Graph:A graph is a data structure that describes a binary relation between elements and has a webby looking graphical representation. Graph plays a significant role in solving a rich class of problems.**Definition:**A graph G = (V, E) consists of a finite set of non-empty set of vertices V and set of edges E.V = { v_1, v_2, …, v_n},E = {e_1, e_2, …e_n}.Each edge e is a pair (v, w) where v, w ϵ V. The edge is also called arc. The vertices are represented by points or circles and the edges are line segments connecting the vertices. If the graph is directed, then the line segments have arrow heads indicating the direction.
Directed Graphs: If every edge (i, j), in E(G) of a graph G is marked by a direction from i to j, then the graph is called directed graph. It is often called diagraph. In edge e = (i, j), we say, e leaves i and enters j. In diagraphs, self loops are allowed. The indegree of a vertex v is the number of edge entering v. The outdegree of a vertex v is the number of edges leaving v. The sum of the indegrees of all the vertices in a graph equals to the sum of outdegree of all the vertices. $\sum_{i=0}^{n} d_{in}(v_i) = \sum_{i=0}^{n} d_{out}(v_i)$ $\quad n = \text{no. of vertices}$

Undirected Graph: If the directions are not marked for any edge, the graph is called undirected graph. The graphs G_1, G_2, G_3 are undirected graphs. In an undirected graph, we say that an edge e = (u, v) is incident on u and v (u and v are connected).
Undirected graph don't have self loops. Incidence is a symmetric relation i.e. if e = (u, v)
Then u is a neighbor of v and vice versa. The degree of a vertex, d(v), is the total number of edges incident on it.

$$\sum_{i=0}^{n} d(v_i) = 2|E|$$

The sum of the degrees of all the vertices in a graph equals twice the number of edges if d(v) = 0, v is isolated. If d(v) = 1, v is pendent.

Weighted Graphs:A graph is said to be weighted graph if every edge in the graph is assigned some weight or value. The weight is a positive value that may represent the cost of moving along the edge, distance between the vertices etc.The two vertices with no edge (path) between them can be thought of having an edge (path) with weight infinite.
Adjacent and Incident: Two vertices i and j are called adjacent if there is an edge between the two. Eg. The vertices adjacent to vertex A, in the fig below are B, C, D. The adjacent vertices of C are A and D.If e(i, j) is an edge on E(G), then we say that the edge e(i, j) is incident on vertices i and j. Eg. in the above figure, e_4 is incident on C and D. If (i, j) is directed edge, then i is adjacent to j and j is adjacent from i.
Path:A path is a sequence or distinct vertices, each adjacent to the next. For e.g. the sequence of vertices e_1, e_3, e_4 (i.e (B,A), (A,C), (C,D)) of the above graph form a path from B to D. The length of the path is the number of edges in the path. So, the path

from B to D has length equal to three.In a weighted graph, the cost of a path is the sum of the costs of its edges. Loops have path length of 1.**Cycle:**A cycle is a path containing at least three vertices such that the last vertex on the path is adjacent to the first. In the above graph G_6, A, C, D, A is a cycle.**Connected:**Any graph is *connected* provided there exists a path (directed or undirected) between any two nodes. A digraph is said to be *strongly connected* if, for any two vertices there is a direct path from i to j.A digraph is said to be weakly connected if, for any two vertices i and j, there is a direct path from i to j or j to i. or A *weakly-connected graph* is a directed graph for which its underlying undirected graph is connected.**Complete graph**:A *complete graph* is a graph in which there is an edge between every pair of vertices.**Trees and Graph:**A tree is a connected graph with no cycle. A tree has **|E| = |V| - 1** edges. Since, it is connected, there is a path between any two vertices.**Representation of Graph:**A graph can be represented in many ways. Some of them are described in this section.**Adjancency matrix:***Adjancency matrix* A for a graph G = (V, E) with n vertices, is n x n matrix, such that A_{ij} = 1, if there is an edge from v_i to v_j A_{ij} = 0, if there is no such edge**.**It is a simple way to represent a graph, but it has following disadvantages:it takes $O(n^2)$ space**,**it takes $O(n^2)$ time to solve most of the problems.**Adjacency List Representation:**The n rows of an adjacency matrix can be represented as n linked lists. This is one list for each node in a graph. Each list will contain adjacent nodes. Each node has two fields, *Vertex* and *Link*. The Vertex field of a node p will contain the nodes that are adjacent to the node p.**Graph Traversals**

A graph traversal means visiting all the nodes of the graph. Basically, there are two methods of graph traversals.Breadth First Traversal,Depth First Traversal.**Breadth First Traversal:**The *Breadth First Traversal* begins with a given vertex and then it next visits all the vertices adjacent to v, putting the vertices adjacent to these in a waiting list to be traversed after all vertices adjacent to v have visited. Breadth First Traversal uses queue. **Algorithm:** void BFTraversal(Graph G)

```
{ for each vertex v in G
  visited[v] = false;
  for each vertex v in G
      if(visited[v]==false)
        { enqueue(V, Q); do
                { v=dequeue(q)
                visited[v]=true;
                        visited(v);
                        for each vertex w adjacent to v
                        if(visited[w]==false) enqueue(w, q);
                }while(isempty(Q)==false)
        }}
```

