Before comparing, let us first understand these algorithms in detail. So, let us begin by learning about binary search.

Binary search

The Binary search is known as one of the most optimal search algorithms. It is used to find an element (say "key") in a sorted list of elements. It utilizes the divide-and-conquer algorithmic pattern to search for the key element. It can be applied to any container as long as the elements are sorted and we know their indices.

Binary search compares the collection's middle element with the key. If the match is found (i.e., middle element = key), it returns the index of that element. If the middle element is greater than the key, it is searched in the sub-portion to the left of the middle element. Otherwise, the algorithm looks for the key in the portion to the right of the middle item. Binary search is applied repeatedly on the sub-portions until the size reduces to zero.

Binary Search Algorithm

The algorithm for binary search is as follows:

Assume that we have an array named "NUM" of size "n," and we have to find an element "key" in that array.

Step 1: START

Step 2: Set Left =0 and Right = n-1.

Step 3: Find the middle using the formula

Middle = Left + (Right - Left)/2.

Step 4: If arr[Middle] = key,

Then return “Middle” and print “element found.”

Step 5: If arr[Middle] > Key,

Then set Right = Middle-1 and GOTO step 3.

Step 6: If arr[Middle] < Key,

Then set Left = Middle + 1 and GOTO step 3.

Step 7: Repeat steps 3 to 6 till Left <= Right.

Step 8: If Left > Right

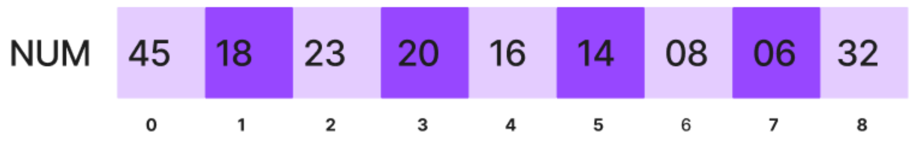
Then Return -1 and print “element not found.”

Step 9: STOP

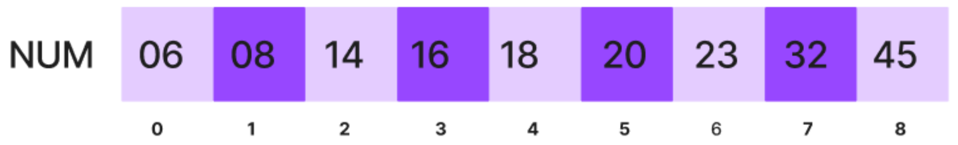
To learn its implementation, you can follow this article.
Now let's see how binary search works. We will consider the working of binary search on an array.

How Binary Search Works

Consider an array named “num” of size “9.” For the binary search to work, the given array must be sorted. If the array is sorted, we can move on to further steps, but if it is not, our first step will be to sort the given array.



Since the array is unsorted, we will sort it first. After sorting, our array looks like this.



Now suppose we have to look for “8” in this array. We will use the binary search for that. Firstly, we will find out the middle element of the array using the following formula:

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

Here, “left” points to the first element of the array (index 0), and “right” points to the last element of the array (index 8).

Let's see what happens when we plug these values into the formula.

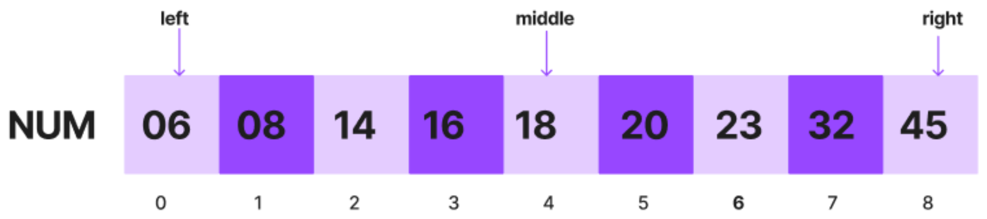
```
middle = 0 + (8-0)/2
middle = 0 + 4
middle = 4
```

Note: *It is important to remember that these calculations are language dependent, and we are doing them according to C++.*

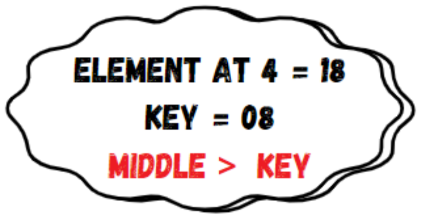
So, we got 4 after putting these values in the given formula. Hence, the middle will now point to the fourth index.

For example, if you get 3.5 as the middle value, use 3 (i.e., the integer value).

