Search Rotated Array

Problem Statement

Search for a given number in a sorted array, with unique elements, that has been rotated by some arbitrary number. Return `-1` if the number does not exist.

> Assume that the array does not contain duplicates

Below is an original array before rotation

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	10	20	47	59	63	75	88	99	107	120	133	155	162	176	188	199	200	210	222

After performing rotation on this array 6 times it changes to:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
176	188	199	200	210	222	1	10	20	47	59	63	75	88	99	107	120	133	155	162

The task is to find a given number in this array.

Hint

- Linear search is not an acceptable solution
- Try to solve the problem using binary search

Try it yourself

C++ | Java | Python | JavaScript | Ruby

```
1  int binary_search_rotated(vector<int>& arr, int key) {
2    // TODO: Write - Your - Code
3    return -1;
4  }
```

Test

Solution

C++ | Java | Python | JavaScript | Ruby

```
  if (start > end) {
    return -1;
  }

  int mid = start + (end - start) / 2;

  if (arr[mid] == key) {
    return mid;
  }

  if (arr[start] <= arr[mid] && key <= arr[mid] && key >= arr[start]) {
    return binary_search(arr, start, mid-1, key);
  }

  else if (arr[mid] <= arr[end] && key >= arr[mid] && key <= arr[end]) {
    return binary_search(arr, mid+1, end, key);
  }

  else if (arr[end] <= arr[mid]) {
    return binary_search(arr, mid+1, end, key);
  }

  else if (arr[start] >= arr[mid]) {
    return binary_search(arr, start, mid-1, key);
  }

  return -1;
}

int binary_search_rotated(vector<int>& arr, int key) {
```

Run

Solution Explanation

Runtime complexity

The runtime complexity of this solution is logarithmic, O(log n).

Memory complexity

The memory complexity of this solution is logarithmic, O(log n).

Solution Breakdown

The solution is essentially a binary search but with some modifications. If we look at the array in the

The solution is essentially a binary search but with some modifications. If we look at the array in the example closely, we notice that at least one half of the array is always sorted. We can use this property t our advantage. If the number ‘n’ lies within the sorted half of the array, then our problem is a basic binary search. Otherwise, discard the sorted half and keep examining the unsorted half. Since we are partitioning the array in half at each step, this gives us O(log n) runtime complexity.