MODULE 4: GRAPHS and INTRODUCTION to SORTING

4.1 Graphs

4.1.1 Introduction

Graph is a nonlinear data structure, it contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices.

A graph is defined as follows

Graph is a collection of vertices and arcs which connects vertices in the graph.

(OR)

Graph is a collection of nodes and edges which connects nodes in the graph.

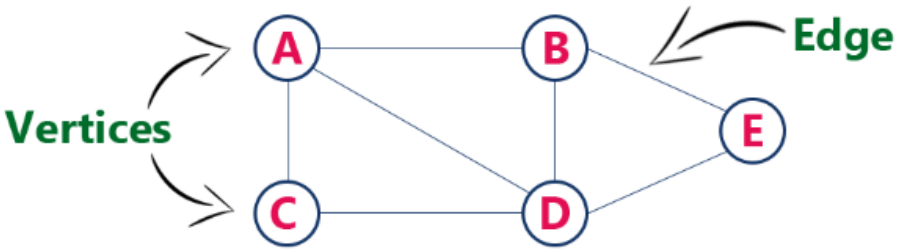
Generally, a graph G is represented as G = (V, E), where V is set of vertices and E is set of edges.

Example

The following is a graph with 5 vertices and 6 edges.

This graph G can be defined as G = (V, E)

Where V = {A, B, C, D, E} and E = {(A, B), (A, C) (A, D), (B, D), (C, D), (B, E), (E, D)}.



Graph Terminology

Vertex: A individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.

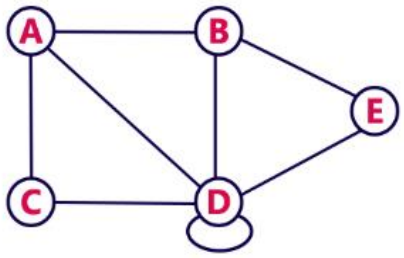
Edge: An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (startingVertex, endingVertex).

For example, in above graph, the link between vertices A and B is represented as (A, B). In above example graph, there are 7 edges (i.e., (A, B), (A, C), (A, D), (B, D), (B, E), (C, D), (D, E)).

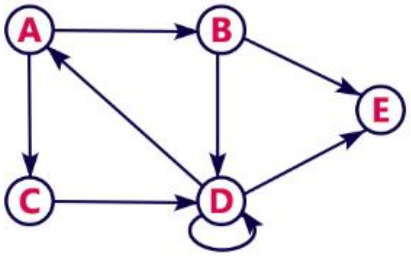
Edges are three types.

1. Undirected Edge - An undirected edge is a bidirectional edge. If there is an undirected edge between vertices A and B then edge (A, B) is equal to edge (B, A).
2. Directed Edge - A directed edge is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A, B) is not equal to edge (B, A).
3. Weighted Edge - A weighted edge is an edge with cost on it.

Undirected Graph: A graph with only undirected edges is said to be undirected graph.

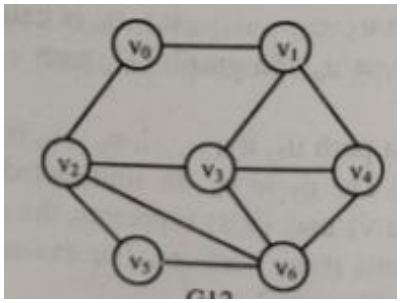


Directed Graph: A graph with only directed edges is said to be directed graph.

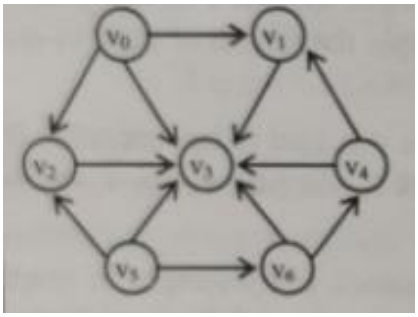


Adjacent: If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, two vertices A and B are said to be adjacent if there is an edge whose end vertices are A and B.

