Unit 1

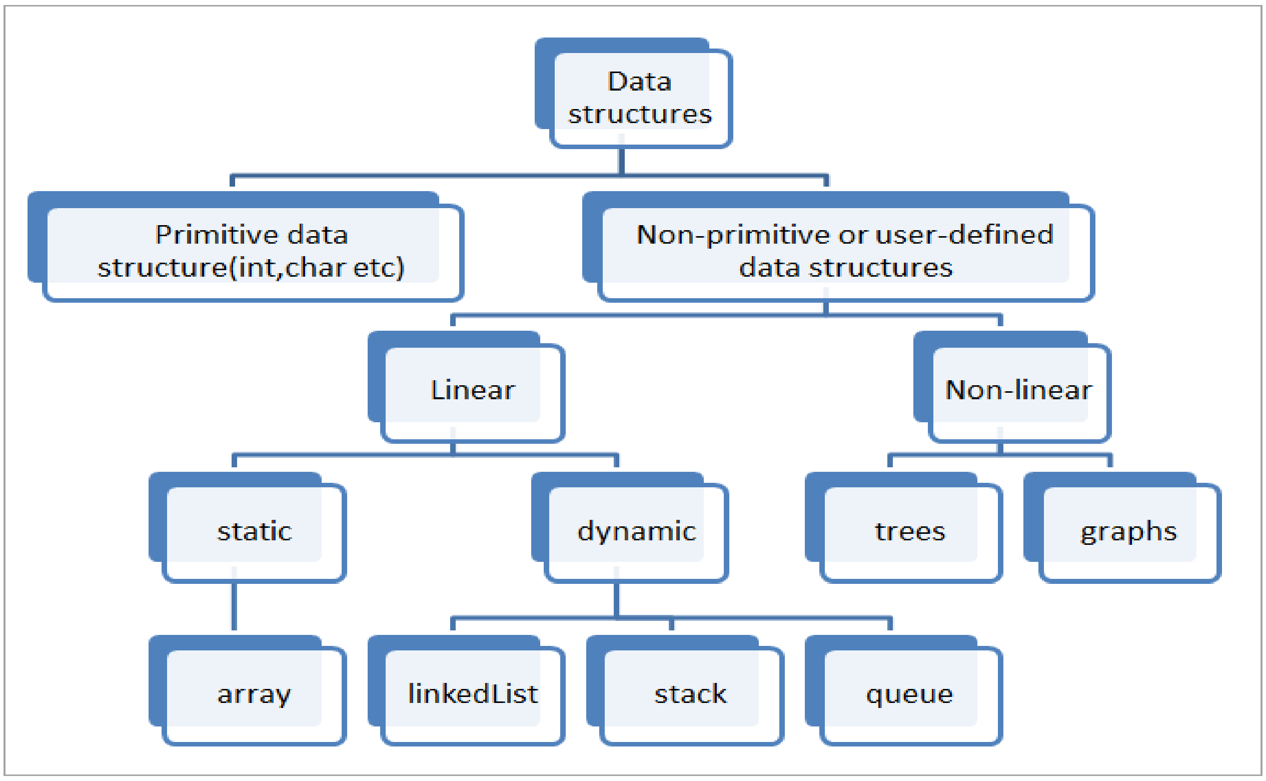
- Data Structure:

  What is Data?
  Anything to give information is called data.
  Eg:- StudentName, StudentRollno, etc..

  What is Structure?
  Representation of data is called strucrure.
  Eg:- Arrays, List, Stack, Graphs, Queues.

- The name of the data structure implies that it is used to organise data in memory.
- There are numerous methods for organising data in memory, as we have already seen with one of the data structures, array in C. An array is a group of memory elements in which data is stored sequentially, that is, one after the other. In other words, an array stores the elements in a continuous fashion. An array of data structures is used to organise the data.
- There are other ways to organise data in memory, and the data structure is not defined by any programming language such as C, C++, java, and so on. It is a set of algorithms that can be used in any programming language to structure data in memory.

- Classification of data structure


Data structures
Primitive data structure(int,char etc)
Non-primitive or user-defined data structures
Linear
Non-linear
static
dynamic
trees
graphs
array
linkedList
stack
queue

- <u>Primitive Data Structure</u>:
  The primitive data structures are primitive data types. The primitive data structures that can hold a single value are int, char, float, double, and pointer.

- <u>Non-primitive Data Structure:</u>
  The non-primitive data structure is divided into two types:

  1)Linear data structure
  2)Non-linear data structure

<u>Linear Data Structure:</u>
A linear data structure is the arrangement of data in a sequential order. Arrays, linked lists, stacks, and queues are the data structures used for this purpose. In these data structures, one element is linearly connected to only one

other element.

Non-linear Data Structure:
A non-linear data structure has no set sequence of connecting all its elements and each element can have multiple paths to connect to other elements. Such data structures support multi-level storage and are frequently inaccessible in a single run. Such data structures are difficult to implement, but they are more efficient in terms of computer memory utilisation. Trees, BSTs, Graphs, and other non-linear data structures are examples.

linear data structure is classified in two part:
1) Static Data structure
2) Dynamic Data structure

Static Data Structure:
The size of the structure in a static data structure is fixed.
The data structure's content can be changed without changing the memory space allotted to it.
Memory is allocated at compile time in a static data structure. As a result, the maximum size is fixed.
Static data structure types include:
[1] array
Benefits:
- Quick Access
Disadvantages:
- Slower inserting and removing

Dynamic Data Structure:
The size of a dynamic data structure is not fixed and can be changed while operations are being performed on it.
Dynamic data structures are intended to allow for data structure changes during runtime.
Memory is allocated at run time in these types of data structures.
Dynamic data structure types include:
1] Stack

2] Waiting List
3] Linking List
Benefits:- Faster insertion and deletion

- Operations On Data Structure:

  1) **Traversing**: Every data structure contains the set of data elements. Traversing the data structure entails visiting each element of the data structure to perform a specific operation such as searching or sorting.

  2)**Insertion**: insertion is the process of adding elements to a data structure at any point in time.

  We can only insert n-1 data elements into a data structure if its size is n.

  3) **Deletion**: Deletion is the process of removing an element from a data structure. We can remove an element from the data structure at any point in time.

  Underflow occurs when we attempt to delete an element from an empty data structure.

  4) **Searching**: Searching refers to the process of locating an element within a data structure. Linear Search and Binary Search are the two algorithms used to perform searching.

  5) **Sorting**: Sorting is the process of arranging the data structure in a specific order. Sorting can be accomplished using a variety of algorithms, such as insertion sort, selection sort, bubble sort, and so on.

