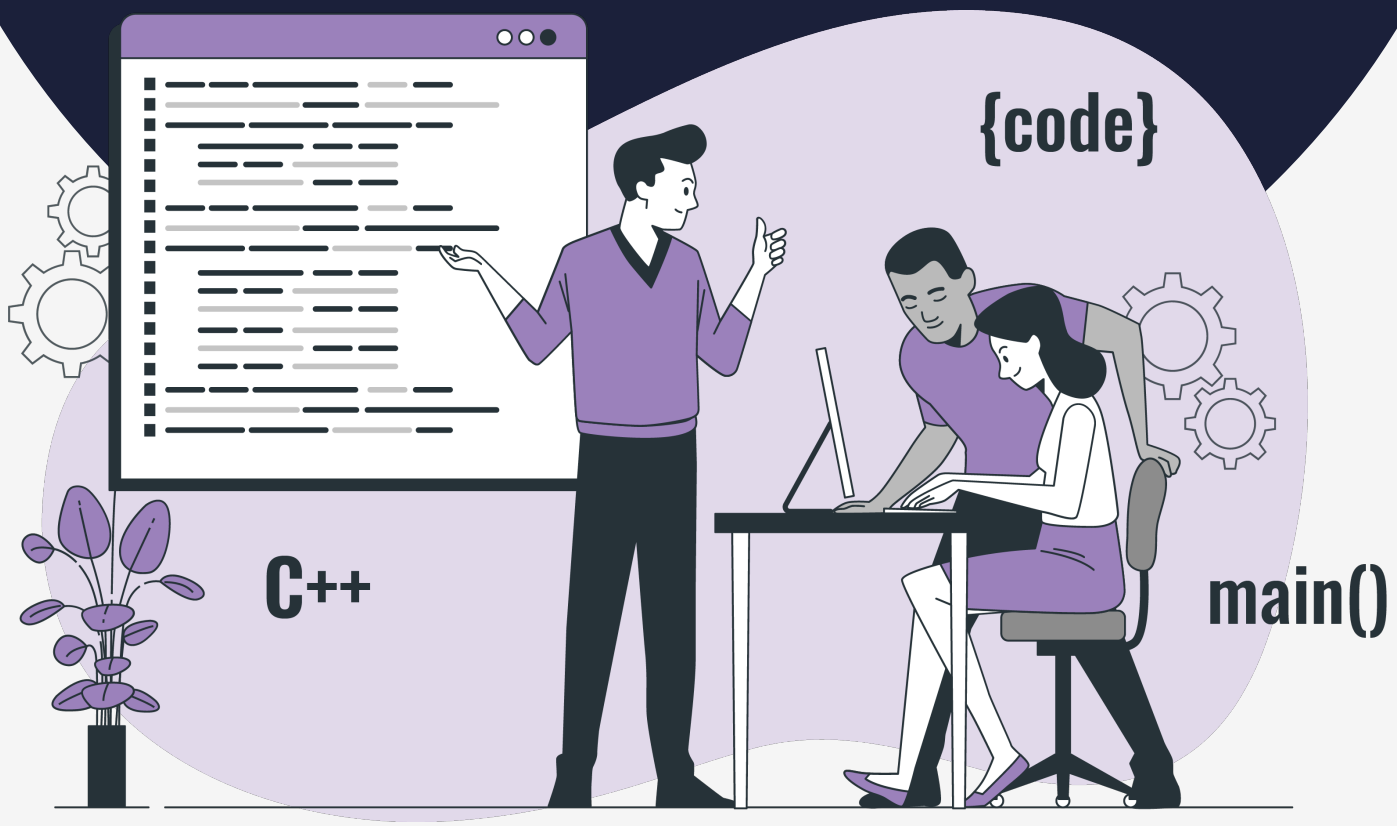
Lesson:

Binary search



Pre-Requisites

- C++ Arrays
- C++ Loops

List of Concepts Involved

- Binary Search Algorithm
- Binary Search Code
- Binary Search Time Complexity Analysis
- Binary Search Space Complexity Analysis
- Problems on Binary Search
- Recursive Binary Search Implementation

Topic 1: Binary Search Algorithm

The algorithm was invented to search an element in a sorted array but can be extended to search in any monotonic space where we can neglect a portion of the search space based on some conditions. The key idea of this algorithm is to reduce the search space at every step into half by comparing the given value to be searched with the middle element of the sorted space of the array. If the middle element is greater than the value we are searching for then the search space becomes the first half of the array and otherwise the search space becomes the second half of the array.

