14 - Search Array

Mark Lesson Complete View Code Prev Next

Algorithms and Data Structures

Suggested Problems

Status	Star	Problem	Difficulty	Video Solution	Code
☐	☆	Binary Search	Easy	▶	C++
☐	☆	Search a 2D Matrix	Medium	▶	C++

Binary Search (Search Array)

Binary search is an efficient way of searching for elements within a sorted array. Typically we are given an array, and an element called the `target` to search for.

At its core, binary search divides the array in the middle, called `mid` and compares the value at `mid` to the `target` value. If the `mid` value is lower than the `target`, it eliminates the left half of the array and searches on the right of `mid`. If `mid` is higher than the `target`, we search to the left. We either find the `target` or determine that the `target` doesn't exist in the array.

In interviews and algorithmic problems, there are two common variations of binary search problems:

1. **Search Array** - a sorted array, and a `target` is given and the task is to determine if the target is found in the array.
2. **Search Range** - a range of numbers is given without a specific `target` .

Mechanics of Binary Search

Now that we know the general idea behind binary search, we can determine how it would work logistically. The `target` value is given as the input but we need to calculate `mid` . `mid` is *initially* calculated by adding the **left-most** index to the **right-most** index and then dividing the result by 2. This allows us to have two equal sections of the array (recall this is exactly what we did with merge sort, except in this case we are not allocating any new space). In the case of binary search, we will have the following:

1. `L` - the left-most index of the current subarray.
2. `R` - the right-most index of the current array.
3. `mid` - `L + R / 2` , the index at which the current sub-array divides itself into two equal halves.

> *`L` and `R` are sometimes referred to as `low` and `high` . It doesn't really matter which one you choose as long as they are understood by the interviewer.*

The idea now is that we will keep searching for the `target` until we either find the `target`, or our `L` pointer crosses the `R` pointer, in which case the `target` doesn't exist.

