<https://leetcode.com/problems/shortest-unsorted-continuous-subarray/>

Given an integer array nums, you need to find one **continuous subarray** that if you only sort this subarray in ascending order, then the whole array will be sorted in ascending order.

Return *the shortest such subarray and output its length*.

**Example 1:**

Input: nums = [2,6,4,8,10,9,15]

Output: 5

Explanation: You need to sort [6, 4, 8, 10, 9] in ascending order to make the whole array sorted in ascending order.

**Example 2:**

Input: nums = [1,2,3,4]

Output: 0

**Example 3:**

Input: nums = [1]

Output: 0

**Constraints:**

* 1 <= nums.length <= 104
* -105 <= nums[i] <= 105

**Follow up:** Can you solve it inO(n)time complexity?

**Attempt 1: 2023-04-02**

**Solution 1: Brute Force (30 min, TLE)**

class Solution {

public int findUnsortedSubarray(int[] nums) {

int len = nums.length;

int result = len;

for(int i = 0; i < len; i++) {

for(int j = i; j <= len; j++) {

int min = Integer.MAX\_VALUE;

int max = Integer.MIN\_VALUE;

for(int k = i; k < j; k++) {

min = Math.min(min, nums[k]);

max = Math.max(max, nums[k]);

}

// Further, the elements in nums[0:i−1] all need to be lesser

// than the min for satisfying the required condition.

// Similarly, all the elements in nums[j:n−1] need to be larger

// than max. We check for these conditions for every possible

// i and j selected.

if((i > 0 && nums[i - 1] > min) || (j < len && nums[j] < max)) {

continue;

}

// Further, we also need to check if nums[0:i−1] and nums[j:n−1]

// are sorted correctly.

int prev = Integer.MIN\_VALUE;

int k = 0;

while(k < i && prev <= nums[k]) {

prev = nums[k];

k++;

}

if(k != i) {

continue;

}

k = j;

while(k < len && prev <= nums[k]) {

prev = nums[k];

k++;

}

if(k == len) {

result = Math.min(result, j - i);

}

}

}

return result;

}

}

Time complexity : O(n^3). Three nested loops are there.

Space complexity : O(1). Constant space is used.

**Refer to**

<https://leetcode.com/problems/shortest-unsorted-continuous-subarray/editorial/>

#### **Approach 1: Brute Force**

**Algorithm**

In the brute force approach, we consider every possible subarray that can be formed from the given array nums. For every subarray nums[i:j]considered, we need to check whether this is the smallest unsorted subarray or not. Thus, for every such subarray considered, we find out the maximum and minimum values lying in that subarray given by max and min respectively.

If the subarrays nums[0:i−1] and nums[j:n−1] are correctly sorted, then only nums[i:j] could be the required subarray. Further, the elements in nums[0:i−1] all need to be lesser than the min for satisfying the required condition. Similarly, all the elements in nums[j:n−1] need to be larger than max. We check for these conditions for every possible i and j selected.

Further, we also need to check if nums[0:i−1] and nums[j:n−1] are sorted correctly. If all the above conditions are satisfied, we determine the length of the unsorted subarray as j−i. We do the same process for every subarray chosen and determine the length of the smallest unsorted subarray found.

public class Solution {

public int findUnsortedSubarray(int[] nums) {

int res = nums.length;

for (int i = 0; i < nums.length; i++) {

for (int j = i; j <= nums.length; j++) {

int min = Integer.MAX\_VALUE, max = Integer.MIN\_VALUE, prev = Integer.MIN\_VALUE;

for (int k = i; k < j; k++) {

min = Math.min(min, nums[k]);

max = Math.max(max, nums[k]);

}

if ((i > 0 && nums[i - 1] > min) || (j < nums.length && nums[j] < max))

continue;

int k = 0;

while (k < i && prev <= nums[k]) {

prev = nums[k];

k++;

}

if (k != i)

continue;

k = j;

while (k < nums.length && prev <= nums[k]) {

prev = nums[k];

k++;

}

if (k == nums.length) {

res = Math.min(res, j - i);

}

}

}

return res;

}

}

**Complexity Analysis**

* Time complexity : O(n^3). Three nested loops are there.
* Space complexity : O(1). Constant space is used.

