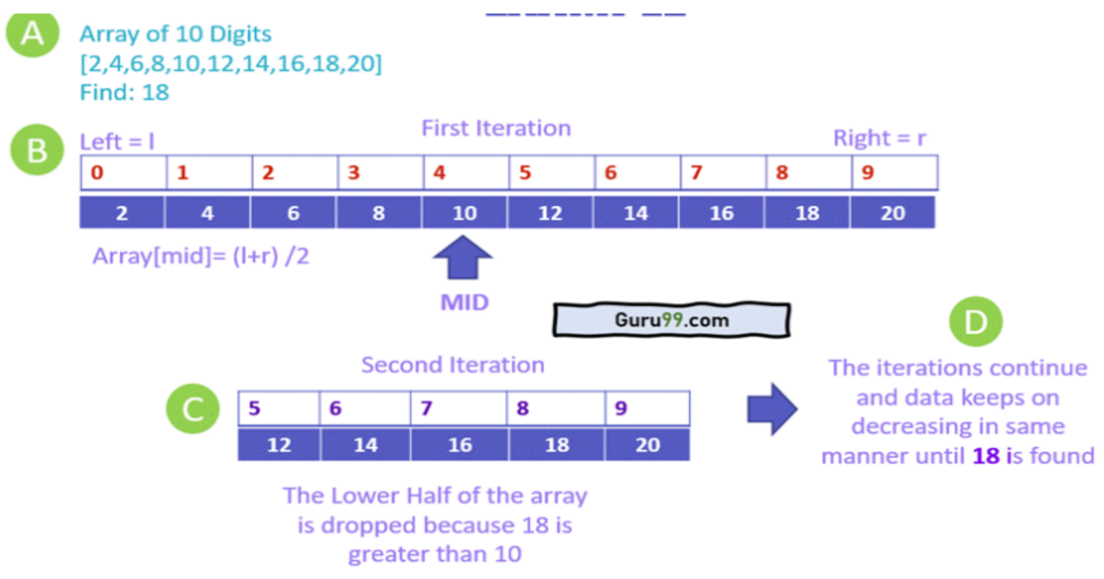
BINARY SEARCH

What is Binary Search?

Search is a utility that enables its user to find documents, files, media, or any other type of data held inside a database. Search works on the simple principle of matching the criteria with the records and displaying them to the user. In this way, the most basic search function works. A binary search is an advanced type of search algorithm that finds and fetches data from a sorted list of items. Its core working principle involves dividing the data in the list into half until the required value is located and displayed to the user in the search result. Binary search is commonly known as a **half-interval search** or a **logarithmic search**. The binary search works in the following manner:

- The search process initiates by locating the middle element of the sorted array of data
- After that, the key value is compared with the element
- If the key value is smaller than the middle element, then searches analyze the upper values of the middle element for comparison and matching
- In case the key value is greater than the middle element then searches analyze the lower values of the middle element for comparison and matching

Example:

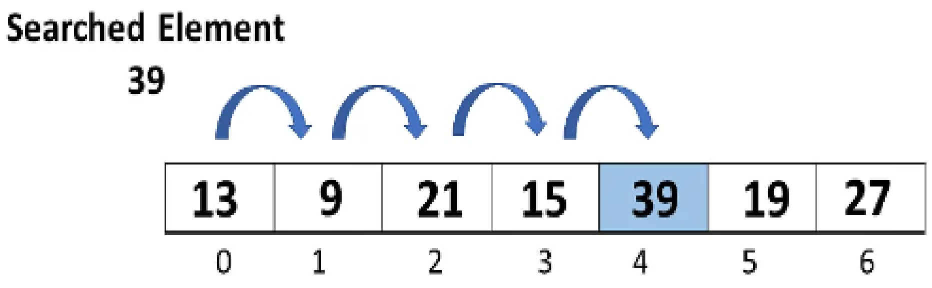


1. You have an array of sorted values ranging from 2 to 20 and need to locate 18.
2. The average of the lower and upper limits is (l + r) / 2 = 4. The value being searched is greater than the mid which is 4.
3. The array values less than the mid are dropped from the search and values greater than the mid-value 4 are searched.
4. This is a recurrent dividing process until the actual item to be searched is found.