  6) **Merging**: Merging occurs when two lists, List A and List B, of size M and N, respectively, of similar type elements are clubbed or joined to produce the third list, List C of size (M+N).

- Abstract Data Types(ADT):
  The abstract datatype is a type of datatype whose behaviour is defined by a set of values and operations. The keyword "Abstract" is used because we can use these datatypes to perform various operations. However, how those operations work is completely hidden from the user. The ADT is composed of primitive datatypes, but the operation logic is hidden.

  Some examples of ADT are Stack, Queue, List etc.

  Let us see some operations of those mentioned ADT –

  Stack –
    - isFull(), This is used to check whether stack is full or not
    - isEmpry(), This is used to check whether stack is empty or not
    - push(x), This is used to push x into the stack
    - pop(), This is used to delete one element from top of the stack
    - peek(), This is used to get the top most element of the stack
    - size(), this function is used to get number of elements present into the stack

  Queue –
    - isFull(), This is used to check whether queue is full or not
    - isEmpry(), This is used to check whether queue is empty or not
    - insert(x), This is used to add x into the queue at the rear end
    - delete(), This is used to delete one element from the

  front end of the queue
  - size(), this function is used to get number of elements present into the queue

List –

- size(), this function is used to get number of elements present into the list
- insert(x), this function is used to insert one element into the list
- remove(x), this function is used to remove given element from the list
- get(i), this function is used to get element at position i
- replace(x, y), this function is used to replace x with y value

- <u>Preliminaries of Algorithm:</u>

An Algorithm is nothing more than a set of instructions. In general, the instructions must be specific enough that the amount of work required to complete each step can be estimated. There may be multiple algorithms to solve the same problem or complete the same task.

Algorithm Characteristics
Not all procedures can be referred to as algorithms. The following characteristics should be present in an algorithm:

Unambiguous – Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.

Input – An algorithm should have 0 or more well-defined inputs.

Output – An algorithm should have 1 or more well-defined outputs, and should match the desired output.

Finiteness – Algorithms must terminate after a finite number of steps.

Feasibility – Should be feasible with the available resources.

Independent – An algorithm should have step-by-step directions, which should be independent of any programming code.

- <u>Time and Space Complexity:</u>

Space Complexity:

The amount of memory space required by an algorithm during its life cycle is represented by its space complexity.

The space required by an algorithm is equal to the sum of the two components listed below.

A fixed part is a space required to store certain data and variables (such as simple variables and constants, programme size, and so on) that are independent of the size of the problem.

A variable part is a space required by variables, the size of which is entirely determined by the size of the problem. For instance, recursion stack space, dynamic memory allocation, and so on.

Example:

```
SUM(P, Q)
Step 1 - START
Step 2 - R ← P + Q + 10
Step 3 – Stop
```

Here we have three variables P, Q and R and one constant. Hence S(p) = 1+3. Now space is dependent on data types of given constant types and variables and it will be multiplied accordingly.

Time Complexity:

The Time Complexity of an Algorithm represents how long it takes the algorithm to complete its execution. Time requirements can be denoted or defined as a numerical function t(N), where t(N) can be measured as the number of steps, as long as each step takes the same amount of time.

When adding two n-bit integers, for example, N steps are taken. As a result, the total computational time is t(N) = c*n, where c is the amount of time required to add two bits. In this case, we can see that t(N) grows linearly as the size of the input increases.

- Searching:

The process of finding a specific element in a list is known as searching. If the element is found in the list, the process is deemed successful, and the location of that element is returned; otherwise, the search is deemed unsuccessful.

There are three popular search methods that are widely used to find an item in a list. However, the algorithm chosen is determined by the order of the list.

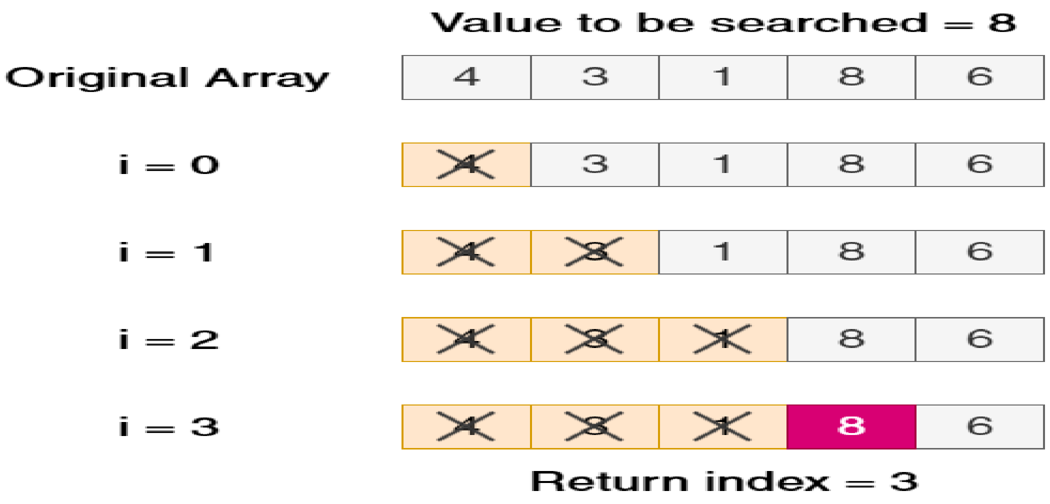
1. Linear Search
2. Binary Search
3. Fibonacci Search

Linear Search:

Linear search is the most basic search algorithm and is also known as sequential search. In this type of search, we simply traverse the list and match each element with the item whose location needs to be found. If a match is found, the item's location is returned; otherwise, the algorithm returns NULL.

Linear search is commonly used to search an unordered list of items.

