# Solution Review: Cyclically Shifted Array

This lesson contains the solution review for the challenge to find the index of the smallest number in a cyclically shifted array.

WE'LL COVER THE FOLLOWING



Let's reiterate the problem statement from the previous challenge.

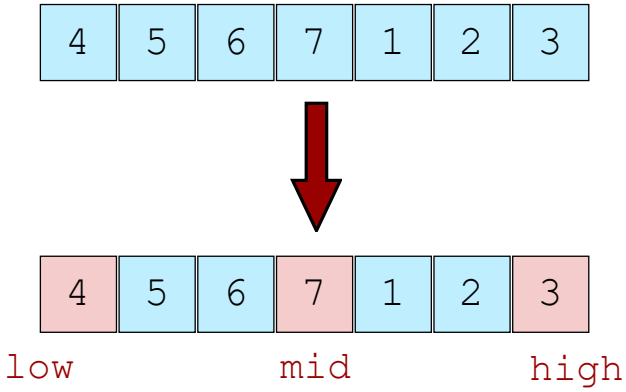
You are required to write a function that determines the index of the smallest element of the cyclically shifted array.

An array is **"cyclically shifted"** if it is possible to shift its entries cyclically so that it becomes sorted.

## Algorithm #

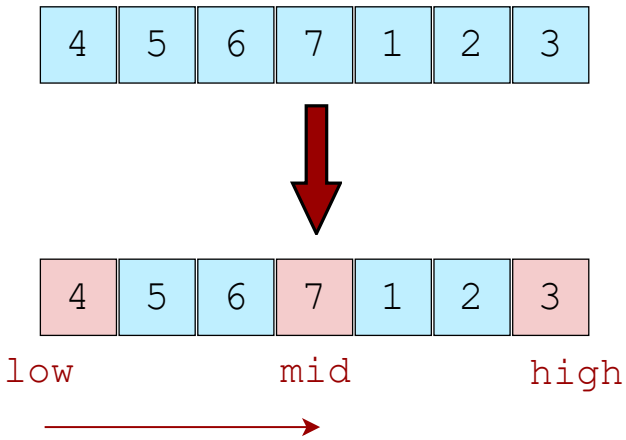
Now we need to come up with a strategy to eliminate parts of the search space. Have a look at the slides below to take note of some observations

## Idea : Binary Search - Example 1



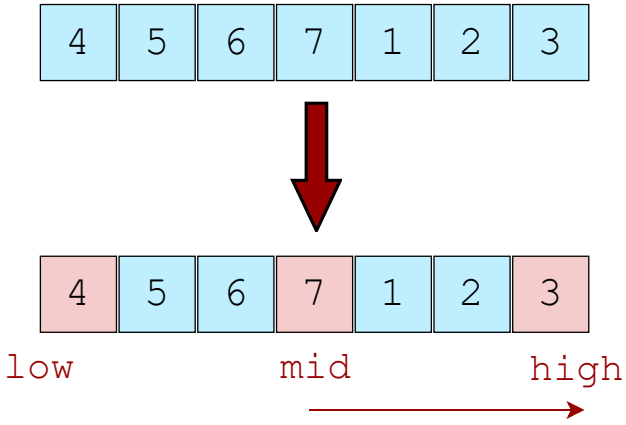



## Idea : Binary Search - Example 1




All the elements from the low to middle are increasing so the smallest element may not be in the first half.

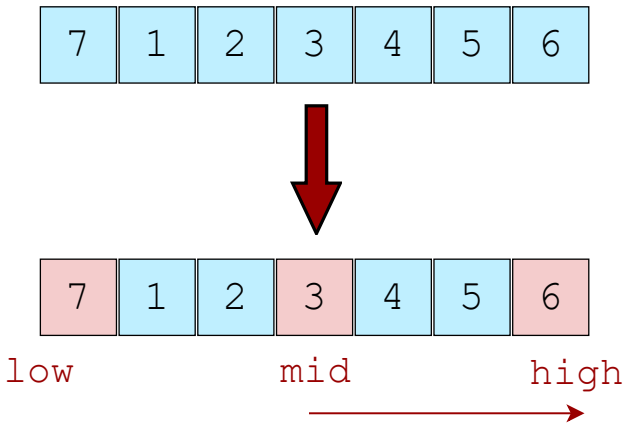
## Idea : Binary Search - Example 1




All the elements from the middle to high are decreasing so the smallest element may be in the second half.

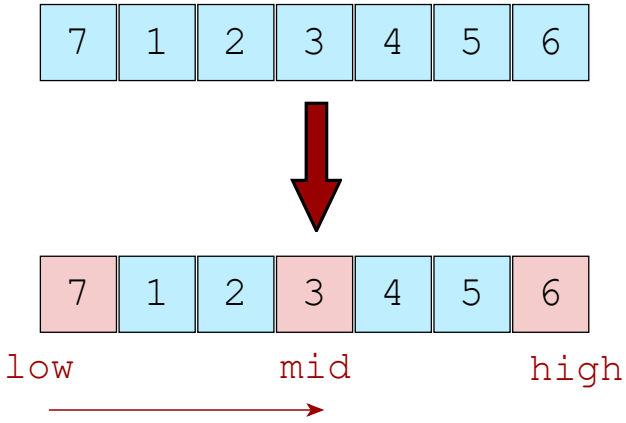


## Idea : Binary Search - Example 2




All the elements from the middle to high are increasing so the smallest element may not be in the second half.