***** NOTE *****
Binary search works efficiently on sorted data no matter the size of the data. Instead of performing the search by going through the data in a sequence, the binary algorithm randomly accesses the data to find the required element. This makes the search cycles shorter and more accurate. Binary search performs comparisons of the sorted data based on an ordering principle than using equality comparisons, which are slower and mostly inaccurate. After every cycle of search, the algorithm divides the size of the array into half hence in the next iteration it will work only in the remaining half of the array.

What is Linear Search?

Linear search, often known as sequential search, is the most basic search technique. In this type of search, you go through the entire list and try to fetch a match for a single element. If you find a match, then the address of the matching target element is returned. On the other hand, if the element is not found, then it returns a NULL value. Following is a step-by-step approach employed to perform Linear Search Algorithm.



The procedures for implementing linear search are as follows:

Step 1: First, read the search element (Target element) in the array.

Step 2: In the second step compare the search element with the first element in the array.

Step 3: If both are matched, display "Target element is found" and terminate the Linear Search function.

Step 4: If both are not matched, compare the search element with the next element in the array.

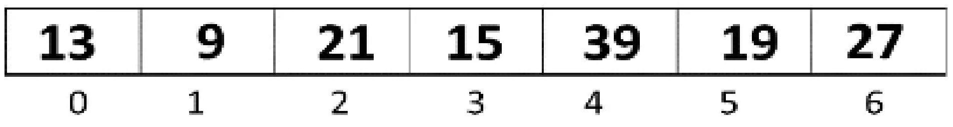
Step 5: In this step, repeat steps 3 and 4 until the search (Target) element is compared with the last element of the array.

Step 6 - If the last element in the list does not match, the Linear Search Function will be terminated, and the message "Element is not found" will be displayed.

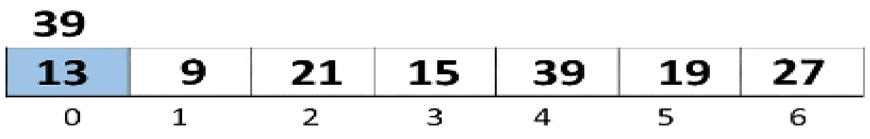
So far, you have explored the fundamental definition and the working terminology of the Linear Search Algorithm.

For Example: Consider an array of size 7 with elements 13, 9, 21, 15, 39, 19, and 27 that starts with 0 and ends with size minus one, 6.

Search element = 39

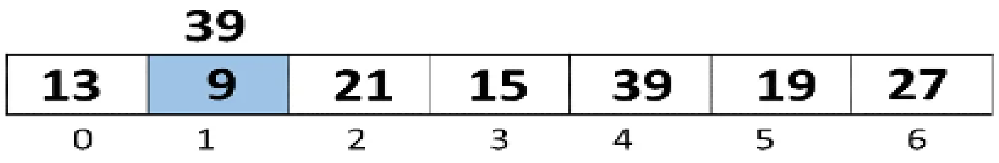


Step 1: The searched element 39 is compared to the first element of an array, which is 13.



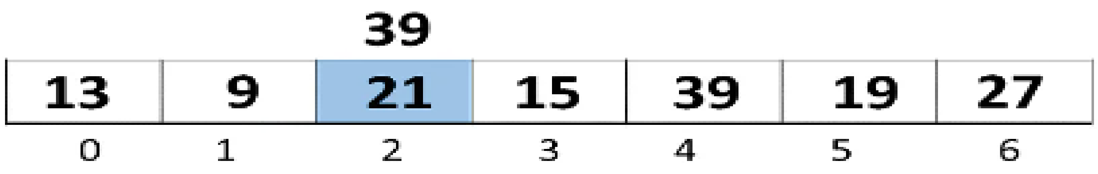
The match is not found, you now move on to the next element and try to implement a comparison.

Step 2: Now, search element 39 is compared to the second element of an array, 9.



