**Insertion Sort Algorthim**

Algorithm

The simple steps of achieving the insertion sort are listed as follows -

**Step 1 -** If the element is the first element, assume that it is already sorted. Return 1.

**Step2 -** Pick the next element, and store it separately in a **key.**

**Step3 -** Now, compare the **key** with all elements in the sorted array.

**Step 4 -** If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

**Step 5 -** Insert the value.

**Step 6 -** Repeat until the array is sorted.

Working of Insertion sort Algorithm

Now, let's see the working of the insertion sort Algorithm.

To understand the working of the insertion sort algorithm, let's take an unsorted array. It will be easier to understand the insertion sort via an example.

Let the elements of array are -

Insertion Sort Algorithm

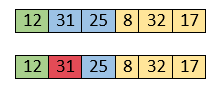
Initially, the first two elements are compared in insertion sort.

Insertion Sort Algorithm

Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.

Insertion Sort Algorithm

Now, move to the next two elements and compare them.

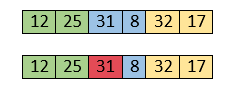


Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.

Insertion Sort Algorithm

Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.



Both 31 and 8 are not sorted. So, swap them.

Insertion Sort Algorithm

After swapping, elements 25 and 8 are unsorted.

Insertion Sort Algorithm

So, swap them.

Insertion Sort Algorithm

Now, elements 12 and 8 are unsorted.

Insertion Sort Algorithm

So, swap them too.

Insertion Sort Algorithm

Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.

Insertion Sort Algorithm

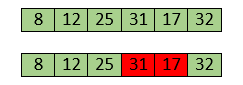
Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

Insertion Sort Algorithm

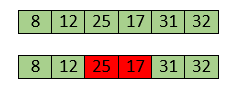
Move to the next elements that are 32 and 17.

Insertion Sort Algorithm

17 is smaller than 32. So, swap them.



Swapping makes 31 and 17 unsorted. So, swap them too.



Now, swapping makes 25 and 17 unsorted. So, perform swapping again.

Insertion Sort Algorithm

Now, the array is completely sorted.

### **Selection sort**

### **Algorithm**

**Step 1** − Set MIN to location 0

**Step 2** − Search the minimum element in the list

**Step 3** − Swap with value at location MIN

**Step 4** − Increment MIN to point to next element

**Step 5** − Repeat until list is sorted

How Selection Sort Works?

Consider the following depicted array as an example.

Unsorted Array

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

Selection Sort

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

Selection Sort

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

Selection Sort

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

Selection Sort

After two iterations, two least values are positioned at the beginning in a sorted manner.

Selection Sort

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −



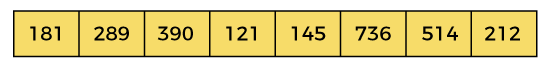
**Radix sort Algorithm**

Algorithm

1. radixSort(arr)
2. max = largest element in the given array
3. d = number of digits in the largest element (or, max)
4. Now, create d buckets of size 0 - 9
5. **for** i -> 0 to d
6. sort the array elements using counting sort (or any stable sort) according to the digits at
7. the ith place

## Working of Radix sort Algorithm

Now let's see the working of radix sort in detail by using an example. To understand it more clearly, let's take an unsorted array and try to sort it using radix sort. It will make the explanation clearer and easier.

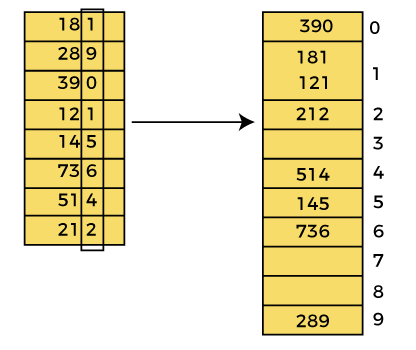


In the given array, the largest element is **736** that have **3** digits in it. So, the loop will run up to three times (i.e., to the **hundreds place**). That means three passes are required to sort the array.

