

# Theory: Binary search in Java

 1 hour

0 / 4 problems solved

Skip this topic

Start practicing

**Binary search** is a fast algorithm for finding the position of an element in a **sorted array**. For an array of size  $n$ , the running time of the algorithm is  $O(\log n)$  in the worst case.

The algorithm begins by comparing the middle element of the array with a target value. If the target value matches the middle element, its position in the array is returned. If the target value is less than or greater than the middle element, the search continues in the left or right subarray, respectively, eliminating the other subarray from consideration. It repeats until the value is found or a new search interval is empty.

Let's implement this algorithm in Java using a `while` loop and recursive calls.

## §1. The iterative implementation

Although the main idea of the binary search looks very simple, an implementation requires some caution with array indexes and conditions.

The iterative implementation uses a loop for iterating over the passed array. If the considered interval is empty the loop stops and the method returns `-1` that indicates the element is not found.

```
1 public static int binarySearch(int[] array, int elem, int left, int right) {
2     while (left <= right) {
3
4         int mid = left + (right - left) / 2; // the index of the middle element
5
6         if (elem == array[mid]) {
7             return mid; // the element is found, return its index
8         } else if (elem < array[mid]) {
9             right = mid - 1; // go to the left subarray
10        } else {
11
12            left = mid + 1; // go the the right subarray
13
14        }
15    }
16
17    return -1; // the element is not found
18 }
```

The method takes an array of int's, a target element and two boundaries of the subarray where we search the element. The last two parameters are not mandatory but they are useful if we also want to be able to search not in the entire array.

## §2. Usage examples

Let's test the method passing an array with the same numbers as we have already considered above.

322 users completed this topic.

Latest completion was 1 day ago

.

### Current topic:


[Binary search in Java](#)

...

### Topic depends on

 [Binary search](#)

...

 [Algorithms in Java](#)

...

### Table of contents:

[↑ Theory: Binary search in Java](#)

[§1. The iterative implementation](#)

[§2. Usage examples](#)

[§3. The recursive implementation](#)

[§4. Ways to calculate the index of the middle element](#)

[Feedback & Comments](#)

```

1  int[] array = { 10, 13, 19, 20, 24, 26, 30, 34, 35 };
2
3  int from = 0, to = array.length - 1;
4
5  int indexOf10 = binarySearch(array, 10, from, to); // 0
6  int indexOf19 = binarySearch(array, 19, from, to); // 2
7  int indexOf26 = binarySearch(array, 26, from, to); // 5
8  int indexOf34 = binarySearch(array, 34, from, to); // 7
9  int indexOf35 = binarySearch(array, 35, from, to); // 8
1
0
1
1  int indexOf5 = binarySearch(array, 5, from, to);    // -1
1
2  int indexOf16 = binarySearch(array, 16, from, to); // -1
1
3  int indexOf40 = binarySearch(array, 40, from, to); // -1

```

If we call the method with other borders for the same array, the results will be different.

```

1  int from = 0, to = 2;
2
3  int indexOf10 = binarySearch(array, 10, from, to); // 0
4  int indexOf19 = binarySearch(array, 19, from, to); // 2
5  int indexOf26 = binarySearch(array, 26, from, to); // -1
6  int indexOf34 = binarySearch(array, 34, from, to); // -1
7  int indexOf35 = binarySearch(array, 35, from, to); // -1

```

## §3. The recursive implementation

The recursive implementation makes a recursive call instead of using a loop. It doesn't throw the `StackOverflowError` because it makes not many recursive calls even for large arrays.

```

1
public static int binarySearch(int[] array, int elem, int left, int right) {
2    if (left > right) {
3        return -1; // search interval is empty, the element is not found
4    }
5
6    int mid = left + (right - left) / 2; // the index of the middle element
7
8    if (elem == array[mid]) {
9        return mid; // the element is found, return its index
1
0    } else if (elem < array[mid]) {
1
1        return binarySearch(array, elem, left, mid - 1); // go to the left subar
ray
1
2    } else {
1
3        return binarySearch(array, elem, mid + 1, right); // go the the right su
barray
1
4    }
1
5    }

```

Make sure that the method returns the same results as the previous one.

In fact, iterative and recursive implementations are equivalent. Use any of them for educational purposes. But remember, the binary search is implemented in the Java standard library, see `java.util.Arrays.binarySearch(...)` for details. It works for different data types including integer numbers, characters, strings and so on.

## §4. Ways to calculate the index of the middle element

There are different ways to calculate the middle element:

- The simplest one is sum both borders and divides them by two:

```
1 | int mid = (left + right) / 2;
```

But this simple formula has one disadvantage: It fails for large values of `left` and `right` when the sum is greater than the maximum positive int value. The sum overflows to a negative value, and the index will be negative when divided by two.

- The longer formula protects us from the int overflow. We used it in our binary search implementations.

```
1 | int mid = left + (right - left) / 2;
```

Actually, it's the same as the previous formula but protects us from the int overflow.

```
1 |  
left + (right - left) / 2 = (2 * left + right - left) / 2 = (left + right) / 2
```

- Using a bitshift operator:

```
1 | int mid = (left + right) >>> 1;
```

When we sum `left` and `right` we may get a negative value because of the type overflow, but the unsigned right shift operator processes the value correctly. Also, it may be faster than division.

35 users liked this theory. 2 didn't like it. What about you?



Start practicing

Show discussion (2)