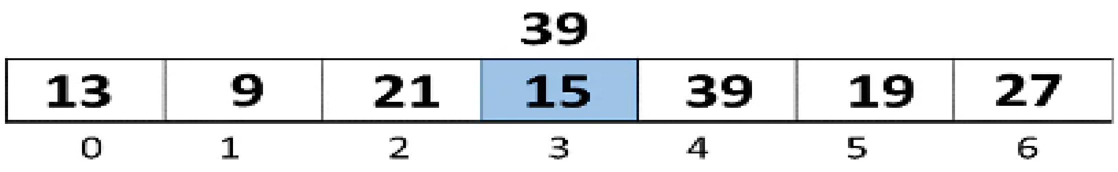
As both are not matching, you will continue the search.

Step 3: Now, search element 39 is compared with the third element, which is 21.\



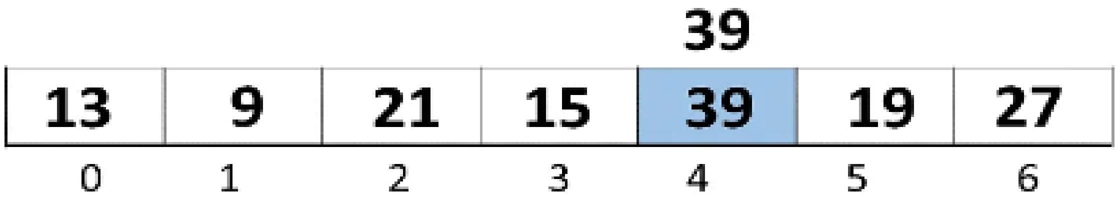
Again, both the elements are not matching, you move on to the next following element.

Step 4; Next, search element 39 is compared with the fourth element, which is 15.



As both are not matching, you move on to the next element.

Step 5: Next, search element 39 is compared with the fifth element 39.



A perfect match is found, you stop comparing any further elements and terminate the Linear Search Algorithm and display the element found at location 4.

What is Quick Sort?

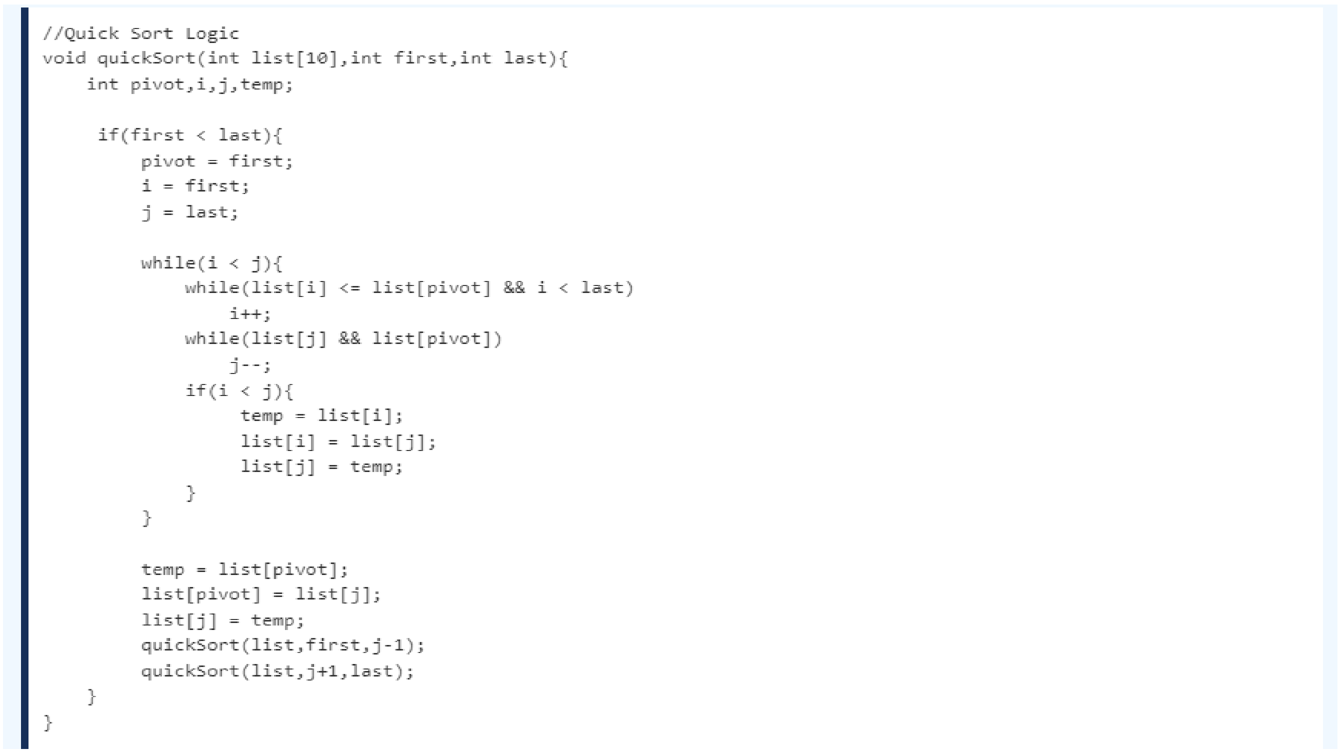
Quick sort is a fast sorting algorithm used to sort a list of elements. The quick sort algorithm is invented by **C. A. R. Hoare**.
The quick sort algorithm attempts to separate the list of elements into two parts and then sort each part recursively. That means it uses a **divide-and-conquer** strategy. In quick sort, the partition of the list is performed based on the element called a ***pivot***. Here pivot element is one of the elements in the list.
The list is divided into two partitions such that **"all elements to the left of the pivot are smaller than the pivot and all elements to the right of the pivot are greater than or equal to the pivot"**.