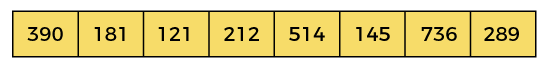
Now, first sort the elements on the basis of unit place digits (i.e., **x = 0**). Here, we are using the counting sort algorithm to sort the elements.

### **Pass 1:**

In the first pass, the list is sorted on the basis of the digits at 0's place.

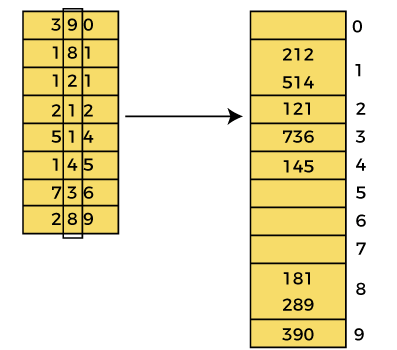


After the first pass, the array elements are -

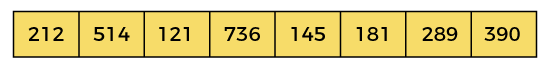


### **Pass 2:**

In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 10th place).

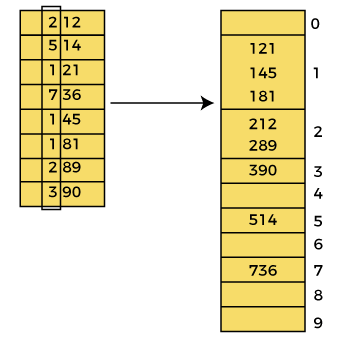


After the second pass, the array elements are -

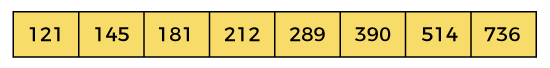


### **Pass 3:**

In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 100th place).



After the third pass, the array elements are -



Now, the array is sorted in ascending order.

**QUICKSORT**

**Algorithm:**

1. QUICKSORT (array A, start, end)
2. {
3. 1 **if** (start < end)
4. 2 {
5. 3 p = partition(A, start, end)
6. 4 QUICKSORT (A, start, p - 1)
7. 5 QUICKSORT (A, p + 1, end)
8. 6 }
9. }

**Partition Algorithm:**

The partition algorithm rearranges the sub-arrays in a place.

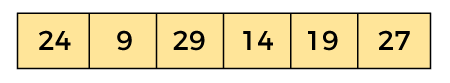
1. PARTITION (array A, start, end)
2. {
3. 1 pivot ? A[end]
4. 2 i ? start-1
5. 3 **for** j ? start to end -1 {
6. 4 **do** **if** (A[j] < pivot) {
7. 5 then i ? i + 1
8. 6 swap A[i] with A[j]
9. 7  }}
10. 8 swap A[i+1] with A[end]
11. 9 **return** i+1
12. }

## Working of Quick Sort Algorithm

Now, let's see the working of the Quicksort Algorithm.

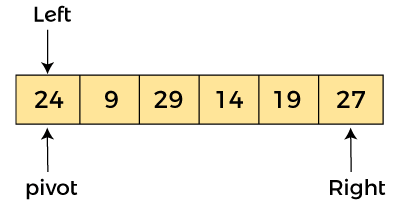
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

Let the elements of array are -

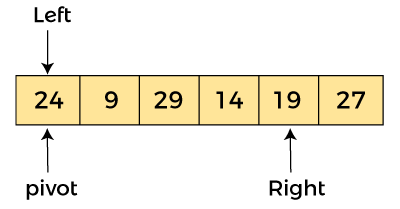


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

Since, pivot is at left, so algorithm starts from right and move towards left.