Depth First Traversal: The *Depth First Traversal* algorithm is roughly analogous to preorder tree traversal, the algorithm follows: suppose that the traversal has just visited a

vertex v, and let w_0, w_1, …w_k be the vertices adjacent to v next, visit w_0 and keep w_1, …w_k waiting after visiting w_0, we traverse all the vertices to which it is adjacent before returning to traverse w_1, …w_k Depth first traversal uses stack to store the adjacent vertices but one.

Algorithm: void DFTraversal(Graph G)

```
{for each vertex v in G

Visited[v]=false;

for each vertex v in G

      If(visited[v]==false)

      Traverse(v);

}
void Traverse(Vertex v)

{
    Visited[v]=true;
    Visited(v);
    For each vertex w adjacent to v
        {
    If(visited[w]==false)
    Traverse(w);
        }
}
```

Subgraph: Let H be a graph with vertex set v(H) and edge set E(H) and, similarly, let G be a graph with vertex set V(G) and edge set E(G). Then H is said to be a subgraph of G,written as H ≤ G if V(H) ≤ V(G) and E(H) ≤E(G).**Spanning Tree:**Let a graph G=(V,E) be a graph. If T is a subgraph of G and contains all the vertices but no cycles/circuits, then 'T' is said to be spanning tree of G.In other words, the spanning tree of an undirected graph G is the free tree formed from graph edges which connects all the vertices of G.

Minimum Spanning Tree: The cost of a spanning tree of a weighted, undirected graph is the sum of the costs (weights) of the edges in tree.The minimum spanning tree of a weighted undirected graph is the spanning tree that connects all the vertices in G at lowest cost. So, if T is the minimum spanning tree of graph G and T' is any other spanning tree of G then, W(T) ≤ W(T').There are several algorithms available to determine the *minimal spanning tree* of a given weighted graph.**Kruskal's Algorithm**:To determine minimum spanning tree consider a graph G with n vertices.Step 1: List all the edges of the graph G with increasing weights Step 2: Proceed sequentially to select one edge at a time joining n vertices of G such that no cycle is formed

until n-1 edges are selected.Step 3: Draw the n-1 edges that were selected forming a minimal spanning tree T of G.**Round Robin Algorithm:**This method provides better performance when the number of edges is low. Initially each node is considered to be a partial tree. Each partial tree is maintained in a queue Q. Priority queue is associated with each partial tree, which contains all the arcs ordered by their weights.**Shortest Path Algorithm:**There are many problems that can be modeled using graph with weight assigned to their edges. Eg, we may set up the basic graph model by representing cities by vertices and flights by edges. Problems involving distances can be modeled by assigning distances between cities to the edges. Problems involving flight time can be modeled by assigning flight times to edges. Problem involving fares can be modeled by assigning fares to the edges.Thus, we can model airplane or other mass transit routes by graphs and use shortest path algorithm to compute the best route between two points.Similarly, if the vertices represent computers; the edges represent a link between computers; and the costs represent communication costs, delay costs, then we can use the shortest-path algorithm to find the cheapest way to send electronic news, data from one computer to a set of other computers.Basically, there are three types of shortest path problems:**Single path**: Given two vertices, s and d, find the shortest path from s to d and its length (weights).**Single source**: Given a vertex, s find the shortest path to all other vertices.**All pairs**: Find the shortest path from all pair of vertices.**Dijkstra's Algorithm to find the shortest path:**Mark all the vertices as unknown**;**for each vertex v keep a distance d_v from source vertex s to v initially set to infinity except for s which is set to $d_s = 0$**;**repeat these steps until all vertices are known**:**select a vertex v, which has the smallest d_v among all the unknown vertices**;**mark v as known**;**for each vertex w adjacent to v;if w is unknown and $d_v + \text{cost}(v, w) < d_w$ update d_w to $d_v + \text{cost}(v, w)$.**Greedy Algorithm:**One of the simplest approaches that often leads to a solution of an optimization problem. This approach selects the best choice at each step, instead of considering all sequences of steps that may lead to an optimal solution. Algorithm that make what seems to be the "best" choice at each step are called greedy algorithm.**Transitive Closure**:In all shortest path, we noticed that every time we have to determine whether there is a path from vertex i to j. This can be accomplished by observing a matrix, say $d^{(n)}$. The path exists if and only if the entry does not have the value infinity. In matrix $d^{(n)}$, we consider only path (not the weight). Thus, (d^k) can be regarded as Boolean matrix.If there is a path from i to j, then (i, j)th entry in d^k is 1 (which means true) otherwise the value is 0. If 'A' is the adjacency matrix of graph 'G', then the transitive closure is defined to be a Boolean matrix with the above property.A^+ Matrix has the following property:1 if there is a path from i to j 0 otherwise.The matrix A^+ is called the transitive closure. The Transitive closure helps to determine which pair or vertices in a digraph are connected by a path. It takes $O(n^4)$ time complexity.To find at least one path between two nodes, we can check adj1, adj2, adj3, …,$adj_{maxnodes1}$. If any of the adjacency matrix give true value for node 'i' and 'j', then there exist at least a path between nodes 'i' and 'j'. Let 'm' is the highest order of length and 'n' is the no. of total nodes. If the graph supports