Algorithm Steps:

- Find the middle element of the search space. To begin with, it's the complete sorted array.
- Compare the middle element with the given value needed to be searched.
- If the middle element is equal to the given element then return the index as we have found the value.
- If the middle element is less than the given value, then definitely the value to be searched will lie in the greater half of the array so we move the search interval to the upper half, and to the lower half if the middle element is greater than the given value.
- Keep on repeating the process until we have found the given value or there is no search space left.

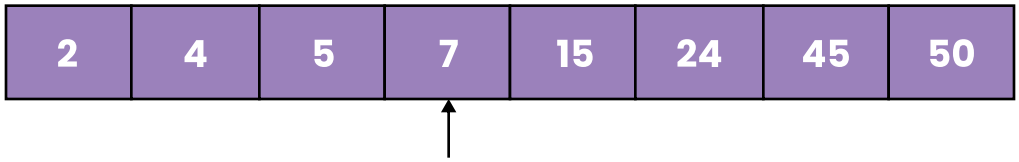
Example - 1: Search if the given value is present in the array or not.
Array = [2, 4, 5, 7, 15, 24, 45, 50], Value = 15

In the above example, we need to search for the value 15 if it is present. A simpler way could be to use linear search directly giving us a time complexity of O(n). But since we are given that the array is sorted, we can use binary search here.

Array-

2	4	5	7	15	24	45	50

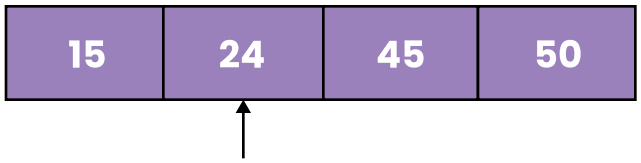
Step 1: For the given array, the middle element is 7.



As we can see 7 is not equal to 15. So we will continue our search.
7 < 15

- our answer lies in the latter half of the array.

Step 2: In the second step, we will consider only the remaining half of the array.



For this half the middle element is 24.
As we can see 24 is not equal to 15. So we will continue our search.
24 > 15

- our answer lies in the first half of the array.

Step 3: We will again consider only the remaining part of the array.

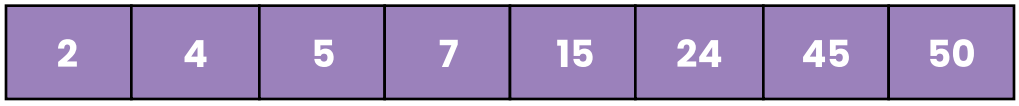
In this case we our middle element will be 15 i.e. the only element in the array.
15 = 15.

- we have found the value that we were looking for.

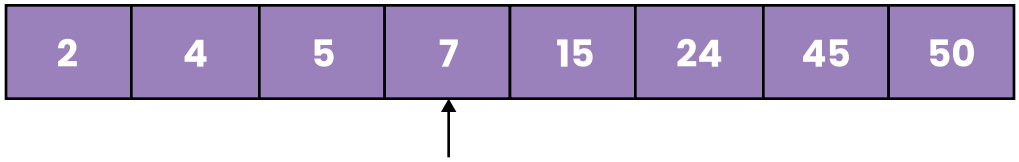
Example - 2: In this let's consider the same array as before but this time we will search for the value 6 in the array.

We can see that 6 doesn't exist in our array. Let's see how our algorithm comes to the same conclusion.

Array-



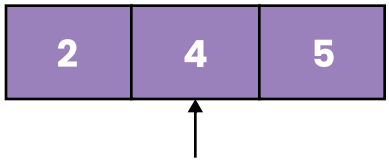
Step 1: For the given array, the middle element is 7.



As we can see 7 is not equal to 6. So we will continue our search.
7 >6

- our answer lies in the first half of the array.

Step 2: In the second step, we will consider only the remaining half of the array.



For this half the middle element is 4.
As we can see, 4 is not equal to 6. So we will continue our search.
4 < 6

- our answer lies in the second half of the array.

Step 3: We will again consider only the remaining part of the array.

In this case we our middle element will be 5 i.e. the only element in the array.
As we can see, 5 is not equal to 6. So we will continue our search.
5 < 6

- our answer lies in the second half of the array.
- but here we can see that there is no second half of the array left to explore.
- 6 doesn't exist in the array.

