# 2855. Minimum Right Shifts to Sort the Array

Easy Array

## Problem Description

In this problem, we are given an array `nums` which contains distinct positive integers in a 0-indexed fashion, meaning the index of the array starts at 0. The goal is to determine the minimum number of right shifts needed to sort the array in ascending order. A right shift means taking an element at index `i` and moving it to index `(i + 1) % n`, where `n` is the length of the array. If the array can be sorted with right shifts, we return the minimum number of shifts required; otherwise, we return `-1` to indicate that sorting is not possible with only right shifts.

A right shift effectively moves the last element of the array to the front while shifting all other elements to the right by one position. This operation can be repeated many times to rotate the array. However, if the array is not a rotated version of a sorted array to begin with, it cannot be sorted by just right shifting.

## Intuition

To solve this problem, we're going to leverage the fact that the array contains distinct positive integers. Given that the array can only be sorted by right shifts (or rotations), the original array must be a rotation of a sorted array. If it is not, then sorting it will be impossible.

The solution starts by stepping through the array while the integers remain in ascending order. The moment this order is violated, we know we've potentially found the point of rotation. From this 'breakpoint', the next step is to check if the array continues to increase up to the first element of the array (i.e., the element before the rotation point). If the order is again violated before reaching the end of the array, it means the array is not a rotation of a sorted array, so we cannot sort it with right shifts alone, and the solution should return `-1`. If we successfully reach the end of the array without any order violations, it means we've confirmed that the array is a rotation of a sorted array.

## Solution Approach

The implementation of the solution follows a straightforward algorithm to identify if the array is a single rotation of a sorted sequence and then calculates the minimum right shifts needed to sort the array.

The complete approach is as follows:

1. Initialize two pointers, `i` and `k`.
   - `i` starts from 1 because we want to compare with the previous element, and it marks the index we're checking for the ascending order.
   - `k` is used to determine if the remaining part of the array is increasing and is smaller than `nums[0]`. It starts from `i + 1`.
2. The first `while` loop finds the first occurrence where `nums[i-1] >= nums[i]`. This is the standard pattern to identify the break in ascending order. If `i` reaches the end of the array `n` without triggering the break condition, it means the array is already sorted, and no right shifts are needed; hence, return `0`.
3. The second `while` loop starts from the index `k` and checks whether every number from that index onwards is less than `nums[0]` and the sequence is still increasing. If `k` reaches `n` without breaking the loop, the array is one rotation away from being sorted.
4. The return value:
   - If `k` has not reached `n`, it means that there's another break in the order, hence the sequence isn't just a single rotation of a sorted array, and sorting it by rotations is impossible. In this case, return `-1`.
   - Otherwise, return `n - i`, which indicates the minimum number of right shifts required to bring `nums[i]` to the 0th index, effectively sorting the array.

Here is how the code encapsulates this logic:

```
class Solution:
    def minimumRightShifts(self, nums: List[int]) -> int:
        n = len(nums)
        i = 1
        while i < n and nums[i - 1] < nums[i]:
            i += 1
        if i == n:  # The array is already sorted.
            return 0
        k = i + 1
        while k < n and nums[k - 1] < nums[k] < nums[0]:
            k += 1
        return -1 if k < n else n - i
```

This solution leverages simple array traversal and comparison without any complex data structures or advanced algorithms.

- No additional space complexity is introduced since we are only using a few integer variables ($i$, $k$, $n$).
- The time complexity is at most O(2n) since both loops could potentially iterate over the whole array, but never more than once per element, simplifying to O(n).

One key point in understanding this algorithm is how the sorted array property is used to verify the possibility of sorting by rotations and then calculating the minimal rotation if possible.