Complexity	Best Case	Average Case	Worst Case
Time	O(1)	O(n)	O(n)



Algorithm:

```
1. Start
2. Read n, a[i], key values as integers
3. Search the list
       while(a[i]!=key && i<=n)
              i=i+1
               repeat step 3
4. Successful search
        if(i = n + 1) then
              print "Element not found in the list"
        Return(0)
       else
               print "Element found in the list"
       Return (i)
5. Stop
```

Program For Linear Search With Non-Recursion Function:

```
#include<stdio.h>
int main()
{
int i,a[10],n,flag=0,key;
printf("enter the size of array\n");
scanf("%d",&n);
printf("enter the array elements\n");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
printf("enter the key value");
scanf("%d",&key);

for(i=0;i<n;i++)
{
if(a[i]==key)
{
flag =1;
break;
}
}
if(flag==1)
{
printf("the key element is found at %d position",i+1);
}
else
{
printf("the key element is not found");
}
return 0;
}
```

Program For Linear Search With Recursion Function:

```
#include<stdio.h>
int linear(int a[],int,int);
int main()
{
int n,a[20],key,flag=0;
printf("enter the n value");
scanf("%d",&n);
printf("Enter the array elements");
for(int i = 0; i < n; i++)
{
scanf("%d", &a[i]);
}
printf("enter key value");
scanf("%d",&key);
flag=linear(a,n,key);
if(flag!=0)
{
printf("element is found at %d position",flag);
}
else
{
printf("element is not found");
}
return 0;
}

int linear(int a[],int n,int key)
{
if(n>0)
{
if(a[n-1]==key)
{
return n;
}
else
{
return linear(a,n-1,key);
}
n--;
}
}
```

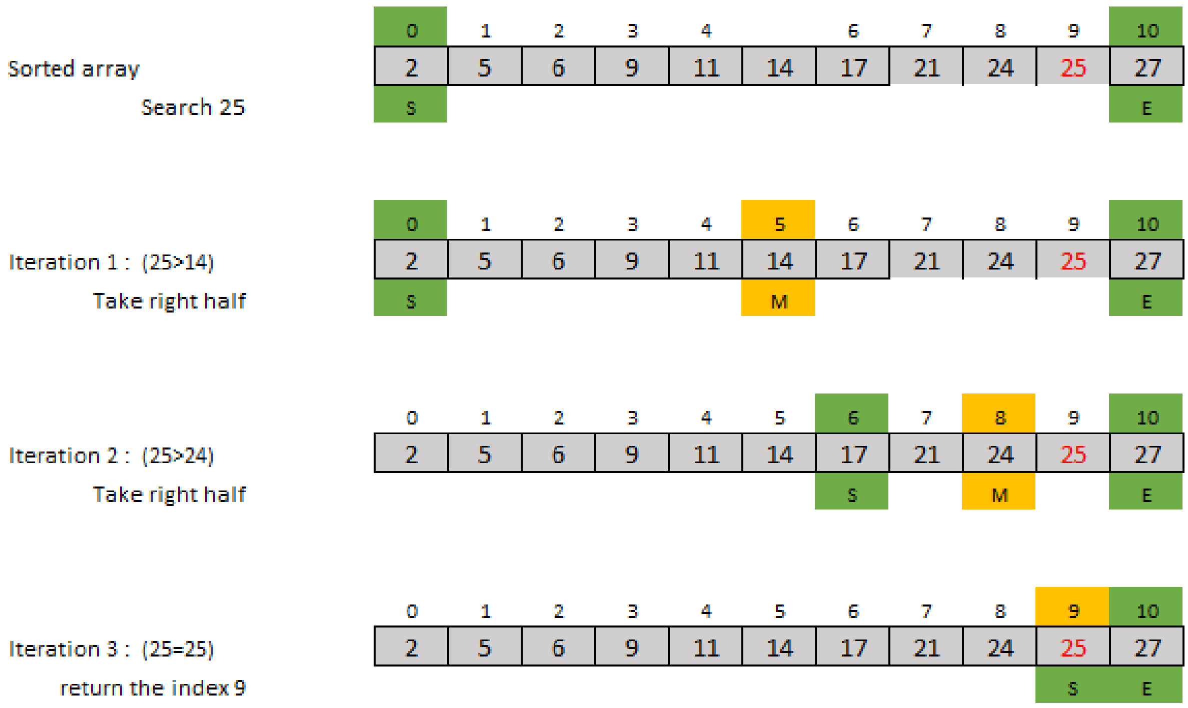
Binary Search:

Binary search is a type of search that works well with sorted lists. As a result, in order to use the binary search technique to find an element in a list, the list must first be sorted.

The divide and conquer strategy is used in binary search, in which the list is divided into two halves and the item is compared to the list's middle element. If a match is found, we return the location of the middle element; otherwise, we search into either half based on the match result.

Complexity:

SN	Performance	Complexity
1	Worst case	O(log n)
2	Best case	O(1)
3	Average Case	O(log n)



Algorithm:

```
 1. Start
 2. Initialize low = 1 high = n
 3: Perform Search While(low <= high)
 4: Obtain index of midpoint of interval
        Middle = (low + high) / 2
  5: Compare if(X < K[middle])
        high = middle – 1
         else
        print "Element found at position"
        Return(middle)
        goto: step 2
        6: Unsuccessful Search
        print "Element found at position"
        Return (middle)
 7: Stop
```

Program Of Binary Search With Non-Recursion Function:

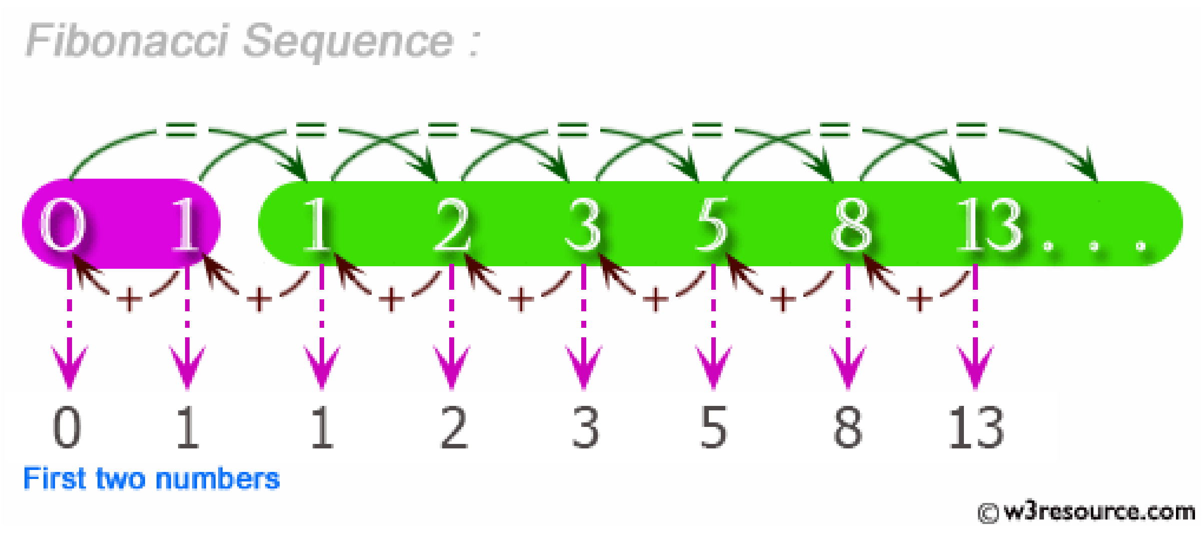
```
#include<stdio.h>
int main()
{

int a[20],i,n,key,start,end,mid;
printf("enter the no of element");
scanf("%d",&n);
printf("enter the elements");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
printf("enter key element");
scanf("%d",&key);

start=0;
end=n-1;

while(start<=end)
{
mid=(start+end)/2;
if(key==a[mid])
{
break;
}
else
{
if(key>a[mid])
{
start=mid+1;
}
else
{
end=mid-1;
}
}
}
if(key==a[mid])
{
printf("element is found at %d position",mid+1);
}
else
{
printf("element is not found");
}
return 0;
}
```

