Minimum in Rotated Sorted Array

Problem Statement: Given an integer array **arr** of size **N**, sorted in ascending order (**with distinct values**). Now the array is rotated between 1 to N times which is unknown. Find the minimum element in the array.

Pre-requisites: Search in Rotated Sorted Array I, Search in Rotated Sorted Array II & Binary Search algorithm

▼ **Examples**

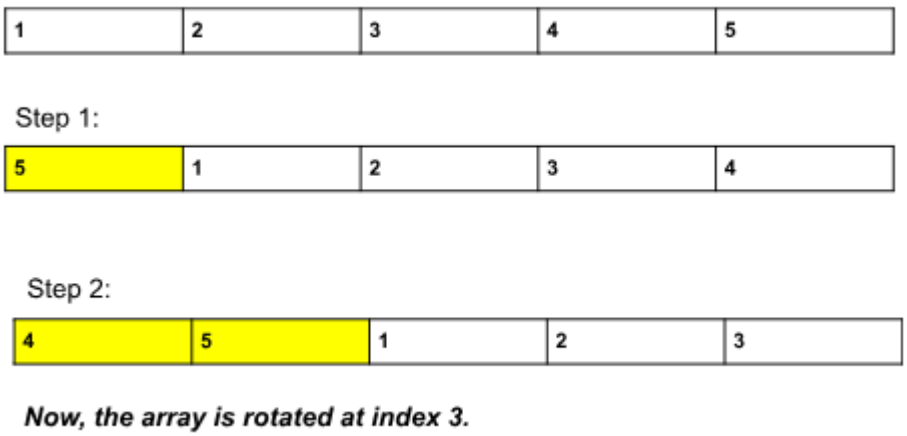
```
Example 1:
Input Format: arr = [4,5,6,7,0,1,2,3]
Result: 0
Explanation: Here, the element 0 is the minimum element in the array.

Example 2:
Input Format: arr = [3,4,5,1,2]
Result: 1
Explanation: Here, the element 1 is the minimum element in the array.
```

Solution:

How does the rotation occur in a sorted array?

Let's consider a sorted array: {1, 2, 3, 4, 5}. If we rotate this array at index 3, it will become: {4, 5, 1, 2, 3}. In essence, we moved the element at the last index to the front, while shifting the remaining elements to the right. We performed this process twice.



Now, the array is rotated at index 3.

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

▼ Solution 1:

▼ Algorithm / Intuition ›

Naive Approach (Brute force):

One straightforward approach, we can consider is using the linear search algorithm. Using this method, we will find the minimum number from the array.

Algorithm:

- First, we will declare a ‘mini’ variable initialized with a large number.
- After that, we will traverse the array and compare each element with the ‘mini’ variable. Each time the ‘mini’ variable will be updated with the minimum value i.e. min(mini, arr[i]).
- Finally, we will return ‘mini’ as our answer.

▼ Code ›

```
import sys
def findMin(arr: [int]):
    n = len(arr)  # size of the array.
    mini = sys.maxsize
    for i in range(n):
        # Always keep the minimum.
        mini = min(mini, arr[i])
    return mini

if __name__ == "__main__":
    arr = [4, 5, 6, 7, 0, 1, 2, 3]
    ans = findMin(arr)
    print("The minimum element is:", ans)
```

Output: The minimum element is: 0

▼ Complexity Analysis ›

Time Complexity: O(N), N = size of the given array.**Reason:** We have to iterate through the entire array to check if the target is present in the array.

Space Complexity: O(1)**Reason:** We have not used any extra data structures, this makes space complexity, even in the worst case as O(1).

▼ Solution 2:

▼ Algorithm / Intuition ›

Optimal Approach(Using Binary Search):

Here, we can easily observe, that we have to find the minimum in a sorted array. That is why, we can think of using the Binary Search algorithm to solve this problem.

The primary objective of the Binary Search algorithm is to efficiently determine the appropriate half to eliminate, thereby reducing the search space by half. It does this by determining a specific condition that ensures that the target is not present in that half.

Key Observation: If an array is rotated and sorted, we already know that for every index, one of the 2 halves of the array will always be sorted.