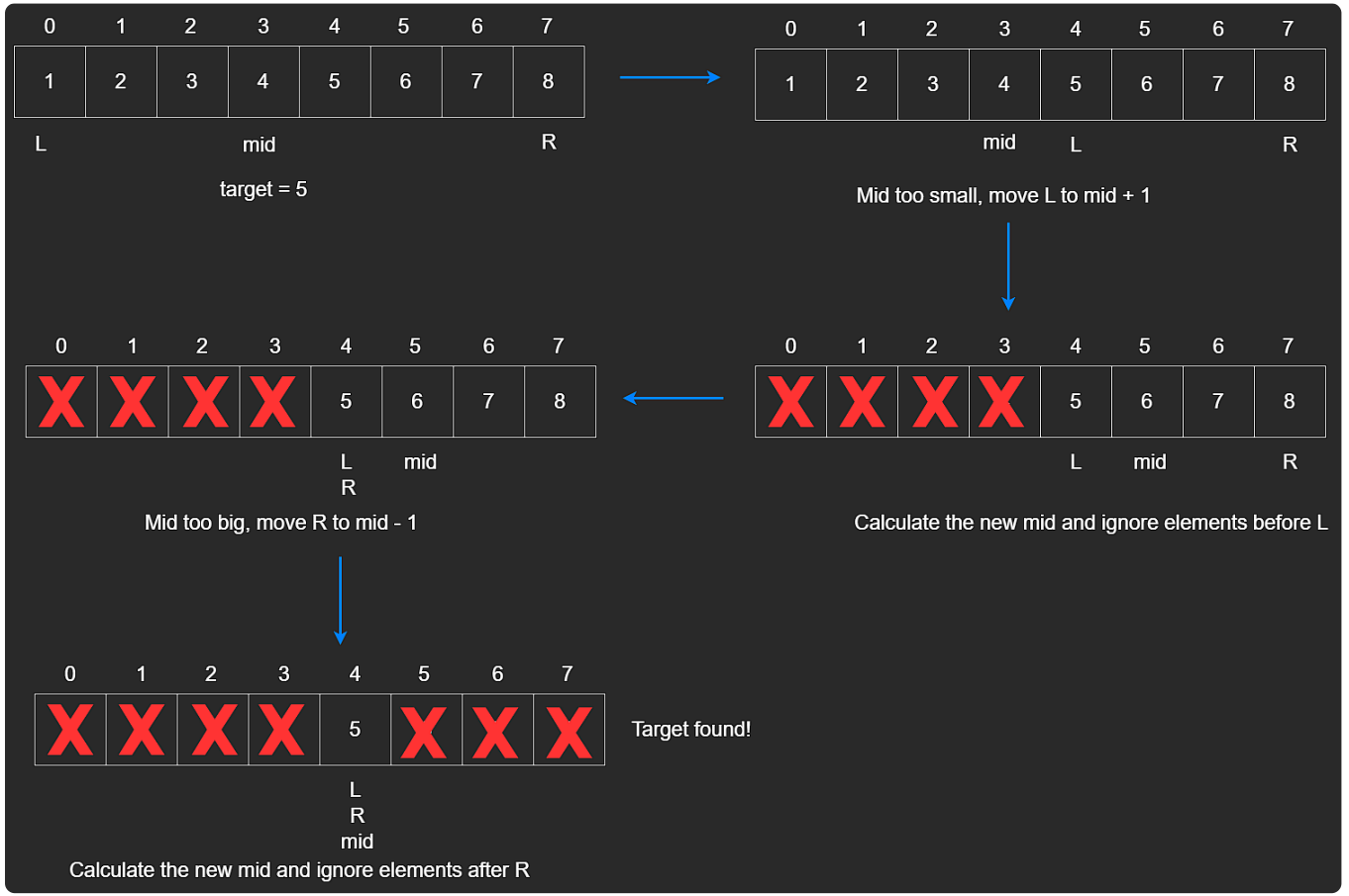
1. Target exists in the array

Let's take the above concept and apply it to an example. Take the array `arr = [1,2,3,4,5,6,7,8]`, and 5 as our `target`.

We know that our `L` will start at the 0th index and `R` will start at 7th index, at `arr.length - 1`. Calculate the `mid` by $(7 + 0) \ / \ 2 = 3$. Now we can compare the value at index 3 to our target element.

4 is less than 5, indicating that we need to look for a larger number, and since the array is sorted, the larger numbers reside on the right. Therefore, we need to move our `L` to `mid + 1`, which determines our lower boundry. The `R` pointer stays where it is.

On the next iteration, calculating `mid` gives us 5. Looking at the 5th index, now our element, 6, is greater than 5. Therefore, our `R` needs to move to `mid - 1` since we need to look for a smaller element. The `L` pointer points to the 4th index and `R` pointer also points to the 4th index. The new `mid` results in 4 and indeed, our target exists at the 4th index, so we can return `mid`.



The pseudocode sums up the process above:

```
arr = [1, 3, 3, 4, 5, 6, 7, 8]

fn binarySearch(arr, target):
    L = 0
    R = arr.length - 1

    while L <= R:
```

