**Binary Search**

**Introduction**

As we know, whenever we are given a sorted **Array** or **LinkedList** or **Matrix**, and we are asked to find a certain element, the best algorithm we can use is the Binary Search.

This pattern describes an efficient way to handle all problems involving **Binary Search**. We will go through a set of problems that will help us build an understanding of this pattern so that we can apply this technique to other problems we might come across in the interviews.

Let’s start with our first problem.

#### **Approach 1: Binary Search**

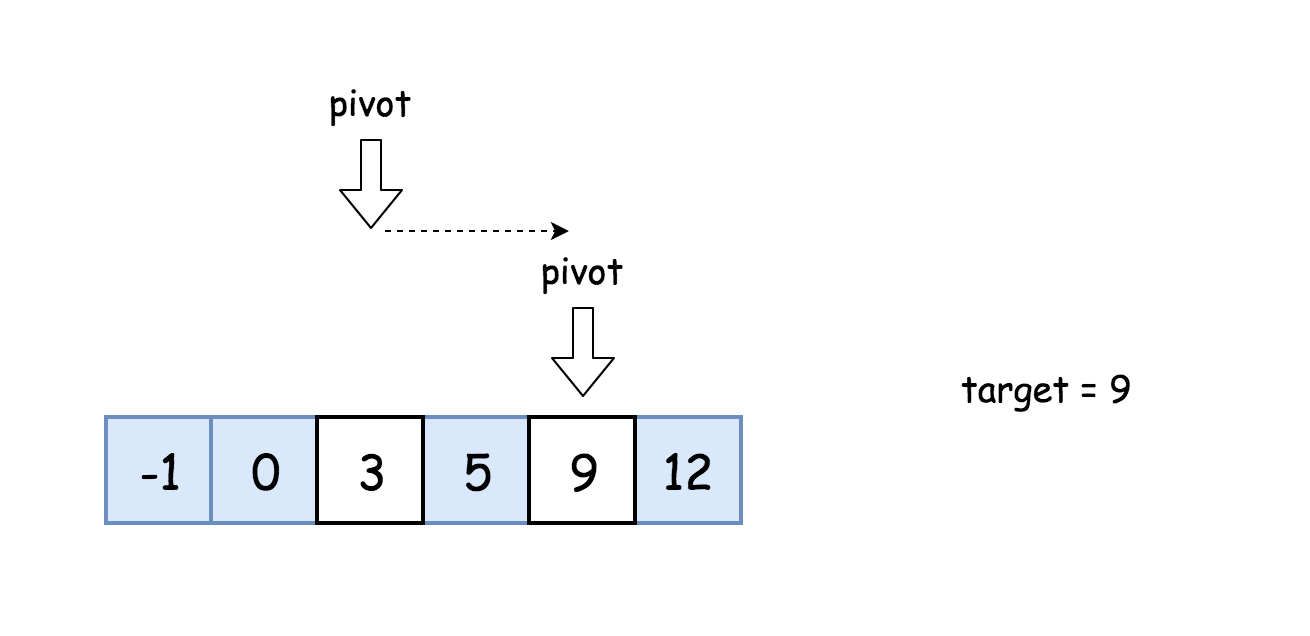
Intuition

Binary search is a textbook algorithm based on the idea to compare the target value to the middle element of the array.

If the target value is equal to the middle element - we're done.

If the target value is smaller - continue to search on the left.

If the target value is larger - continue to search on the right.



**Algorithm**

* Initialise left and right pointers : left = 0, right = n - 1.
* While left <= right :
  + Compare middle element of the array nums[pivot] to the target value target.
    - If the middle element *is* the target target = nums[pivot] : return pivot.
    - If the target is not yet found :
      * If target < nums[pivot], continue the search on the left right = pivot - 1.
      * Else continue the search on the right left = pivot + 1.

**Implementation**

class Solution {

  public int search(int[] nums, int target) {

    int pivot, left = 0, right = nums.length - 1;

    while (left <= right) {

      pivot = left + (right - left) / 2;

      if (nums[pivot] == target) return pivot;

      if (target < nums[pivot]) right = pivot - 1;

      else left = pivot + 1;

    }

    return -1;

  }

}

**Complexity Analysis**

* Time complexity : O(log*N*).

Let's compute time complexity with the help of [master theorem](https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)) *T*(*N*)=*aT*(*bN*​)+Θ(*Nd*). The equation represents dividing the problem up into a*a* subproblems of size  ​  Θ(*Nd*) time. Here at step there is only one subproblem a = 1, its size is a half of the initial problem b = 2, and all this happens in a constant time d = 0. That means that log*b*​*a*=*d* and hence we're dealing with [case 2](https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)#Case_2_example) that results in O(*n*log*b*​*a*log*d*+1*N*) = O(log*N*) time complexity.

* Space complexity : O(1) since it's a constant space solution.

# Order-agnostic Binary Search

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/R8LzZQlj8lO#problem-statement)

Given a sorted array of numbers, find if a given number ‘key’ is present in the array. Though we know that the array is sorted, we don’t know if it’s sorted in ascending or descending order. You should assume that the array can have duplicates.

Write a function to return the index of the ‘key’ if it is present in the array, otherwise return -1.

**Example 1:**

Input: [4, 6, 10], key = 10  
Output: 2

**Example 2:**

Input: [1, 2, 3, 4, 5, 6, 7], key = 5  
Output: 4

**Example 3:**

Input: [10, 6, 4], key = 10  
Output: 0

**Example 4:**

Input: [10, 6, 4], key = 4  
Output: 2

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/R8LzZQlj8lO#solution)

To make things simple, let’s first solve this problem assuming that the input array is sorted in ascending order. Here are the set of steps for **Binary Search**:

1. Let’s assume start is pointing to the first index and end is pointing to the last index of the input array (let’s call it arr). This means:

    int start = 0;  
    int end = arr.length - 1;