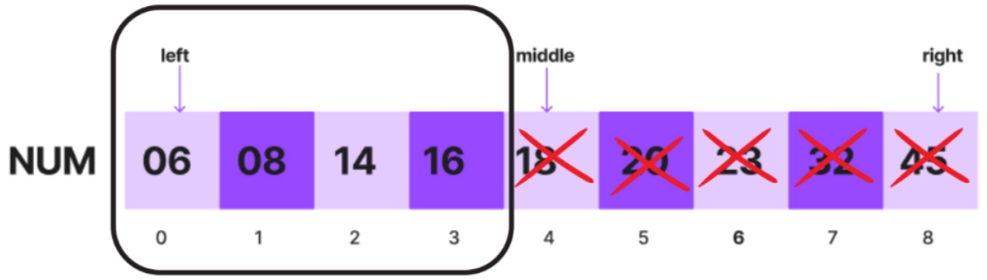
Look at the following image to visualize the positions of the left, right, and middle in our array.



Now we'll see if the value indicated by the middle equals the key.



Since the value of the key is less than the element pointed by the middle, we will look for the key in the left sub-array. The left sub-array will be between the current middle and the current left.



To search this subarray, we will make the following changes to the right and middle.

```
right = middle - 1
middle = left + (right-left) / 2
```

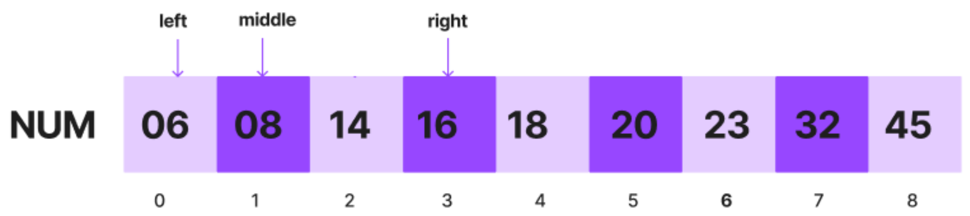
Here, “left” is pointing to the first element of the array (index 0), and “right” is pointing to the current middle -1 (index 3).

Let’s calculate the new value of the middle.

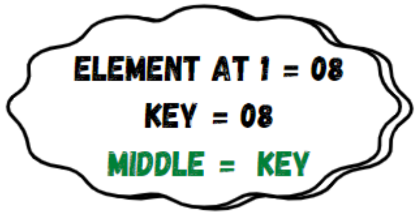
```
middle = 0 + (3 - 0)/2
middle = 0 + 1
middle = 1
```

The new middle will point at index 1.

Now let's look at the new positions of left, right, and middle in the following image.



Now we'll see if the value indicated by the middle equals the key.



Since the element pointed by the middle is equal to the key, our search operation was successful. Hence, the key is found at index 2. We found the key in two iterations.

You must have noticed that the binary search significantly reduces the number of comparisons.

Complexity Analysis of Binary Search

Time Complexity

O($\log_2 n$) is the time complexity. Here, n is the size of the sorted array.

Reason: The search is decreased to half the array in each iteration or recursive call. So there are $\log_2(n)$ calls for n elements in the array.

Space Complexity

O(log n) is the space complexity in recursive implementation, and in iterative implementation, it is **O(1)** Here, n is the size of the array.

Reason: Implementing binary search using Recursion uses extra space in the stack while making the recursive calls. Since we make log(n) calls in recursive implementation, the space complexity becomes O(log n).

In the case of iterative implementation, we do not use any extra space. That is why the space complexity remains O(1).

Read More - Time Complexity of Sorting Algorithms

Let us now learn about the ternary search.

Ternary Search

Ternary search is another search algorithm based on the divide and conquer technique. It is quite similar to the binary search. To search for an element in a container using ternary search, we divide it into three parts. We can use it on any container if the elements are sorted, and we know their indices.

Ternary Search Algorithm

Assume that we have an array named "NUM" of size "n," and we have to find an element "key" in that array. The Algorithm of the ternary search is as follows:

Step 1: Start

Step 2: Initialize left =0, right=n-1, mid1=0, and mid2=0.

Step 3: Find mid1 and mid 2 using the formula:

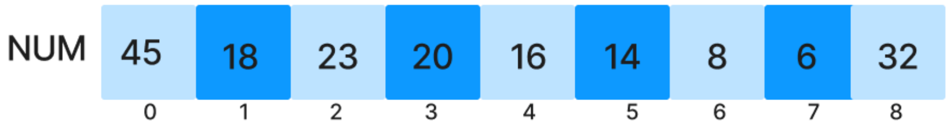
mid1 = left + (right - left) /3

```
         mid2 = right - (right - left) / 3

Step 4: If arr[mid1] = key
        Then return mid1, and print element found.

Step 5: If arr[mid2] = key
        Then return mid2, and print element found.

Step 6: If key < arr[mid1]
        Then set right = mid1-1 and go to step 3.

Step 7: If key > array[mid2]
        Then set left = mid2+1 and go to step 3.

Step 8: If key > arr[mid1]  && key < arr[mid2]
         Then set left = mid1+1, right = mid2-1, and goto step 3.

Step 9: Repeat steps 3 to 8 till Left <= Right.

Step 10: If Left > Right
         Then return -1 and print element not found.

Step 11: STOP
```

