Linear Search Algorithm

```
Linear_search(values[],target, n)
Begin
    i
    for (i < n)
        if
          values[i] = target
          return i
        end if
    end for
    return -1;
End
```

Example

Elements n =5
Search Target element =7

0	1	2	3	4
100	11	12	7	21

Code View

```
int linearSearch(int values[], int target, int n)
{
    for(int i = 0; i < n; i++)
    {
        if (values[i] == target)
        {
            return i;
        }
    }
    return -1;
}
```

Binary Search Algorithm

```
Binary_Search(a[], low, high, key)
Begin
   while(low<=high)

     mid=(low+high)/2
     if
        a[mid]<key

          low=mid+1

     else if
        a[mid]>key

          high=mid-1

     else

          return mid

     end if

   end while

   return -1                //key not found
 End
```

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.

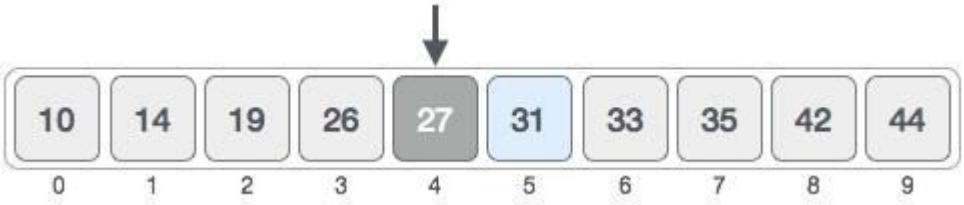
First, we shall determine half of the array by using this formula –

Low=0, high=9

```
mid = (low + high) / 2
```

Here it is, (0 + 9) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array. Mid=4, a[mid]=a[4]=27,27<31

Low=mid+1=4+1=5



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. High=9,low=5

```
low = mid + 1
mid = low + high / 2=(5+9)/2=7
```

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.

High=mid-1=7-1=6

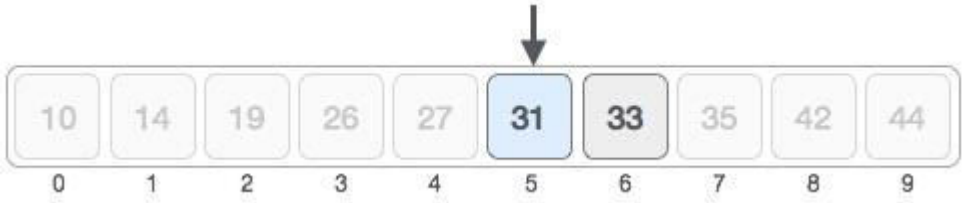
Low=5

Mid=(5+6)/2=5

Hence, we calculate the mid again. `high= mid - 1`

```
mid = low + high / 2=(5+6)/2=5
```

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.

Insertion Sort

INSERTION-SORT(A)

for j ← 2 to length[A]

do key ← A[j]

▸ Insert A[j] into the sorted sequence A[1 to j - 1].

i←j - 1

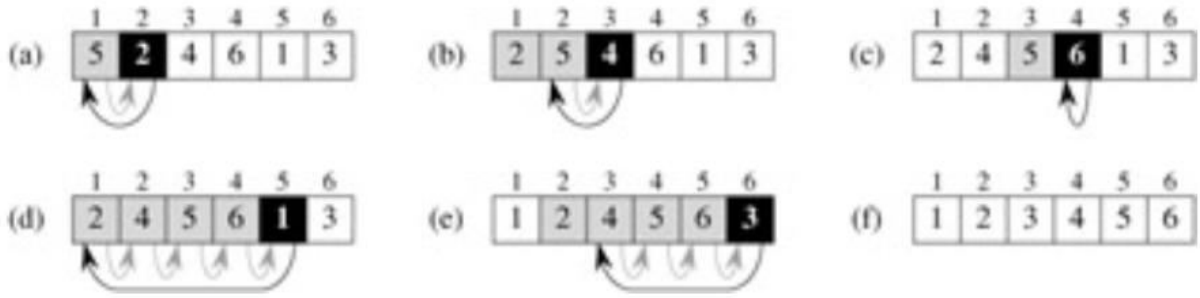
while i > 0 and A[i] > key

do A[i + 1] ←A[i]