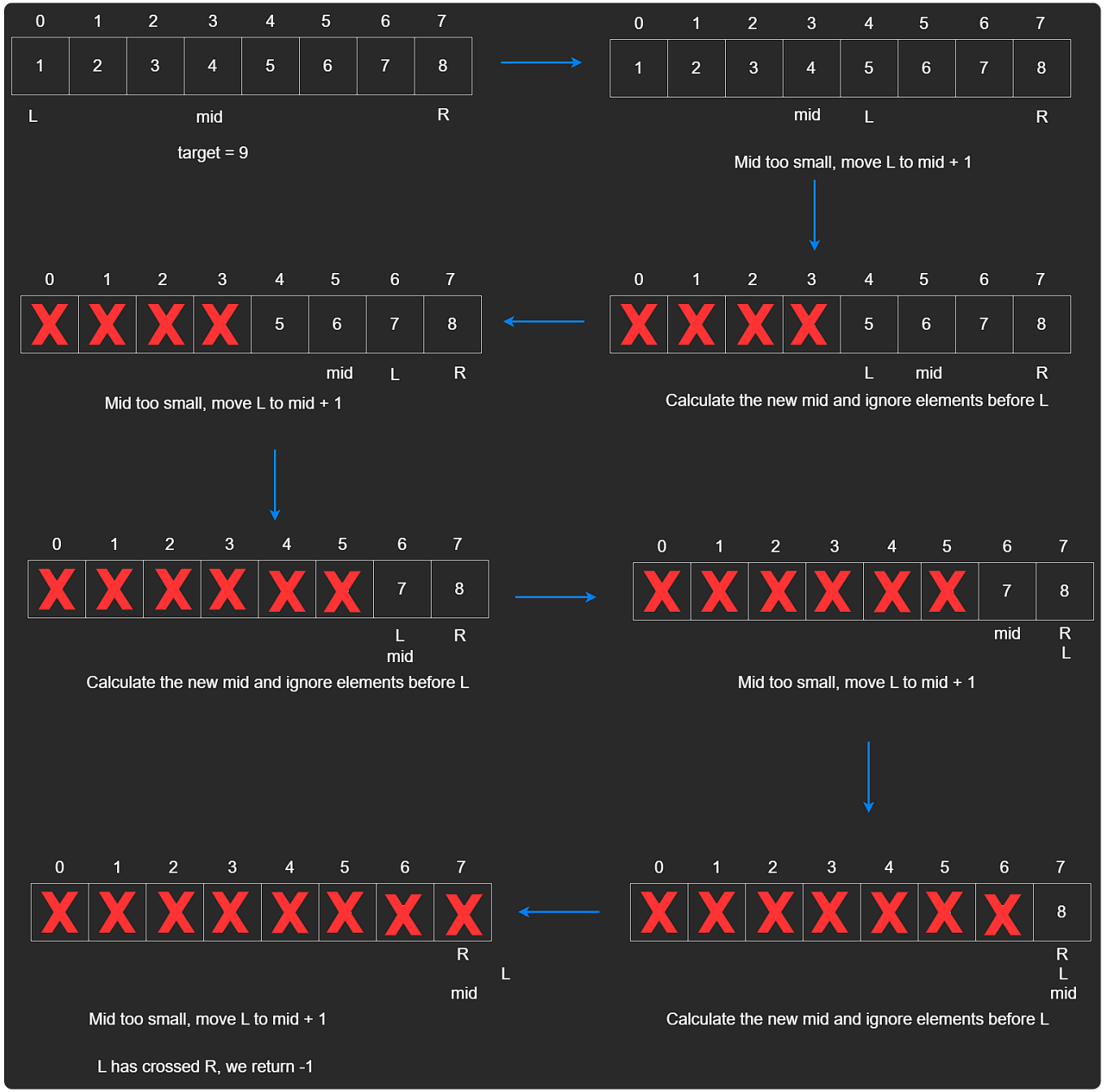
```
        // At each iteration, calculate the mid
        mid = (L + R) / 2
        if target > arr[mid]:
            L = mid + 1
        else if target < arr[mid]:
            R = mid - 1
        else:
            return mid
    return -1
```

> *A better formula for calculating the `mid` value is `L + (R - L)\ /\ 2` . This formula guarentees that our `mid` doesn't exceed the maximum integer value but also making sure that it isn't negative. This article from **Google Research** provides an intuitive explanation.*

2. Target does not exist in the array

Let's dig a little deeper into what happens if our `target` does not exist in the array. Let's take the same array, `arr = [1,2,3,4,5,6,7,8]` and the `target` 9.

Our left pointer would end up out of bounds.


0 1 2 3 4 5 6 7
1 2 3 4 5 6 7 8
L mid R
target = 9
Mid too small, move L to mid + 1
mid L R
Calculate the new mid and ignore elements before L
L mid R
Mid too small, move L to mid + 1
mid L R
Calculate the new mid and ignore elements before L
L mid R
Mid too small, move L to mid + 1
mid R L
L has crossed R, we return -1
R L mid

Time Space Complexity

The work being done is very similar to that of the merge-sort algorithm where we are dividing the array in half until we reach an array of size 1. The time complexity for binary search will be $O(log\ n)$. If you are confused about how we arrived at this answer, refer back to the **Time Complexity section** of merge-sort chapter where we extensively covered how $O(log\ n)$ comes about.

Github Privacy Terms