Based on this observation, we adopted a straightforward two-step process to eliminate one-half of the rotated sorted array.

- First, we identify the sorted half of the array.
- Once found, we determine if the target is located within this sorted half.
    - If not, we eliminate that half from further consideration.
    - Conversely, if the target does exist in the sorted half, we eliminate the other half.

Let's observe how we identify the sorted half:

We basically compare arr[mid] with arr[low] and arr[high] in the following way:

- **If arr[low] <= arr[mid]:** In this case, we identified that the left half is sorted.
- **If arr[mid] <= arr[high]:** In this case, we identified that the right half is sorted.

Let's observe how we will find the minimum element:

In this situation, we have two possibilities to consider. The sorted half of the array may or may not include the minimum value. However, we can leverage the property of the sorted half, specifically that the leftmost element of the sorted half will always be the minimum element within that particular half.

During each iteration, we will select the leftmost element from the sorted half and discard that half from further consideration. Among all the selected elements, the minimum value will serve as our answer.

To facilitate this process, we will declare a variable called 'ans' and initialize it with a large number. Then, at each step, after selecting the leftmost element from the sorted half, we will compare it with 'ans' and update 'ans' with the smaller value (i.e., min(ans, leftmost_element)).

Note:*If, at any index, both the left and right halves of the array are sorted, we have the flexibility to select the minimum value from either half and eliminate that particular half (in this case, the left half is chosen first). The algorithm already takes care of this case, so there is no need for explicit handling.*

Algorithm:

The steps are as follows:

We will declare the 'ans' variable and initialize it with the largest value possible. With that, as usual, we will declare 2 pointers i.e. low and high.

1. **Place the 2 pointers i.e. low and high:** Initially, we will place the pointers like this: low will point to the first index and high will point to the last index.

2. **Calculate the ‘mid’:** Now, inside a loop, we will calculate the value of ‘mid’ using the following formula:
   mid = (low+high) // 2 (‘//’ refers to integer division)
3. Identify the sorted half, and after picking the leftmost element, eliminate that half.
    1. **If arr[low] <= arr[mid]:** This condition ensures that the left part is sorted. So, we will pick the leftmost element i.e. **arr[low]**. Now, we will compare it with 'ans' and update 'ans' with the smaller value (i.e., **min(ans, arr[low])**). Now, we will eliminate this left half(i.e. low = mid+1).
    2. **Otherwise, if the right half is sorted:** This condition ensures that the right half is sorted. So, we will pick the leftmost element i.e. **arr[mid]**. Now, we will compare it with 'ans' and update 'ans' with the smaller value (i.e., **min(ans, arr[mid])**). Now, we will eliminate this right half(i.e. high = mid-1).
4. This process will be inside a loop and the loop will continue until low crosses high. Finally, we will return the ‘ans’ variable that stores the minimum element.

Dry-run: *Please refer to the video for a detailed explanation.*

```
Example:{4,5,1,2,3}
low=0,high=4,mid=2
Check if arr[low] <= arr[mid], its not,
So right part is sorted.
We take ans=min(ans,arr[2]) => ans=1, and high = mid-1.

low=0,high=1,mid=0;
arr[low]<=arr[mid] = true.
So we update ans as min(ans,arr[0]) => ans=1;
Since the left part was sorted low=mid+1. Which makes low = high = 1.

low=1,high=1,mid=1
arr[low] <= arr[mid] = true.
So we update ans as min(ans,arr[1]) => ans=1;
Since the left part was sorted low=mid+1. Which makes low = 2. Loop Stops.
```

▼ Code ›

```
import sys
def findMin(arr: [int]):
    low = 0
    high = len(arr) - 1
    ans = sys.maxsize

    while low <= high:
        mid = (low + high) // 2

        if arr[low] <= arr[mid]:  # if left part is sorted
            ans = min(ans, arr[low])  # keep the minimum
            low = mid + 1  # eliminate left half

        else:  # if right part is sorted
            ans = min(ans, arr[mid])  # keep the minimum
            high = mid - 1  # eliminate right half

    return ans

if __name__ == "__main__":
    arr = [4, 5, 6, 7, 0, 1, 2, 3]
    ans = findMin(arr)
    print("The minimum element is:", ans)
```

Output: The minimum element is: 0

▼ Complexity Analysis ›
Time Complexity: O(logN), N = size of the given array.**Reason:** We are basically using binary search to find the minimum.