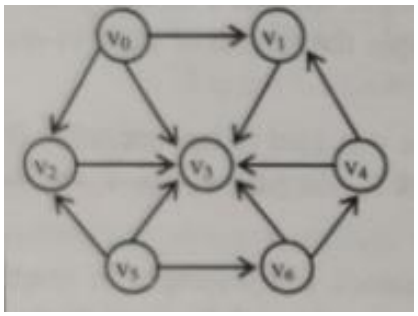
Degree: Total number of edges connected to a vertex is said to be degree of that vertex. In the following graph the degree of vertex v_0 is 2, degree of v_2 is 4.



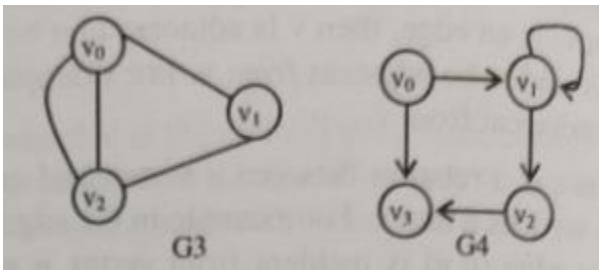
Indegree: Total number of incoming edges connected to a vertex is said to be indegree of that vertex. In the following graph, the indegree of vertex v_0 is 0 and for v_1 it is 2.



Outdegree: Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex. In the following graph the outdegree of vertex v_0 is 3 and for v_3 it is 0.

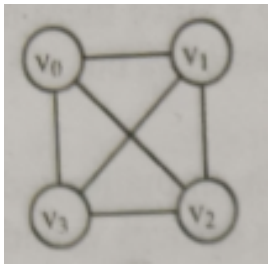


Multigraph: A graph which contains loop or multiple edges is known as multigraph. Following are the examples of multigraph.

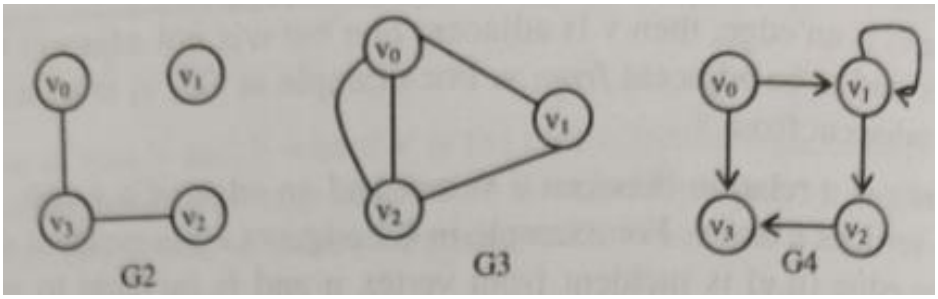


Simple graph: A graph which does not have loop or multiple edges is known as simple graph.

Regular graph: A graph is regular if every vertex is adjacent to the same number of vertices. Following is the regular graph since very vertex is adjacent to 3 vertices.



Planar graph: A graph is called planar if it can be drawn in a plane without any two edges intersecting. Following graphs are planar graphs.



4.1.2 Representation of Graphs

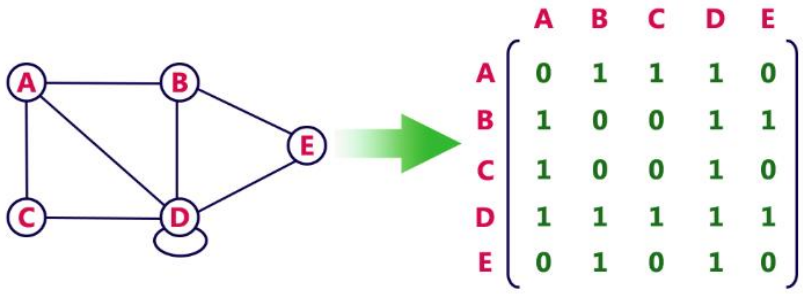
Graph data structure is represented using following representations.