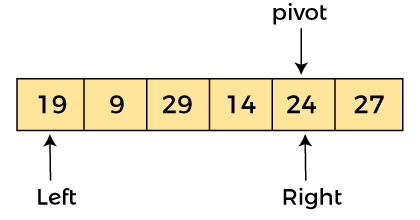


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



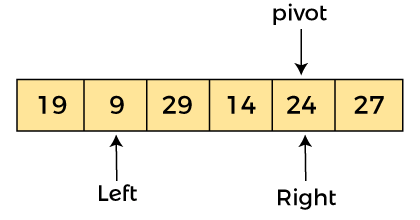
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -

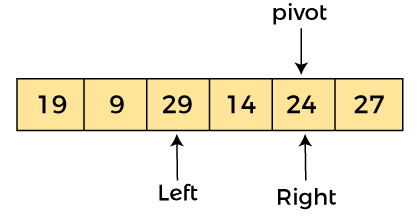


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

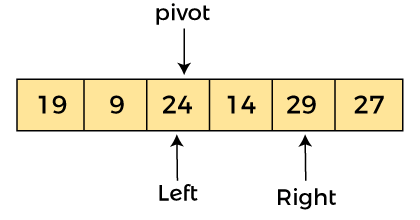
As a[pivot] > a[left], so algorithm moves one position to right as -



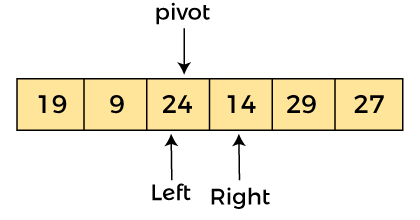
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



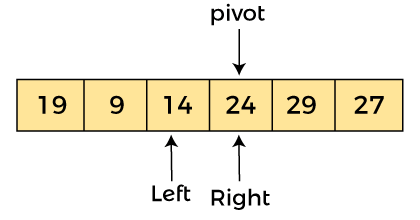
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



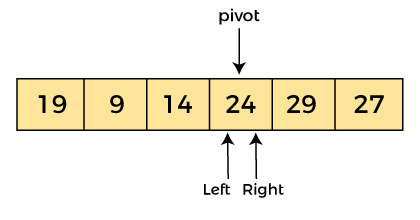
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



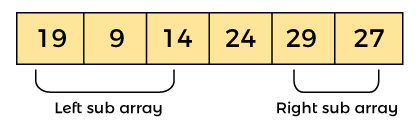
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



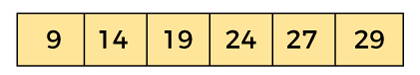
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -



**Heap sort**

Algorithm

1. HeapSort(arr)
2. BuildMaxHeap(arr)
3. for i = length(arr) to 2
4. swap arr[1] with arr[i]
5. heap\_size[arr] = heap\_size[arr] ? 1
6. MaxHeapify(arr,1)
7. End

**BuildMaxHeap(arr)**

BuildMaxHeap(arr)

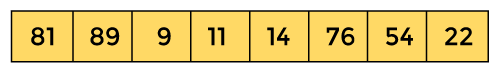
1. heap\_size(arr) = length(arr)
2. for i = length(arr)/2 to 1
3. MaxHeapify(arr,i)
4. End

**MaxHeapify(arr,i)**

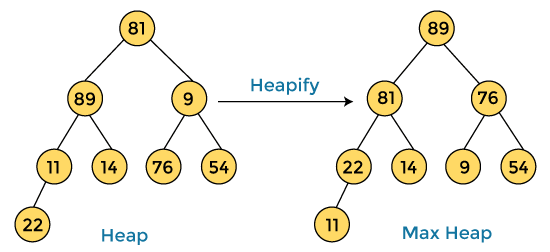
1. MaxHeapify(arr,i)
2. L = left(i)
3. R = right(i)
4. if L ? heap\_size[arr] and arr[L] **>** arr[i]
5. largest = L
6. else
7. largest = i
8. if R ? heap\_size[arr] and arr[R] **>** arr[largest]
9. largest = R
10. if largest != i
11. swap arr[i] with arr[largest]
12. MaxHeapify(arr,largest)
13. End

Working of Heap sort Algorithm

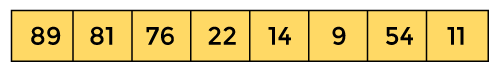
Now let's see the working of heap sort in detail by using an example. To understand it more clearly, let's take an unsorted array and try to sort it using heap sort. It will make the explanation clearer and easier.