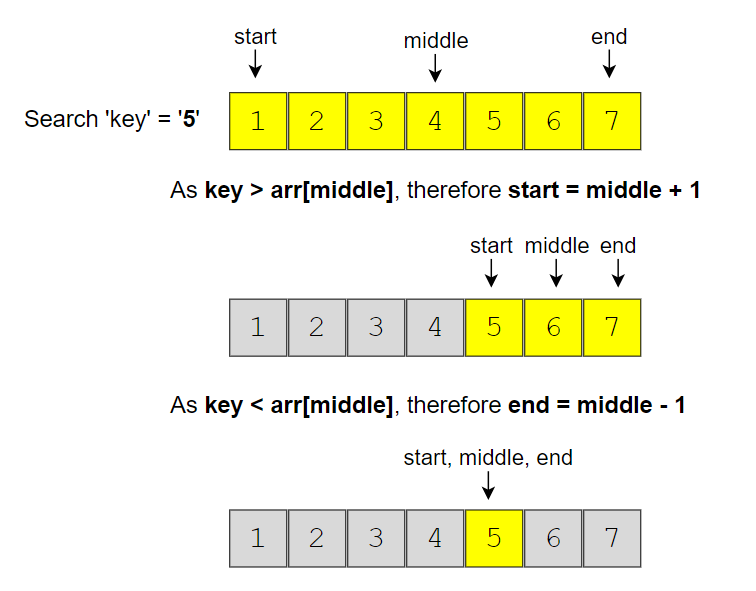
1. First, we will find the middle of start and end. An easy way to find the middle would be: middle=(start+end)/2*middle*=(*start*+*end*)/2. For **Java and C++**, this equation will work for most cases, but when start or end is large, this equation will give us the wrong result due to integer overflow. Imagine that end is equal to the maximum range of an integer (e.g. for Java: int end = Integer.MAX\_VALUE). Now adding any positive number to end will result in an integer overflow. Since we need to add both the numbers first to evaluate our equation, an overflow might occur. The safest way to find the middle of two numbers without getting an overflow is as follows:

     middle  = start + (end-start)/2

The above discussion is not relevant for **Python**, as we don’t have the integer overflow problem in pure Python.

1. Next, we will see if the ‘key’ is equal to the number at index middle. If it is equal we return middle as the required index.
2. If ‘key’ is not equal to number at index middle, we have to check two things:
   * If key < arr[middle], then we can conclude that the key will be smaller than all the numbers after index middle as the array is sorted in the ascending order. Hence, we can reduce our search to end = mid - 1.
   * If key > arr[middle], then we can conclude that the key will be greater than all numbers before index middle as the array is sorted in the ascending order. Hence, we can reduce our search to start = mid + 1.
3. We will repeat steps 2-4 with new ranges of start to end. If at any time start becomes greater than end, this means that we can’t find the ‘key’ in the input array and we must return ‘-1’.

Here is the visual representation of **Binary Search** for the Example-2:



If the array is sorted in the descending order, we have to update the step 4 above as:

* If key > arr[middle], then we can conclude that the key will be greater than all numbers after index middle as the array is sorted in the descending order. Hence, we can reduce our search to end = mid - 1.
* If key < arr[middle], then we can conclude that the key will be smaller than all the numbers before index middle as the array is sorted in the descending order. Hence, we can reduce our search to start = mid + 1.

Finally, how can we figure out the sort order of the input array? We can compare the numbers pointed out by start and end index to find the sort order. If arr[start] < arr[end], it means that the numbers are sorted in ascending order otherwise they are sorted in the descending order.

### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/R8LzZQlj8lO#code)

class BinarySearch {

  public static int search(int[] arr, int key) {

    int start = 0, end = arr.length - 1;

    boolean isAscending = arr[start] < arr[end];

    while (start <= end) {

      // calculate the middle of the current range

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

      if (key == arr[mid])

        return mid;

      if (isAscending) { // ascending order

        if (key < arr[mid]) {

          end = mid - 1; // the 'key' can be in the first half

        } else { // key > arr[mid]

          start = mid + 1; // the 'key' can be in the second half

        }

      } else { // descending order

        if (key > arr[mid]) {

          end = mid - 1; // the 'key' can be in the first half

        } else { // key < arr[mid]

          start = mid + 1; // the 'key' can be in the second half

        }

      }

    }

    return -1; // element not found

  }

  public static void main(String[] args) {

    System.out.println(BinarySearch.search(new int[] { 4, 6, 10 }, 10));

    System.out.println(BinarySearch.search(new int[] { 1, 2, 3, 4, 5, 6, 7 }, 5));

    System.out.println(BinarySearch.search(new int[] { 10, 6, 4 }, 10));

    System.out.println(BinarySearch.search(new int[] { 10, 6, 4 }, 4));

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/R8LzZQlj8lO#time-complexity)

Since, we are reducing the search range by half at every step, this means that the time complexity of our algorithm will be O(logN)*O*(*logN*) where ‘N’ is the total elements in the given array.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/R8LzZQlj8lO#space-complexity)

The algorithm runs in constant space O(1)*O*(1).

# Ceiling of a Number

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qA5wW7R8ox7#problem-statement)

Given an array of numbers sorted in an ascending order, find the ceiling of a given number ‘key’. The ceiling of the ‘key’ will be the smallest element in the given array greater than or equal to the ‘key’.

Write a function to return the index of the ceiling of the ‘key’. If there isn’t any ceiling return -1.

**Example 1:**

Input: [4, 6, 10], key = 6  
Output: 1  
Explanation: The smallest number greater than or equal to '6' is '6' having index '1'.

**Example 2:**

Input: [1, 3, 8, 10, 15], key = 12  
Output: 4  
Explanation: The smallest number greater than or equal to '12' is '15' having index '4'.

**Example 3:**

Input: [4, 6, 10], key = 17  
Output: -1  
Explanation: There is no number greater than or equal to '17' in the given array.

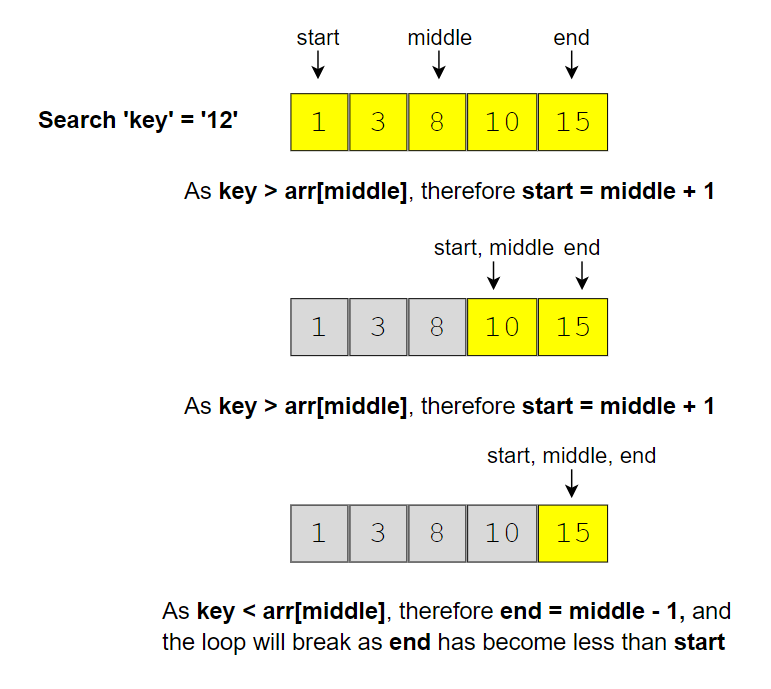
**Example 4:**

Input: [4, 6, 10], key = -1  
Output: 0  
Explanation: The smallest number greater than or equal to '-1' is '4' having index '0'.

