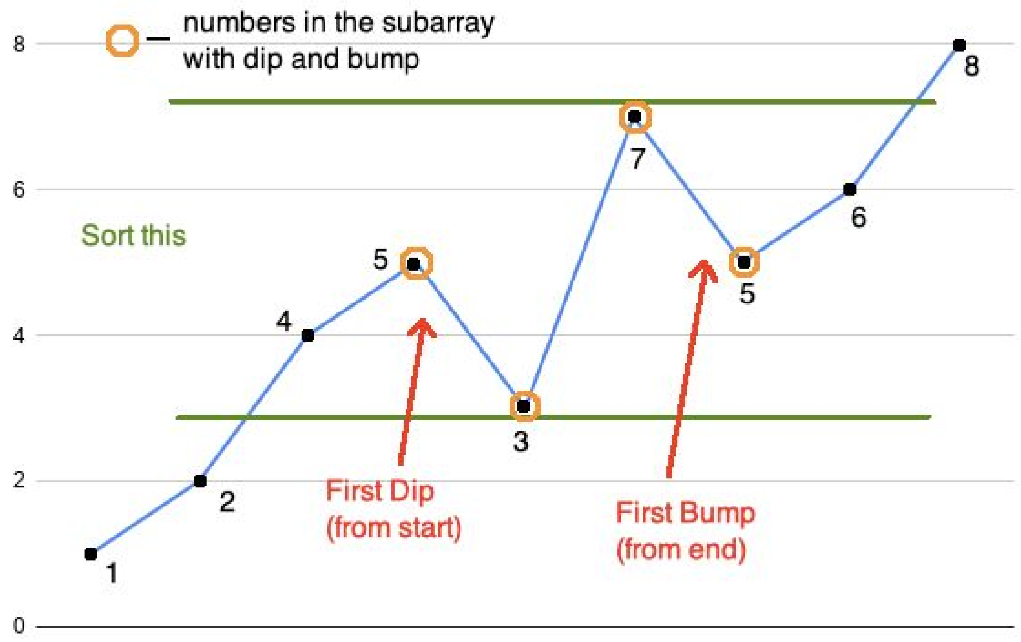


So, we can make an initial estimate of the unsorted portion, by traversing from both ends and finding a subarray between the first dip and bump. This subarray can be the answer for some inputs, for example:



Above, we found the dip and bump, and the subarray between them is good enough to sort the entire array. The result is the subarray [6,4,7,4].

However, it will not work in certain cases. For example, look at the following:



Just finding the dip and bump doesn't work above. Just by finding the dip and bump, our subarray is [5,3,7,5]. However, look at the numbers **4** and **6** - they need to be in the result as well.

Look at the green lines along 3 and 7. Everything between those lines should be in the result. Only then will the array be sorted.

So 4 should be included in our result - so that 3 can appear on it's left after sorting the subarray. Similarly, 7 is greater than 6, so 6 should be included - so that 7 can appear on it's right after sorting.

Now the question becomes: How do we add the leftover numbers that are between the green lines?

Here's how we fix this - we need to expand the subarray so that it covers all the numbers less than the minimum. The minimum of our subarray [5,3,7,5] is 3. So, we know that we should expand left if we encounter numbers greater than 3. In this case, 4.

Similarly, the maximum of our subarray [5,3,7,5] is 7. So, we should expand right while we encounter numbers less than 7. In this case, 6.

This will bring all the numbers within the green lines into our result.

Here is the overall algorithm that will do the job:

Start from 0. Find the first dip in value, call that *start*.
For example: [1,3,5,2,6,4,7,8,9] --> First dip starts at index 2.

Next, start from the end of the array. Find the first bump in value, call that *end*.
For example: [1,3,5,2,6,4,7,8,9] --> First bump starts at index 5.

We now know that the subarray [*start*..*end*] is causing the array to be unsorted. But simply sorting this subarray will not sort the array. After all, sorting [5,2,6,4] will not do the trick, because there is a 3 before it. We need to expand this subarray. Let's find the min of subarray [*start*..*end*].

In this case, the *min* is 2. In order for the entire array to be sorted, we will have to expand subarray [*start*..*end*] to include all numbers greater than 2. So we expand from {5,2,6,4} to {**3**,5,2,6,4}.

We do the same for *max*. In this case, we don't need to expand *max*, so that is the answer.

Pseudocode:

```
(Note: Never write pseudocode in an actual interview. Unless you're writing a
few lines quickly to plan out your solution. Your actual solution should be in
a real language and use good syntax.)

  start = first dip (when traversing from 0 to end)
  if no dip, array is already sorted, return

  end = first bump (when traversing from reverse order - end to 0)

  find max and min in subarray [start,end]
  expand start left, until you find an element less than min
  expand end right, until you find an element greater than max

  return start, end
```

<u>Test Cases:</u>
Edge Cases: empty array, null array
Base Case: one element, 2 elements (sorted and unsorted)
Regular Case: array already sorted, unsorted portion at beginning/end etc.

<u>Time Complexity: O(n)</u>
<u>Space Complexity: O(1)</u>

```
public static Pair<Integer> shortestUnsortedSubarray(int[] a) {
    if (a == null || a.length == 0)
        return null

    int start, end;

    // find dip
    for (start = 0; start < a.length - 1; start++) {
        if (a[start + 1] < a[start])
            break;
    }

    // no dip found, array is already sorted
    if (start == (a.length - 1))
        return null;

    // find bump
    for (end = a.length - 1; end > 0; end--) {
        if (a[end - 1] > a[end])
            break;
    }

    // find min and max of a[start..end]
    int max = Integer.MIN_VALUE, min = Integer.MAX_VALUE;
    for (int k = start; k <= end; k++) {
        if (a[k] > max)
            max = a[k];

        if (a[k] < min)
            min = a[k];
    }

    // expand start and end outward
    while (start > 0 && a[start - 1] > min) {
        start--;
    }
    while (end < (a.length - 1) && a[end + 1] < max) {
        end++;
    }

    return new Pair<Integer>(start, end);
}
```