i←i - 1

A[i + 1] ←key

Consider the following array: 5, 2, 4, 6, 1, 3



A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
5	2	4	6	1	3

First Iteration:

for j ← 2 to 6

j=2

A[j]=key

→ A[2]=2 key=2

i=j-1=2-1=1 → i=1

A[1]= 5

while

i>0 →1>0 && A[i] > key → 5>2 T

A[i+1] ← A[i] → A[1+1] ← A[1] → A[2] ← 5

i←i - 1 → i←1 – 1 → i=0

A[i + 1] ←key

→ A[0 + 1] ←2

→ A[1] ←2

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
2	5	4	6	1	3

Second Iteration

j=3
A[j]=key
→ A[3]=4
key=4

i=j-1=
3-1=2 → i=2
A[2]=5
while
i>0 →1>0 && A[i] > key → 5>4 T
A[i+1] ← A[i]
→ A[2+1] ← A[2]
→ A[3] ← 5
i←i - 1 → i←2 – 1 → i=1

A[i]=2
While i>0 →1>0 && A[i] > key → 2>4 F

A[i + 1] ←key → A[1 + 1] ←4 → A[2] ←4

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
2	4	5	6	1	3

Third Iteration

j=4
A[j]=key
→ A[4]=6
key=6

i=j-1
=4-1=3 → i=3

A[i]=5 →A[3]=5

while
i>0 →3>0 && A[i] > key → 5>6 T
A[i+1] ← A[i]
→ A[2+1] ← A[2]
→ A[3] ← 5
i←i - 1 → i←2 - 1 → i=1

A[i + 1] ←key → A[3 + 1] ←6→ A[2] ←4

//

j=4
A[j]=key→ A[4]=6 key=6
i=j-1=4-1=3 → i=3
A[i]=5 →A[3]=5

while
i>0 →3>0 && A[i] > key → 5>6 T
A[i+1] ← A[i] → A[2+1] ← A[2] → A[3] ← 5
i←i - 1 → i←2 - 1 → i=1

A[3 + 1] ←key

A[4]=6

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
1	2	4	5	6	3

Fourth Iteration

j=6
A[j]=key→ A[6]=1 key=3
i=j-1=5-1=4 → i=5
A[i]=5 →A[5]=6

while
i>0 →1>0 && A[1] > key → 2>1 T
A[i+1] ← A[i]

→ A[1+1] ← A[2]

→ A[2] ← 2
i←i - 1 → i←1- 1 → i=0

A[0 + 1] ←key

A[1]=1

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
1	2	4	5	6	3

So, after Fifth Iteration

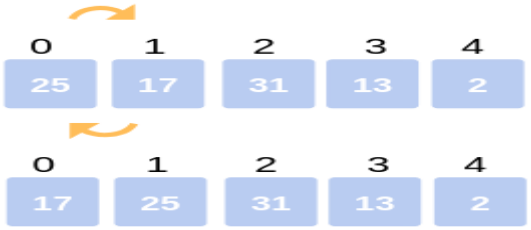
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
1	2	3	4	5	6

//

****Consider the following array: 25, 17, 31, 13, 2

Compare 25 with 17. The comparison shows 17< 25. Hence swap 17 and 25. The array now looks like:

17, 25, 31, 13, 2



First Iteration

Second Iteration: Begin with the second element (25), but it was already swapped on for the correct position, so we move ahead to the next element.
Now hold on to the third element (31) and compare with the ones preceding it.

Since 31> 25, no swapping takes place.

Also, 31> 17, no swapping takes place and 31 remains at its position.

The array after the Second iteration looks like:

17, 25, 31, 13, 2

Second Iteration

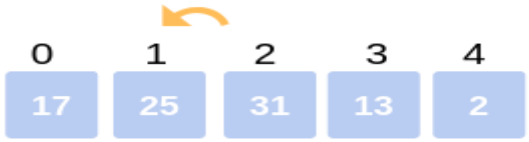
Third Iteration: Start the following Iteration with the fourth element (13), and compare it with its preceding elements.
Since 13< 31, we swap the two.

Array now becomes: 17, 25, 13, 31, 2.