Program For Binary Search In Recursion Function:

```
#include<stdio.h>
#include<stdlib.h>
#define size 10
int binsearch(int[], int, int, int);
int main()
{
  int num, i, key, position;
  int low, high, list[size];
  printf("\nEnter the total number of elements");
  scanf("%d", &num);
  printf("\nEnter the elements of list :");
  for (i = 0; i < num; i++) {
    scanf("%d", &list[i]);
  }
  low = 0;
  high = num - 1;
  printf("\nEnter element to be searched : ");
  scanf("%d", &key);
  position = binsearch(list, key, low, high);
  if (position != -1)
 {
        printf("\nNumber present at %d", (position + 1));
    }
else
        printf("\n The number is not present in the list");
    return (0);
}
int binsearch(int a[], int x, int low, int high)
{
    int mid;
    if (low > high)
        return -1;
    mid = (low + high) / 2;
    if (x == a[mid]) {
        return (mid);
    } else if (x < a[mid]) {
        binsearch(a, x, low, mid - 1);
    }
    else {
        binsearch(a, x, mid + 1, high);
    }
}
```

Fibonacci Search:

- Fibonacci searching method uses Fibonacci series to search an element.
- This techniques divides the array into unequal parts.
- It doesn't use / to divide the array, but uses + and -. Since the division operator may be costly on some CPU systems.
- It is a comparison based technique that uses Fibonacci numbers to search an element in sorted array.



Algorithm of Fibonacci search:

- Find the smallest Fibonacci Number greater than or equal to n. Let this number be fibM [m'th Fibonacci Number]. Let the two Fibonacci numbers preceding it be fibMm1 [(m-1)'th Fibonacci Number] and fibMm2 [(m-2)'th Fibonacci Number].
- While the array has elements to be inspected:
- Compare x with the last element of the range covered by fibMm2
- If x matches, return index
- Else If x is less than the element, move the three Fibonacci variables two Fibonacci down, indicating elimination of approximately rear two-third of the

remaining array.
- **Else** x is greater than the element, move the three Fibonacci variables one Fibonacci down. Reset offset to index. Together these indicate the elimination of approximately front one-third of the remaining array.
- Since there might be a single element remaining for comparison, check if fibMm1 is 1. If Yes, compare x with that remaining element. If match, return index.

Program for Fibonacci search:

```
#include<stdio.h>
int fibonaccy_search(int[],int,int);
int min(int,int);
int main()
{
int arr[]={10,22,35,40,56,78,90,100};
int n=sizeof(arr)/sizeof(arr[0]);
int x=35;
int result= fibonaccy_search(arr,x,n);
if(result>=0)
{
printf("found at index %d",result);
}
else
{
printf("elment not found");
}
return 0;
}
int fibonaccy_search(int arr[],int x,int n)
{
int fibm2=0;//m-2th
int fibm1=1;//m-1th
int fibm=fibm2+fibm1;
```

```
while(fibm<n)
{
fibm2=fibm1;
fibm1=fibm;
fibm=fibm1+fibm2;
}

int offset=-1;
while(fibm>1)
{
int i=min(offset+fibm2,n-1);
if(arr[i]<x)
{
fibm=fibm1;
fibm1=fibm2;
fibm2=fibm-fibm1;
offset=i;
}
else if(arr[i]>x)
{
fibm=fibm2;
fibm1=fibm1-fibm2;
fibm2=fibm-fibm1;
}
else
return i;

}
if(fibm1 && arr[offset+1]==x)
return offset+1;
return -1;
}



int min(int x,int y)
{
return (x<=y)?x:y;
```

}

- <u>Sorting:</u>

  Sorting is the process of arranging or arranging a list of elements from a collection. It is simply data storage in sorted order. Sorting is possible in both ascending and descending order. It arranges the data in a logical order, making searching easier.

Sorting can be done in a variety of ways, including the following:

1) Insertion Sort
2) Selection Sort
3) Bubble Sort
4) Quick Sort
5) Radix Sort
6) Merge Sort

1)Insertion Sort:

Insertion sort is a simple sorting algorithm that works in the same way that you would sort playing cards in your hands. The array is divided into two parts: sorted and unsorted. Values from the unsorted part are selected and assigned to the appropriate position in the sorted part.

