Theory: Binary search in Java

36 minutes 0 / 4 problems solved

Skip this topic Start practicing

750 users solved this topic. Latest completion was about 23 hours ago.

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.

In this lesson, we will learn how this algorithm can be implemented in Java. We will consider two implementations: iterative and recursive.

§1. 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 an array. If the considered interval is empty the loop stops and the method returns `-1` that indicates the element is not found.

```
public static int binarySearch(int[] array, int elem, int left, int right) {
    while (left <= right) {
        int mid = left + (right - left) / 2; // the index of the middle element

        if (elem == array[mid]) {
            return mid; // the element is found, return its index
        } else if (elem < array[mid]) {
            right = mid - 1; // go to the left subarray
        } else {
            left = mid + 1;  // go the the right subarray
        }
    }
    return -1; // the element is not found
}
```

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

The code snippet below illustrates how the method can be used:

Current topic:

Binary search in Java

Topic depends on:

- ✓ Binary search
- ✗ Algorithms in Java
- ✗ Linear search in Java

Table of contents:

```
int[] array = { 10, 13, 19, 20, 24, 26, 30, 34, 35 };

int from = 0, to = array.length - 1;

int indexOf10 = binarySearch(array, 10, from, to); // 0
int indexOf19 = binarySearch(array, 19, from, to); // 2
int indexOf26 = binarySearch(array, 26, from, to); // 5
int indexOf34 = binarySearch(array, 34, from, to); // 7
int indexOf35 = binarySearch(array, 35, from, to); // 8

int indexOf5 = binarySearch(array, 5, from, to);   // -1
int indexOf16 = binarySearch(array, 16, from, to); // -1
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.

```
int from = 0, to = 2;

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

§3. Recursive implementation

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

```
public static int binarySearch(int[] array, int elem, int left, int right) {
    if (left > right) {
        return -1; // search interval is empty, the element is not found
    }

    int mid = left + (right - left) / 2; // the index of the middle element

    if (elem == array[mid]) {
        return mid; // the element is found, return its index
    } else if (elem < array[mid]) {
        return binarySearch(array, elem, left, mid - 1); // go to the left subarray
    } else {
        return binarySearch(array, elem, mid + 1, right); // go the the right subarray
    }
}
```

Try using this method yourself and make sure that it returns the same results as the previous one.

§4. Calculating the index of a middle element

There are different ways to calculate the index of a middle element:

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

```
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.

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

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

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

- Using a bitshift operator:

```
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.

§5. Summary

In this topic, we have learned how binary search can be implemented in Java. We have covered two versions: iterative and recursive. Also, we have discussed different ways to calculate the index of a middle element during the search.

Note that 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.

Report a typo

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

Start practicing

Comments (6) Hints (0) Useful links (0) Show discussion