Technique: Cyclically Sorted Arrays

Level: Easy

Given an array that is cyclically sorted, find the minimum element. A cyclically sorted array is a sorted array rotated by some number of elements. Assume all elements are unique.

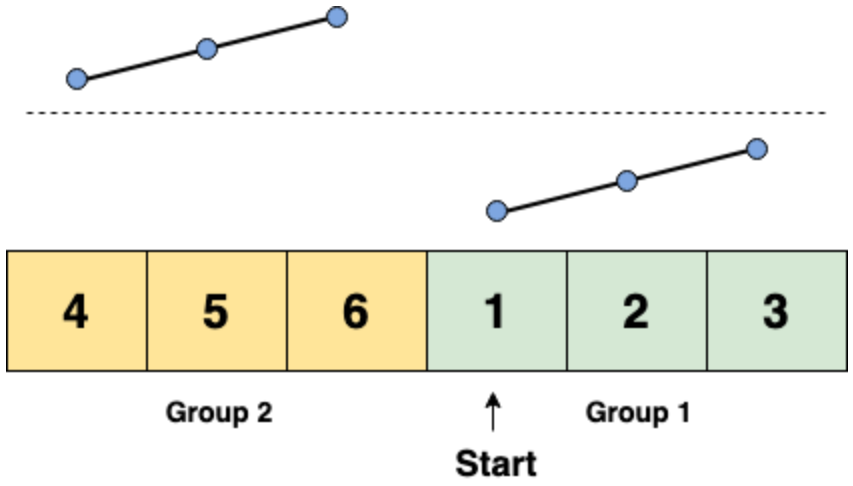
For example:
A = [4,5,1,2,3], which is just [1,2,3,4,5] rotated by 2
Result = index 2

Questions to Clarify:
Q. How do you want the input
A. Return the index of the min item.

Q.Can the array be rotated by 0 elements? i.e, a normal sorted array?
A. Yes

Solution:
Let's say A = [4,5,6,1,2,3]
Look at the last element in the array (3).

1. All elements that were rotated and came back around are > 3. We call these **Group 2** in the diagram
2. All elements that moved right but didn't go back around are <= 3. We call these **Group 1**.



To find the minimum, we are looking for the first element in Group 1. How can we utilize binary search for this?

Remember, our target is Start. If we can look at any item in the array, and check which group it's in, we know which direction to go. If it is in Group 2, we need to go right. If it is in Group 1, we go left. If we

reach Start, we've found our answer.

How do we identify this? By looking at the last element. Our last element is 3.

```
For any a[i] we encounter:

if a[i] <= 3, it is in Group 1. We go left.
If a[i] > 3, it is in Group 2. We go right.
```

Now, how do we tell if we've reached Start? Start has a unique property that no other element has. Notice that for the minimum element, the element to its left is greater. This is the only element with this property - because it's the turning point of rotation. If we find such an element, we've found the result.

But what if the array is not rotated - a possible input. In that case, Start will have nothing on it's left. It will be at index 0. We need to check for this too.

At this point, we have our 3 essential conditions for Binary Search. We know when to go left, when to go right, and when to return the result. We can now write code for this.

> Duplicates: Keep in mind that this algorithm will not work if duplicates are allowed. With duplicates, you can have a situation like this: A = [1,1,1,0,1,1]. It is difficult to divide the input into halves. So you can not do it in O(log(n)) time. With duplicates, you will have to go through the entire array and that will take O(n) time in the worst case.

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.)


right = a[a.length - 1] // last element
while start <= end
     find mid
     if mid <= right and left is greater or no left
         return mid
     if mid > right
         go right, start = mid + 1
     else
         go left, end = mid - 1
```

Test Cases:
Edge Cases: empty array, null array
Base Cases: one element, two elements (rotated by 0 & 1)
Regular Cases: rotated, not rotated

Time Complexity: O(log(n))

Space Complexity: O(1)

```
public static int cyclicallySortedMin(int[] a) {
    if (a == null) {
        return -1;
    }

    int low = 0, high = a.length - 1, right = a[a.length - 1];
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (a[mid] <= right && (mid == 0 || a[mid - 1] > a[mid])) {
            return mid;
        } else if (a[mid] > right) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    return -1;
}
```

Level: Medium

Search Array of Unknown length
You are given an array, but you don't know the length. Write a program to find a target element in the array.

Questions to Clarify:
Q. What happens if we try to access an index that is out of bounds?
A. An exception is thrown

Q. How do you want the output?
A. Return the index of target.

Q. What to return if target is not found?
A. Return -1

Solution:
This problem is different because we actively catch exceptions and use them in the algorithm. This is not usually the case.

The most obvious solution is to keep going through the array until we find the target or an exception is thrown. That takes O(n) time. We can use Binary Search to shorten it to O(log(n)).

The first step is to find the end of the array in O(log(n)) time.

While looking for the end of the array, instead of querying indices `0,1,2,3,4..`, we can query indices `1,2,4,8,16,32,`... Since we double this every time, we will encounter an exception in log(n) steps. Let's say the array length is 30. We encounter an exception at a[32]. Now, we know that the end is between index 16 and index 32. So we binary search between those to find the end.

Now that we know the length, we can do a regular binary search to find the target.

Both steps are O(log(n)), so the algorithm takes O(log(n)) time.

Optional Optimization: While not better than O(log(n)), you can make it a bit more efficient. In the first step, let's say you are querying indices 1,2,4,8, etc. At index 8, you find that a[8] is greater than target. This means that the target is between indices 4 and 8, and you can simply search there. This avoids searching the rest of the array again. The time complexity will still be O(log(n)), so both solutions are acceptable.

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.)

high = 1;
```

```
lastIndex = -1
while true:
     if a[high] throws exception
         binary search over (high/2+1, high) for lastIndex of the array
         break loop
     else
         high = high * 2

binary search from 0 to lastIndex for target element
```

Test Cases:
Edge Cases: empty array, null array
Base Cases: single element, two elements
Regular Cases: power of 2 +- 1 / not a power of 2

Time Complexity: O(log(n))

Space Complexity: O(1)

```
public static int findWithUnknownLength(int[] a, int target) {
        if (a == null || a.length == 0) {
            return -1;
        }

        int high = 1; // 1,2,4,8,16,32..
        int lastIndex = -1;

        // Consider putting a sanity limit here, for e.g,don't go more
        // than index 1 million. Discuss this with the interviewer.
        while (true) {
            try {
                int temp = a[high];
            } catch (ArrayIndexOutOfBoundsException e) {
                lastIndex = binarySearchForLastIndex(a, high/2, high);
                break;
            }
            high *= 2;
        }

        return binarySearchOverRange(a, target, 0, lastIndex);
    }

    private static int binarySearchForLastIndex(int[] a, int low, int high) {
        while (low <= high) {
            int mid = low + (high - low)/2;
            try {
                int temp = a[mid];
            } catch (ArrayIndexOutOfBoundsException e) {
```

```
                // mid is out of bounds, go to lower half
                high = mid - 1;
                continue;
            }

            try {
                int temp = a[mid+1];
            } catch (ArrayIndexOutOfBoundsException e) {
                // mid + 1 is out of bounds, mid is last index
                return mid;
            }

            // both mid and mid + 1 are inside array, mid is not last index.
            low = mid + 1;
        }

        return -1; // this subarray does not have end of the array
    }

    private static int binarySearchOverRange(int[] a, int target, int low, int high)
    {
        while (low <= high) {
            int mid = low + (high - low)/2;
            if (a[mid] == target) {
                return mid;
            } else if (a[mid] < target) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        return -1;
    }
```