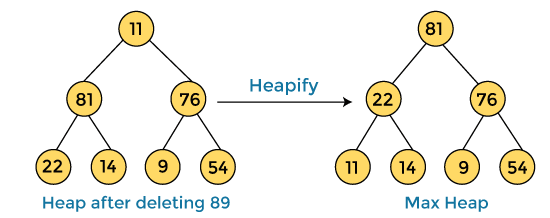
First, we have to construct a heap from the given array and convert it into max heap.



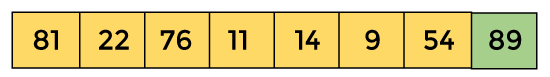
After converting the given heap into max heap, the array elements are -



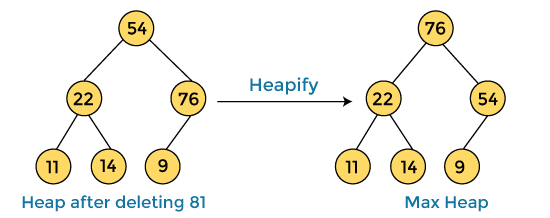
Next, we have to delete the root element **(89)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



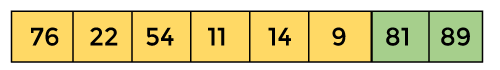
After swapping the array element **89** with **11,** and converting the heap into max-heap, the elements of array are -



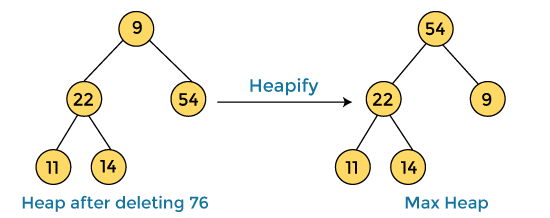
In the next step, again, we have to delete the root element **(81)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(54).** After deleting the root element, we again have to heapify it to convert it into max heap.



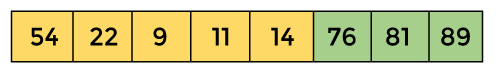
After swapping the array element **81** with **54** and converting the heap into max-heap, the elements of array are -



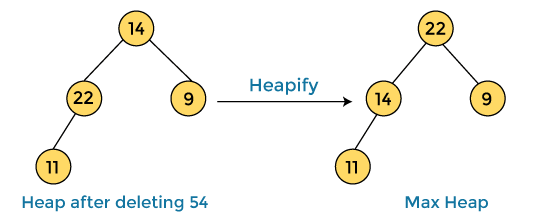
In the next step, we have to delete the root element **(76)** from the max heap again. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.



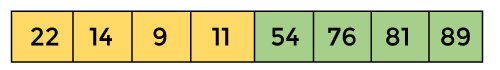
After swapping the array element **76** with **9** and converting the heap into max-heap, the elements of array are -



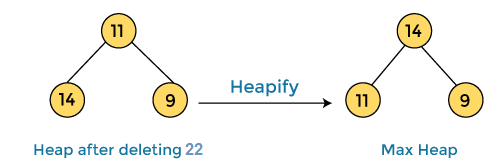
In the next step, again we have to delete the root element **(54)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(14).** After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **54** with **14** and converting the heap into max-heap, the elements of array are -



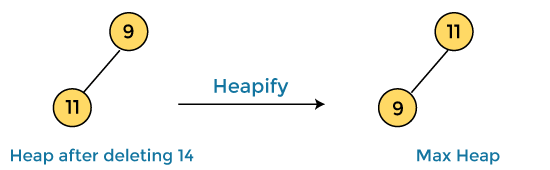
In the next step, again we have to delete the root element **(22)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **22** with **11** and converting the heap into max-heap, the elements of array are -



In the next step, again we have to delete the root element **(14)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.



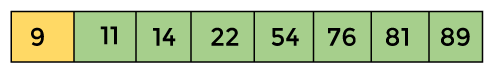
After swapping the array element **14** with **9** and converting the heap into max-heap, the elements of array are -



In the next step, again we have to delete the root element **(11)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.



