**Solution: Problem Challenge 3: Minimum Window Sort**

**Problem Statement**

Given an array, find the length of the smallest subarray in it which when sorted will sort the whole array.

**Example 1**:

Input: [1, 2, 5, 3, 7, 10, 9, 12]

Output: 5

Explanation: We need to sort only the subarray [5, 3, 7, 10, 9] to make the whole array sorted

**Example 2**:

Input: [1, 3, 2, 0, -1, 7, 10]

Output: 5

Explanation: We need to sort only the subarray [1, 3, 2, 0, -1] to make the whole array sorted

**Example 3**:

Input: [1, 2, 3]

Output: 0

Explanation: The array is already sorted

**Example 4**:

Input: [3, 2, 1]

Output: 3

Explanation: The whole array needs to be sorted.

**Constraints:**

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

**Solution**

As we know, once an array is sorted (in ascending order), the smallest number is at the beginning and the largest number is at the end of the array. So if we start from the beginning of the array to find the first element which is out of sorting order i.e., which is smaller than its previous element, and similarly from the end of array to find the first element which is bigger than its previous element, will sorting the subarray between these two numbers result in the whole array being sorted?

Let’s try to understand this with Example-2 mentioned above. In the following array, what are the first numbers out of sorting order from the beginning and the end of the array:

[1, 3, 2, 0, -1, 7, 10]

Starting from the beginning of the array the first number out of the sorting order is ‘2’ as it is smaller than its previous element which is ‘3’. Starting from the end of the array the first number out of the sorting order is ‘0’ as it is bigger than its previous element which is ‘-1’ As you can see, sorting the numbers between ‘3’ and ‘-1’ will not sort the whole array. To see this, the following will be our original array after the sorted subarray:

[1, -1, 0, 2, 3, 7, 10]

The problem here is that the smallest number of our subarray is ‘-1’ which dictates that we need to include more numbers from the beginning of the array to make the whole array sorted. We will have a similar problem if the maximum of the subarray is bigger than some elements at the end of the array. To sort the whole array we need to include all such elements that are smaller than the biggest element of the subarray.

**Step-by-step Algorithm**

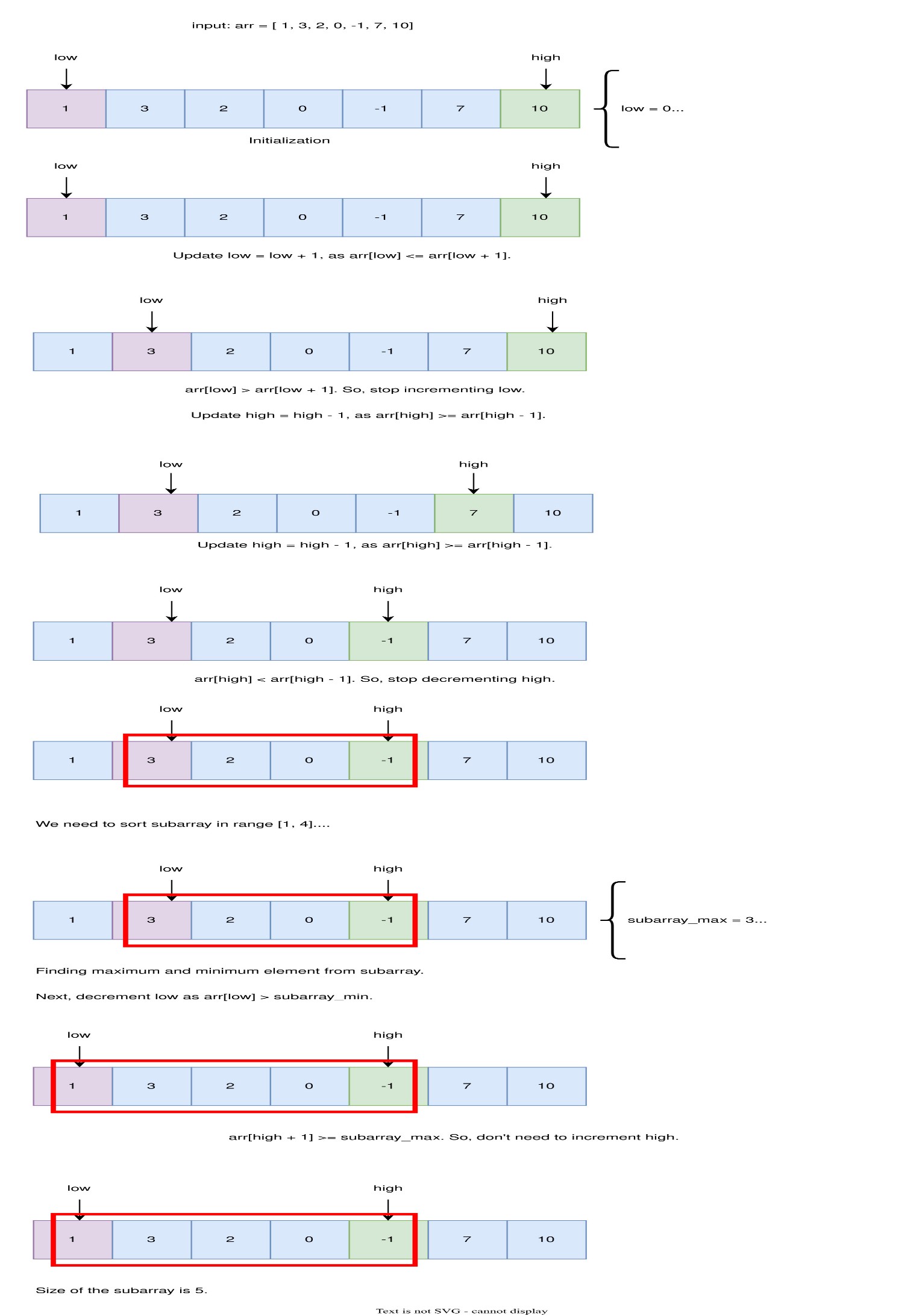
1. **Initialize Pointers**: Set low to 0 and high to the last index of the array.
2. **Find Left Boundary**:
   * Move low to the right while the current element is less than or equal to the next element.
3. **Check If Sorted**:
   * If low reaches the end, the array is already sorted. Return 0.
4. **Find Right Boundary**:
   * Move high to the left while the current element is greater than or equal to the previous element.
5. **Find Min and Max**:
   * Iterate from low to high to find the minimum and maximum values in this subarray.
6. **Extend Left Boundary**:
   * Move low to the left while the previous element is greater than the subarray's minimum.
7. **Extend Right Boundary**:
   * Move high to the right while the next element is less than the subarray's maximum.
8. **Calculate Length**:
   * The length of the subarray to be sorted is high - low + 1.