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qA5wW7R8ox7#solution)

This problem follows the **Binary Search** pattern. Since Binary Search helps us find a number in a sorted array efficiently, we can use a modified version of the Binary Search to find the ceiling of a number.

We can use a similar approach as discussed in Order-agnostic Binary Search. We will try to search for the ‘key’ in the given array. If we find the ‘key’, we return its index as the ceiling. If we can’t find the ‘key’, the next big number will be pointed out by the index start. Consider Example-2 mentioned above:



Since we are always adjusting our range to find the ‘key’, when we exit the loop, the start of our range will point to the smallest number greater than the ‘key’ as shown in the above picture.

We can add a check in the beginning to see if the ‘key’ is bigger than the biggest number in the input array. If so, we can return ‘-1’.

### Code

class CeilingOfANumber {

  public static int searchCeilingOfANumber(int[] arr, int key) {

    if (key > arr[arr.length - 1]) // if the 'key' is bigger than the biggest element

      return -1;

    int start = 0, end = arr.length - 1;

    while (start <= end) {

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

      if (key < arr[mid]) {

        end = mid - 1;

      } else if (key > arr[mid]) {

        start = mid + 1;

      } else { // found the key

        return mid;

      }

    }

    // since the loop is running until 'start <= end', so at the end of the while loop, 'start == end+1'

    // we are not able to find the element in the given array, so the next big number will be arr[start]

    return start;

  }

  public static void main(String[] args) {

    System.out.println(CeilingOfANumber.searchCeilingOfANumber(new int[] { 4, 6, 10 }, 6));

    System.out.println(CeilingOfANumber.searchCeilingOfANumber(new int[] { 1, 3, 8, 10, 15 }, 12));

    System.out.println(CeilingOfANumber.searchCeilingOfANumber(new int[] { 4, 6, 10 }, 17));

    System.out.println(CeilingOfANumber.searchCeilingOfANumber(new int[] { 4, 6, 10 }, -1));

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qA5wW7R8ox7#time-complexity)

Since we are reducing the search range by half at every step, this means that the time complexity of our algorithm will be *O*(*logN*) where ‘N’ is the total elements in the given array.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qA5wW7R8ox7#space-complexity)

The algorithm runs in constant space *O*(1).

# Floor of a Number

### Problem Statement

#### [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qA5wW7R8ox7#problem-1)

Given an array of numbers sorted in ascending order, find the floor of a given number ‘key’. The floor of the ‘key’ will be the biggest element in the given array smaller than or equal to the ‘key’

Write a function to return the index of the floor of the ‘key’. If there isn’t a floor, return -1.

**Example 1:**

Input: [4, 6, 10], key = 6  
Output: 1  
Explanation: The biggest number smaller than or equal to '6' is '6' having index '1'.

**Example 2:**

Input: [1, 3, 8, 10, 15], key = 12  
Output: 3  
Explanation: The biggest number smaller than or equal to '12' is '10' having index '3'.

**Example 3:**

Input: [4, 6, 10], key = 17  
Output: 2  
Explanation: The biggest number smaller than or equal to '17' is '10' having index '2'.

**Example 4:**

Input: [4, 6, 10], key = -1  
Output: -1  
Explanation: There is no number smaller than or equal to '-1' in the given array.

#### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/qA5wW7R8ox7#code-2)

class FloorOfANumber {

  public static int searchFloorOfANumber(int[] arr, int key) {

    if (key < arr[0]) // if the 'key' is smaller than the smallest element

      return -1;

    int start = 0, end = arr.length - 1;

    while (start <= end) {

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

      if (key < arr[mid]) {

        end = mid - 1;

      } else if (key > arr[mid]) {

        start = mid + 1;

      } else { // found the key

        return mid;

      }

    }

    // since the loop is running until 'start <= end', so at the end of the while loop, 'start == end+1'

    // we are not able to find the element in the given array, so the next smaller number will be arr[end]

    return end;

  }

  public static void main(String[] args) {

    System.out.println(FloorOfANumber.searchFloorOfANumber(new int[] { 4, 6, 10 }, 6));

    System.out.println(FloorOfANumber.searchFloorOfANumber(new int[] { 1, 3, 8, 10, 15 }, 12));

    System.out.println(FloorOfANumber.searchFloorOfANumber(new int[] { 4, 6, 10 }, 17));

    System.out.println(FloorOfANumber.searchFloorOfANumber(new int[] { 4, 6, 10 }, -1));

  }

}

# Next Letter

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/g2w6QPBA2Nk#problem-statement)

Given an array of lowercase letters sorted in ascending order, find the **smallest letter** in the given array **greater than a given ‘key’**.

Assume the given array is a **circular list**, which means that the last letter is assumed to be connected with the first letter. This also means that the smallest letter in the given array is greater than the last letter of the array and is also the first letter of the array.

Write a function to return the next letter of the given ‘key’.

**Example 1:**

Input: ['a', 'c', 'f', 'h'], key = 'f'  
Output: 'h'  
Explanation: The smallest letter greater than 'f' is 'h' in the given array.

**Example 2:**

Input: ['a', 'c', 'f', 'h'], key = 'b'  
Output: 'c'  
Explanation: The smallest letter greater than 'b' is 'c'.

**Example 3:**

Input: ['a', 'c', 'f', 'h'], key = 'm'  
Output: 'a'  
Explanation: As the array is assumed to be circular, the smallest letter greater than 'm' is 'a'.