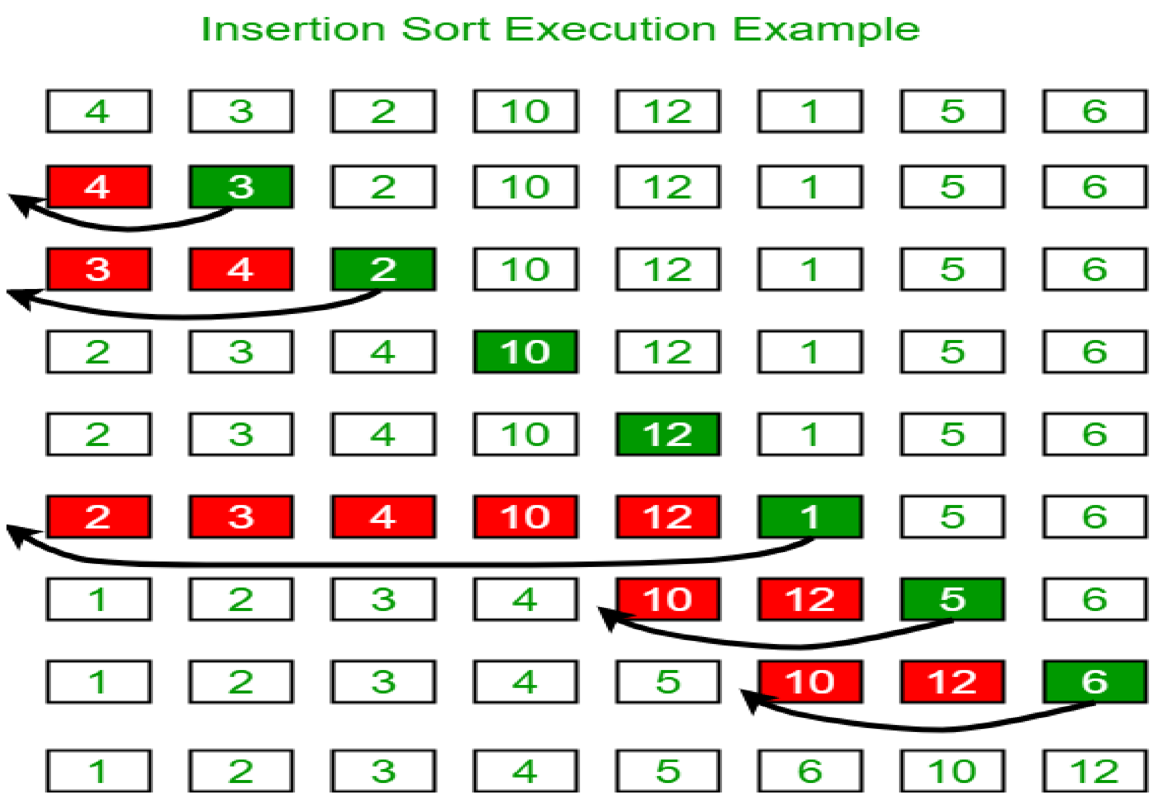
Algorithm

To sort an array of size n in ascending order, use the following syntax:

1: Iterate through the array from arr[1] to arr[n].

2: Contrast the current element (key) with its forerunner.

3: If the key element is smaller than the one before it, compare it to the elements before it. To make room for the swapped element, move the larger elements one position up.



Complexities of Insertion sort:

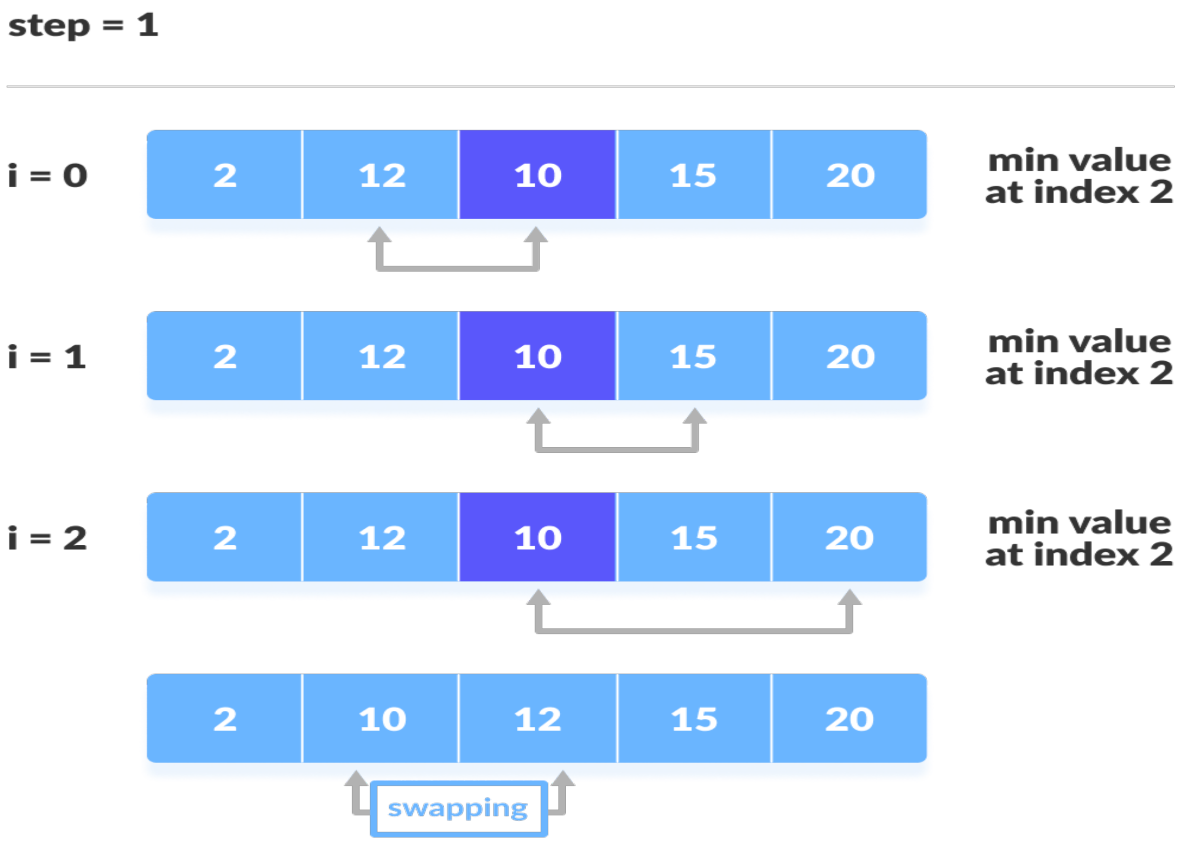
Best Case = O(n)
Average Case =O(n^2)
Worst Case = O(n^2)

Program For Insertion Sort:

```
#include<stdio.h>
void insertion_sort(int [],int);
int main()
{
int arr[]={12,11,13,8,3,7};
int n=sizeof(arr)/sizeof(arr[0]);
insertion_sort(arr,n);

for(int i=0;i<n;i++)
{
printf("%d  ",arr[i]);
}

}
void insertion_sort(int arr[],int n)
{
int i,j,key;
for(i=0;i<n;i++)
{
key=arr[i];
j=i-1;

while(j>=0 && arr[j]>key)
{
arr[j+1]=arr[j];
j=j-1;
}

arr[j+1]=key;
}
}
```

2)selection sort:

The selection sort algorithm sorts an array by repeatedly finding the smallest element (in ascending order) from the unsorted part and inserting it at the start. In a given array, the algorithm keeps two subarrays.
1) The subarray that has already been sorted.
2) The remaining unsorted subarray
Every iteration of selection sort selects the smallest element (in ascending order) from the unsorted subarray and moves it to the sorted subarray.