Step-by-Step Process

In the Quick sort algorithm, partitioning of the list is performed using the following steps...

- Step 1 - Consider the first element of the list as the **pivot** (i.e., the Element at the first position in the list).
- Step 2 - Define two variables i and j. Set i and j to the first and last elements of the list respectively.
- Step 3 - Increment i until list[i] > pivot then stop.
- Step 4 - Decrement j until list[j] < pivot then stop.
- Step 5 - If i < j then exchange list[i] and list[j].
- Step 6 - Repeat steps 3,4 & 5 until i > j.
- Step 7 - Exchange the pivot element with the list[j] element.

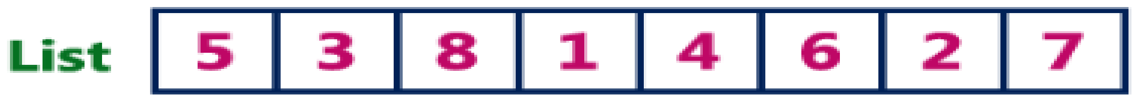
Following is the sample code for Quick sort…



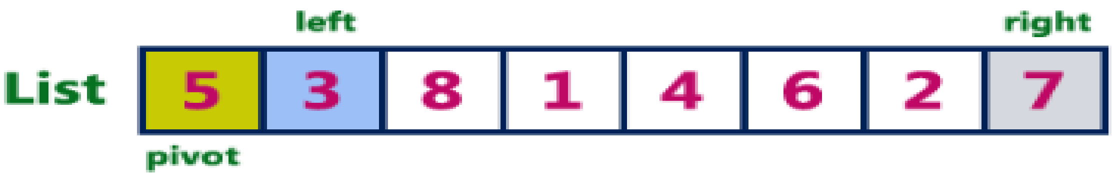
```
//Quick Sort Logic
void quickSort(int list[10],int first,int last){
    int pivot,i,j,temp;

     if(first < last){
         pivot = first;
         i = first;
         j = last;

         while(i < j){
             while(list[i] <= list[pivot] && i < last)
                 i++;
             while(list[j] && list[pivot])
                 j--;
             if(i < j){
                  temp = list[i];
                  list[i] = list[j];
                  list[j] = temp;
             }
         }

         temp = list[pivot];
         list[pivot] = list[j];
         list[j] = temp;
         quickSort(list,first,j-1);
         quickSort(list,j+1,last);
    }
}
```

For Example:

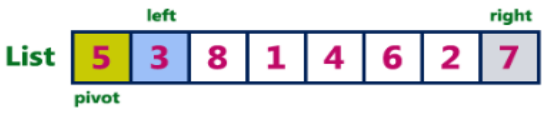
Consider the following list of unsorted elements…..



