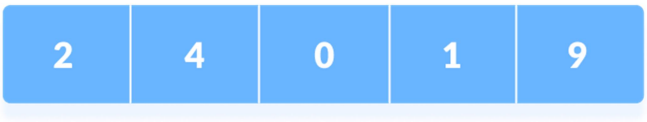
Linear Search

Linear search is the simplest searching algorithm that searches for an element in a list in sequential order. We start at one end and check every element until the desired element is not found.

Input- An array of n-element , K
Ouput- position of K

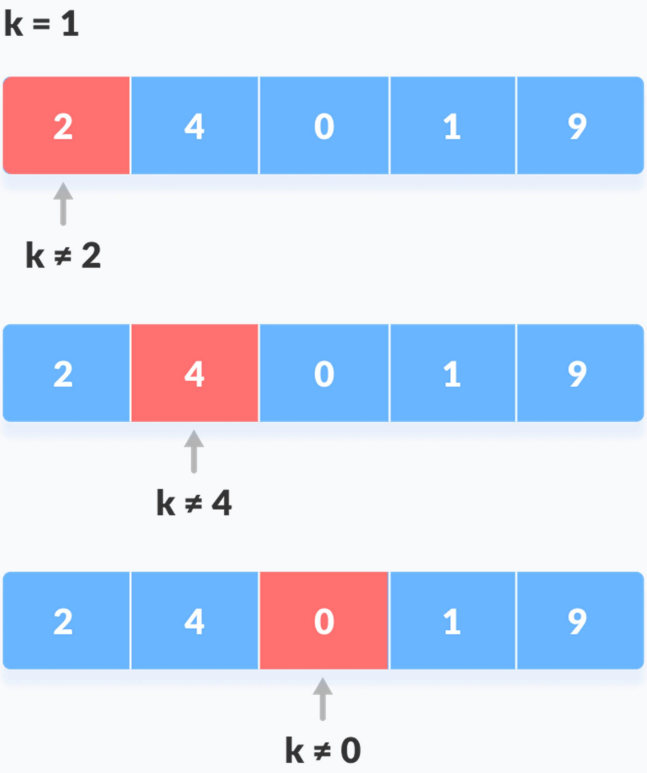
How Linear Search Works?

The following steps are followed to search for an element `k = 1` in the array below.

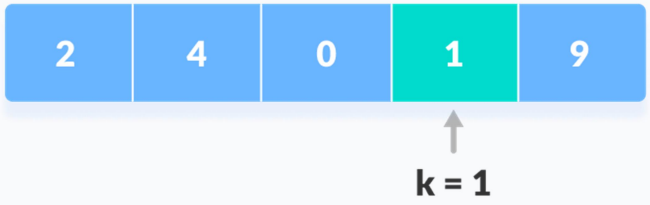


Array to be searched for

1. Start from the first element, compare `k` with each element `x`.



Compare with each element



2. If `x == k`, return the index. Element found
3. Else, return `not found`.

Time Complexity of Linear Search

The **best-case complexity** is **O(1)** if the element is found in the first iteration of the loop.

The **worst-case time complexity is O(n)**, if the search element is found at the end of the array, provided the size of the array is n.

Binary Search

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

Input: Sorted Array of n-element

Output: Position of x

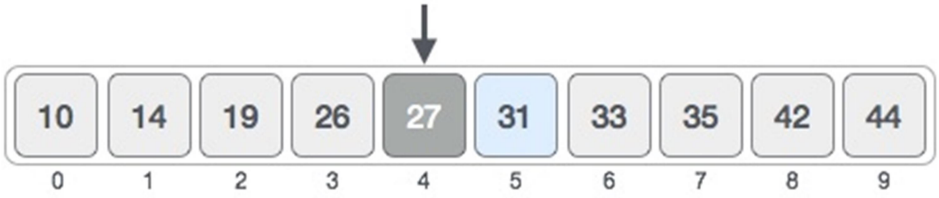
How Binary Search Works?

For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

First, we shall determine half of the array by using this formula −

mid = low + (high - low) / 2

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

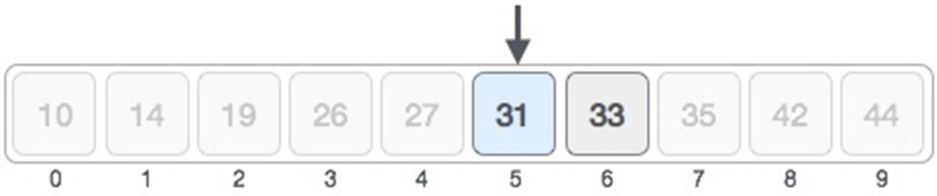
We change our low to mid + 1 and find the new mid value again.

low = mid + 1
mid = low + (high - low) / 2

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Algorithm of Binary Search

```
BinarySearch(a,i,j,x)
{
  if(i==j)
  {
    if(a[i]==x)
      return(i);
    else
     return(-1);
  }
  else
  {
    mid = (i+j)/2 ⟶ O(1)
    if(a[mid]==x) ⟶ 1-comparison O(1)
      return(mid); ⟶ Best case
    else
      if(x<a[mid]) ⟶ 1-comparision = O(1)
        BinarySearch(a,i,mid-1,x) ⟶ T(n/2)
      Else
        BinarySearch(a,mid+1,j,x) ⟶ T(n/2)
  }
}
```

There is else so we can not take 2T(n/2)

Let T(n) be the time complexity of above algorithm

$T(n)=T(n/2) + c \quad T(1) = 1$

$T(n)=O(\log_2 n)$

BASIS FOR COMPARISON	LINEAR SEARCH	BINARY SEARCH
Time Complexity	O(N)	O($\log_2$ N)
Best case time	First Element O(1)	Center Element O(1)
Prerequisite for an array	No required	Array must be in sorted order
Worst case for N number of elements	N comparisons are required	Can conclude after only $\log_2$N comparisons
Can be implemented on	Array and Linked list	Cannot be directly implemented on linked list
Insert operation	Easily inserted at the end of list	Require processing to insert at its proper place to maintain a sorted list.
Algorithm type	Iterative in nature	Divide and conquer in nature
Usefulness	Easy to use and no need for any ordered elements.	Anyhow tricky algorithm and elements should be organized in order.
Lines of Code	Less	More