**Algorithm Walkthrough**

Using the input [1, 3, 2, 0, -1, 7, 10]:

* **Initialize Pointers**: low = 0, high = 6.
* **Find Left Boundary**:
  + Compare 1 and 3, move low to 1.
  + Compare 3 and 2, stop. low = 1.
* **Find Right Boundary**:
  + Compare 10 and 7, move high to 5.
  + Compare 7 and -1, move high to 4.
  + Compare -1 and 0, stop at 4.
* **Find Min and Max**:
  + Subarray is [3, 2, 0, -1].
  + Minimum is -1, Maximum is 3.
* **Extend Left Boundary**:
  + 1 is greater than -1, low stays 1.
* **Extend Right Boundary**:
  + 7 is not less than 3, high stays 4.
* **Calculate Length**:
  + Length is high - low + 1 = 4 - 0 + 1 = 5.

Here is the visual representation of this algorithm for Example 1:



**import math**

**class Solution:**

**def sort(self, arr):**

**low, high = 0, len(arr) - 1**

**# find the first number out of sorting order from the beginning**

**while (low < len(arr) - 1 and arr[low] <= arr[low + 1]):**

**low += 1**

**if low == len(arr) - 1: # if the array is sorted**

**return 0**

**# find the first number out of sorting order from the end**

**while (high > 0 and arr[high] >= arr[high - 1]):**

**high -= 1**

**# find the maximum and minimum of the subarray**

**subarray\_max = -math.inf**

**subarray\_min = math.inf**

**for k in range(low, high+1):**

**subarray\_max = max(subarray\_max, arr[k])**

**subarray\_min = min(subarray\_min, arr[k])**

**# extend the subarray to include any number which is bigger than the minimum of**

**# the subarray**

**while (low > 0 and arr[low-1] > subarray\_min):**

**low -= 1**

**# extend the subarray to include any number which is smaller than the maximum of**

**# the subarray**

**while (high < len(arr)-1 and arr[high+1] < subarray\_max):**

**high += 1**

**return high - low + 1**

**def main():**

**sol = Solution()**

**print(sol.sort([1, 2, 5, 3, 7, 10, 9, 12]))**

**print(sol.sort([1, 3, 2, 0, -1, 7, 10]))**

**print(sol.sort([1, 2, 3]))**

**print(sol.sort([3, 2, 1]))**

**main()**

**Complexity Analysis**

**Time Complexity**

* **First and second while loops (finding**low**and**high**)**: The first two while loops each scan a portion of the array once to identify the first and last indices (low and high) where the array is out of order. Both loops run in  time, where N is the length of the array.
* **Subarray max and min calculation**: The for loop that finds the maximum and minimum values within the subarray between low and high runs at most  because it scans the array between low and high. In the worst case, this could be the entire array.
* **Third and fourth while loops (extending the subarray)**: These loops extend the subarray by comparing the elements outside the subarray with the subarrayMin and subarrayMax. Both of these loops run in  time because they only scan the array once.

**Overall time complexity**: Each operation is , so the overall time complexity of the algorithm is .

**Space Complexity**

* **Constant space**: The algorithm uses only a few variables (low, high, subarrayMax, subarrayMin, etc.), all of which require constant space, .
* **In-place modification**: The algorithm does not use any additional data structures that scale with the input size, and it processes the array in place.

**Overall space complexity**: , since only a constant amount of extra space is used.