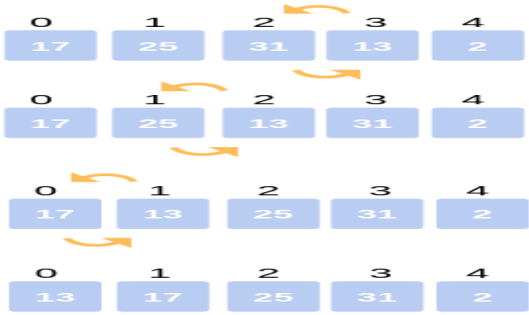
But there still exist elements that we haven't yet compared with 13. Now the comparison takes place between 25 and 13. Since, 13 < 25, we swap the two.

The array becomes **17, 13, 25, 31, 2.**
The last comparison for the iteration is now between 17 and 13. Since 13 < 17, we swap the two.

The array now becomes **13, 17, 25, 31, 2.**



Third Iteration



Fourth Iteration: The last iteration calls for the comparison of the last element (2), with all the preceding elements and make the appropriate swapping between elements.
Since, 2< 31. Swap 2 and 31.

Array now becomes: 13, 17, 25, 2, 31.

Compare 2 with 25, 17, 13.

Since, 2< 25. Swap 25 and 2.

13, 17, 2, 25, 31.
Compare 2 with 17 and 13.

Since, 2<17. Swap 2 and 17.

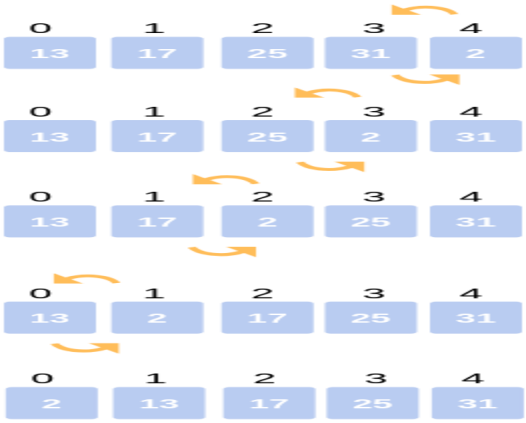
Array now becomes:

13, 2, 17, 25, 31.
The last comparison for the Iteration is to compare 2 with 13.

Since 2< 13. Swap 2 and 13.

The array now becomes:

2, 13, 17, 25, 31.
This is the final array after all the corresponding iterations and swapping of elements.



Fourth Iteration

Bubble Sort Algorithm

```
BubbleSort(list)
begin
   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for

   return list

end
```

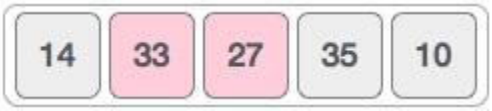
How Bubble Sort Works?

We take an unsorted array for our example.

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this –

Next we compare 33 and 35. We find that both are in already sorted positions.

Then we move to the next two values, 35 and 10.

We know then that 10 is smaller 35. Hence they are not sorted.

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this –

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this –

Notice that after each iteration, at least one value moves at the end.

And when there's no swap required, bubble sorts learns that an array is completely sorted.

Now we should look into some practical aspects of bubble sort.

Code View:

```
void bubbleSort(int arr[], int n)
{
   int i, j;
   for (i = 0; i < n-1; i++)
       for (j = 0; j < n-i-1; j++)
           if (arr[j] > arr[j+1])
              swap(&arr[j], &arr[j+1]);
}
```

Merge Sort

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

38	27	43	3	9	82	10

p=1,r=7, q=(p+r)/2 → q=4

MERGE-SORT(A,1,4)-→MS MS M

MERGE-SORT(A,5,7)

MERGE(A,1,4,7)