**Solution 2: Two Pass (30 min)**

class Solution {

public int findUnsortedSubarray(int[] nums) {

if(nums == null) {

return 0;

}

int len = nums.length;

if(len == 0 || len == 1) {

return 0;

}

int max = Integer.MIN\_VALUE;

// Iterate from beginning of array

// find the last element which is smaller than the last seen max from

// its left side and mark it as end

int end = -2;

for(int i = 0; i < len; i++) {

max = Math.max(nums[i], max);

if(nums[i] < max) {

end = i;

}

}

int min = Integer.MAX\_VALUE;

// Iterate from end of array

// find the last element which is bigger than the last seen min from

// its right side and mark it as begin

int start = -1;

for(int i = len - 1; i>= 0; i--) {

min = Math.min(nums[i], min);

if(nums[i] > min) {

start = i;

}

}

return end - start + 1;

}

}

Time complexity : O(n).

Space complexity : O(1). Constant space is used.

**Refer to**

<https://leetcode.com/problems/shortest-unsorted-continuous-subarray/solutions/103057/java-o-n-time-o-1-space/comments/106306>

if(nums == null) return 0;

if(nums.Length == 0 || nums.Length == 1) return 0;

int max = Int32.MinValue;

int end = -2;

//iterate from beginning of array

//find the last element which is smaller than the last seen max from

//its left side and mark it as end

for(int i = 0; i < nums.Length; i ++){

max = Math.Max(max, nums[i]);

if(nums[i] < max)

end = i;

}

int min = Int32.MaxValue;

int begin = -1;

//iterate from end of array

//find the last element which is bigger than the last seen min from

//its right side and mark it as begin

for(int i = nums.Length - 1; i >= 0; i --){

min = Math.Min(min, nums[i]);

if(nums[i] > min)

begin = i;

}

return end - begin + 1;

**Some explanations:**

endIdx = The most right element having greater elements on the left side.

begIdx = The most left element having smaller elements on the right side.

**Prove it is effective:**

According to the definition, we can know that all elements on the right side of endIdx do not have greater elements on the left side and all elements on the left side of the begIdx do not have greater elements on the right side. Therefore, these two parts are good and we only need to sort the elements between begIdx and endIdx.

**Prove the bounds are tight:**

According to the definition, the two elements at begIdx and endIdx are "illegal", so the range to be sort should at least include these two elements.

**Solution 3: Sorting (30 min)**

class Solution {

public int findUnsortedSubarray(int[] nums) {

if(nums == null) {

return 0;

}

int len = nums.length;

if(len == 0 || len == 1) {

return 0;

}

int[] clone\_nums = nums.clone();

Arrays.sort(clone\_nums);

int start = len - 1;

int end = 0;

for(int i = 0; i < len; i++) {

if(clone\_nums[i] != nums[i]) {

start = Math.min(start, i);

end = Math.max(end, i);

}

}

// Test case: {1,2,3,4} -> in case already sorted,

// then 'start' and 'end' not changed, return 0

// return end - start + 1;

return end - start >= 0 ? end - start + 1 : 0;

}

}

Time complexity : O(nlog⁡n). Sorting takes nlog⁡n time.

Space complexity : O(n). We are making copy of original array.

**Refer to**

<https://leetcode.com/problems/shortest-unsorted-continuous-subarray/editorial/>

#### **Approach 3: Using Sorting**

**Algorithm**

Another very simple idea is as follows. We can sort a copy of the given array nums, say given by nums\_sorted. Then, if we compare the elements of nums and nums\_sorted, we can determine the leftmost and rightmost elements which mismatch. The subarray lying between them is, then, the required shorted unsorted subarray.

public class Solution {

public int findUnsortedSubarray(int[] nums) {

int[] snums = nums.clone();

Arrays.sort(snums);

int start = snums.length, end = 0;

for (int i = 0; i < snums.length; i++) {

if (snums[i] != nums[i]) {

start = Math.min(start, i);

end = Math.max(end, i);

}

}

return (end - start >= 0 ? end - start + 1 : 0);

}

}

**Complexity Analysis**

* Time complexity : O(nlog⁡n). Sorting takes nlog⁡n time.
* Space complexity : O(n). We are making copy of original array.

**Solution 4: Monotonic Stack (30 min)**

class Solution {

public int findUnsortedSubarray(int[] nums) {

if(nums == null) {

return 0;

}

int len = nums.length;

if(len == 0 || len == 1) {

return 0;

}

Stack<Integer> stack = new Stack<Integer>();

int start = len;

for(int i = 0; i < len; i++) {

while(!stack.isEmpty() && nums[stack.peek()] > nums[i]) {

start = Math.min(stack.pop(), start);

}

stack.push(i);

}

stack.clear();

int end = 0;

for(int i = len - 1; i >= 0; i--) {

while(!stack.isEmpty() && nums[stack.peek()] < nums[i]) {

end = Math.max(stack.pop(), end);

}

stack.push(i);

}

return end - start > 0 ? end - start + 1 : 0;

}

}

Time complexity : O(n). Stack of size n is filled.