1. Adjacency Matrix
2. Adjacency List
3. Adjacency Multilists
4. Weighted Edges

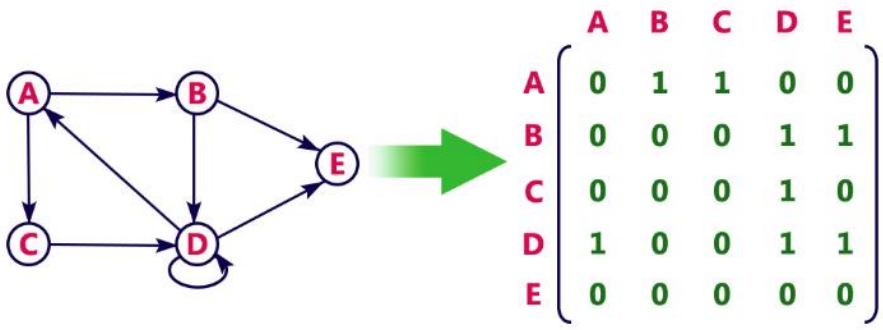
Adjacency Matrix

In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices. That means if a graph with 4 vertices can be represented using a matrix of 4X4 class. In this matrix, rows and columns both represents vertices. This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

Consider the following undirected graph representation:



Consider the following Directed graph representation:

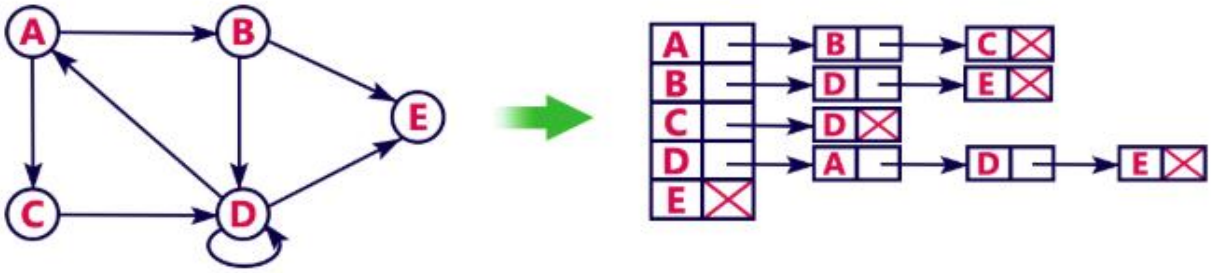


Adjacency List

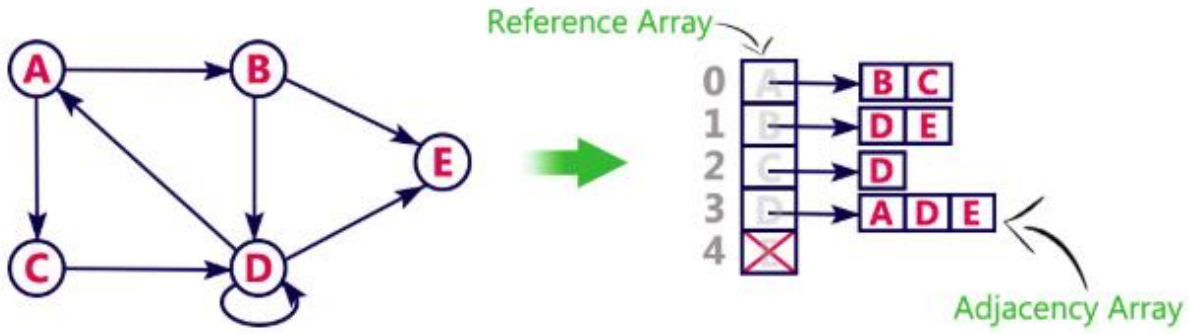
In this representation, every vertex of graph contains list of its adjacent vertices.