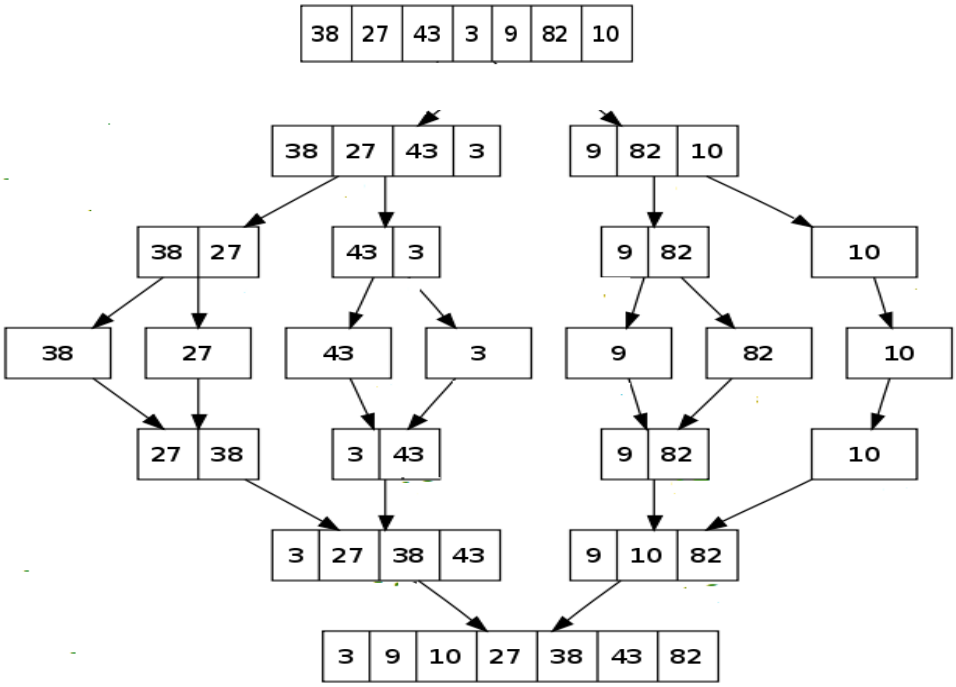
```
MERGE(A, p, q, r)
1   n1 = q − p + 1
2   n2 = r − q
3   let L[1 . . n1 + 1] and R[1 . . n2 + 1] be new arrays
4   for i = 1 to n1
5       L[i] = A[p + i − 1]
6   for j = 1 to n2
7       R[j] = A[q + j]
8   L[n1 + 1] = ∞
9   R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

38	27	43	3	9	82	10

n1=q-p+1 =4-1+1 =4

n2=r-q=7-4=3

```
for(i=1 to n1)
L[i]=A[p+i-1]

i=1,
L[1]=A[1+1-1]
L[1]=A[1]=3

i=2
L[i]=A[p+i-1]
L[2]=A[1+2-1]
L[2]=A[2]=27 So on…

for(j=1 to n2)

R[j]=A[q+j]
j=1,
R[1]=A[4+1]
R[1]=A[5]=9

j=2
R[2]=A[4+2]
R[2]=A[6]=10

j=3
R[3]=A[4+3]
R[3]=A[7]=82
```

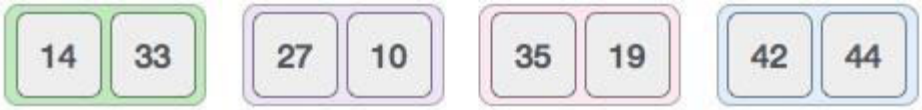


How Merge Sort Works?

To understand merge sort, we take an unsorted array as the following −

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

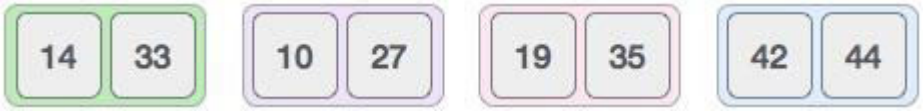
This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.



We further divide these arrays and we achieve atomic value which can no more be divided.

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

After the final merging, the list should look like this −

Now we should learn some programming aspects of merge sorting.

Quick Sort Algorithm