**Example 4:**

Input: ['a', 'c', 'f', 'h'], key = 'h'  
Output: 'a'  
Explanation: As the array is assumed to be circular, the smallest letter greater than 'h' is 'a'.

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/g2w6QPBA2Nk#solution)

The problem follows the **Binary Search** pattern. Since **Binary Search** helps us find an element in a sorted array efficiently, we can use a modified version of it to find the next letter.

We can use a similar approach as discussed in Ceiling of a Number. There are a couple of differences though:

1. The array is considered circular, which means if the ‘key’ is bigger than the last letter of the array or if it is smaller than the first letter of the array, the key’s next letter will be the first letter of the array.
2. The other difference is that we have to find the next biggest letter which can’t be equal to the ‘key’. This means that we will ignore the case where key == arr[middle]. To handle this case, we can update our start range to start = middle +1.

In the end, instead of returning the element pointed out by start, we have to return the letter pointed out by start % array\_length. This is needed because of point 2 discussed above. Imagine that the last letter of the array is equal to the ‘key’. In that case, we have to return the first letter of the input array.

### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/g2w6QPBA2Nk#code)

class NextLetter {

  public static char searchNextLetter(char[] letters, char key) {

    int n = letters.length;

    if (key < letters[0] || key > letters[n - 1])

      return letters[0];

    int start = 0, end = n - 1;

    while (start <= end) {

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

      if (key < letters[mid]) {

        end = mid - 1;

      } else { //if (key >= letters[mid]) {

        start = mid + 1;

      }

    }

    // since the loop is running until 'start <= end', so at the end of the while loop, 'start == end+1'

    return letters[start % n];

  }

  public static void main(String[] args) {

    System.out.println(NextLetter.searchNextLetter(new char[] { 'a', 'c', 'f', 'h' }, 'f'));

    System.out.println(NextLetter.searchNextLetter(new char[] { 'a', 'c', 'f', 'h' }, 'b'));

    System.out.println(NextLetter.searchNextLetter(new char[] { 'a', 'c', 'f', 'h' }, 'm'));

    System.out.println(NextLetter.searchNextLetter(new char[] { 'a', 'c', 'f', 'h' }, 'h'));

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/g2w6QPBA2Nk#time-complexity)

Since, we are reducing the search range by half at every step, this means that the time complexity of our algorithm will be *O*(*logN*) where ‘N’ is the total elements in the given array.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/g2w6QPBA2Nk#space-complexity)

The algorithm runs in constant space *O*(1).

# Number Range

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/R1B78K9oBEz#problem-statement)

Given an array of numbers sorted in ascending order, find the range of a given number ‘key’. The range of the ‘key’ will be the first and last position of the ‘key’ in the array.

Write a function to return the range of the ‘key’. If the ‘key’ is not present return [-1, -1].

**Example 1:**

Input: [4, 6, 6, 6, 9], key = 6  
Output: [1, 3]

**Example 2:**

Input: [1, 3, 8, 10, 15], key = 10  
Output: [3, 3]

**Example 3:**

Input: [1, 3, 8, 10, 15], key = 12  
Output: [-1, -1]

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/R1B78K9oBEz#solution)

The problem follows the **Binary Search** pattern. Since Binary Search helps us find a number in a sorted array efficiently, we can use a modified version of the Binary Search to find the first and the last position of a number.

We can use a similar approach as discussed in Order-agnostic Binary Search. We will try to search for the ‘key’ in the given array; if the ‘key’ is found (i.e. key == arr[middle) we have two options:

1. When trying to find the first position of the ‘key’, we can update end = middle - 1 to see if the key is present before middle.
2. When trying to find the last position of the ‘key’, we can update start = middle + 1 to see if the key is present after middle.

In both cases, we will keep track of the last position where we found the ‘key’. These positions will be the required range.

### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/R1B78K9oBEz#code)

class FindRange {

  public static int[] findRange(int[] arr, int key) {

    int[] result = new int[] { -1, -1 };

    result[0] = search(arr, key, false);

    if (result[0] != -1) // no need to search, if 'key' is not present in the input array

      result[1] = search(arr, key, true);

    return result;

  }

  // modified Binary Search

  private static int search(int[] arr, int key, boolean findMaxIndex) {

    int keyIndex = -1;

    int start = 0, end = arr.length - 1;

    while (start <= end) {

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

      if (key < arr[mid]) {

        end = mid - 1;

      } else if (key > arr[mid]) {

        start = mid + 1;

      } else { // key == arr[mid]

        keyIndex = mid;

        if (findMaxIndex)

          start = mid + 1; // search ahead to find the last index of 'key'

        else

          end = mid - 1; // search behind to find the first index of 'key'

      }

    }

    return keyIndex;

  }

  public static void main(String[] args) {

    int[] result = FindRange.findRange(new int[] { 4, 6, 6, 6, 9 }, 6);

    System.out.println("Range: [" + result[0] + ", " + result[1] + "]");

    result = FindRange.findRange(new int[] { 1, 3, 8, 10, 15 }, 10);

    System.out.println("Range: [" + result[0] + ", " + result[1] + "]");

    result = FindRange.findRange(new int[] { 1, 3, 8, 10, 15 }, 12);

    System.out.println("Range: [" + result[0] + ", " + result[1] + "]");

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/R1B78K9oBEz#time-complexity)

Since, we are reducing the search range by half at every step, this means that the time complexity of our algorithm will be *O*(*logN*) where ‘N’ is the total elements in the given array.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/R1B78K9oBEz#space-complexity)

The algorithm runs in constant space *O*(1).

# Search in a Sorted Infinite Array

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B1ZW38kXJB2#problem-statement)

Given an infinite sorted array (or an array with unknown size), find if a given number ‘key’ is present in the array. Write a function to return the index of the ‘key’ if it is present in the array, otherwise return -1.

Since it is not possible to define an array with infinite (unknown) size, you will be provided with an interface ArrayReader to read elements of the array. ArrayReader.get(index) will return the number at index; if the array’s size is smaller than the index, it will return Integer.MAX\_VALUE.

**Example 1:**

Input: [4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30], key = 16  
Output: 6  
Explanation: The key is present at index '6' in the array.

**Example 2:**

Input: [4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30], key = 11  
Output: -1  
Explanation: The key is not present in the array.

**Example 3:**

Input: [1, 3, 8, 10, 15], key = 15  
Output: 4  
Explanation: The key is present at index '4' in the array.