### Example Walkthrough

Let's use an example to illustrate the solution approach. Consider the following array:

```
nums = [3, 4, 5, 1, 2]
```

Now, let's walk through the approach step by step:

1. We initialize `i` to `1` as we want to start comparing from the first element with the previous one (index 0). Variable `k` is not used until later.
2. The initial state is `i = 1`, and we compare `nums[i - 1]` and `nums[i]`. The pair we are comparing is therefore `(3, 4)` which satisfies `nums[i - 1] < nums[i]`, so we increment `i` to `2`.
3. Continuing the loop, `i` now is `2`, and we compare `(4, 5)`. This also satisfies the ascending order so `i` is incremented to `3`.
4. At `i = 3`, we compare `(5, 1)` and find that `nums[i - 1] >= nums[i]`, which breaks the ascending order. According to our logic, this indicates where the array was potentially rotated.
5. We set `k = i + 1` which is `4` in this case and enter the second `while` loop to verify that the remaining part of the array is still in ascending order and each element is less than `nums[0]`.
6. Now, `k = 4` and we compare `(1, 2)`. The condition `nums[k - 1] < nums[k] < nums[0]` is satisfied as `(1 < 2 < 3)`. Since `k < n`, we continue and increment `k` to `5`.
7. As `k` equals to `n` (end of array), which means the loop was never broken and confirms that the array is a rotation of a sorted array.
8. Finally, we calculate the minimum number of right shifts required to sort the array which is `n - i`. In this case, `n` is `5` and `i` is `3`, so we need `5 - 3 = 2` right shifts.

Therefore, the array `[3, 4, 5, 1, 2]` requires a minimum of `2` right shifts to be sorted in ascending order.

## Solution Implementation

### Python

```
from typing import List

class Solution:

    def minimum_right_shifts(self, nums: List[int]) -> int:
        # Determine the length of the input list `nums`
        length = len(nums)

        # Initialize the index `i` to start checking the ascending order from the beginning of the list
        i = 1

        # Find the first breaking point where ascending order is violated
        while i < length and nums[i - 1] < nums[i]:
            i += 1

        # If there was no breaking point, the list is already in ascending order
        if i == length:
            return 0

        # Initialize `k` for the second part of the task to identify if the right-shift requirement is met.
        k = i + 1

        # Check if the rest of the list after the breaking point is in ascending order and
        # that it is less than the first element (to ensure a proper shifting sequence)
        while k < length and nums[k - 1] < nums[k] < nums[0]:
            k += 1

        # If `k` did not reach end of the list, it means the right-shift to fix is impossible
        return -1 if k < length else length - i  # Return the number of shifts or -1 if not possible

# Example of creating an instance of the solution and using the method
solution_instance = Solution()
print(solution_instance.minimum_right_shifts([3, 4, 5, 1, 2]))  # Example output: 3 because it takes three right shifts for the list
```

### Java

```
class Solution {
    public int minimumRightShifts(List<Integer> nums) {
        // Get the size of the list
        int listSize = nums.size();

        // Initialize an index to track how many elements are
        // in non-decreasing order starting from the beginning of the list
        int nonDecreasingIndex = 1;

        // Iterate through the list until elements are no longer in
        // non-decreasing order. Starting from the second element
        while (nonDecreasingIndex < listSize && nums.get(nonDecreasingIndex - 1) < nums.get(nonDecreasingIndex)) {
            ++nonDecreasingIndex;
        }

        // Once the non-decreasing sequence is broken, start another
        // index to track the rest of the list if the elements are in non-decreasing order
        // and also less than the first element of the list (to ensure proper rotation)
        int checkIndex = nonDecreasingIndex + 1;
        while (checkIndex < listSize && nums.get(checkIndex - 1) < nums.get(checkIndex) && nums.get(checkIndex) < nums.get(0)) {
            ++checkIndex;
        }

        // If the checkIndex hasn't reached the end of the list, it means that the list can't be
        // sorted by right rotations alone, so we return -1.
        // Otherwise, return the number of right shifts required, which is the size of the list minus
        // the index of the initial non-decreasing sequence
        return checkIndex < listSize ? -1 : listSize - nonDecreasingIndex;
    }
}
```