If the graph is not dense i.e., the number of edges is less, then it is efficient to represent the graph through adjacency list.

For example, consider the following directed graph representation implemented using linked list.



This representation can also be implemented using array as follows.



Adjacency Multilists

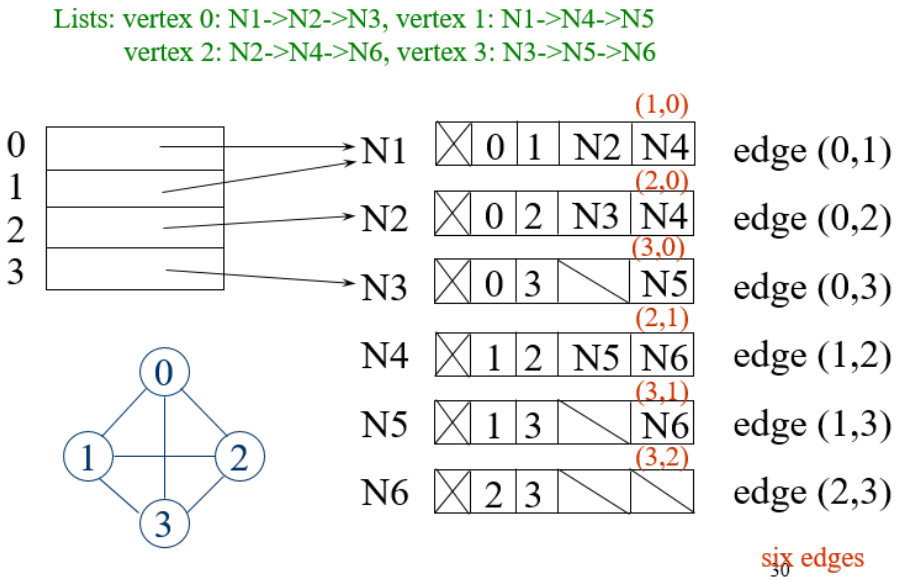
In the adjacency list representation of an undirected graph each edge (v_i,v_j) is represented by two entries, one on the list for v_i and the other on the list for v_j.

As we shall see, in some situations it is necessary to be able to determine the second entry for a particular edge and mark that edge as already having been examined. This can be accomplished easily if the adjacency lists are actually maintained as multilists (i.e., lists in which nodes may be shared among several lists).

For each edge there will be exactly one node, but this node will be in two lists, i.e., the adjacency lists for each of the two nodes it is incident to. The node structure now becomes where

marked	vertex1	vertex2	path1	path2

Example



Weighted Edges

In many applications, the edges of a graph have weights assigned to them. These weights may represent the distance from one vertex to another or the cost of going from one vertex to an adjacent vertex. In these applications the adjacency matrix entries a[i][j] would keep this information too.

When adjacency lists are used, the weight information may be kept I the list nodes by including an additional field, weight.

A graph with weighted edges is called a network.

4.1.3 Graph Traversals

Graph traversal is technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices to be visit in the search process. A graph traversal finds the edges to be used in the search process without creating loops that means using graph traversal we visit all vertices of graph without getting into looping path.

There are two graph traversal techniques and they are as follows.

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

DFS (Depth First Search)

DFS traversal of a graph, produces a **spanning tree** as final result.

Spanning Tree is a graph without any loops.

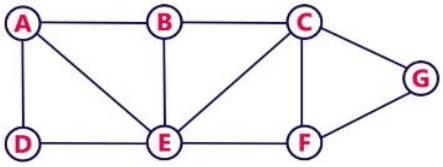
We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS.