**Example 4:**

Input: [1, 3, 8, 10, 15], key = 200  
Output: -1  
Explanation: The key is not present in the array.

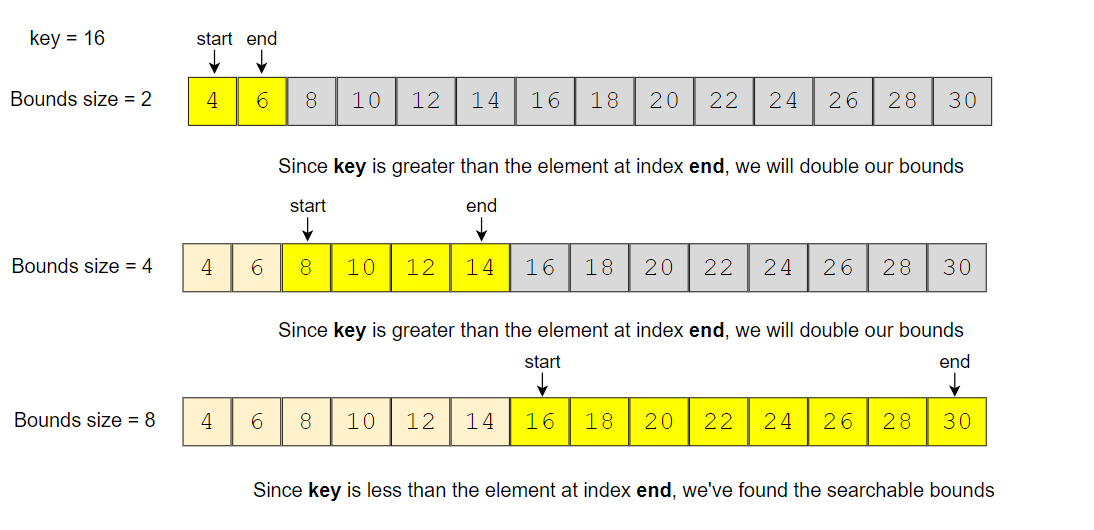
### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B1ZW38kXJB2#solution)

The problem follows the **Binary Search** pattern. Since Binary Search helps us find a number in a sorted array efficiently, we can use a modified version of the Binary Search to find the ‘key’ in an infinite sorted array.

The only issue with applying binary search in this problem is that we don’t know the bounds of the array. To handle this situation, we will first find the proper bounds of the array where we can perform a binary search.

An efficient way to find the proper bounds is to start at the beginning of the array with the bound’s size as ‘1’ and exponentially increase the bound’s size (i.e., double it) until we find the bounds that can have the key.

Consider Example-1 mentioned above:



Once we have searchable bounds we can apply the binary search.

### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B1ZW38kXJB2#code)

class ArrayReader {

  int[] arr;

  ArrayReader(int[] arr) {

    this.arr = arr;

  }

  public int get(int index) {

    if (index >= arr.length)

      return Integer.MAX\_VALUE;

    return arr[index];

  }

}

class SearchInfiniteSortedArray {

  public static int search(ArrayReader reader, int key) {

    // find the proper bounds first

    int start = 0, end = 1;

    while (reader.get(end) < key) {

      int newStart = end + 1;

      end += (end - start + 1) \* 2; // increase to double the bounds size

      start = newStart;

    }

    return binarySearch(reader, key, start, end);

  }

  private static int binarySearch(ArrayReader reader, int key, int start, int end) {

    while (start <= end) {

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

      if (key < reader.get(mid)) {

        end = mid - 1;

      } else if (key > reader.get(mid)) {

        start = mid + 1;

      } else { // found the key

        return mid;

      }

    }

    return -1;

  }

  public static void main(String[] args) {

    ArrayReader reader = new ArrayReader(new int[] { 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30 });

    System.out.println(SearchInfiniteSortedArray.search(reader, 16));

    System.out.println(SearchInfiniteSortedArray.search(reader, 11));

    reader = new ArrayReader(new int[] { 1, 3, 8, 10, 15 });

    System.out.println(SearchInfiniteSortedArray.search(reader, 15));

    System.out.println(SearchInfiniteSortedArray.search(reader, 200));

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B1ZW38kXJB2#time-complexity)

There are two parts of the algorithm. In the first part, we keep increasing the bound’s size exponentially (double it every time) while searching for the proper bounds. Therefore, this step will take O(logN) assuming that the array will have maximum ‘N’ numbers. In the second step, we perform the binary search which will take O(logN), so the overall time complexity of our algorithm will be O(logN + logN) which is asymptotically equivalent to O(logN).

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/B1ZW38kXJB2#space-complexity)

The algorithm runs in constant space O(1).

# Minimum Difference Element

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/mymvP915LY9#problem-statement)

Given an array of numbers sorted in ascending order, find the element in the array that has the minimum difference with the given ‘key’.

**Example 1:**

Input: [4, 6, 10], key = 7  
Output: 6  
Explanation: The difference between the key '7' and '6' is minimum than any other number in the array

**Example 2:**

Input: [4, 6, 10], key = 4  
Output: 4

**Example 3:**

Input: [1, 3, 8, 10, 15], key = 12  
Output: 10

**Example 4:**

Input: [4, 6, 10], key = 17  
Output: 10

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/mymvP915LY9#solution)

The problem follows the **Binary Search** pattern. Since Binary Search helps us find a number in a sorted array efficiently, we can use a modified version of the Binary Search to find the number that has the minimum difference with the given ‘key’.

We can use a similar approach as discussed in Order-agnostic binary search. We will try to search for the ‘key’ in the given array. If we find the ‘key’ we will return it as the minimum difference number. If we can’t find the ‘key’, (at the end of the loop) we can find the differences between the ‘key’ and the numbers pointed out by indices start and end, as these two numbers will be closest to the ‘key’. The number that gives minimum difference will be our required number.

### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/mymvP915LY9#code)

class MinimumDifference {

  public static int searchMinDiffElement(int[] arr, int key) {

    if (key < arr[0])

      return arr[0];

    if (key > arr[arr.length - 1])

      return arr[arr.length - 1];

    int start = 0, end = arr.length - 1;

    while (start <= end) {

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

      if (key < arr[mid]) {

        end = mid - 1;

      } else if (key > arr[mid]) {

        start = mid + 1;

      } else {

        return arr[mid];

      }

    }

    // at the end of the while loop, 'start == end+1'

    // we are not able to find the element in the given array

    // return the element which is closest to the 'key'

    if ((arr[start] - key) < (key - arr[end]))

      return arr[start];

    return arr[end];

  }

  public static void main(String[] args) {

    System.out.println(MinimumDifference.searchMinDiffElement(new int[] { 4, 6, 10 }, 7));

    System.out.println(MinimumDifference.searchMinDiffElement(new int[] { 4, 6, 10 }, 4));

    System.out.println(MinimumDifference.searchMinDiffElement(new int[] { 1, 3, 8, 10, 15 }, 12));

    System.out.println(MinimumDifference.searchMinDiffElement(new int[] { 4, 6, 10 }, 17));

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/mymvP915LY9#time-complexity)

Since, we are reducing the search range by half at every step, this means the time complexity of our algorithm will be O(logN) where ‘N’ is the total elements in the given array.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/mymvP915LY9#space-complexity)

The algorithm runs in constant space O(1).

# Bitonic Array Maximum

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/RMyRR6wZoYK#problem-statement)

Find the maximum value in a given Bitonic array. An array is considered bitonic if it is monotonically increasing and then monotonically decreasing. Monotonically increasing or decreasing means that for any index i in the array arr[i] != arr[i+1].

**Example 1:**

Input: [1, 3, 8, 12, 4, 2]  
Output: 12  
Explanation: The maximum number in the input bitonic array is '12'.

**Example 2:**

Input: [3, 8, 3, 1]  
Output: 8

**Example 3:**

Input: [1, 3, 8, 12]  
Output: 12

**Example 4:**

Input: [10, 9, 8]  
Output: 10

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/RMyRR6wZoYK#solution)

A bitonic array is a sorted array; the only difference is that its first part is sorted in ascending order and the second part is sorted in descending order. We can use a similar approach as discussed in Order agnostic binary search. Since no two consecutive numbers are same (as the array is monotonically increasing or decreasing), whenever we calculate the middle, we can compare the numbers pointed out by the index middle and middle+1 to find if we are in the ascending or the descending part. So:

1. If arr[middle] > arr[middle + 1], we are in the second (descending) part of the bitonic array. Therefore, our required number could either be pointed out by middle or will be before middle. This means we will be doing: end = middle.
2. If arr[middle] < arr[middle + 1], we are in the first (ascending) part of the bitonic array. Therefore, the required number will be after middle. This means we will be doing: start = middle + 1.

We can break when start == end. Due to the two points mentioned above, both start and end will be pointing at the maximum number of the bitonic array.

### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/RMyRR6wZoYK#code)

class MaxInBitonicArray {

  public static int findMax(int[] arr) {

    int start = 0, end = arr.length - 1;

    while (start < end) {

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

      if (arr[mid] > arr[mid + 1]) {

        end = mid;

      } else {

        start = mid + 1;

      }

    }

    // at the end of the while loop, 'start == end'

    return arr[start];

  }

  public static void main(String[] args) {

    System.out.println(MaxInBitonicArray.findMax(new int[] { 1, 3, 8, 12, 4, 2 }));

    System.out.println(MaxInBitonicArray.findMax(new int[] { 3, 8, 3, 1 }));

    System.out.println(MaxInBitonicArray.findMax(new int[] { 1, 3, 8, 12 }));

    System.out.println(MaxInBitonicArray.findMax(new int[] { 10, 9, 8 }));

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/RMyRR6wZoYK#time-complexity)

Since we are reducing the search range by half at every step, this means that the time complexity of our algorithm will be O(logN) where ‘N’ is the total elements in the given array.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/RMyRR6wZoYK#space-complexity)

The algorithm runs in constant space O(1).

### Search Bitonic Array [**#**](https://www.educative.io/courses/grokking-the-coding-interview/mEGORlpZYJE#search-bitonic-array-medium)

Given a Bitonic array, find if a given ‘key’ is present in it. An array is considered bitonic if it is monotonically increasing and then monotonically decreasing. Monotonically increasing or decreasing means that for any index i in the array arr[i] != arr[i+1].

Write a function to return the index of the ‘key’. If the ‘key’ is not present, return -1.

**Example 1:**

Input: [1, 3, 8, 4, 3], key=4  
Output: 3

**Example 2:**

Input: [3, 8, 3, 1], key=8  
Output: 1

**Example 3:**

Input: [1, 3, 8, 12], key=12  
Output: 3

**Example 4:**

Input: [10, 9, 8], key=10  
Output: 0

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/7n3BlOvqW0r#solution)

The problem follows the **Binary Search** pattern. Since Binary Search helps us efficiently find a number in a sorted array we can use a modified version of the Binary Search to find the ‘key’ in the bitonic array.

Here is how we can search in a bitonic array:

1. First, we can find the index of the maximum value of the bitonic array, similar to Bitonic array maximum. Let’s call the index of the maximum number maxIndex.
2. Now, we can break the array into two sub-arrays:
   * Array from index ‘0’ to maxIndex, sorted in ascending order.
   * Array from index maxIndex+1 to array\_length-1, sorted in descending order.
3. We can then call **Binary Search** separately in these two arrays to search the ‘key’. We can use the same Order-agnostic binary search for searching.

### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/7n3BlOvqW0r#code)

class SearchBitonicArray {

  public static int search(int[] arr, int key) {

    int maxIndex = findMax(arr);

    int keyIndex = binarySearch(arr, key, 0, maxIndex);

    if (keyIndex != -1)

      return keyIndex;

    return binarySearch(arr, key, maxIndex + 1, arr.length - 1);

  }

  // find index of the maximum value in a bitonic array

  public static int findMax(int[] arr) {

    int start = 0, end = arr.length - 1;

    while (start < end) {

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

      if (arr[mid] > arr[mid + 1]) {

        end = mid;

      } else {

        start = mid + 1;

      }

    }

    // at the end of the while loop, 'start == end'

    return start;

  }

  // order-agnostic binary search

  private static int binarySearch(int[] arr, int key, int start, int end) {

    while (start <= end) {

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

      if (key == arr[mid])

        return mid;

      if (arr[start] < arr[end]) { // ascending order

        if (key < arr[mid]) {

          end = mid - 1;

        } else { // key > arr[mid]

          start = mid + 1;

        }

      } else { // descending order

        if (key > arr[mid]) {

          end = mid - 1;

        } else { // key < arr[mid]

          start = mid + 1;

        }

      }

    }

    return -1; // element is not found

  }

  public static void main(String[] args) {

    System.out.println(SearchBitonicArray.search(new int[] { 1, 3, 8, 4, 3 }, 4));

    System.out.println(SearchBitonicArray.search(new int[] { 3, 8, 3, 1 }, 8));

    System.out.println(SearchBitonicArray.search(new int[] { 1, 3, 8, 12 }, 12));

    System.out.println(SearchBitonicArray.search(new int[] { 10, 9, 8 }, 10));

  }

}

#### Time complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/7n3BlOvqW0r#time-complexity)

Since we are reducing the search range by half at every step, this means that the time complexity of our algorithm will be O(logN) where ‘N’ is the total elements in the given array.