Idea : Binary Search - Example 2

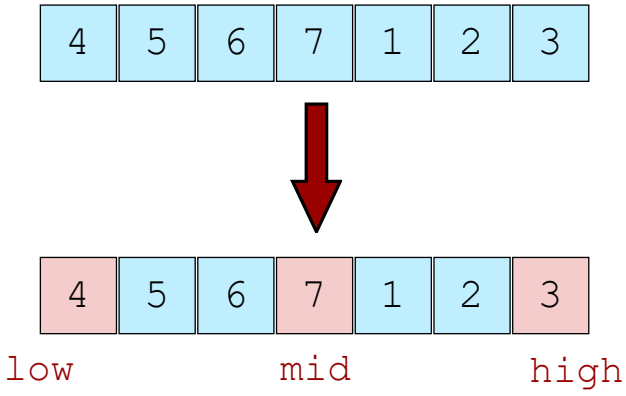



All the elements from the low to mid are decreasing so the smallest element may be in the first half.

**5** of 5

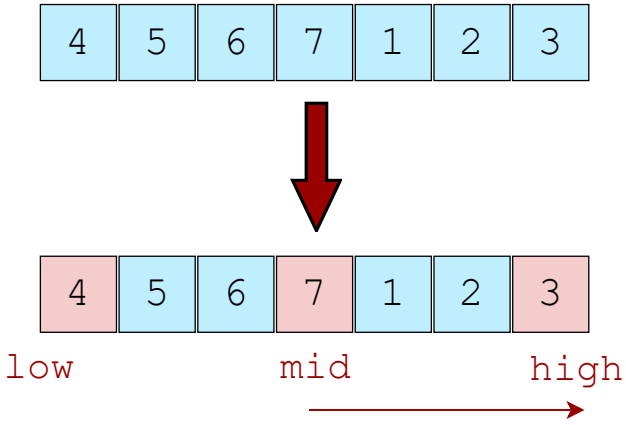
At this point, you will have a basic idea of how to solve this problem. Let's step more into the algorithm which is applied to *Example 1* from the slides above:

Idea : Binary Search




**1** of 4

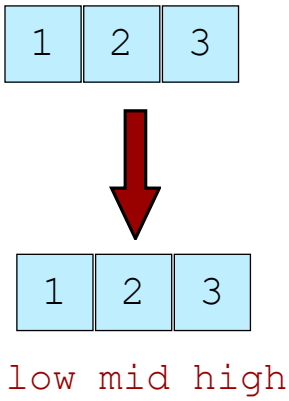
## Idea : Binary Search




All the elements from the middle to high are decreasing so the smallest element may be in the second half.

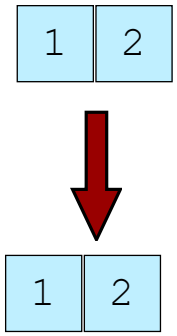


## Idea : Binary Search




2 < 3,
smallest element may be to the left
so we dismiss all the elements to the right of the mid.

Idea : Binary Search




low high

When we come to a case like this, all we need to do is to return the low.



# Implementation #

Now that you have a complete idea of the algorithm, let's jump to the implementation in Python:

```
def find(A):
    low = 0
    high = len(A) - 1

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

        if A[mid] > A[high]:
            low = mid + 1
        elif A[mid] <= A[high]:
            high = mid

    return low

A = [4, 5, 6, 7, 1, 2, 3]
idx = find(A)
print(A[idx])
```

## Explanation #

`low` and `high` are set to `0` and `len(A) - 1` respectively on **lines 2-3**. The code on **lines 5-6** is the same as the code in the standard binary search implementation that we covered at the beginning of the chapter. According to the algorithm, we check on **line 8** if the middle element is greater than `A[high]`. If it is, then it implies that the elements are decreasing from the middle to the high element. To reduce the search space, `low` is set equal to `mid + 1` on **line 9**. **Line 10** is evaluated in case the condition on **line 8** is not `True`, so we check if the middle element is less than or equal to `A[high]`. If this condition evaluates to `True`, it implies that the elements are increasing from `mid` position to `high` position and the smallest element may be somewhere between `low` position to `mid` position. Therefore, `high` is set to `mid` to eliminate the space from `mid` position to the previous `high` position. After the `while` loop terminates, `low` will be the index of the smallest integer in the list.

That's all on what we have for Binary Search. In the next chapter, we'll learn to solve a few problems using recursion. Stay tuned!