We've now understood how the algorithm works. So let's see its code too.

Code link - https://pastebin.com/4FgVU8ns

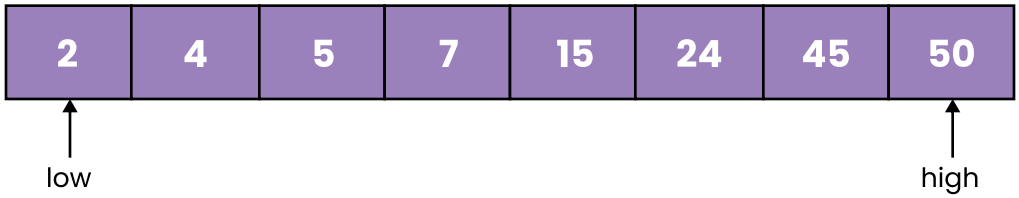
Important Note:

- This algorithm only works if the given array is sorted.
- In some cases, finding the middle element using the formula `(low + high) / 2`, may lead to integer overflow i.e. when the value of low and high is close to `INT_MAX`.To avoid this we instead use- `low + (high - low) / 2.`

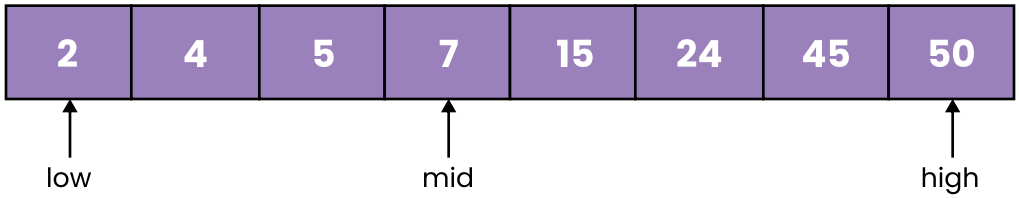
Let's dry run this code on the same examples as taken before.

Example - 1: Array = [2, 4, 5, 7, 15, 24, 45, 50], Value = 15

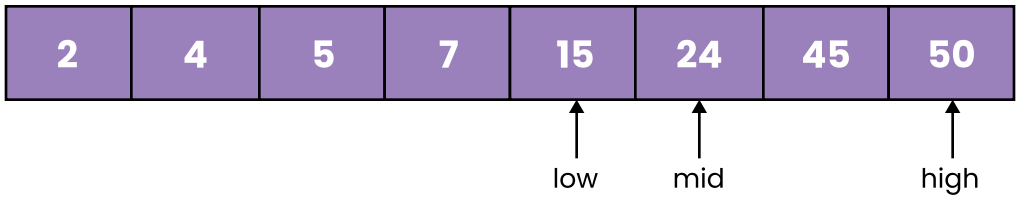
Initially, low = 0 and high = 7. Here low and high represent the min and max index present in the array.



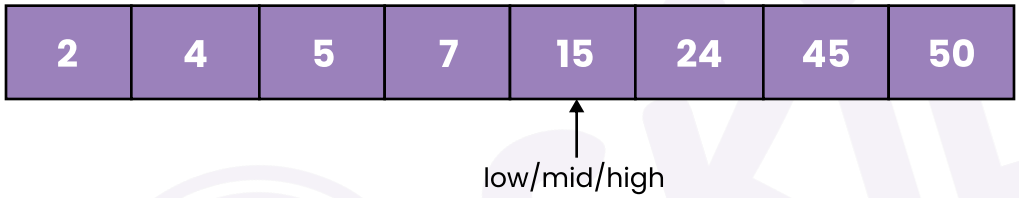
Step 1: Find the middle element.
`mid = low + (high - low) / 2 = 0 + (7 - 0) / 2 = 3`
Value of mid element, arr[mid] = 7.
Here, comparing it with the value to be searched i.e. 15, we get, 15 > 7, so we need to search in the upper half of the sorted array. Move the low pointer to low = mid+1 i.e., the index of the low pointer will be 4 now.



Step 2: Finding the middle element again, middle element index = `low + (high - low) / 2 = 4 + (7-4)/2 = 5`, hence the value at index 5 is 24, comparing it with value to be searched i.e. 15, we get 24 > 15, so this time we move high pointer to mid - 1, high pointer becomes 4.