m>n, then at least one of the node is visited twice or more times in the path. It means, we can discard the cycling path from the node to the same node. The removal process is repeated until no node is repeated in the path. This ensures that the paths that exist in the graph be less than or equal to n. The final matrix path in which no path exists that has length greater than 'n' is known as transitive closure of the matrix adj.**Warshall's Algorithm:**Transitive closure method of finding path between nodes is quite inefficient because it processes the adjacency matrix lots of times. An Efficient method for calculating transitive closure is by Warshall's algorithm whose time complexity is only $O(n^3)$.**Warshall's Algorithm** procedure warshall (M_R : n x n zero one matrix){D = M_R

for k = 1 to n
begin for i = 1 to n begin
for j = 1 to n
$d_{ij} = d_{ij} \vee (d_{ik} \wedge d_{kj})$
end
}

D = [d_{ij}] is the transitive closure.**Topological Sorting:**A topological sort is an ordering of vertices in a directed acyclic (graph with no cycle) graph (dag), such that if there is a path from v_i to v_j, then vi appears before v_j in the ordering. A topological sort of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.**Applications:**Consider the course available at university as the vertices of a directed graph; where there is an edge from one course to another if the first is a prerequisite for the second.A topological ordering is then a listing of all the courses such that all prerequisite for a course appear before it does.**Algorithms to find out topological ordering**:**Breadth First Ordering:**In Breadth First Topological ordering of a directed graph with no cycles, we start by finding the vertices that should be first in the topological order and then apply the fact that every vertex must come before its successors in the topological order. The vertices that come first are that are not successors of any other vertex.Determining the Breadth First Topological ordering:Fill the information from the graph into the table(Vertices, Predecessors and Predecessor count)**,**Visit the vertex that have *Predecessor count* zero (0).Decrement the *Predecessor count* value by 1 where ever the visited vertex appears in the *Adjacent Predecessor vertices* column.Repeat until all the vertices has been visited.**2. Depth First Ordering:**Start by finding a vertex that has no successors and place it last in the order. After, we have, by recursion placed all the successors of a vertex into the topological order, then place the vertex itself in the order (before any of its successors).Determining the Depth First Topological ordering:Fill the information from the graph into the table(Vertices, Successors and Successor count)**,**Push all the vertices into stack and visit the vertex that is on the top.Decrement the *Successor count* value by 1 where ever the visited vertex appears in the *Adjacent Successor vertices* column.Repeat until all the vertices has been visited.

Searching is a process of finding an element within the list of elements stored in any order. Searching is divided into two categories. They are **Linear** Search and **Binary** Search. Linear Searching is the basic and simple method of searching. Binary Search takes some less time to search an element from the **sorted list** of element. we can say that binary search method is more efficient then the linear search only drawback is that binary search work on the sorted list where there is no prerequisite for the linear search. **Types of Searching** Linear (Sequential) Searching, Binary Searching.**Linear or sequential searching:** In linear search, we access each element of an array one by one sequentially and see whether it is desired element or not. A search will be unsuccessful if all the elements are accessed and the desired element is not found. Therefore linear search can be defined as the technique which traverses the array sequentially to locate the given item.**Algorithm:** Let '**a**' be the linear array with '**n**' elements, and '**item**' is an element to be searched. The algorithm finds the location '**loc**' of item in '**a**' or give the failure message.Read the item to be searched.[Initialize counter] set **loc= -1,j=0**[Search for item]Repeat while j<n If a[j] = item Set loc = j Break Else Set j = j+1 Wend [Successful] if **loc>=0** Print the searched value's (**item's**) position (**loc**) [Unsuccessful] else Print the searching **item is not found.** [END]

Binary Search: Binary search is an extremely efficient algorithm. This search technique searches the given item in minimum possible comparisons. To do the binary search, first we have to sort the array elements. The logic behind this technique is given below.First find the middle element of the array. Compare the middle element with an item.There are three cases.If it is a desired element then search is successful,If it is less than the desired item then search only in the first half of the array.If it is greater than the desired item, search in the second half of the array.Repeat the same steps until an element is found or search area is exhausted.In this way, at each step we reduce the length of the list to be searched by half.**Requirements** The list must be ordered,Rapid random access is required, so we cannot use binary search for a linked list.**Algorithm** Let **'a'** be the array of size **'Maxsize'** and **'LB', 'UB'** and **'mid'** are variables to denote first, last and middle location of a segment. This algorithm finds the location **'Loc'** of '**item**' in array '**a**' or return fail(sets loc=NULL).[Initialize segment variables] set beg = LB, end = UB and mid = int((beg + end)/2) Repeat steps 3 and 4 while beg<=end and a[mid]!=item if item<a[mid],then set end=mid – 1 Else set beg=mid + 1 [End if] set mid=int((beg + end)/2)

if a[mid]=item then set loc=mid
print the value and its position else
set loc=NULL
print search unsuccessful
[End if]
Exit

