```
    {
    return binary search (a, beg, mid-1, item);
    }
  }
  return -1;
  }
```

Output:
```
Enter item which you want to search
19
Item found at location 2.
```

Sorting Algorithm :-

1). Bubble sort Algorithm :-

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

It is not suitable for large data sets. The average and worst case complexity of bubble sort is $O(n^2)$, where n is number of items.

Bubble sort is majorly used where :

- complexity does not matter
- simple and shortcode is preferred.

Algorithm :-

```
begin Bubblesort (arr)
  for all array elements
  if arr[i] > arr(i+1]
    swap (arr[i], arr[i+1]]
  end if
 end for
 return arr
end Bubble sort.
```

Bubble sort complexity :-

case	Time complexity	space complexity
Best case	O(n)	O(1).
Average case	O(n²)	
worst case	O(n²)	

Implementation of Bubble Sort :-

C language Implementation :-

```
#include <stdio.h>
void print(int a[],int n)
 {
  int i;
  for (i=0;i<n;i++)
   {
    printf("%d",a[i]);
   }
}
void bubble (int a[], int n)
 {
 int i,j,temp;
 for(i=0;i<n;i++)
  {
  For (j=i+1 ; j<n ; j++)
   {
   if (a [j] < a[i])
    {
    temp = a[i];
    a[i] = a[j];
    a[j] = temp;
    }
  }
 }
}
void main ()
 {
  int i,j,temp;
  int a[5]= { 10,35,32,13,26};
  int n = size of (a) / size of (a0);
  printf ("Before sorting array elements are :\n");
  print (a,n);
  bubble (a,n);
  printf ("\n after sorting array elements -\n");
  print (a,n);
}
```

output :-

```
Before sorting array elements are -
10  35  32  13  26
After sorting array elements are -
10  13  26  32  35.
```

Bucket Sort Algorithm :-

The data items in the bucket sort are distributed in form of buckets.

Bucket sort is a sorting algorithm that seprates elements into multiple groups said to be buckets. Elements in bucket sort are first uniformly divided into groups called buckets, and then they are sorted by any other sorting algorithm. After that, elements are gathered in sorted manner.

Advantages of bucket sort are :-

- Bucket sort reduces no. of comparisons
- It is asymptotically fast because of uniform distribution of elements.

limitations of bucket sort are :

- It may or may not be a stable sorting algorithm
- It is not useful if we have a large array bcz it increases the cost.
- It is not an in-place sorting algorithm, because more some extra space is required to sort the buckets.

The best and average-case complexity of bucket sort is $O(n+k)$, worst-case complexity of bucket sort is $O(n^2)$, where n is number of items.

Bucket sort is commonly used :

- with floating - point values.
- when input is distributed uniformly over a range.

Algorithm :-

```
Bucket sort (A [])
1.  let B [0 ....n-1] be a new array.
2.  n = length [A].
3.  for i=0 to n-1
4.  make B[i] an empty list
5.  for i=1 to n
6.  do insert A[i] into list B[n a[i]]
7.  for i=0 to n-1
8.  do sort list B[i] with insertion sort.
9.  concatenate lists B[0], B[1] .... B[n-1] together
    in order.
10. END.
```

Complexity :-

1. Time complexity :-

case	time complexity
Best case	$O(n+k)$.

Average case	$O(n+k)$
worst case	$O(n^2)$

2. space complexity :-

Space complexity	$O(n*k)$
stable	YES

Implementation of bucket sort in c:-

```
#include <stdio.h>
int getmax (int a[],int n)
 {
  int max =a[0];
  for( int i=1; i<n; i ++)
    if (a[i] > max)
       max= a[i];
   return max;
 }
void bucket(int a[], int n)
 {
  int max = getmax (a,n)
  int bucket[max],i;
  for( int i = 0 ;i<=max ;i++)
   {
        bucket [i]=0 ;
    }
   for(int i=0 ;i<n ;i++)
     {
      bucket [a[i]]++;
     }
   for (int i= 0; j=0 ;i<=max ;i++)
     {
     while (bucket [i] >0)
       {
        a[j++] =i ;
        bucket [i]-- ;
       }
   }
  }
  void printArr (int a[],int n)
   {
    for (int i=0; i<n ; i++)
    printf("%d", a[i]);
   }
  int main ()
  {
   int a[]= {54, 12, 84, 57, 69, 41, 9, 5};
   int n= size of(a) / size of a[0]);
   printf ("Before sorting array elements are:-\n");
   printArr(a,n);
    bucket (a,n);
   printf( "\n After sorting array elements are:-\n");
    printArr(a,n);
   }
```

output :-

```
Before sorting array elements are :-
54   12   84   57   69   41   9   5
After sorting array elements are :-
5   9   12   41   54   57   69   84.
```

Heap Sort Algorithm :-

Heap sort processes the elements by creating min-heap or max-heap using the elements of the given array.

Heap sort basically recursively performs two main operations :

- Build a heap H, using the element of array.
- Repeatedly delete the root element of heap formed in 1st phase.

What is heap ?

A heap is a complete binary tree, and binary tree is a tree in which node can have utmost two children.