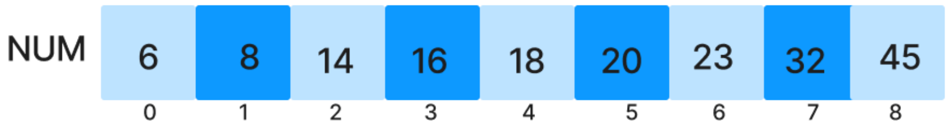
To learn its implementation, you can follow this article.
Let us now try to understand the working of this algorithm, taking the example of searching in an array.

How Ternary Search Works

Consider an array named "NUM" of size "9." For the ternary search to work, we must ensure that the given array is sorted. If the array is sorted, we can move on to further steps, but if it is not, our first step will be to sort the given array. Assume that the following is the array we have to apply ternary search.



Since the given array is unsorted, our first step will be to sort it. After sorting, the array will look like this:



Now, suppose **we have to look for element 32** in this array. For that, we'll have to find two indices that can divide our array into three equal parts. Let's call these points mid1 and mid2.

To calculate the indices mid1 and mid2, we will use the following formula:

```
mid1 = left + (right-left)/3
mid2 = right - (right-left)/3
```

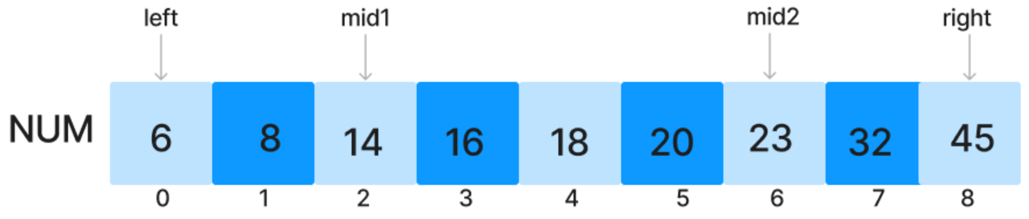
Here, "left" points to the first element of the array (index 0), and "right" is pointing to the last element of the array (index 8).

Let's see what we get after plugging these values into the formula.

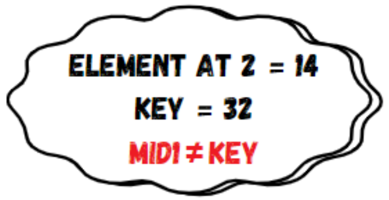
```
mid1 = 0 + (8-0)/3
mid1 = 0 + 2
mid1 = 2

mid2 = 8 - (8-0)/3
mid2 = 8 - 2
mid2 = 6
```

Note: *It is important to remember that these calculations are language dependent, and we are doing them according to C++.*

The **mid1** will point to **index 2**, and the **mid2** will point to **index 6**. Look at the following image to visualize the positions of left, right, mid1, and mid2.



Now, we will compare the key with mid1 first.



The key element was not present at mid1. So we'll now compare the key to mid2.

ELEMENT AT 6 = 23

KEY = 32

MID2≠ KEY

The key element wasn’t present at mid2 either. So now, we’ll check if the key element is less than the element at mid1.

ELEMENT AT 2 = 14

KEY = 32

MID1 < KEY

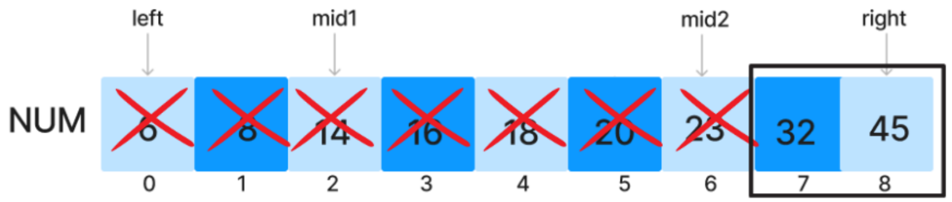
The key element is greater than mid1. Now, we’ll compare the key element with mid2.

ELEMENT AT 6 = 23

KEY = 32

MID2 < KEY

Now that we know that the key element is greater than mid2, we will apply the ternary search on the sub-array to the right of mid2.



To apply ternary search in this sub-array, we will make the following changes to left, mid1, and mid2.