Sequential search efficiency: The number of comparisons of keys done in sequential search of a list of length ***n*** is Unsuccessful search: ***n*** comparisons **,**Successful search, best case: **1** comparison ,Successful search, worst case: ***n*** comparisons , Successful search, average case: **(*n* + 1)/2** comparisons,In any case, the number of comparison is ***O*(*n*)**

Binary search efficiency:**In all cases, the no. of comparisons in proportional to n.** Hence, no. of comparisons in binary search is *O* (log n), where n is the no of items in the list, Obviously binary search is faster then sequential search, but there is an extra overhead in maintaining the list ordered,For small lists, better to use sequential search
Binary search is best suited for lists that are constructed and sorted once, and then repeatedly searched **Binary Search Procedure**

```
void binsrch( int a[], int beg, int end, int item)
{ int mid; int count = 0; mid =
        (beg + end)/2; count ++;
        while(beg <= end && item != a[mid])
        {If(item < a[mid])
                        end = end + 1;
                else
                        beg = beg + 1;
                mid = (beg + end)/2; count ++;
}if(item == a[mid])
{ printf("\nSearch Successful!!!\n It took %d iterations to find this item", cnt); printf("\nThe position is %d",mid);
        }
        else printf("\n Search Unsuccessful");
        getch();
}
```

Hashing: "Derive a number from the key information given to you and use that number to access all information related to the key". This is the basic principle of hashing. This way, we can access any record in a specific time without making any comparison, irrespective of the number of record in the file. Hashing is a scheme of searching that will use some function say hash to directly find out the location of the record in a constant search time, no matter where the record is in the file. In order words the process of mapping large amount of data into a smaller table is called hashing. This searching scheme is easy to program compared to trees. But it is difficult to expand, since it is based on arrays.**Hash Table:** A hash table is simply an array that is address via a hash function. It is a data structure made up of A table of some fixed size to hold a collection of records each uniquely identified by some key,A function called hash function that is used to generate index values in the table. **Hash Function:** The basic idea in hashing is the transformation of a key into the corresponding location in the hash table. This is done by a hash function. A hash function can be defined as a function that takes key as input and transforms it into a hash table index usually denoted by H. Some popular hash functions are Folding, Mid square method,Division method. **Hash of Key** Let H be a hash function and k is a key then H(k) is called hash-of-key. The hash-of-key is the index at which a record with the key values k must be kept.**Popular Hashing Functions**:The two principle criteria in selection of a

good hash function are:It must be very easy and quick to compute hash value and It must minimize collision.**What is collision?** There are a finite number of indices in a table. But there are large numbers of keys so it is clearly impossible to get two different indexes for two distinct keys. A situation in which two different keys k1 and k2 hash to the same index of the table i.e. h(k1) = h(k2), it is called collision or hash clash. We have, in general following methods.Folding method,Mid square method,Division method.**Folding method** Eg. Let the keys be of four digits, chopping the key into two parts and adding yields Hash(5421) = 54 + 21 = 75 So 75 is the index at which w e should store or retrieve record with key 5421. **Mid square method**: The key is squared. We defined the hash function in this case as Hash (key) = p; Where p is obtained by deleting digits from both ends of $(key)^2$. We emphasize that the same positions of $(key)^2$ must be used for all the keys.**Division Remainder method**
Convert the key to an integer, divide by the size of the index range and take the remainder as the result.**Collision Resolution Techniques:** When a collision occurs, alternative locations in the table are tried until an empty location is found. Looking for the next available position is called probing.**Techniques are as follows:**Linear probing,Quadratic probing,Double hashing,Chaining.**Linear probing:** A simple approach to resolve collision is to store the colliding element in the next available space. This technique is known as linear probing.**Disadvantages of linear probing:** One main disadvantage of linear probing is that, records tend to cluster, that is, the records appear next to one another when the table is about half full.
Clustering occurs when a hash function is biased towards the placement of keys into a given region within the storage space. When the linear method is used to resolve collisions, this clustering problem is compounded, because keys that collide are loaded relatively close to the initial collision point.**Chaining:** It is another technique to deal with collision. In this method, for each location in the table, we keep a linked list of records that hash to the same index. The hash table contains pointers to linked list nodes; we can view these as the head pointers. Each time a record is inserted, it is added to the list at the location given by the hash function. When a collision occurs, the record is simply added to the list at the collision site.**Quadratic Probing:** This method makes an attempt to correct the problem of clustering with linear probing. It forces the problem key to move quickly a considerable distance from the initial collision. When a key value hashes and collision occurs for key, this method probes the table location at$(h(k) + 1^2)$ % table-size,$(h(k) + 2^2)$ % table-size,$(h(k) + 3^2)$ % table-size and so on.**Double Hashing**:When collision occurs, a new hash function is defined.$H_2(key) = R - (key\%R)$Where R is the nearest prime number from its table size.**Rehashing:** If at any stage the hash table become almost full (when packing density is more than 70%) then it will be difficult to find the free slot which will increase execution time. If the hash function produces collisions, we create a new hash table of double size. We may use the old hash value on input to a rehash function and compute a new hash value. For rehashing with linear probing, we can use the rehash function as:(old hash value + constant) % array-size Where constant and array size are relatively prime, i.e, the largest number that divides both of them is 1. For eg, given 100 slots array, we may use constant in rehash function:
(old hash value + 3) % 100.**Procedures of some Hash Resolutions:**

