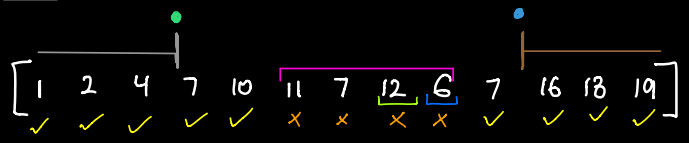
Input: An array of at least two integers

array = [1 2 4 7 10 11 7 12 6 7 16 18 19]
0 1 2 3 4 5 6 7 8 9 10 11 12

Output: An array of the starting and ending indices of the smallest sub array in the input array to be sorted in place
[3,9]

```
// O(n) time | O(1) space
function subarraySort(array) {
  let smallestNum = Infinity;
  let largestNum = -Infinity;

  for (let i = 0; i < array.length; i++) {
    const currNum = array[i];
    if (isOutOfOrder(i, currNum, array)) {
      smallestNum = Math.min(currNum, smallestNum);
      largestNum = Math.max(currNum, largestNum);
    }
  }

  if (smallestNum === Infinity) {
    return [-1, -1];
  }

  let numStartIdx = 0;
  while (smallestNum >= array[numStartIdx]) numStartIdx++;

  let numEndIdx = array.length - 1;
  while (largestNum <= array[numEndIdx]) numEndIdx--;

  return [numStartIdx, numEndIdx];
}

function isOutOfOrder(i, num, array) {
  if (i === 0) return num > array[i + 1];
  if (i === array.length - 1) return num < array[i - 1];
  return num > array[i + 1] || num < array[i - 1];
}
```

Idea:



Sorted values (value to right is greater; value to left smaller)
Not sorted (10 < 11 but 11 ≯ 7)
Find smallest and largest number in our unsorted sub array and get their positions
Loop from right to find spot to place smallest number (b/w 4 and 7)
Loop from left to find spot to place largest number (b/w 7 and 16)
return location of smallestIdx and largestIdx (ans: [3,9])

Time: O(n) (where n is the # of elements in the input array) since we iterate through the array n + n + n (for, while, while loop) at worst. This is 3n or O(n). All other operations are constant time.

Space: O(1) since no additional space is used as input size grows