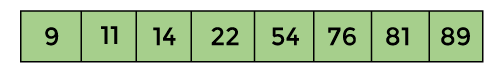
After swapping the array element **11** with **9,** the elements of array are -



Now, heap has only one element left. After deleting it, heap will be empty.



After completion of sorting, the array elements are -



Now, the array is completely sorted.

Merge Sort

Algorithm

In the following algorithm, **arr** is the given array, **beg** is the starting element, and **end** is the last element of the array.

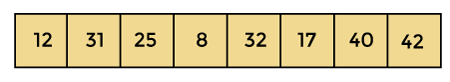
1. MERGE\_SORT(arr, beg, end)
3. **if** beg < end
4. set mid = (beg + end)/2
5. MERGE\_SORT(arr, beg, mid)
6. MERGE\_SORT(arr, mid + 1, end)
7. MERGE (arr, beg, mid, end)
8. end of **if**
10. END MERGE\_SORT

## Working of Merge sort Algorithm

Now, let's see the working of merge sort Algorithm.

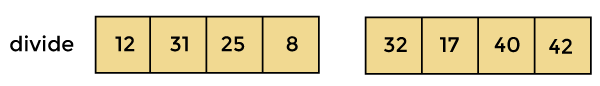
To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.

Let the elements of array are -

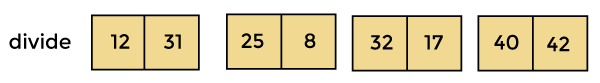


According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

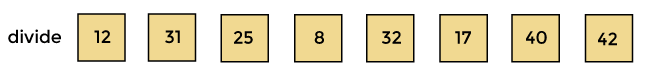
As there are eight elements in the given array, so it is divided into two arrays of size 4.



Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.



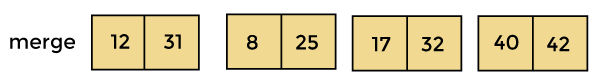
Now, again divide these arrays to get the atomic value that cannot be further divided.



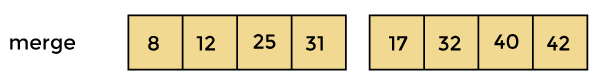
Now, combine them in the same manner they were broken.

In combining, first compare the element of each array and then combine them into another array in sorted order.

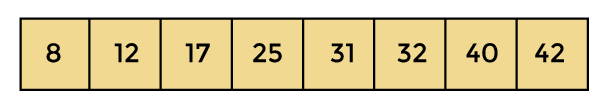
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.



Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -



Now, the array is completely sorted.

# **Linear Search Algorithm**

Linear search is also called as **sequential search algorithm.** It is the simplest searching algorithm. In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.

It is widely used to search an element from the unordered list, i.e., the list in which items are not sorted. The worst-case time complexity of linear search is **O(n).**

### **Algorithm**

Linear\_Search(a, n, val) // 'a' is the given array, 'n' is the size of given array, 'val' is the value to search

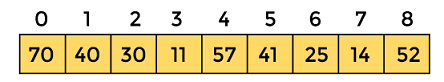
1. Step 1: set pos = -1
2. Step 2: set i = 1
3. Step 3: repeat step 4 while i **<**= n
4. Step 4: if a[i] == val
5. set pos = i
6. print pos
7. go to step 6
8. [end of if]
9. set ii = i + 1
10. [end of loop]
11. Step 5: if pos = -1
12. print "value is not present in the array "
13. [end of if]
14. Step 6: exit

## Working of Linear search

Now, let's see the working of the linear search Algorithm.

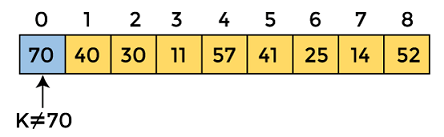
To understand the working of linear search algorithm, let's take an unsorted array. It will be easy to understand the working of linear search with an example.