An array is declared and each cell is assigned -1 to denote free cell.

```
int keys[10]; for (int i=0;
i<10; i++)

keys[i] = -1;

voidlinear_probing(int keys,
int target)

{ int rem; rem = target%10; if(keys[rem]
        == -1; keys[rem] = target;
        else
        { while(keys[rem] != -1)
                {If(rem >= 10) rem = 0;
                        rem++; }
                keys[rem] = target;
        }
}
```

void double_hashing(int keys[], int target)

```
{ int rem; rem = key%10;
        if(keys[rem] == -1) keys[rem] =
                target;
        else
        {Rem = 7 – (key%7)
                while(keys[rem] != -1)
                { if(rem >= 10) rem = 0;
                        rem++; }
                keys[rem]=target;
        }
```

} **void quadratic_probing(int keys[], int target)**

```
{ int rem;rem = key%10;

        if(keys[rem] == -1) keys[rem] =
        target;
        else
        { int x = 0; int h = rem;
                while(keys[rem] != -1)
                { x++; int rem1 = h + pow(x,2); rem =
                        rem1%10;
                }
```

```
                keys[rem] = target;
        }
}
```

Procedure for chaining: struct chain

```
{ int data; struct chain *next;
};
```

typedef struct chain node; node *table[10]; Here, initially all the node pointers of the table are initialized to NULL. **void chaining(int key)**

```
{ int rem; rem = key%10;
        node *p;
        p = (node *)malloc(sizeof(node));
        p->data = key; p->next =
        NULL; if(table[rem] ==
        NULL) table[rem] = p;
        else
        { node *temp; temp = table[rem];
                while(temp->next != NULL) temp
                = temp->next;
                temp->next = p;
        }
}
```

Sorting: sorting is one of the most common ingredients of programming systems. The process of rearranging the items in a list according to some linear order is termed as sorting.**Types of sorting:Internal sorting :** Records to be sorted are in main memory.**External sorting:** Records to be sorted, or some of them are kept in auxiliary storage(disk/tape).**Stable Sort:**It is possible for two records in a list to have the same key.A sorting algorithm is *stable* if for all records *i* and *j* such that *k*[*i*] equals *k*[*j*], if *r*[*i*] preceded *r*[*j*] in the original list, *r*[*i*] precedes *r*[*j*] in the sorted list.But the stable sort keeps track of original pattern unless specified.**Sorting algorithm:Exchange sort:**Comparison based. The basic idea is to compare two elements; if out of order, swap them or move one of the elements. E.g. Bubble sort, Quick sort.**Selection sort:**An element is selected and is placed in its correct position. e. g .Selection sort, Heap sort.**Insertion sort:**Sorts by inserting an element into a sorted list. E. g. insertion sort, merge sort.**Bubble sort:** Pass through the list sequentially several times. At each pass, each element in the list is compared to its successor and they are interchanged if they are not in proper order. At each pass, one element will be in its proper positions. In general, A[n-i] will be in its place after pass i. Since each pass place a new element in its proper position, a total of N-1 passes are required for a list of N elements. Also, all the elements in positions greater than or equal to N-i are already in proper position after pass i, so they need not be considered in succeeding passes.