Algorithm :-

```
Heap Sort (arr)
BuildMaxHeap(arr)
for i = length (arr) to 2
    swap arr [i] with arr [1]
     heap-size [arr] = heap-size [arr] ? 1
      MaxHeapify (arr, 1)
End.
```

```
BuildmaxHeap(arr):
    BuildmaxHeap (arr)
        heap-size (arr)= length (arr)
        for i = length (arr)/2 to 1.
    maxHeapify (arr, i)
    End.
```

complexity :-

case	Time complexity	space complexity.
Best	o(n log n)	o(1).
Average	o(n log n)	
worst	o(n log n)	

Implementation of Heap sort :-

```
#include<stdio.h>
/* function to heapify a subtree. Here is 'i' the
index of root node in array a[], and 'n' is size
of heap */
void heapify (int a[], int n, int i)
 {
  int largest = i
  int left = 2*i+1
  int right = 2* i+2
  if(left <n && a[left]> a[largest]
     largest= left;
```

```
if (right <n && a[right] > a[largest]
  largest = right ;
if (largest != i)
{ int temp = a[i] ;
   a[i] = a[largest];
   a[largest] = temp;
   heapify (a, n, largest);
 }
}
void heapsort (int a[], int n)
{
  for(int i = n/2 -1 ; i>=0, i--)
    heapify (a, n, i);
  for (int i= n-1; i>=0 , i--)
  { int temp = a[0];
     a[0] = a[i];
     a[i] = temp;
   heapify (a, i, 0);
  }
} void printArr (int arr[], int n)
{
   for(int i= 0; i<n ; ++i)
     {
       printf("%d", arr[i]);
       printf(" ");
     }
} int main ()
 {
 int a[] = {48, 10, 23, 43, 28, 26, 1};
 int n = sizeof (a) / sizeof (a[0]);
 printf ("Before sorting array elements are - \n");
```

```
    printArr(a,n);
    heapSort(a,n);
    printf("\nAfter sorting array elements are -\n");
    printArr(a,n);
    return 0;
}
```

output:

```
Before sorting array elements are :-
 48  10  23  43  28  26  1
After sorting array elements are -
  1  10  23  26  28  43  48.
```

Insertion sort Algorithm :-

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card. The idea behind the insertion sort is that first make /take one element, iterate it through sorted array. complexity of insertion sort in the average case and worst case is $O(n^2)$, where n is number of items.

Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort and 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.

complexity :-

case time complexity :

Best case o(n)

Avergge case o(n^2)

worst case o(n^2).

space complexity o(1).

Implementation of insertion sort :-

```
#include <stdio.h>
void insert (int a[], int n)
{
  int i, j, temp;
  for (i= 1; i<n ,i++) {
    temp =a[i];
     j = i-1 ;
   while (j>=0 && temp <= a[j])
    {
     a[j+1] = a[j];
     j = j-1;
    }
    a[j+1] = temp;
 }
}
void printArr (int a[], int n)
 {
   int i;
   for(i=0; i<n; i++)
   printf ("%d", a[i]);
 }
int main()
{
  int a[] = {12,31, 25, 8, 32, 17};
```

```
    int n = size of (a) / size of a[0]);
    printf ("Before sorting array elements are -\n");
    printArr (a, n);
    insert (a, n);
    printf("\nAfter sorting array elements are -\n");
    printArr(a, n);
    return 0;
}
```

output:

```
Before sorting array elements are -
12 31 25 8 32 17
After sorting array elements are -
8 12 17 25 31 32.
```

Merge sort Algorithm :-

Merge sort is the sorting technique that follows divide and conquer approach. This will be very helpful and interesting.

Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort elements.

Algorithm :-

arr is given array, beg is starting element and end is last element of array.

```
MERGE-SORT (arr, beg, end)
if beg < end
set mid = (beg + end)/2.
MERGE-SORT (arr, beg, mid)
MERGE_SORT (arr, mid+1, end)
MERGE (arr, beg, mid, end)
```

end of if
End MERGE_SORT.

implementation of merge sort:-

```
/* function of merge the subarrays of a[] */
void merge (int a[], int beg, int mid, int end)
 {
  int i, j, k;
  int n1 = mid - beg + 1;
  int n2 = end - mid;
  int LeftArray [n1], RightArray [n2];
  /* copy data to temp arrays */
  for(int i=0; i<n1; i++)
  LeftArray [i] = a[beg + i];
  for (int j=0; j<n2; j++)
  RightArray[j] = a[mid+1 +j];
  i = 0;
  j = 0;
 k = beg;
 while (i<n1 && j<n2)
 {
  if(LeftArray [i] <= RightArray [j]).
  {
   a[k] = LeftArray[i];
   i++;
  }
 else
 {
 a[k]=RightArray [j];
 j++;
 }
```

```
    }
    k++;
    }
   while (i<n1)
   {
   a[k]= LeftArray[i];
    i++;
   k++;
  }
 while (j<n2)
 {
   a[k] = RightArray [j];
   j++;
   k++;
  }
}
```

Complexity :-

case	Time complexity	space complexity
Best case	$o(n*\log n)$	$o(n)$.
Avg. case	$o(n*\log n)$	
worst case	$o(n*\log n)$	