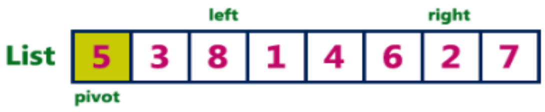
Define Pivot, left & right. Set Pivot = 0, left = 1 & right = 7. Here ‘7’ indicates ‘size - 1’



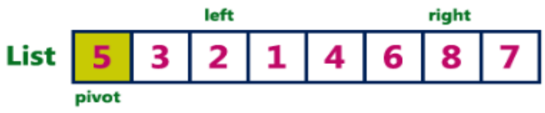
Compare List[left] with List[pivot]. If **List[left]** is greater than **List[pivot]** then stop left otherwise move left to the next.
Compare List[right] with List[pivot]. If List[right] is smaller than List[pivot] then stop right otherwise move right to the previous.
Repeat the same until **left>=right**.
If both left & right are stoped but left<right then swap List[left] with List[right] and countinue the process.
If left>=right then swap List[pivot] with List[right].



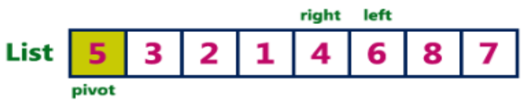
Compare List[left]<List[pivot] as it is true increment left by one and repeat the same, left will stop at 8.
Compare List[right]>List[pivot] as it is true decrement right by one and repeat the same, right will stop at 2.



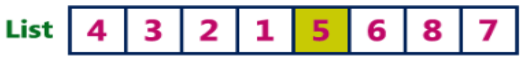
Here left & right both are stoped and left is not greater than right so we need to swap List[left] and List[right]



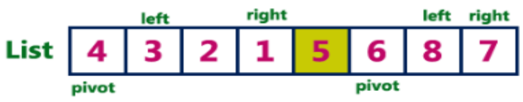
Compare List[left]<List[pivot] as it is true increment left by one and repeat the same, left will stop at 6.
Compare List[right]>List[pivot] as it is true decrement right by one and repeat the same, right will stop at 4.



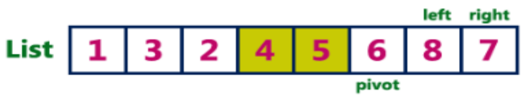
Here left & right both are stoped and left is greater than right so we need to swap List[pivot] and List[right]



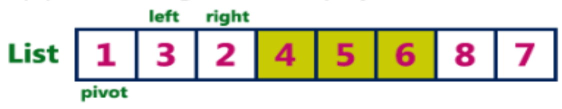
Here we can observe that all the numbers to the left side of 5 are smaller and right side are greater. That means 5 is placed in its correct position.
Repeat the same process on the left sublist and right sublist to the number 5.



In the left sublist as there are no smaller number than the pivot left will keep on moving to the next and stops at last number. As the List[right] is smaller, right stops at same position. Now left and right both are equal so we swap pivot with right.



In the right sublist left is grester than the pivot, left will stop at same position.
As the List[right] is greater than List[pivot], right moves towords left and stops at pivot number position.
Now left > right so we swap pivot with right. (6 is swap by itself).



Repeat the same recursively on both left and right sublists until all the numbers are sorted.
The final sorted list will be as follows...

List
1 2 3 4 5 6 7 8

The complexity of the Quick Sort Algorithm

To sort an unsorted list with **'n'** number of elements, we need to make **((n-1)+(n-2)+(n-3)+......+1) = (n (n-1))/2** number of comparisons in the worst case. If the list is already sorted, then it requires an **'n'** number of comparisons.

Worst Case: O(n2)

Best Case: O (n log n)

Average Case: O (n log n)

What is Bubble Sort

The working procedure of bubble sort is simplest. This article will be very helpful and interesting to students as they might face bubble sort as a question in their examinations. So, it is important to discuss the topic.

Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order. It is called bubble sort because the movement of array elements is just like the movement of air bubbles in the water. Bubbles in water rise up to the surface; similarly, the array elements in bubble sort move to the end in each iteration.

Although it is simple to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world. It is not suitable for large data sets. The average and worst-case complexity of Bubble sort is $O(n^2)$, where n is a number of items.