Procedure: void bubblesort(int a[], int N)

```
{
```

```
int pass, j; for(pass=0; pass<N-1;
        pass++)
        {       for(j=0; j<N-pass-1; j++)
                        if(a[j]>a[j+1] swap(&a[j],
                                &a[j+1]);
        }}
```

Algorithm: Given a list A of size N, the following algorithm uses bubble sort to sort the list

- For ***pass* = 0** To ***N* – 2**
  - For ***j* = 0** To ***N* – *pass* – 2**
    - If ***A*[*j*] > *A*[*j* + 1]**
      Swap the elements ***A*[*j*]** and ***A*[*j* + 1]**
      End If
  - End For
- End For

Efficiency: This algorithm is good for small n usually less than 100 elements.**No. of comparisons**= $O(n^2)$**No. of Interchanges**:This cannot be greater than no. of comparisons,In the best case, there are no interchanges,In the worst case, this equals no of comparisons.The average and worse case running time of bubble sort is $O(n^2)$**.**It is actually the no of interchanges which takes up most time of the program's execution than the no of comparisons.When elements are large and interchange operation is expensive, it is better to maintain an array of pointers to the elements. One can then interchange pointers rather then the elements itself.**Insertion Sort**: Sorts a list of record by inserting new element into an existing sorted list. An initial list with only one item is considered to be sorted list. For a list of size N, N-1 passes are made, and for each pass the elements from a[0] through a[i-1] are sorted.Take the element a[i], find the proper place to insert a[i] within 0, 1, …, i-1 and insert a[i] at that place.To insert new item into the list:Search the position in the sorted sublist from last toward first,While searching, move elements one position right to make a room to insert a[i] ,Place a[i] in its proper place.**C-Procedure**:

```
void InsertionSort(int a[], int N){int i, j;

        int hold; /* the current element to insert */

        for (i = 1; i < N; i++) // Insert a[i] into the sorted list

        { hold = a[i]; /* hold the element to be inserted */
                for (j = i-1; j >= 0 && a[j] > hold; j--)        //Move right 1 position all
                        a[j+1] = a[j];                          //elements greater than hold

                a[j+1] = hold; /* Place hold in its proper place
        */ }}
```

Efficiency:No of comparisons: Best case: $n - 1$ Worst case: $n^2/2 + O(n)$ Average case: $n^2/4 + O(n)$.**No of assignments (movements)**:Best case: $2*(n-1)$ // moving from a[i] to hold and

back,Worst case: $n^2/2 + O(n)$,Average case: $n^2/4 + O(n)$.Hence running time of insertion sort is $O(n^2)$ in worst and average case and $O(n)$ in best case and space requirement is $O(1)$. **Advantages:**It is an excellent method whenever a list is nearly in the correct order and few items are removed from their correct location.Since there is no swapping, it is twice as faster than bubble sort.**Disadvantage:**It makes a large amount of shifting of sorted elements when inserting later elements.**Selection Sort:**The selection sort algorithm sorts a list by selecting successive elements in order and placing into their proper sorted positions.A list of size N require N-1 passes:For each pass I,Find the position of i^{th} largest (or smallest) element.To place the i^{th} largest (of smallest) in its proper position, swap this element with the element currently in the position of its largest (or smallest) element.

C – Procedure

```
void SelectionSort(int a[], int N)
{
        int i, j; int
        maxpos;
        for (i = N-1; i > 0; i--)          //Find the position of largest element from 0 to i
        { maxpos = 0; for (j = 1; j <= i; j++)
                if (a[j] > a[maxpos])
                maxpos = j;
                if(maxpos != i) swap(&a[maxpos], &a[i]);    //Place the ith largest element
        }                                                   // in its place
}
```

Efficiency:No of comparisons: Best, average and worst case: $n(n-1)/2$.**No of assignments (movements)**:Best, average and worst case: $3(n-1)$, (total $n-1$ swaps).If we include a test, to prevent interchanging an element with itself, the number of interchanges in the best case would be 0.Hence running time of selection sort is $O(n^2)$ and additional space requirements is $O(1)$. **Advantages:**It is the best algorithm in regard to data movement.An element that is in its correct final position will never be moved and only one swap is needed to place an element in its proper position.**Disadvantages**:In case of number of comparisons, it pays no attention to the original ordering of the list. For a list that is nearly correct to begin with, selection sort is slower than insertion sort.**Divide and Conquer Sorting Algorithms**:The idea of dividing a problem into smaller but similar subproblems is called *divide and conquer*.Divide and Conquer Sorting:Procedure :Sort(list) if (list has length greater than 1),Partition the list into two sublists lowlist, highlist;Sort(lowlist);Sort(highlist);Combine (lowlist, highlist);End If;End Procedure **Quick Sort:**It is the fastest known sorting algorithms used in practice.Divide the list into two sublists such that all elements in the first list is less than some pivot key and all elements in the second list is greater than the pivot key, and finally sort the sublists independently and combine them.**Algorithm:**If size of list *A* is greater than 1:Pick any element *v* from *A*. This is called the *pivot*.Partition the list *A* by placing *v* in some position *j*, such that o all elements before position *j* are less than or equal to *v* o all elements after position *j* are greater than or equal to

v.Recursively sort the sublists *A*[0] through *A*[*j*-1] and *A*[*j*+1] through *A*[*N*-1].Return *A*[0] through *A*[*j*-1] followed by *A*[*j*] (the pivot) followed by *A*[*j*+1] through *A*[*N*-1].