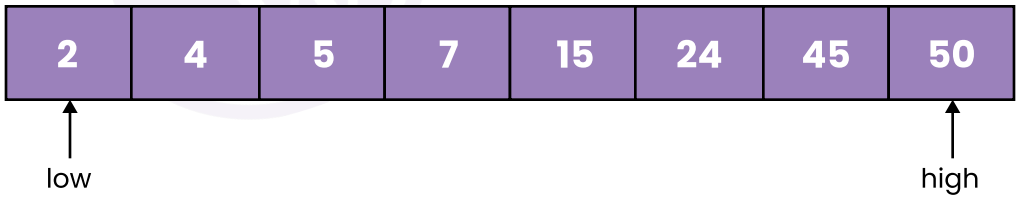


Step 3: Find the middle element index , `low + (high - low) / 2 = 4+(4-4)/2 = 4`, hence here the middle element is 15, so we compare it with the element to be searched 15 = 15. We have found the required value and now we will simply return true.

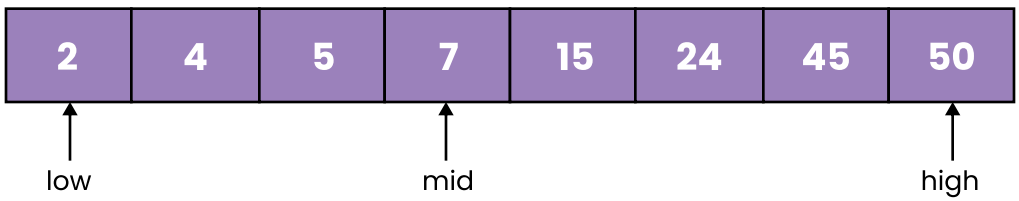


Example - 2: Array = [2, 4, 5, 7, 15, 24, 45, 50], Value = 6

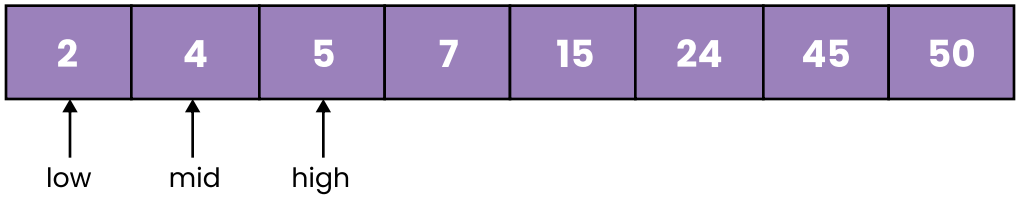
Initially, low = 0 and high = 7. Here low and high represent the min and max index present in the array.



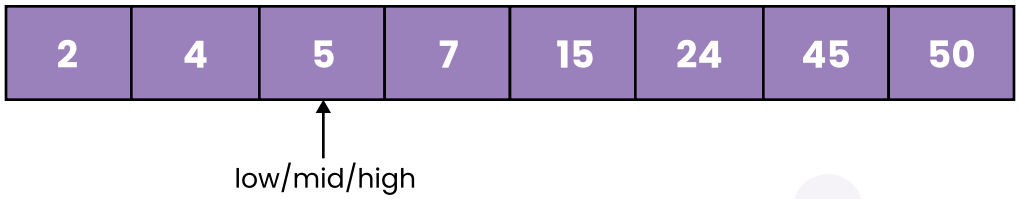
Step 1: Find the middle element.
`mid = (low + high) / 2 = (0 + 7) / 2 = 3`
Value of mid element, arr[mid] = 7.
Here, comparing it with the value to be searched i.e. 45, we get, 7 > 6, so we need to search in the lower half of the sorted array. Move the low pointer to high = mid - 1 i.e. the index of the high pointer will be 2 now.