Space complexity : O(n). Stack size grows up to n.

**Refer to**

<https://leetcode.com/problems/shortest-unsorted-continuous-subarray/editorial/>

#### **Approach 4: Using Stack**

**Algorithm**

The idea behind this approach is also based on selective sorting. We need to determine the correct position of the minimum and the maximum element in the unsorted subarray to determine the boundaries of the required unsorted subarray.

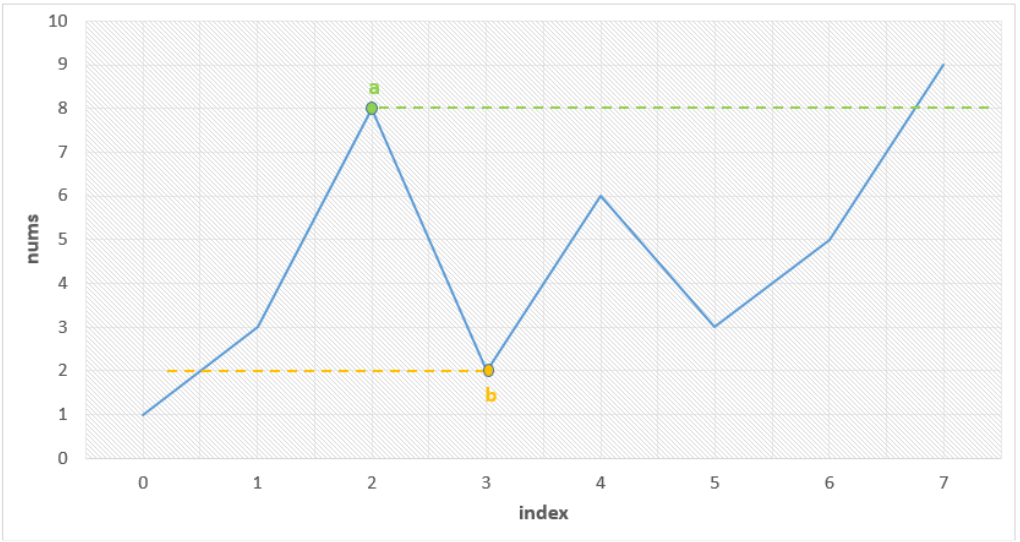
To do so, in this implementation, we make use of a stack. We traverse over the nums array starting from the beginning. As we go on facing elements in ascending order(a rising slope), we keep on pushing the elements' indices over the stack. This is done because such elements are in the correct sorted order(as it seems till now). As soon as we encounter a falling slope, i.e. an element nums[j] which is smaller than the element on the top of the stack, we know that nums[j] isn't at its correct position.

In order to determine the correct position of nums[j], we keep on popping the elements from the top of the stack until we reach the stage where the element(corresponding to the index) on the top of the stack is lesser than nums[j]. Let's say the popping stops when the index on stack's top is k. Now, nums[j] has found its correct position. It needs to lie at an index k+1k + 1*k*+1.

We follow the same process while traversing over the whole array, and determine the value of minimum such k. This marks the left boundary of the unsorted subarray.

Similarly, to find the right boundary of the unsorted subarray, we traverse over the nums array backwards. This time we keep on pushing the elements if we see a falling slope. As soon as we find a rising slope, we trace forwards now and determine the larger element's correct position. We do so for the complete array and thus, determine the right boundary.

We can look at the figure below for reference. We can observe that the slopes directly indicate the relative ordering. We can also observe that the point b needs to lie just after index 0 marking the left boundary and the point a needs to lie just before index 7 marking the right boundary of the unsorted subarray.



public class Solution {

public int findUnsortedSubarray(int[] nums) {

Stack < Integer > stack = new Stack < Integer > ();

int l = nums.length, r = 0;

for (int i = 0; i < nums.length; i++) {

while (!stack.isEmpty() && nums[stack.peek()] > nums[i])

l = Math.min(l, stack.pop());

stack.push(i);

}

stack.clear();

for (int i = nums.length - 1; i >= 0; i--) {

while (!stack.isEmpty() && nums[stack.peek()] < nums[i])

r = Math.max(r, stack.pop());

stack.push(i);

}

return r - l > 0 ? r - l + 1 : 0;

}

}

**Complexity Analysis**

* Time complexity : O(n). Stack of size n is filled.
* Space complexity : O(n). Stack size grows up to n.