### C++

```
class Solution {
public:
    int minimumRightShifts(vector<int>& nums) {
        // Get the size of the vector nums.
        int numsSize = nums.size();

        // Initialize an index to keep track of the sequence.
        int index = 1;

        // Loop through the vector to find when the ascending sequence breaks.
        while (index < numsSize && nums[index - 1] < nums[index]) {
            ++index;
        }

        // If the entire vector is in ascending order, no right shift is needed.
        if (index == numsSize) {
            return 0;
        }

        // Initialize an index for the second part of the sequence.
        int secondIndex = index + 1;

        // Loop through the second part to validate ascending order
        // till an element is less than the first element of the vector.
        while (secondIndex < numsSize &&
               nums[secondIndex - 1] < nums[secondIndex] &&
               nums[secondIndex] < nums[0]) {
            ++secondIndex;
        }

        // If secondIndex did not reach the end, sequence is not strictly ascending,
        // return -1 indicating it's impossible to achieve the condition.
        if (secondIndex < numsSize) {
            return -1;
        }

        // Return the number of right shifts needed which is the length of the vector
        // minus the length of the initial ascending sequence.
        return numsSize - index;
    }
};
```

### TypeScript

```
/**
 * Computes the minimum number of right shifts required to sort the array in non-decreasing order.
 * If it's impossible to sort the array with only right shifts, the function returns -1.
 * @param {number[]} nums - The input array of numbers.
 * @return {number} The minimum number of shifts required, or -1 if impossible.
 */
function minimumRightShifts(nums: number[]): number {
    // Get the length of the array.
    const length = nums.length;

    // Initialize the first index to 1.
    let currentIndex = 1;

    // Find the first element that is smaller than its previous element (breaks the increasing order).
    while (currentIndex < length && nums[currentIndex - 1] < nums[currentIndex]) {
        currentIndex++;
    }

    // If we have reached the end of the array, no right shifts are needed.
    if (currentIndex === length) {
        return 0;
    }

    // Define a variable to start checking for the correct position of first element after shifting.
    let indexToCheck = currentIndex;

    // Check that the remaining elements are still in increasing order and smaller than the first element.
    while (indexToCheck < length && nums[indexToCheck - 1] < nums[indexToCheck] && nums[indexToCheck] < nums[0]) {
        indexToCheck++;
    }

    // If indexToCheck has not reached the end, sorting is impossible.
    return indexToCheck < length ? -1 : length - currentIndex;
}
```

```
from typing import List

class Solution:

    def minimum_right_shifts(self, nums: List[int]) -> int:
        # Determine the length of the input list `nums`
        length = len(nums)

        # Initialize the index `i` to start checking the ascending order from the beginning of the list
        i = 1

        # Find the first breaking point where ascending order is violated
        while i < length and nums[i - 1] < nums[i]:
            i += 1

        # If there was no breaking point, the list is already in ascending order
        if i == length:
            return 0

        # Initialize `k` for the second part of the task to identify if the right-shift requirement is met.
        k = i + 1

        # Check if the rest of the list after the breaking point is in ascending order and
        # that it is less than the first element (to ensure a proper shifting sequence)
        while k < length and nums[k - 1] < nums[k] < nums[0]:
            k += 1

        # If `k` did not reach end of the list, it means the right-shift to fix is impossible
        return -1 if k < length else length - i  # Return the number of shifts or -1 if not possible

# Example of creating an instance of the solution and using the method
solution_instance = Solution()
print(solution_instance.minimum_right_shifts([3, 4, 5, 1, 2]))  # Example output: 3 because it takes three right shifts for the list
```

## Time and Space Complexity

### Time Complexity

The given Python function involves checking each element of the array exactly once in the two `while` loops. The first loop continues until it encounters a non-ascending pair (`nums[i - 1] >= nums[i]`), and the second loop continues until it reaches an element that does not satisfy the condition (`nums[k - 1] < nums[k] < nums[0]`) or until it reaches the end of the array.

Both loops have a worst-case scenario where they traverse the entire list ($n$ being the length of the list).

- The first loop runs in $O(n)$ time in the worst case.
- The second loop also runs in $O(n)$ time in the worst case.

As both loops run sequentially, the overall time complexity of the function is $O(n) + O(n) = O(n)$.

### Space Complexity

The function only uses a few integer variables ($i$, $k$, $n$) and does not allocate any additional space that grows with the size of the input. Therefore, the space complexity of the function is $O(1)$.