Procedure:

We use the following steps to implement DFS traversal

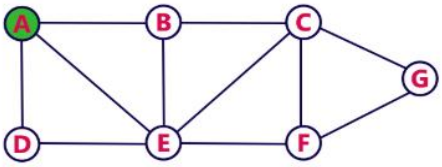
- **Step 1:** Define a Stack of size total number of vertices in the graph.
- **Step 2:** Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
- **Step 3:** Visit any one of the **adjacent** vertex of the vertex which is at top of the stack which is not visited and push it on to the stack.
- **Step 4:** Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.
- **Step 5:** When there is no new vertex to be visit then use **back tracking** and pop one vertex from the stack.
- **Step 6:** Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7:** When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

Consider the following example graph to perform DFS traversal



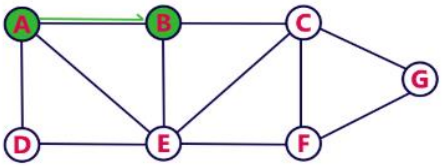
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



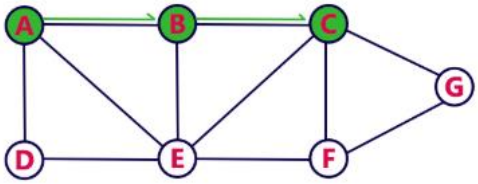
Step 2:

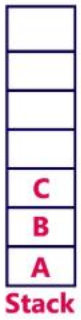
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



Step 3:

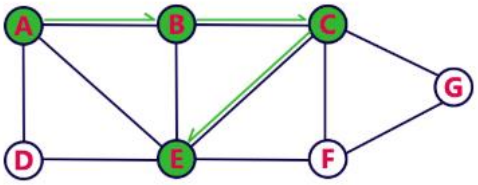
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.





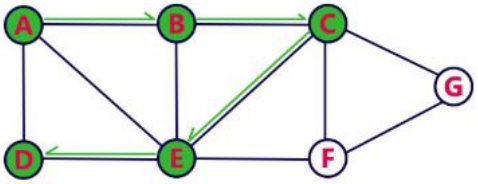
Step 4:

- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack



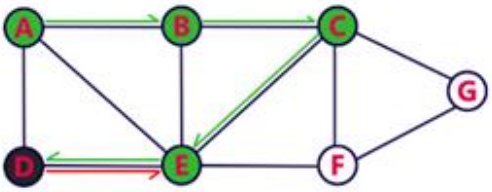
Step 5:

- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack



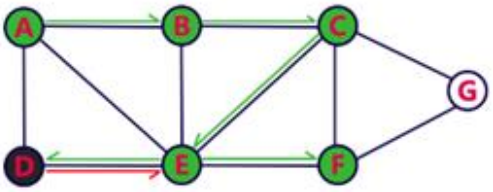
Step 6:

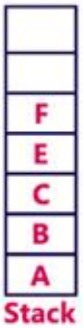
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.



Step 7:

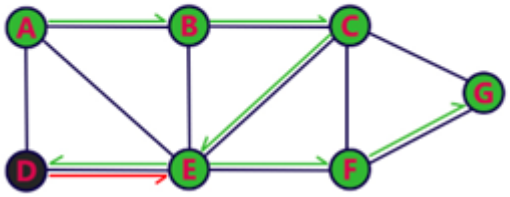
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.





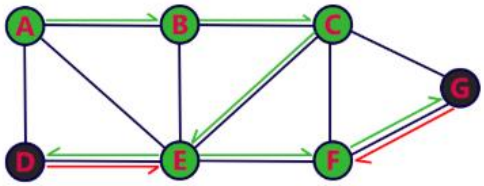
Step 8:

- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.



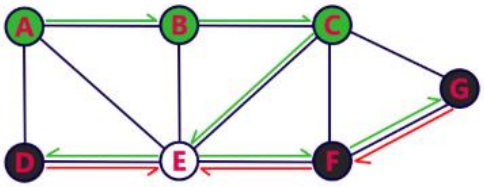
Step 9:

- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.



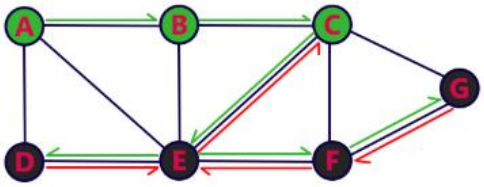
Step 10:

- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.