```
QuickSort(A,p,r)
{
if (p<r)
{
q=Partiton(A,p, r)
QuickSort(A,p,q-1)
QuickSort(A,q+1,r)
}
}
Partition(A,p,r)
x=A[r]
i=p-1
for(j=p to r-1)
{

if (A[j]<=x)
{
i=i+1
exchange A[i] with A[j]
}


}
exchange A[i+1] with A[r]
return i+1
}
```

We will consider the following array: 50, 23, 9, 18, 61, 32

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
50	23	9	18	61	32

p=1, r=6
pivot=x=A[r]=A[6]=32
pivot=32

i=p-1=1-1=0
//r-1 =6-1 =5
for(j= 1 to 5)
j=1
if(A[j]<=x→ A[1]<=32→50<=32)

{
//i=i+1
//exchange A[i] with A[j]
}

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
50	23	9	18	61	32

for(j= 1 to 5)
j=2
if(A[j]<=x→ A[2]<=32→23<=32)
{
i=i+1 → i=0+1=1
A[i]=A[1]=50
A[j]=A[2]=23
exchange A[i] with A[j]
}

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
23	50	9	18	61	32

for(j= 1 to 5)
j=3
A[j]=A[3]=9
if(A[j]<=x→ A[3]<=32→9<=32)
{
i=i+1 → i=1+1=2
A[i]=A[2]=50
A[j]=A[3]=9
exchange A[i] with A[j]
}

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
23	9	50	18	61	32

for(j= 1 to 5)
j=4
A[j]=A[4]=18
if(A[j]<=x→ A[3]<=32→18<=32)
{
i=i+1 → i=2+1=3
A[i]=A[3]=50
A[j]=A[4]=18
exchange A[i] with A[j]→ 50 with 18
}

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
23	9	18	50	61	32

for(j= 1 to 5)
j=5
if(A[j]<=x→ A[5]<=32→61<=32)
{
//i=i+1
//exchange A[i] with a[j]
}

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
23	9	18	50	61	32

exchange A[i+1] with A[r]
last i=3
A[i+1]= A[3+1]=A[4]=50
A[r]=A[6]=32
Exchange 50 with 32
return i+1

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
23	9	18	32	61	50

//
Pivot=32 p=1 r=6

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
50	23	9	18	61	32

50<=32 F

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
50	23	9	18	61	32

23<=32 T

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
23	50	9	18	61	32

9<=32 T

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
23	9	50	18	61	32

18<=32 T

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
23	9	18	50	61	32

61<=32 F

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
23	9	18	50	61	32

After the exchange of 50 and 32

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
23	9	18	32	61	50

Here, we will get two parts to continue the quick sort.
So, rest of the part will follow the above procedure again and finally we will get the following sorted array:

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
9	18	23	32	50	61

Example-2

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
5	23	9	18	61	32

p=1, r=6
pivot=x=A[r]=A[6]=32
pivot=32

i=p-1=1-1=0
//r-1 =6-1 =5
for(j= 1 to 5)
j=1
if(A[j]<=x→ A[1]<=32→5<=32)
{
i=i+1= 0+1=1 → A[i]=A[1]=5 and A[j]=A[1]=5
exchange A[i] with A[j]→ exchange 5 with 5, so 5 will remain at the same position.
}

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
5	23	9	18	61	32

for(j= 1 to 5)
j=2
if(A[j]<=x→ A[2]<=32→23<=32)
{
i=i+1 → i=1+1=2
A[i]=A[2]=23
A[j]=A[2]=23
exchange A[i] with A[j]

}

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
5	23	9	18	61	32

for(j= 1 to 5)
j=3
A[j]=A[3]=9

if(A[j]<=x→ A[3]<=32→9<=32)
{
i=i+1 → i=2+1=3
A[i]=A[3]=9
A[j]=A[3]=9
exchange A[i] with A[j]
}

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
5	23	9	18	61	32

for(j= 1 to 5)
j=4
A[j]=A[4]=18
if(A[j]<=x→ A[3]<=32→18<=32)
{
i=i+1 → i=3+1=4
A[i]=A[4]=18
A[j]=A[4]=18
}

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
5	23	9	18	61	32

for(j= 1 to 5)
j=5
if(A[j]<=x→ A[5]<=32→61<=32)
{
//i=i+1
//exchange A[i] with a[j]
}

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
23	9	18	50	61	32

exchange A[i+1] with A[r]
last i=4
A[i+1]= A[4+1]=A[5]=61
A[r]=A[6]=32
Exchange 61 with 32
return i+1=4+1
After the exchange of 61 and 32

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
23	9	18	50	32	61

Here, we will get two parts to continue the quick sort.
So, rest of the part will follow the above procedure again and finally we will get the following sorted array:

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
9	18	23	32	50	61