Complexity of Selection sort:

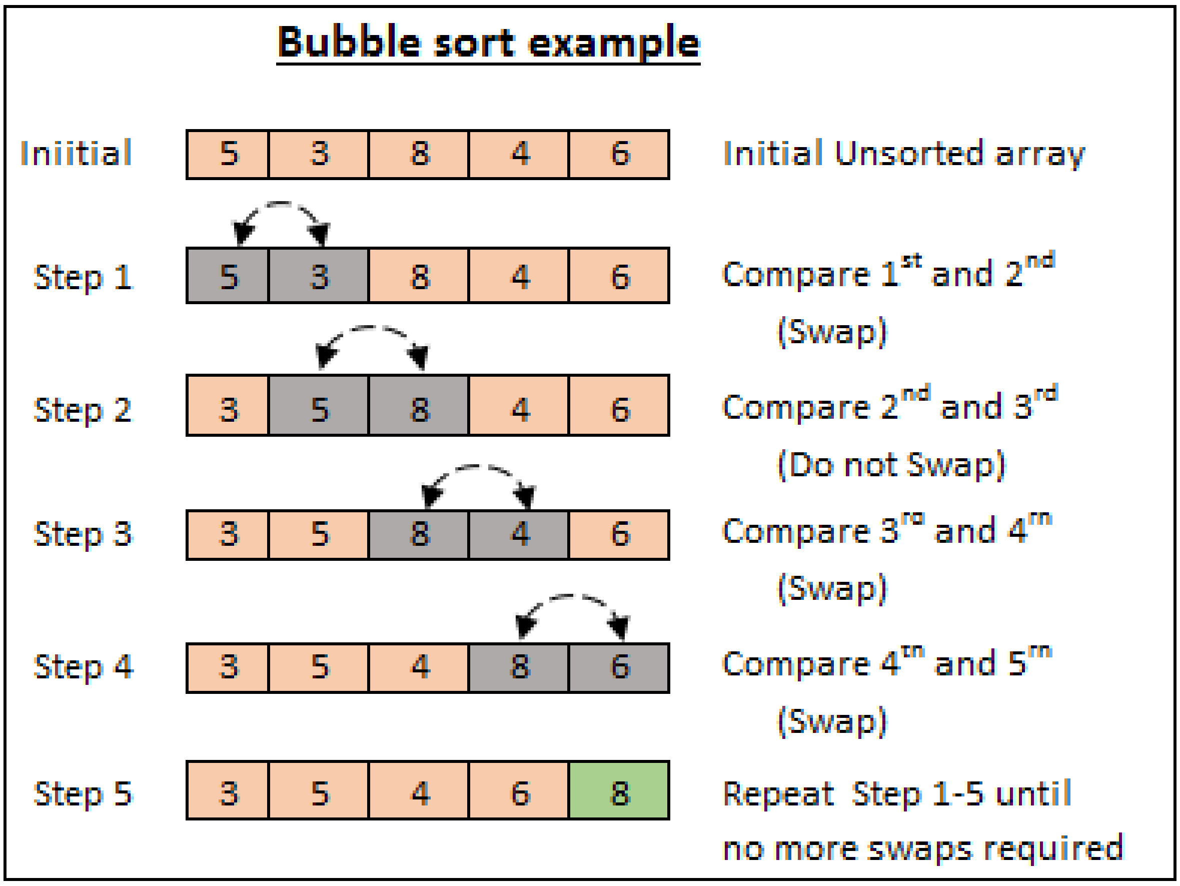
Best Case: O(n^2)
Average Case: O(n^2)
Worst Case: O(n^2)

Program For Selection Sort:

```
#include<stdio.h>
int main()
{
	 int i,j,min,temp;
	int arr[]={2,13,5,1,0};
	int n=sizeof(arr)/sizeof(arr[0]);

	for(i=0;i<n-1;i++)
	{
		min=i;
		for(j=i+1;j<n;j++)
		{
			if(arr[j]<arr[min])
			{
				min=j;
			}
		}
		temp=arr[min];
		arr[min]=arr[i];
		arr[i]=temp	;
	}

	printf("sorted array \n");
	for(i=0;i<n;i++)
	{
		printf("%d  ",arr[i]);
	}
	return 0;
}
```

3)Bubble Sort:

Bubble Sort is the most basic sorting algorithm, and it works by repeatedly swapping adjacent elements if they are out of order.


Bubble sort example
Iniitial 5 3 8 4 6 Initial Unsorted array
Step 1 5 3 8 4 6 Compare 1st and 2nd (Swap)
Step 2 3 5 8 4 6 Compare 2nd and 3rd (Do not Swap)
Step 3 3 5 8 4 6 Compare 3rd and 4th (Swap)
Step 4 3 5 4 8 6 Compare 4th and 5th (Swap)
Step 5 3 5 4 6 8 Repeat Step 1-5 until no more swaps required

Complexity of Bubble Sort:

Best Case: O(n)
Average Case: O(n^2)
Worst Case: O(n^2)

Program For Bubble Sort:

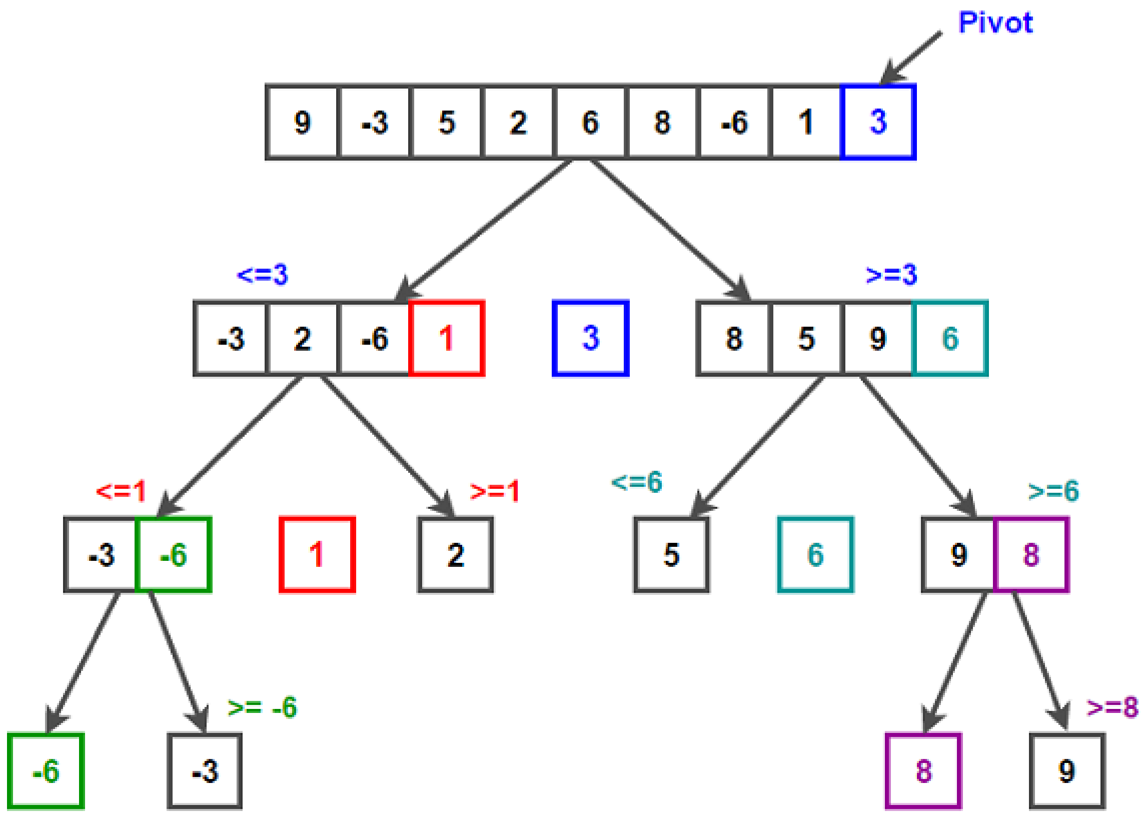
```
#include<stdio.h>
#define size 50
 int main()
 {
        int a[size];
        int i,j,n,temp;
        printf("enter the value of n:\n");
        scanf("%d",&n);
        printf("enter the array element");
        for(i=0;i<n;i++)
        {
                scanf("%d",&a[i]);
        }
        for(i=0;i<n;i++)
        {
                for(j=0;j<(n-i-1);j++)
                {
                        if(a[j]>a[j+1])
                        {
                                temp=a[j];
                                a[j]=a[j+1];
                                a[j+1]=temp;
                        }

                }
        }

        printf("sorted array");
        for(i=0;i<n;i++)
        {
                printf("%d",a[i]);
        }
 }
```

4)Quick Sort:

- Quick sort is also known as partition Exchange Sort.
- Quick sort is divides the main lists into two lists by choosing pivot element.
- Pivot element you can choose as following ways:
  Pivot element as a first element of array.
  Pivot element as last element of array.
  Pivot element as medium element of array.

  Complexity of quick sort is:

  Best case: O(n log n)
  Average Case: O(n log n)
  Worst Case:O(n^2)



Program for quick sort:

```
#include<stdio.h>
void quicksort(int number[25],int first,int last){
int i, j, pivot, temp;

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

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

temp=number[pivot];
number[pivot]=number[j];
number[j]=temp;
quicksort(number,first,j-1);
quicksort(number,j+1,last);

}
}

int main(){
int i, n, number[25];

printf("How many elements are u going to enter?: ");
scanf("%d",&n);
printf("Enter %d elements: ", n);
for(i=0;i<n;i++)
scanf("%d",&number[i]);

quicksort(number,0,n-1);

printf("Order of Sorted elements: ");
for(i=0;i<n;i++)
printf(" %d",number[i]);

return 0;
}
```

5)Radix Sort:

- Radix sort generally used for sorting files with very large records and small keys.
- Radix sort works by sorting each digit from least significant digit to most significant digit.

Complexity of Radix Sort:
Best Case: O(kn)
Average Case: O(kn)
Worst Case : O(kn)

Algorithm:

Step 1: In first pass/iteration the elements are picked up and kept in various buckets checking their unit digits.

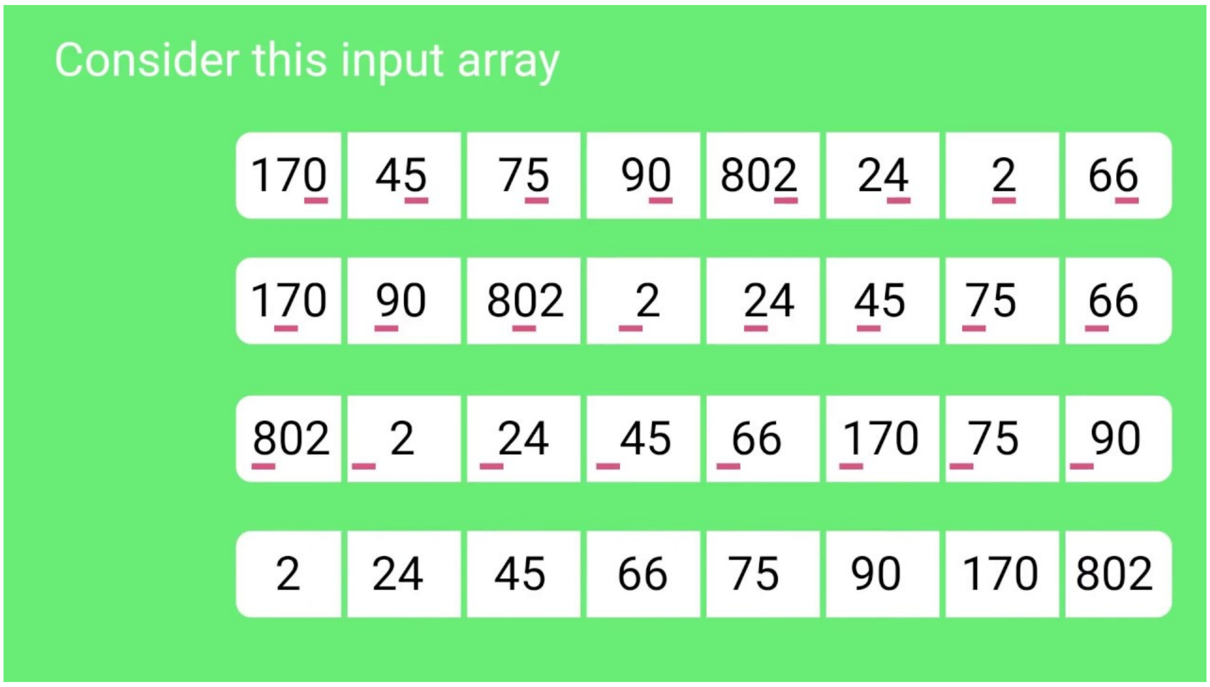
Step 2: The element are collect from bucket 0-9 and again they are given as input for sorting.

Step 3: In 2nd pass / iteration , the digits which are in ten’s place are sorted.

