educative

Solution Review: Problem Challenge 3

We'll cover the following

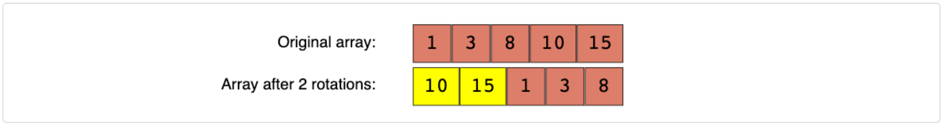
Rotation Count (medium)

Given an array of numbers which is sorted in ascending order and is rotated 'k' times around a pivot, find 'k'.

You can assume that the array does not have any duplicates.

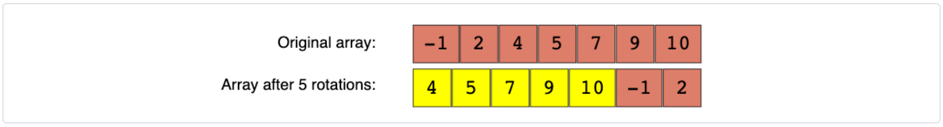
Example 1:

```
Input: [10, 15, 1, 3, 8]
Output: 2
Explanation: The array has been rotated 2 times.
```



Example 2:

```
Input: [4, 5, 7, 9, 10, -1, 2]
Output: 5
Explanation: The array has been rotated 5 times.
```



Example 3:

```
Input: [1, 3, 8, 10]
Output: 0
Explanation: The array has not been rotated.
```

Solution

This problem follows the **Binary Search** pattern. We can use a similar strategy as discussed in Search in Rotated Array.

In this problem, actually, we are asked to find the index of the minimum element. The number of times the minimum element is moved to the right will be equal to the number of rotations. An interesting fact about the minimum element is that it is the only element in the given array which is smaller than its previous element. Since the array is sorted in ascending order, all other elements are bigger than their previous element.

After calculating the `middle`, we can compare the number at index `middle` with its previous and next number. This will give us two options:

1. If `arr[middle] > arr[middle + 1]`, then the element at `middle + 1` is the smallest.
2. If `arr[middle - 1] > arr[middle]`, then the element at `middle` is the smallest.

To adjust the ranges we can follow the same approach as discussed in Search in Rotated Array. Comparing the

To adjust the ranges we can follow the same approach as discussed in Search in Rotated Array. Comparing the numbers at indices `start` and `middle` will give us two options:

1. If `arr[start] < arr[middle]`, the numbers from `start` to `middle` are sorted.
2. Else, the numbers from `middle + 1` to `end` are sorted.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```java
class RotationCountOfRotatedArray {

  public static int countRotations(int[] arr) {
    int start = 0, end = arr.length - 1;
    while (start < end) {
      int mid = start + (end - start) / 2;

      if (mid < end && arr[mid] > arr[mid + 1]) // if mid is greater than the next element
        return mid + 1;
      if (mid > start && arr[mid - 1] > arr[mid]) // if mid is smaller than the previous element
        return mid;

      if (arr[start] < arr[mid]) { // left side is sorted, so the pivot is on right side
        start = mid + 1;
      } else { // right side is sorted, so the pivot is on the left side
        end = mid - 1;
      }
    }

    return 0; // the array has not been rotated
  }

  public static void main(String[] args) {
    System.out.println(RotationCountOfRotatedArray.countRotations(new int[] { 10, 15, 1, 3, 8 }));
    System.out.println(RotationCountOfRotatedArray.countRotations(new int[] { 4, 5, 7, 9, 10, -1, 2 }));
    System.out.println(RotationCountOfRotatedArray.countRotations(new int[] { 1, 3, 8, 10 }));
  }
}
```

Time complexity

Since we are reducing the search range by half at every step, this means that the time complexity of our algorithm will be $O(logN)$ where 'N' is the total elements in the given array.

Space complexity

The algorithm runs in constant space $O(1)$.

Similar Problems

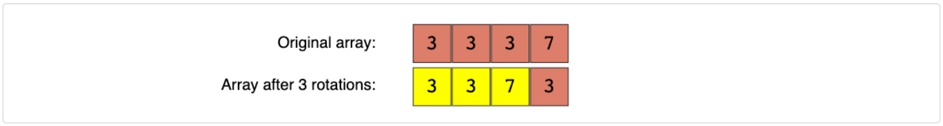
Problem 1

How do we find the rotation count of a sorted and rotated array that has duplicates too?

The above code will fail on the following example!

Example 1:

```
Input: [3, 3, 7, 3]
Output: 3
Explanation: The array has been rotated 3 times
```

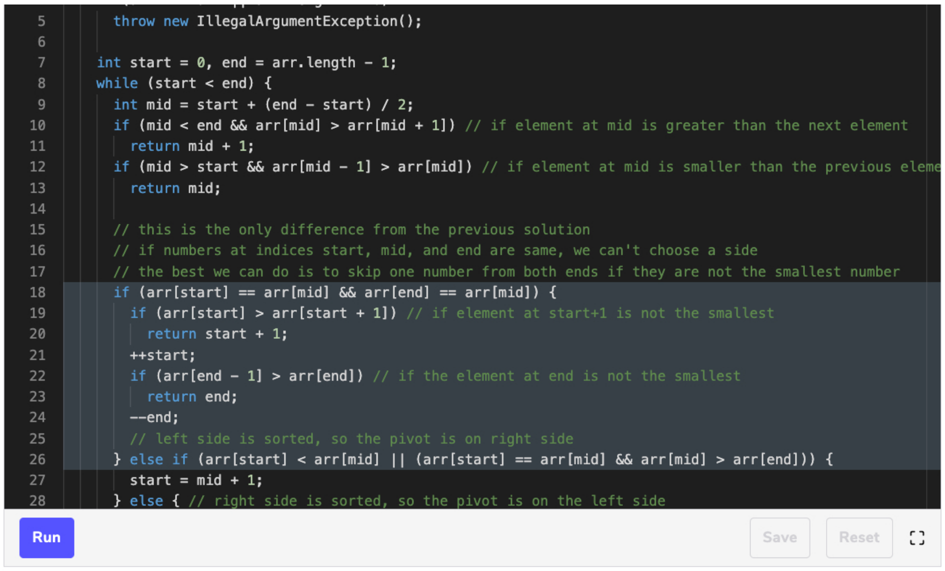


Solution
We can follow the same approach as discussed in Search in Rotated Array. The only difference is that before incrementing `start` or decrementing `end`, we will check if either of them is the smallest number.

Java | Python3 | C++ | JS

```java
class RotationCountWithDuplicates {

  public static int countRotations(int[] arr) {
    if (arr == null || arr.length == 0)
```



```
      throw new IllegalArgumentException();

    int start = 0, end = arr.length - 1;
    while (start < end) {
      int mid = start + (end - start) / 2;
      if (mid < end && arr[mid] > arr[mid + 1]) // if element at mid is greater than the next element
        return mid + 1;
      if (mid > start && arr[mid - 1] > arr[mid]) // if element at mid is smaller than the previous eleme
        return mid;

      // this is the only difference from the previous solution
      // if numbers at indices start, mid, and end are same, we can't choose a side
      // the best we can do is to skip one number from both ends if they are not the smallest number
      if (arr[start] == arr[mid] && arr[end] == arr[mid]) {
        if (arr[start] > arr[start + 1]) // if element at start+1 is not the smallest
          return start + 1;
        ++start;
        if (arr[end - 1] > arr[end]) // if the element at end is not the smallest
          return end;
        --end;
        // left side is sorted, so the pivot is on right side
      } else if (arr[start] < arr[mid] || (arr[start] == arr[mid] && arr[mid] > arr[end])) {
        start = mid + 1;
      } else { // right side is sorted, so the pivot is on the left side
```

Time complexity

This algorithm will run in $O(logN)$ most of the times, but since we only skip two numbers in case of duplicates instead of the half of the numbers, therefore the worst case time complexity will become $O(N)$.

Space complexity

The algorithm runs in constant space $O(1)$.

Back
Problem Challenge 3

Next
Introduction

Mark as Completed

Report an Issue Ask a Question