finds the position of a target value within a [sorted array](https://en.wikipedia.org/wiki/Sorted_array);

Binary search runs in at worst [logarithmic time](https://en.wikipedia.org/wiki/Time_complexity#Logarithmic_time), making *O*(log *n*) comparisons;

Binary search takes constant (*O*(1)) space, meaning that the space taken by the algorithm is the same for any number of elements in the array;

Given an array *A* of *n* elements with values or [records](https://en.wikipedia.org/wiki/Record_(computer_science)) *A*0, *A*1, ..., *An*−1, sorted such that *A*0 ≤ *A*1 ≤ ... ≤ *An*−1, and target value *T*, the following [subroutine](https://en.wikipedia.org/wiki/Subroutine) uses binary search to find the index of *T* in *A*.[[7]](https://en.wikipedia.org/wiki/Binary_search_algorithm#cite_note-FOOTNOTEKnuth1998%C2%A76.2.1_(%22Searching_an_ordered_table%22),_subsection_%22Algorithm_B%22-7)

1. Set *L* to 0 and *R* to *n* − 1.
2. If *L* > *R*, the search terminates as unsuccessful.
3. Set *m* (the position of the middle element) to the [floor](https://en.wikipedia.org/wiki/Floor_and_ceiling_functions), or the greatest integer less than (*L* + *R*) / 2.
4. If *Am* < *T*, set *L* to *m* + 1 and go to step 2.
5. If *Am* > *T*, set *R* to *m* − 1 and go to step 2.
6. Now *Am* = *T*, the search is done; return *m*.

In the above procedure, the algorithm checks whether the middle element (*m*) is equal to the target (*T*) in every iteration. Some implementations leave out this check during each iteration. The algorithm would perform this check only when one element is left (when *L* = *R*). The modified subroutine would be as follows:

1. Set *L* to 0 and *R* to *n* − 1.
2. Set *m* (the position of the middle element) to the [ceiling](https://en.wikipedia.org/wiki/Floor_and_ceiling_functions), or the least integer greater than (*L* + *R*) / 2.
3. If *Am* > *T*, set *R* to *m* − 1 and go to step 2.
4. Otherwise, if *Am* ≤ *T*, set *L* to *m* and go to step 2.
5. Now *L* = *R*, the search is done. If *T* = *Am*, return *m*. Otherwise, the search terminates as unsuccessful.

*L* is set to *m* instead of *m* + 1 if *Am* ≤ *T*, as otherwise the algorithm can eliminate the target. This results in a faster comparison loop, as one comparison is eliminated per iteration. However, it requires one more iteration on average.[[8]](https://en.wikipedia.org/wiki/Binary_search_algorithm#cite_note-bottenbruch-8) The above variant procedure will always return the index of the rightmost element if the array contains duplicate elements. For example, if the array to be searched was [1, 2, 3, 4, 4, 5, 6, 7] and the target was 4, then this variant algorithm would return the 5th element (index 4) instead of the 4th element (index 3).[[9]](https://en.wikipedia.org/wiki/Binary_search_algorithm#cite_note-FOOTNOTEKnuth1998%C2%A76.2.1_(%22Searching_an_ordered_table%22),_subsection_%22History_and_bibliography%22-9)

[二分查找法](http://en.wikipedia.org/wiki/Binary_search_algorithm)主要是解决在“一堆数中找出指定的数”这类问题。

而想要应用二分查找法，这“一堆数”必须有一下特征：

* 存储在数组中
* 有序排列

**二分查找法找寻边界值**

之前的都是在数组中找到一个数要与目标相等，如果不存在则返回-1。我们也可以用二分查找法找寻边界值，也就是说在**有序数组**中找到“正好大于（小于）目标数”的那个数。

用数学的表述方式就是：

     在集合中找到一个大于（小于）目标数t的数x，使得集合中的任意数要么大于（小于）等于x，要么小于（大于）等于t

二分查找法的O(log n)让它成为十分高效的算法。不过它的缺陷却也是那么明显的。就在它的限定之上：

**必须有序**，我们很难保证我们的数组都是有序的。当然可以在构建数组的时候进行排序，可是又落到了第二个瓶颈上：它必须是[数组](http://en.wikipedia.org/wiki/Array_data_structure)。

数组读取效率是O(1)，可是它的插入和删除某个元素的效率却是O(n)。因而导致构建有序数组变成低效的事情。

Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky

*You know the difference between in theory and in practice? In theory there’s no difference but in practice there are.*

public static int binSearch(int srcArray[], int start, int end, int key) {

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

if (srcArray[mid] == key) {

return mid;

}

if (start >= end) {

return -1;

} else if (key > srcArray[mid]) {

return binSearch(srcArray, mid + 1, end, key);

} else if (key < srcArray[mid]) {

return binSearch(srcArray, start, mid - 1, key);

}

return -1;

}

// 二分查找普通循环实现

public static int binSearch(int srcArray[], int key) {

int mid = srcArray.length / 2;

if (key == srcArray[mid]) {

return mid;

}

int start = 0;

int end = srcArray.length - 1;

while (start <= end) {

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

if (key < srcArray[mid]) {

end = mid - 1;

} else if (key > srcArray[mid]) {

start = mid + 1;

} else {

return mid;

}

}

return -1;

}

int SearchInRotatedSortedArray(int array[], int low, int high, int target)   
{  
 while(low <= high)  
 {  
 int mid = (low + high) / 2;  
 if (target < array[mid])  
 if (array[mid] < array[high])//the higher part is sorted  
 high = mid - 1; //the target would only be in lower part  
 else //the lower part is sorted  
 if(target < array[low])//the target is less than all elements in low part  
 low = mid + 1;  
 else  
 high = mid - 1;  
  
 else if(array[mid] < target)  
 if (array[low] < array[mid])// the lower part is sorted  
 low = mid + 1; //the target would only be in higher part  
 else //the higher part is sorted  
 if (array[high] < target)//the target is larger than all elements in higher part  
 high = mid - 1;  
 else  
 low = mid + 1;  
 else //if(array[mid] == target)  
 return mid;  
 }  
  
 return -1;  
}