Step 4: Repeat step 2

Step 5: In 3rd pass/iteration the digits which are in the 100’s place are sorted.

Step 6: Repeat step 2



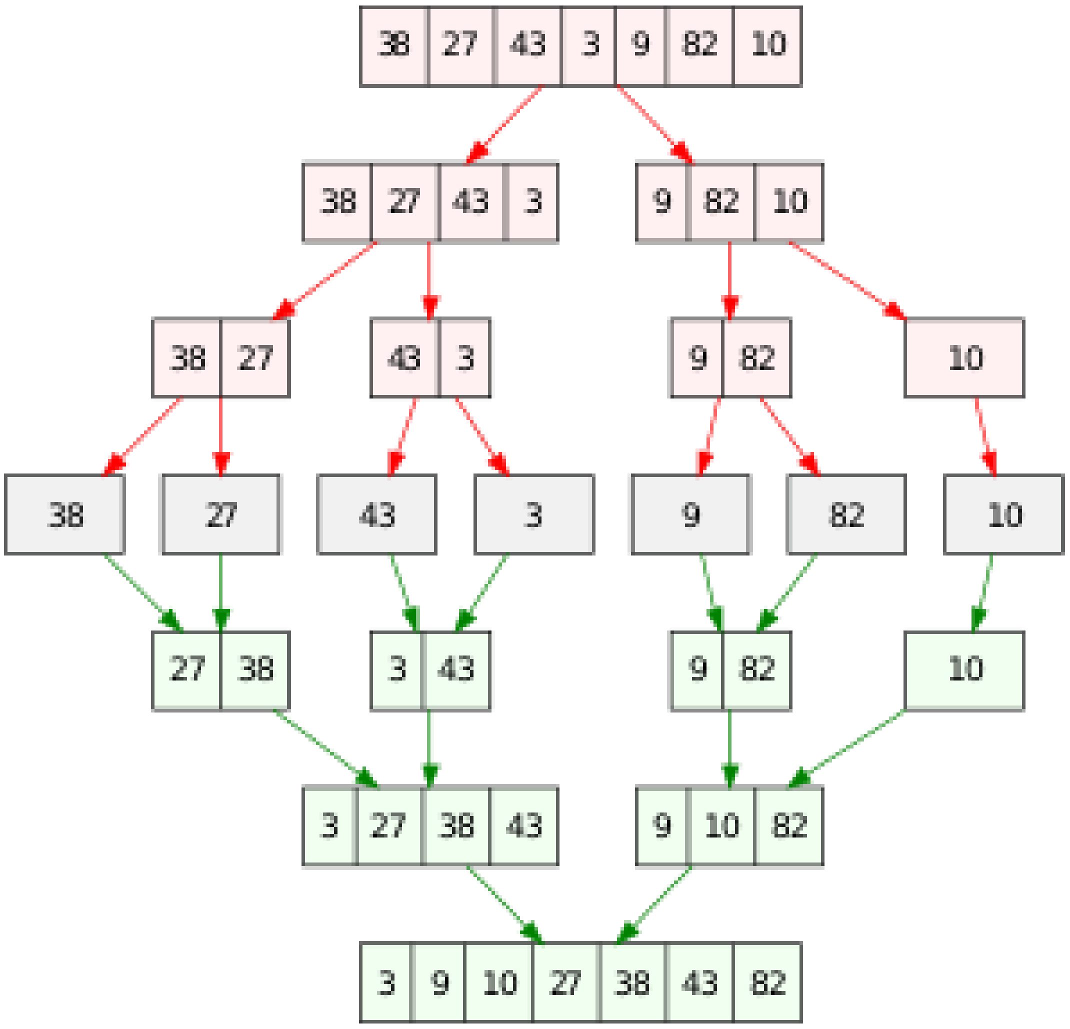
Program for radix sort:

```
#include<stdio.h>
int getMax(int arr[], int num)
{
int x = arr[0];
int a;
for (a= 1; a < num; a++)
if (arr[a] > x)
x = arr[a];
return x;
}
void countSort(int arr[], int num, int exp)
{
int output[num+1]; // output array
int a, count[10] = { 0 };
for (a = 0; a < num; a++)
{
count[(arr[a] / exp) % 10]++;
}
for (a = 1; a < 10; a++)
{
count[a] += count[a- 1];
}
for (a = num - 1; a >= 0; a--)
{
output[count[(arr[a] / exp) % 10] - 1] = arr[a];
count[(arr[a] / exp) % 10]--;
}
for (a = 0; a < num; a++)
arr[a] = output[a];
}


void radixsort(int arr[], int num) {
int m = getMax(arr, num);
int exp;
for (exp = 1; m / exp > 0; exp *= 10)
countSort(arr, num, exp);
}
void print(int arr[], int num) {
int a;
for (a = 0; a< num; a++)
printf("%d ", arr[a]);
}
int main()
{
int arr[] = { 200,84,02,96,52,46,150,63 };
int num = sizeof(arr) / sizeof(arr[0]);
radixsort(arr, num);
print(arr, num);
return 0;
}
```

6)Merge Sort:

The Merge Sort algorithm is a Divide and Conquer algorithm. It divides the input array in half, calls itself for each half, and then merges the two sorted halves.

Complexity of Merge Sort: O(n log n)



Programe For Merge Sort:

```
#include <stdio.h>
void mergesort(int a, int b, int arr[], int aux[])
{
   if (b<= a)
```

```
{
    return;
 }
 int mid = (a + b) / 2;
 mergesort(a, mid, arr, aux);
 mergesort(mid + 1, b, arr, aux);
 int left = a;
 int right = mid + 1;
 int k;
 for (k = a; k <= b; k++)
{
    if (left == mid + 1)
    {
        aux[k] = arr[right];
        right++;
    }
      else if (right == b + 1)
     {
        aux[k] = arr[left];
        left++;
    }
     else if (arr[left] < arr[right])
   {
        aux[k] = arr[left];
        left++;
    }
    else
    {
        aux[k] = arr[right];
        right++;
    }
 }

 for (k = a; k <= b; k++)
{
    arr[k] = aux[k];
 }
```

```
}

int main()
{
  int arr[100], aux[100], num, a, d;
  printf("Enter value of num:\n");
  scanf("%d", &num);
  printf("Enter the elements in an array:\n", num);
  for (a = 0; a < num; a++)
    scanf("%d", &arr[a]);
  mergesort(0, num - 1, arr, aux);
  printf("Sorted array:\n");
  for (a = 0; a<num; a++)
     printf("%d", arr[a]);
  return 0;
}
```