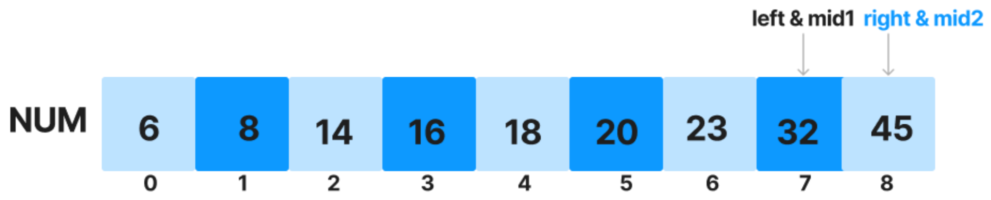
```
left = mid2 + 1
mid1 = left + (right-left)/3
mid2 = right - (right-left)/3
```

Here, “right” is pointing to the last element of the array (index 8), and “left” is pointing to the current mid2 + 1 (index 7).

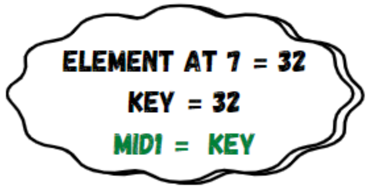
Let’s find out the new values of mid1 and mid2.

```
mid1 = 7 + (8-7)/3
mid1 = 7 + 0
mid1 = 7

mid2 = 8 - (8-7)/3
mid2 = 8 - 0
mid2 = 8
```

The **mid1** will point to **index 7**, and the **mid2** will point to **index 8**. Look at the following image to visualize the positions of left, right, mid1, and mid2.



Now we'll see if the element pointed to by mid1 is the same as the key element.



Since the element pointed by the mid1 is equal to the key, our search operation was successful. Hence, the key is found at index 7.

You must have noticed that even though we got the answer in fewer iterations, the number of comparisons we had to make was much more compared to the binary search. In the ternary search, we need to make a maximum of four comparisons in each iteration.

Check out this array problem - Merge 2 Sorted Arrays

Complexity Analysis of Ternary Search

Time Complexity

O($\log_3$n) is the worst-case time complexity. Here, n is the size of the sorted array.

Reason: The search is decreased to one-third of the array in each iteration or recursive call. So there are $\log_3$(n) calls for n elements in the array.

Space Complexity

O(log n) is the space complexity in recursive implementation. In the iterative implementation, it is **O(1).** Here, n is the size of the array.

Reason: Implementing ternary search using recursion uses extra space in the stack while making the recursive calls. Since we make log(n) calls in recursive implementation, the space complexity becomes O(log n).

In the case of iterative implementation, we do not use any extra space. That is why the space complexity remains O(1).

Now that we understand how these algorithms work, it will be easy to compare them.

Must Read Recursive Binary Search.

Comparison

Since in ternary search, we divide our array into three parts and discard the two-thirds of space at iteration each time, you might think that its time complexity is O($\log_3$n) which is faster as compared to that of binary search, which has a complexity of O($\log_2$n), if the size of the array is n. But you’re in a delusion if you think so.

Ternary search makes four comparisons, whereas, in binary search, we only make a maximum of two comparisons in each iteration.

In binary search, $T1(n) = 2*c\log_2(n) + O(1)$ (c = constant)

In ternary search, $T2(n) = 4*c\log_3(n) + O(1)$ (c=constant)

It implies that ternary search will make more comparisons and thus have more time complexity.

Let us list some of the differences between the binary search and the ternary search.

Comparison Factor	**Binary Search**	**Ternary Search**
Best Case Time Complexity	O(1)	O(1)
Worst Case Time Complexity	$O(\log_2 n)$	$O(\log_3 n)$
Average Case Time Complexity	$O(\log_2 n)$	$O(\log_3 n)$
Space Complexity	$O(\log_2 n)$	$O(\log_3 n)$
Maximum No. of Comparisons in each call	2	4
Pre-requisites	Elements of the array should be sorted.	Elements of the array should be sorted.

Let us now address some frequently asked questions.

Frequently asked questions

What is the time complexity of the ternary search?

The ternary search has a time complexity of O ($\log_3$ n).

Is binary search better than ternary search?

Yes, binary search is better than ternary search in terms of complexity.

Is ternary search useful?

Ternary search is handy when the function can’t be easily differentiated. We can use it to find a function's extremum (minimum and maximum).

Is linear search better than binary search?

With a time complexity of O(log n), binary search is faster than linear. The array must be in sorted order for it to work.

What are the different search algorithms?

Linear, binary, and ternary search are the three most commonly used search algorithms.

Conclusion

In this article, we learned about binary search and ternary search. We also made a comparison between these two search algorithms.