#### Space complexity [**#**](https://www.educative.io/courses/grokking-the-coding-interview/7n3BlOvqW0r#space-complexity)

The algorithm runs in constant space O(1).

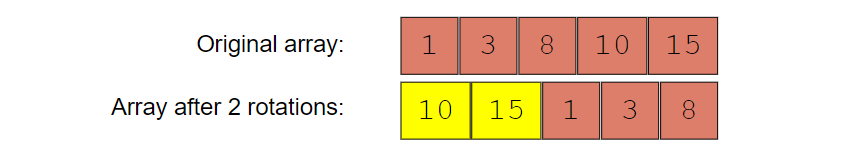
### Search in Rotated Array

Given an array of numbers which is sorted in ascending order and also rotated by some arbitrary number, find if a given ‘key’ is present in it.

Write a function to return the index of the ‘key’ in the rotated array. If the ‘key’ is not present, return -1. You can assume that the given array does not have any duplicates.

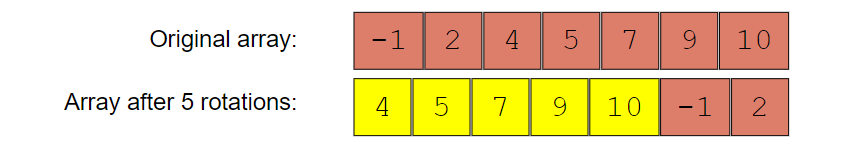
**Example 1:**

Input: [10, 15, 1, 3, 8], key = 15  
Output: 1  
Explanation: '15' is present in the array at index '1'.



**Example 2:**

Input: [4, 5, 7, 9, 10, -1, 2], key = 10  
Output: 4  
Explanation: '10' is present in the array at index '4'.



### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/RMPVM2Y4PW0#solution)

The problem follows the **Binary Search** pattern. We can use a similar approach as discussed in Order-agnostic binary search and modify it similar to Search bitonic array to search for the ‘key’ in the rotated array.

After calculating the middle, we can compare the numbers at indices start and middle. This will give us two options:

1. If arr[start] <= arr[middle], the numbers from start to middle are sorted in ascending order.
2. Else, the numbers from middle+1 to end are sorted in ascending order.

Once we know which part of the array is sorted, it is easy to adjust our ranges. For example, if option-1 is true, we have two choices:

1. By comparing the ‘key’ with the numbers at index start and middle we can easily find out if the ‘key’ lies between indices start and middle; if it does, we can skip the second part => end = middle -1.
2. Else, we can skip the first part => start = middle + 1.

class SearchRotatedArray {

  public static int search(int[] arr, int key) {

    int start = 0, end = arr.length - 1;

    while (start <= end) {

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

      if (arr[mid] == key)

        return mid;

      if (arr[start] <= arr[mid]) { // left side is sorted in ascending order

        if (key >= arr[start] && key < arr[mid]) {

          end = mid - 1;

        } else { //key > arr[mid]

          start = mid + 1;

        }

      } else { // right side is sorted in ascending order

        if (key > arr[mid] && key <= arr[end]) {

          start = mid + 1;

        } else {

          end = mid - 1;

        }

      }

    }

    // we are not able to find the element in the given array

    return -1;

  }

  public static void main(String[] args) {

    System.out.println(SearchRotatedArray.search(new int[] { 10, 15, 1, 3, 8 }, 15));

    System.out.println(SearchRotatedArray.search(new int[] { 4, 5, 7, 9, 10, -1, 2 }, 10));

  }

}

### Similar Problems [**#**](https://www.educative.io/courses/grokking-the-coding-interview/RMPVM2Y4PW0#similar-problems)

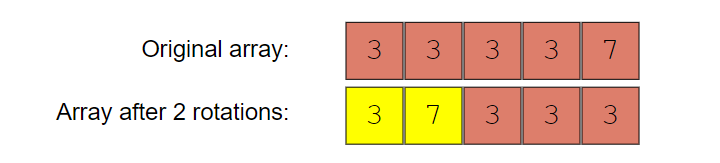
#### Problem 1 [**#**](https://www.educative.io/courses/grokking-the-coding-interview/RMPVM2Y4PW0#problem-1)

How do we search in a sorted and rotated array that also has duplicates?

The code above will fail in the following example!

**Example 1:**

Input: [3, 7, 3, 3, 3], key = 7  
Output: 1  
Explanation: '7' is present in the array at index '1'.



#### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/RMPVM2Y4PW0#solution-2)

The only problematic scenario is when the numbers at indices start, middle, and end are the same, as in this case, we can’t decide which part of the array is sorted. In such a case, the best we can do is to skip one number from both ends: start = start + 1 & end = end - 1.

#### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/RMPVM2Y4PW0#code-2)

class SearchRotatedWithDuplicate {

  public static int search(int[] arr, int key) {

    int start = 0, end = arr.length - 1;

    while (start <= end) {

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

      if (arr[mid] == key)

        return mid;

      // the only difference from the previous solution,

      // if numbers at indexes start, mid, and end are same, we can't choose a side

      // the best we can do, is to skip one number from both ends as key != arr[mid]

      if ((arr[start] == arr[mid]) && (arr[end] == arr[mid])) {

        ++start;

        --end;

      } else if (arr[start] <= arr[mid]) { // left side is sorted in ascending order

        if (key >= arr[start] && key < arr[mid]) {

          end = mid - 1;

        } else { //key > arr[mid]

          start = mid + 1;

        }

      } else { // right side is sorted in ascending order

        if (key > arr[mid] && key <= arr[end]) {

          start = mid + 1;

        } else {

          end = mid - 1;

        }

      }

    }

    // we are not able to find the element in the given array

    return -1;

  }

  public static void main(String[] args) {

    System.out.println(SearchRotatedWithDuplicate.search(new int[] { 3, 7, 3, 3, 3 }, 7));

  }

}

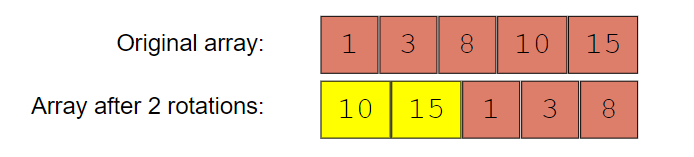
### Rotation Count[**#**](https://www.educative.io/courses/grokking-the-coding-interview/7nPmB8mZ6vj#rotation-count-medium)

Given an array of numbers which is sorted in ascending order and is rotated ‘k’ times around a pivot, find ‘k’.