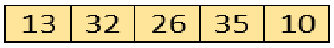
Bubble short is majorly used where -

- **complexity does not matter**

- simple and shortcode is preferred

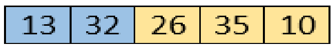
Now, let's see the working of Bubble sort Algorithm.
To understand the working of bubble sort algorithm, let's take an unsorted array. We are taking a short and accurate array, as we know the complexity of bubble sort is **$O(n^2)$.**

Let say the elements of array are -

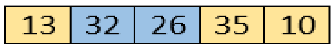


First Pass

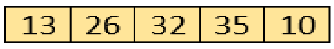
Sorting will start from the initial two elements. Let compare them to check which is greater.



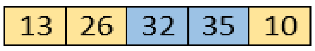
Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.



Here, 26 is smaller than 36. So, swapping is required. After swapping new array will look like -

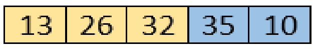


Now, compare 32 and 35.

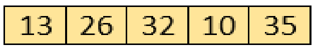


Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.

Now, the comparison will be in between 35 and 10.



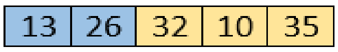
Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be -

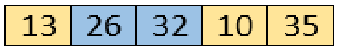


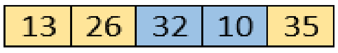
Now, move to the second iteration.

Second Pass

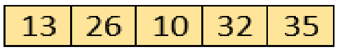
The same process will be followed for second iteration.

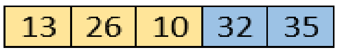






Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be -

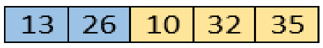


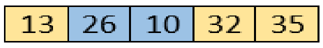


Now, move to the third iteration.

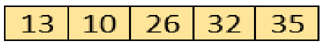
Third Pass

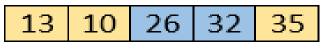
The same process will be followed for third iteration.

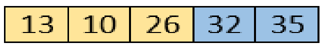




Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be -



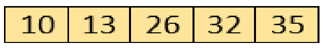




Now, move to the fourth iteration.

Fourth pass

Similarly, after the fourth iteration, the array will be -



Hence, there is no swapping required, so the array is completely sorted.

Bubble sort complexity

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is **O(n).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is **$O(n^2)$.**
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is **$O(n^2)$.**

Insertion Sort Algorithm

In this article, we will discuss the Insertion sort Algorithm. The working procedure of insertion sort is also simple. This article will be very helpful and interesting to students as they might face insertion sort as a question in their examinations. So, it is important to discuss the topic. Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place. The same approach is applied in insertion sort. The idea behind the insertion sort is that first take one element, iterate it through the sorted array. Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is $O(n^2)$, where n is the number of items. Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

Insertion sort has various advantages such as

- Simple implementation
- Efficient for small data sets
- Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

Now, let's see the algorithm of insertion sort.

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.

Insertion sort complexity

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is **O(n)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is **O(n^2)**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is **O(n^2)**.

Merge Sort Algorithm

In this article, we will discuss the merge sort Algorithm. Merge sort is the sorting technique that follows the divide and conquer approach. This article will be very helpful and interesting to students as they might face merge sort as a question in their examinations. In coding or technical interviews for software engineers, sorting algorithms are widely asked. So, it is important to discuss the topic. Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the merge() function to perform the merging. The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

Merge sort complexity

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is **O(n*logn)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is **O(n*logn)**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of merge sort is **O(n*logn)**.

*****NOTE*****

In C#, arguments can be passed to parameters either by value or by reference. Remember that C# types can be either reference types (class) or value types (struct):

- *Pass by value* means passing a copy of the variable to the method.

- *Pass by reference* means passing access to the variable to the method.
- A variable of a *reference type* contains a reference to its data.
- A variable of a *value type* contains its data directly.

Because a struct is a value type, when you pass a struct by value to a method, the method receives and operates on a copy of the struct argument. The method has no access to the original struct in the calling method and therefore can't change it in any way. The method can change only the copy.

A class instance is a reference type, not a value type. When a reference type is passed by value to a method, the method receives a copy of the reference to the class instance. That is, the called method receives a copy of the address of the instance, and the calling method retains the original address of the instance. The class instance in the calling method has an address, the parameter in the called method has a copy of the address, and both addresses refer to the same object. Because the parameter contains only a copy of the address, the called method cannot change the address of the class instance in the calling method. However, the called method can use the copy of the address to access the class members that both the original address and the copy of the address reference. If the called method changes a class member, the original class instance in the calling method also changes.