Quick Sort Code:

```
void QuickSort(int A[], int N)
{
        QSort(A, 0, N - 1);
}

void QSort(ItemType A[], int low, int high)
{ int pivotloc; if (low
        < high)
        { pivotloc = partition(A, low, high);
                QSort(A, low, pivotloc - 1);
                QSort(A, pivotloc + 1, high);
        }} int partition(int A[], int low, int high) { int down, up;
        int pivot;
        pivot = A[low];         /* choose first element as the
        pivot down = low;       /* Initialize pointers */ up =
        high;

        while (down < up)
        { while (A[down] <= pivot && down < high)          /* move right */
                down++;
                while (A[up] > pivot)                      /* move left */
                        up--;
                if (down < up)                  /* exchange element at up and down */
                        swap (&A[down], &A[up]); }
        swap (&A[low], &A[up]);                 /* Place pivot at its proper position
                                                */
  return up;                                    /* return the pivot location */
}
```

Description of partition procedure: Choose any element as the pivot, here we choose the first element as the first item in the list. Initialize two pointers, ***up*** and ***down*** to the upper bound (***low***) and lower bound (***high***) of the array; While ***down*** is left of ***up***, repeat these steps: Move ***down*** right, skipping over elements that are smaller than or equal to pivot. Move ***up*** left, skipping over elements that are larger than the pivot; When ***down*** and ***up*** have stopped, ***down*** is pointing at a large element and ***up*** is pointing at a small element; If ***down*** is left of ***up***, swap the elements at ***down*** and ***up*** (The effect is to push a large element to the right and a small element to the left);Swap the first element (***pivot***) with the element at ***up***. Return ***up*** as the pivot location.**Method for choosing pivot: First element**: Choose the first item in the list.

Random element: Choose any item form the list. Swap it with the first item to apply the algorithm. **Median**: Pick three elements randomly and use their median as pivot.
Efficiency:*No. of comparisons:* Average case: $O(n \log n)$ Worst case: $O(n^2)$.***No of interchanges (swaps)*** Average case: $O(n \log n)$ Worst case: $O(n^2)$.Hence, the time complexity of QuickSort is $O(n \log n)$ for average case and $O(n^2)$ for worst case.**Merge Sort:**
The merge sort also uses divide and conquer approach. It divides the list into sub lists. Then merge two sorted into a single list.**Algorithm outline**:If size of list is greater than 1 then divide the list into two sublists of sizes nearly equal as possible;recursively sort the sublist separately.Merge the two sorted sublists into a single sorted list End if.

C-Code:

```
void main()
{
 clrscr(); int n,i; int x[N]; int temp[N];
 printf("\nEnter no. of elements to sort:
 "); scanf("%d",&n); printf("\nEnter
 elements to sort:\n"); for (i = 0; i < n;
 i++) scanf("%d",&x[i]); //perform
 merge sort on array msort(x,temp,0,n-1);
void msort(int x[], int temp[], int left, int
right)
{ int mid;
 if(left<right)
 {

 mid = (right + left) / 2; msort(x,
  temp, left, mid); msort(x,
  temp, mid+1, right);

  merge(x, temp, left, mid+1, right);
 }
}
 printf("Sorted List \n");
 for (i = 0; i < n; i++)
 printf("%d\n", x[i]);
getch(); }
 { x[right] = temp[right];
  right = right - 1;
 }
}
```

```
void merge(int x[], int temp[], int left, int
mid, int right)
{ int i, lend, no_element, tmpos;

 lend = mid - 1; tmpos = left;
 no_element = right - left + 1;

 while ((left <= lend) && (mid <= right))
 { if (x[left] <= x[mid])
  { temp[tmpos] = x[left];
  tmpos = tmpos + 1; left =
  left +1; }
  else
  { temp[tmpos] = x[mid];
       tmpos = tmpos + 1;
       mid = mid + 1;
  }
 }

 while (left <= lend)
 { temp[tmpos] =
 x[left]; left = left + 1;
 tmpos = tmpos + 1; }
 while (mid <= right)
 { temp[tmpos] = x[mid]; mid
 = mid + 1; tmpos = tmpos +
 1; } for (i=0; i <= no_element;
 i++)
```