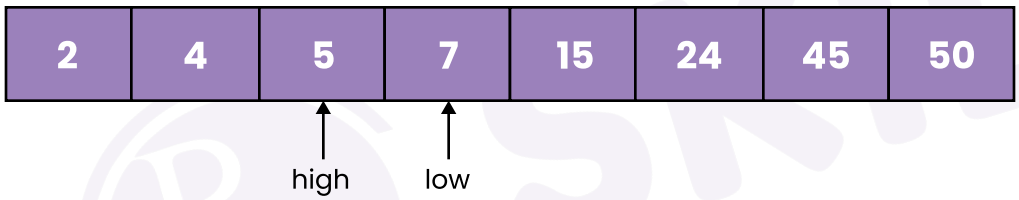
Step 2: Finding the middle element again, middle element index = `(low + high)/2 = (0+2)/2 = (2)/2 = 1`, hence the value at index 1 is 4, comparing it with value to be searched i.e. 6, we get 4 < 6, so we move the low pointer to mid + 1, low becomes 2.



Step 3: Find the middle element index, `(low + high)/ 2 = (1+1)/2 = 2/2 = 1`, hence here the middle element is 5, so we compare it with the element to be searched 5 < 6. so we move the low pointer to mid + 1, low becomes 2.



Step 4: here we can see that the value of low has become greater than the value of high. So we will exit our loop and conclude that 6 is not present in our array. Thus return false.



Topic 2: Time Complexity Analysis

At every step we reduce the interval ranging from low to high to half its size. Initial size of the search space is N where N represents the length of the given array and every step, that search space is reduced by half.

`N → N/2 → N/4 → … → N/(2^k)`

Here k represents the total number of iterations.

Hence the value of k will be logarithmic base 2 of N, giving us a time complexity of **`O(logN)`**.

`O(logN)` is a better time complexity than **`O(n)`** i.e. the time complexity for linear search. Hence we can see why it is preferred over linear search.

At this point many may ask- why are we dividing our array only into 2 parts and not more? Won't it be more efficient?

Let's take the example of ternary search for this. In ternary search we divide our array into 3 parts and discard 1/3 or ⅔ of it based on comparisons of the 2 middle points with respect to the required answer. At first glance it may look like ternary search is faster as it will have a time complexity of $\mathbf{O(\log_3 n)}$, that is better than the time complexity of binary search, $\mathbf{O(\log_2 n)}$. However this is not the case. In case of binary search we have only 2 comparisons being made.

- $T1(n) = 2 * \log_2 n + c = \log_{1.414} n + c$

But in case of ternary search we have 4 comparisons to make.

- $T2(n) = 4 * \log_3 n + c = \log_{1.316} n + c$
- $T2(n) > T1(n)$

Thus the time complexity of ternary search is more than that of binary search.
In conclusion, with an increase in the number of parts, the number of comparisons also increases. This leads to a higher time complexity.

Topic 3: Space Complexity Analysis

In the iterative version of the binary search, we are generating only 3 variables- low, high and mid. Therefore the space complexity of binary search is O(1).

Topic 4: Problems on Binary Search

Problem 1: Find the index of the first occurrence of a given element x in an array. It is given that the array is sorted. If no occurrence of x is found then return -1.

Input
arr = [2, 5, 5, 5, 6, 6, 8, 9, 9, 9]
x = 5

Output
1

Explanation: Given that the array is sorted, so to search the given element we can use Binary Search Algorithm.

The main difference between this question and the examples that we have discussed is that in this question, the given element may have multiple occurrences and we need to find the index of the first occurrence. To tackle this problem, we need to modify only the case where we arrive at the given value. In this case we first consider the current occurrence of the value to be the first occurrence and then reduce the array to its first half. We do this to see if there is any other occurrence of the same value before the current one as then that will be the first occurrence.

Code link - https://pastebin.com/srWyCVMQ

Problem 2: Find the square root of the given non negative value x. Round it off to the nearest floor integer value.

Input: x = 4
Output: 2

Input: x = 11
Output: 3

Explanation: Here we can find the square root of the value x using binary search. Definitely the square root of a non negative number x will lie between [0,x]. Hence we need to search in that sorted range for finding the square root of x. Every single iteration we will find the value of the square of middle in range and compare it to the given value x and change the range accordingly.

Code link - https://pastebin.com/rCF8WHD9

Topic 5: Recursive Binary Search Implementation

Code link - https://pastebin.com/LfGMm7ph

Explanation: The algorithm used in the recursive function is the same, just the implementation varies here. At every point, we check if the middle element is equal to the target value to be searched, if it satisfies then just return the middle index. If the target value is not equal to the value at the middle index, then we need to reduce the search space accordingly. Keep on doing this till the high pointer is greater than or equal to low pointer, else target value not found, so return -1.

Upcoming Class Teasers

- Problems on Binary search - 1