Space Complexity: O(1)**Reason:** We have not used any extra data structures, this makes space complexity, even in the worst case as O(1).

▼ Solution 3:
▼ Algorithm / Intuition ›

Further Optimization(Using Binary Search):

If both the left and right halves of an index are sorted, it implies that the entire search space between the low and high indices is also sorted. In this case, there is no need to conduct a binary search within that segment to determine the minimum value. Instead, we can simply select the leftmost element as the minimum.

The condition to check will be **arr[low] <= arr[mid] && arr[mid] <= arr[high]**. We can shorten this into **arr[low] <= arr[high]** as well.

If arr[low] <= arr[high]: In this case, the array from index low to high is completely sorted. Therefore, we can simply select the minimum element, arr[low], and update the 'ans' variable with the minimum value i.e. min(ans, arr[low]). Once this is done, there is no

need to continue with the binary search algorithm.

Algorithm:

The steps are as follows:

We will declare the ‘ans’ variable and initialize it with the largest value possible. With that, as usual, we will declare 2 pointers i.e. low and high.

1. **Place the 2 pointers i.e. low and high:** Initially, we will place the pointers like this: low will point to the first index, and high will point to the last index.
2. **Calculate the ‘mid’:** Now, inside a loop, we will calculate the value of ‘mid’ using the following formula:
   mid = (low+high) // 2 (‘//’ refers to integer division)
3. **If arr[low] <= arr[high]:** In this case, the array from index low to high is completely sorted. Therefore, we can select the minimum element, arr[low], and update the 'ans' variable with the minimum value i.e. min(ans, arr[low]). Once this is done, there is no need to continue with the binary search algorithm. So, we will **break** from this step.
4. Identify the sorted half, and after picking the leftmost element, eliminate that half.
    1. **If arr[low] <= arr[mid]:** This condition ensures that the left part is sorted. So, we will pick the leftmost element i.e. **arr[low]**. Now, we will compare it with 'ans' and update 'ans' with the smaller value (i.e., **min(ans, arr[low])**). Now, we will eliminate this left half(i.e. low = mid+1).
    2. **Otherwise, if the right half is sorted:** This condition ensures that the right half is sorted. So, we will pick the leftmost element i.e. **arr[mid]**. Now, we will compare it with 'ans' and update 'ans' with the smaller value (i.e., **min(ans, arr[mid])**). Now, we will eliminate this right half(i.e. high = mid-1).
5. This process will be inside a loop and the loop will continue until low crosses high. Finally, we will return the ‘ans’ variable that stores the minimum element.

Dry-run: *Please refer to the video for a detailed explanation.*

Note: *Though the time complexity of the following code is the same as the previous one, this code will run slightly faster.*

▼ Code ›

```
import sys
def findMin(arr: [int]):
    low = 0
    high = len(arr) - 1
    ans = sys.maxsize

    while low <= high:
        mid = (low + high) // 2

        # search space is already sorted
        # then arr[low] will always be
        # the minimum in that search space:
        if arr[low] <= arr[high]:
            ans = min(ans, arr[low])
            break

        if arr[low] <= arr[mid]:  # if left part is sorted
            ans = min(ans, arr[low])  # keep the minimum
            low = mid + 1  # eliminate left half

        else:  # if right part is sorted
            ans = min(ans, arr[mid])  # keep the minimum
            high = mid - 1  # eliminate right half

    return ans

if __name__ == "__main__":
    arr = [4, 5, 6, 7, 0, 1, 2, 3]
    ans = findMin(arr)
    print("The minimum element is:", ans)
```

Output: The minimum element is: 0

▼ Complexity Analysis ›

Time Complexity: O(logN), N = size of the given array.
Reason: We are basically using binary search to find the minimum.

Space Complexity: O(1)
Reason: We have not used any extra data structures, this makes space complexity, even in the worst case as O(1).