- **Efficiency:No. of Comparisons:**For all cases the number of comparisons is to be $O(n*\log n)$, the constant term is different for different cases. On average, it requires fewer than $n*\log n - n + 1$ comparisons.**No. of assignments:**For our implementation, it is twice the no of comparisons, merging in the temporary array and copying back to the original array which is still $O(n*\log n)$.**Space Complexity:**In contrast to other sorting algorithms, we have studied, Merge Sort requires $O(n)$ extra space for the temporary memory used while merging. Algorithm has been developed for performing in-place merge in $O(n)$ time, but this would increase the no of assignments If recursive version is used, additional space is required for the implicit stack, which is $O(\log n)$. Hence, Space complexity for Merge Sort is $O(n)$.**Notes on Merge sort:**Even though the worst-case running-time of Merge Sort is $\boldsymbol{O(n\log n)}$, it is **not** an algorithm of choice for sorting contiguous lists. Merge Sort can prove superior over other sorting algorithms when used with linked lists.**Binary Tree Sort:**General idea is to create a binary search tree and access the elements either in LVR and RVL for ascending and descending order.But, in case of imbalanced tree (right skewed and left skewed), the search time goes approximately n2. Therefore, to minimize the search time, AVL trees are maintained. This will increase performance up to nlogn. Still, BST requires some time to search and retrieve the data. After deletion of elements, there are some burden to maintain the BST property. It mean, the tree is accessed 2 times. To minimize the time for retrieval, heap is created. In heap sort, the heap creation takes time, but the retrieval takes no time.**Heap Sort**:The heap sort algorithm sorts by representing its input as a heap in the array.Two phases in sorting:Converts the array representation of the tree into a heap;Repeatedly moves the largest element to the last position by swapping the firstelement with the last element and adjusts the heap property at each stage in the remaining elements. **First Phase:**The entries in the array being sorted are interpreted as a binary tree in array implementation; Tree with only one node automatically satisfies the heap property. So, we don't need to worry about any of the leaves.Start from the level above the leaf nodes, and work out backward towards the root.**Second Phase:**Note that the root (the first element of the array) has the largest key.Repeat these steps until the size of heap becomes 1;Move the largest key at root to the last position of the heap, replacing an entry$x$ currently at the last position;Decrease a counter i that keeps track of the size of the heap, thereby excluding the largest key from further sorting;The element x may not belong to the root of the heap, so insert x into theproper position to restore the heap property.**Adjusting the Heap Properties:**When the last element, $\boldsymbol{x}$, is replaced by the largest element at the root, a hole is created at the root and the heap size becomes smaller by **1**;We must move $\boldsymbol{x}$ somewhere to restore the heap property;First we look if $\boldsymbol{x}$ can be placed in the hole, by looking at the two children of that hole;If $\boldsymbol{x}$ belongs to the hole, then we put $\boldsymbol{x}$ there and we are done;Else we slide the larger of the two children to the hole, thus pushing the hole downone level;We repeat this process on the subtree until $\boldsymbol{x}$ can be placed in the hole or there are no children;Thus, our action is to place $\boldsymbol{x}$ in its correct spot along a path from the root containing minimum children. **Heap Sort Efficiency:** No of comparisons and assignments Worst-case: $\boldsymbol{O(n\log n)}$;Average case: $\boldsymbol{O(n\log n)}$**.** Hence time complexity of heap sort is $\boldsymbol{O(n\log n)}$ for both worst case and average case and space complexity is $\boldsymbol{O(1)}$**.** In average case, it is not as efficient as quick sort,

however, it is far superior to quick sort in worse case. Generally, heap sort is used for large amount of data.

Heap as Priority Queue:A ***priority queue*** is a data structure with the following primitive operations:Insert an item;Remove the item having the largest (or smallest) key.Implementations:Use a sorted contiguous list, removal takes $O(1)$ but insertion takes $O(n)$;Use an unsorted list, insertion takes $O(1)$ but removal takes $O(n)$ **Efficiency Priority Queue:**Consider the properties of heap:The item with largest key is on the top and can be removed immediately. However it will take time $O(\log n)$ to restore the heap property for remaining keys.For insertion we shift the new item from down to up which also takes $O(\log n)$.Hence, implementation of a priority queue as a heap proves advantageous for large n.It efficiently represents in contiguous storage and is guaranteed to require only logarithmic time for both insertions and deletions.**Shell Sort**:Significant improvement on simple insertion sort can be achieved by using shell sort (or diminishing increment sort). This method separates original file into subfiles. These subfiles contain every k^{th} element of the original file. The value of k is called an increment. Eg. If k=5, then subfile consists of x[0], x[5], x[10],… is first sorted.After the first k subfiles are sorted (usually by simply insertion), a new smaller value of k is chosen and the file is again partitioned into a new set of subfiles. Each of these larger subfiles is sorted and the process is repeated yet again, until eventually the value of the k is set to 1.The decreasing sequence of increments can Either be fixed at the start of the entire process. The last value must be 1 Or take the first increment to hk = floor(N/2) and hk = floor(hk / 2) until hk = 1. hk = subsequent increment. **Efficiency**: Worse case : $O(n^2)$.Average case : $O(n(\log n)^2)$ (if appropriate increment sequent is used). **Radix Sort**:The sorting is based on the values of the actual digits in the positional representations of the numbers being sorted.**Process**:Beginning with the least-significant digit and ending with the most-significant digit, perform the following action,Take each number in the order in which it appears in the file and place it into one of the ten queues, depending on the value of the digit currently being processed.Then restore each queue to the original file starting with the queue of numbers with a 0 digit and ending with the queue of numbers with a 9 digit.When these actions have been performed for each digit, starting with the least significant digit and ending with the most significant, the file is sorted.Efficiency : **O(n.logn)**