You can assume that the array does not have any duplicates.

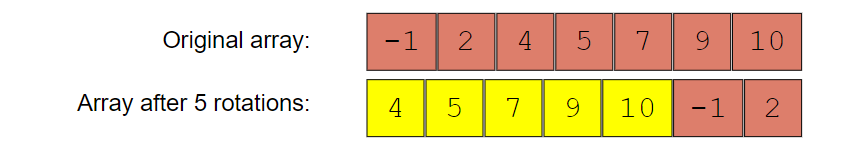
**Example 1:**

Input: [10, 15, 1, 3, 8]  
Output: 2  
Explanation: The array has been rotated 2 times.



**Example 2:**

Input: [4, 5, 7, 9, 10, -1, 2]  
Output: 5  
Explanation: The array has been rotated 5 times.



**Example 3:**

Input: [1, 3, 8, 10]  
Output: 0  
Explanation: The array has been not been rotated.

### Solution [**#**](https://www.educative.io/courses/grokking-the-coding-interview/R1v4P0R7VZw#solution)

This problem follows the **Binary Search** pattern. We can use a similar strategy as discussed in [Search in Rotated Array](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5141325911425024/).

In this problem, actually, we are asked to find the index of the minimum element. The number of times the minimum element is moved to the right will be equal to the number of rotations. An interesting fact about the minimum element is that it is the only element in the given array which is smaller than its previous element. Since the array is sorted in ascending order, all other elements are bigger than their previous element.

After calculating the middle, we can compare the number at index middle with its previous and next number. This will give us two options:

1. If arr[middle] > arr[middle + 1], then the element at middle + 1 is the smallest.
2. If arr[middle - 1] > arr[middle], then the element at middle is the smallest.

To adjust the ranges we can follow the same approach as discussed in Search in rotated array. Comparing the numbers at indices start and middle will give us two options:

1. If arr[start] < arr[middle], the numbers from start to middle are sorted.
2. Else, the numbers from middle + 1 to end are sorted.

### Code [**#**](https://www.educative.io/courses/grokking-the-coding-interview/R1v4P0R7VZw#code)

class RotationCountOfRotatedArray {

  public static int countRotations(int[] arr) {

    int start = 0, end = arr.length - 1;

    while (start < end) {

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

      if (mid < end && arr[mid] > arr[mid + 1]) // if mid is greater than the next element

        return mid + 1;

      if (mid > start && arr[mid - 1] > arr[mid]) // if mid is smaller than the previous element

        return mid;

      if (arr[start] < arr[mid]) { // left side is sorted, so the pivot is on right side

        start = mid + 1;

      } else { // right side is sorted, so the pivot is on the left side

        end = mid - 1;

      }

    }

    return 0; // the array has not been rotated

  }

  public static void main(String[] args) {

    System.out.println(RotationCountOfRotatedArray.countRotations(new int[] { 10, 15, 1, 3, 8 }));

    System.out.println(RotationCountOfRotatedArray.countRotations(new int[] { 4, 5, 7, 9, 10, -1, 2 }));

    System.out.println(RotationCountOfRotatedArray.countRotations(new int[] { 1, 3, 8, 10 }));

  }

}

### Similar Problems [**#**](https://www.educative.io/courses/grokking-the-coding-interview/R1v4P0R7VZw#similar-problems)

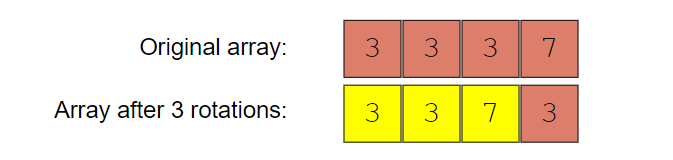
#### Problem 1 [**#**](https://www.educative.io/courses/grokking-the-coding-interview/R1v4P0R7VZw#problem-1)

How do we find the rotation count of a sorted and rotated array that has duplicates too?

The above code will fail on the following example!

**Example 1:**

Input: [3, 3, 7, 3]  
Output: 3  
Explanation: The array has been rotated 3 times



**Solution**  
We can follow the same approach as discussed in search in rotated array. The only difference is that before incrementing start or decrementing end, we will check if either of them is the smallest number.

class RotationCountWithDuplicates {

  public static int countRotations(int[] arr) {

    if (arr == null || arr.length == 0)

      throw new IllegalArgumentException();

    int start = 0, end = arr.length - 1;

    while (start < end) {

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

      if (mid < end && arr[mid] > arr[mid + 1]) // if element at mid is greater than the next element

        return mid + 1;

      if (mid > start && arr[mid - 1] > arr[mid]) // if element at mid is smaller than the previous element

        return mid;

      // this is the only difference from the previous solution

      // if numbers at indices start, mid, and end are same, we can't choose a side

      // the best we can do is to skip one number from both ends if they are not the smallest number

      if (arr[start] == arr[mid] && arr[end] == arr[mid]) {

        if (arr[start] > arr[start + 1]) // if element at start+1 is not the smallest

          return start + 1;

        ++start;

        if (arr[end - 1] > arr[end]) // if the element at end is not the smallest

          return end;

        --end;

        // left side is sorted, so the pivot is on right side

      } else if (arr[start] < arr[mid] || (arr[start] == arr[mid] && arr[mid] > arr[end])) {

        start = mid + 1;

      } else { // right side is sorted, so the pivot is on the left side

        end = mid - 1;

      }

    }

    return 0; // the array has not been rotated

  }

  public static void main(String[] args) {

    System.out.println(RotationCountWithDuplicates.countRotations(new int[] { 3, 3, 7, 3 }));

  }

}