Let the elements of array are -

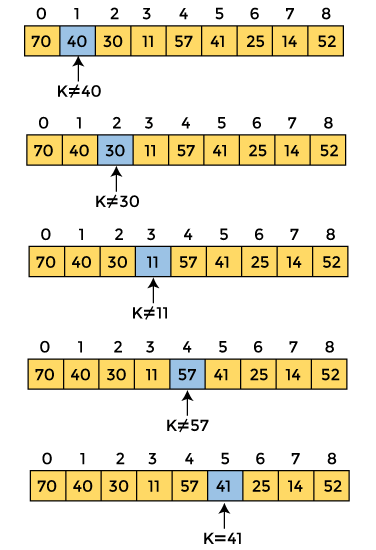


Let the element to be searched is **K = 41**

Now, start from the first element and compare **K** with each element of the array.



The value of **K,** i.e., **41,** is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.



Now, the element to be searched is found. So algorithm will return the index of the element matched.

# **Binary Search Algorithm**

Binary search is the search technique that works efficiently on sorted lists. Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted.

Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match.

Algorithm

1. Binary\_Search(a, lower\_bound, upper\_bound, val) // 'a' is the given array, 'lower\_bound' is the index of the first array element, 'upper\_bound' is the index of the last array element, 'val' is the value to search
2. Step 1: set beg = lower\_bound, end = upper\_bound, pos = - 1
3. Step 2: repeat steps 3 and 4 while beg **<**=end
4. Step 3: set mid = (beg + end)/2
5. Step 4: if a[mid] = val
6. set pos = mid
7. print pos
8. go to step 6
9. else if a[mid] **>** val
10. set end = mid - 1
11. else
12. set beg = mid + 1
13. [end of if]
14. [end of loop]
15. Step 5: if pos = -1
16. print "value is not present in the array"
17. [end of if]
18. Step 6: exit

Working of Binary search

Now, let's see the working of the Binary Search Algorithm.

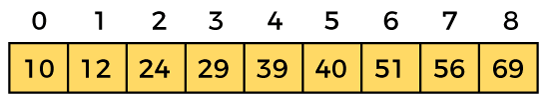
To understand the working of the Binary search algorithm, let's take a sorted array. It will be easy to understand the working of Binary search with an example.

There are two methods to implement the binary search algorithm -

* Iterative method
* Recursive method

The recursive method of binary search follows the divide and conquer approach.

Let the elements of array are -



Let the element to search is, **K = 56**

We have to use the below formula to calculate the **mid** of the array -

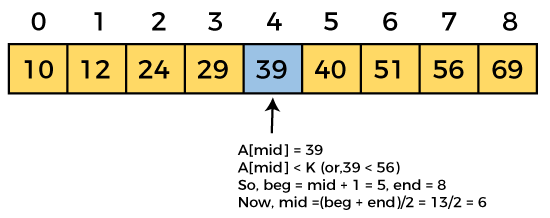
1. mid = (beg + end)/2

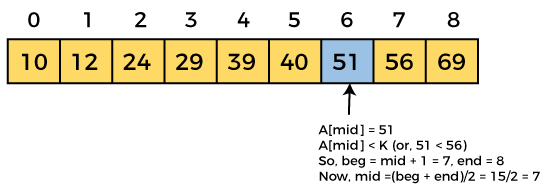
So, in the given array -

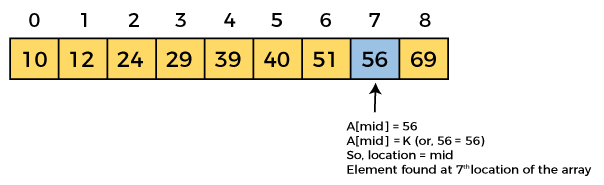
**beg** = 0

**end** = 8

**mid** = (0 + 8)/2 = 4. So, 4 is the mid of the array.







Now, the element to search is found. So algorithm will return the index of the element matched.

**Hashing**

## Introduction

Hashing is an important [data structure](https://www.upgrad.com/blog/data-structure-project-ideas-beginners/) designed to solve the problem of efficiently finding and storing data in an array.

For example, if you have a list of 20000 numbers, and you have given a number to search in that list- you will scan each number in the list until you find a match.

It requires a significant amount of your time to search in the entire list and locate that specific number.

This manual process of scanning is not only time-consuming but inefficient too.

With hashing in the data structure, you can narrow down the search and find the number within seconds

## What is Hashing in Data Structure?

**Hashing in the** [**data structure**](https://www.upgrad.com/blog/graphs-in-data-structure/) is a technique of mapping a large chunk of data into small tables using a hashing function.

It is also known as the message digest function.

It is a technique that uniquely identifies a specific item from a collection of similar items.

**Hash Function**

The hash function in a data structure maps arbitrary size of data to fixed-sized data. It returns the following values:  a small integer value (also known as hash value), hash codes, and hash sums.

**hash = hashfunc(key)**

**index = hash % array\_size**

The has function must satisfy the following requirements:

* A good hash function is easy to compute.
* A good hash function never gets stuck in clustering and distributes keys evenly across the hash table.
* A good hash function avoids collision when two elements or items get assigned to the same hash value.

**Hash Table**

**Hashing in data structure** uses hash tables to store the key-value pairs. The hash table then uses the hash function to generate an index. Hashing uses this unique index to perform insert, update, and search operations.

**Overflow Handling**

An overflow occurs at the time of the home bucket for a new pair (key, element) is full.

We may tackle overflows by

Search the hash table in some systematic manner for a bucket that is not full.

* Linear probing (linear open addressing).
* Quadratic probing.
* Random probing.

Eliminate overflows by allowing each bucket to keep a list of all pairs for which it is the home bucket.

* Array linear list.
* Chain.

Open addressing is performed to ensure that all elements are stored directly into the hash table, thus it attempts to resolve collisions implementing various methods.

Linear Probing is performed to resolve collisions by placing the data into the next open slot in the table.

**Performance of Linear Probing**

* Worst-case find/insert/erase time is θ(m), where m is treated as the number of pairs in the table.
* This occurs when all pairs are in the same cluster.

**Problem of Linear Probing**

* Identifiers are tending to cluster together
* Adjacent clusters are tending to coalesce
* Increase or enhance the search time

**Quadratic Probing**

Linear probing searches buckets (H(x)+i2)%b; H(x) indicates Hash function of x

Quadratic probing implements a quadratic function of i as the increment

Examine buckets H(x), (H(x)+i2)%b, (H(x)-i2)%b, for 1<=i<=(b-1)/2

b is indicated as a prime number of the form 4j+3, j is an integer

**Random Probing**

Random Probing performs incorporating with random numbers.

H(x):= (H’(x) + S[i]) % b

S[i] is a table along with size b-1

S[i] is indicated as a random permutation of integers [1, b-1].

**comparison of binary search and linear search**

|  |  |  |
| --- | --- | --- |
| **Basis of comparison** | **Linear search** | **Binary search** |
| **Definition** | The linear search starts searching from the first element and compares each element with a searched element till the element is not found. | It finds the position of the searched element by finding the middle element of the array. |
| **Sorted data** | In a linear search, the elements don't need to be arranged in sorted order. | The pre-condition for the binary search is that the elements must be arranged in a sorted order. |
| **Implementation** | The linear search can be implemented on any linear data structure such as an array, linked list, etc. | The implementation of binary search is limited as it can be implemented only on those data structures that have two-way traversal. |
| **Approach** | It is based on the sequential approach. | It is based on the divide and conquer approach. |
| **Size** | It is preferrable for the small-sized data sets. | It is preferrable for the large-size data sets. |
| **Efficiency** | It is less efficient in the case of large-size data sets. | It is more efficient in the case of large-size data sets. |
| **Worst-case scenario** | In a linear search, the worst- case scenario for finding the element is O(n). | In a binary search, the worst-case scenario for finding the element is O(log2n). |
| **Best-case scenario** | In a linear search, the best-case scenario for finding the first element in the list is O(1). | In a binary search, the best-case scenario for finding the first element in the list is O(1). |
| **Dimensional array** | It can be implemented on both a single and multidimensional array. | It can be implemented only on a multidimensional array. |