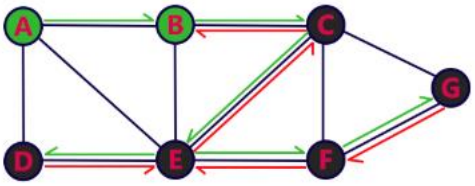
Step 11:

- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.



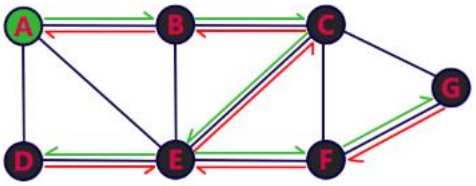
Step 12:

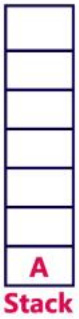
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.



Step 13:

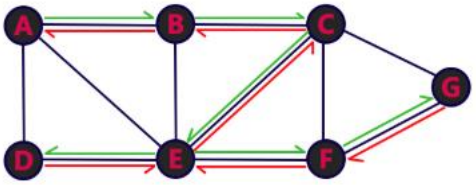
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.



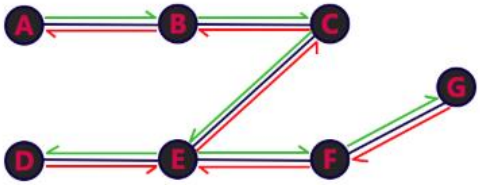


Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.



- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



Depth First Search (DFS) Algorithm

```
n ← number of nodes
Initialize visited [ ] to false (0)
for (i=0; i<n; i++)
        visited[i] = 0;
void DFS (vertex i) [DFS starting from i]
{
   visited[i]=1;
```

```
   for each w adjacent to i
      if (! visited[w])
         DFS(w);
}
```

Depth First Search (DFS) Program in C [Adjacency Matrix]

```
#include<stdio.h>
void DFS(int);
int G [10] [10], visited [10], n;   //n is no of vertices and graph is sorted in array G [10] [10]
void main ()
{
   int i, j;
   printf ("Enter number of vertices:");
   scanf ("%d", &n);
  //read the adjacency matrix
  printf ("\n Enter adjacency matrix of the graph:");
  for (i=0; i<n; i++)
     for (j=0; j<n; j++)
         scanf ("%d", &G[i][j]);
   //visited is initialized to zero
 for (i=0; i<n; i++)
      visited[i]=0;
    DFS (0);
}
void DFS (int i)
{
   int j;
   printf ("\n %d", i);
   visited[i]=1;
   for (j=0; j<n; j++)
     if (! visited[j]&&G[i][j] ==1)
         DFS(j);
}
```

BFS (Breadth First Search)

BFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

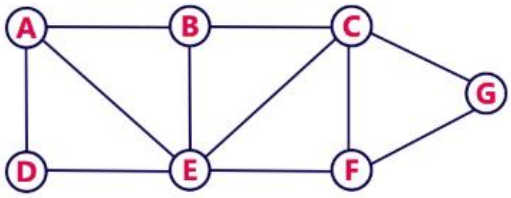
Procedure:

We use the following steps to implement BFS traversal.

- *Step 1: Define a Queue of size total number of vertices in the graph.*
- *Step 2: Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.*

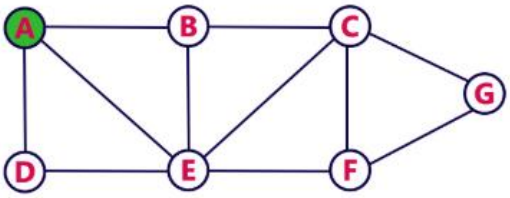
- *Step 3: Visit all the adjacent vertices of the verex which is at front of the Queue which is not visited and insert them into the Queue.*
- *Step 4: When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.*
- *Step 5: Repeat step 3 and 4 until queue becomes empty.*
- *Step 6: When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph*

Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

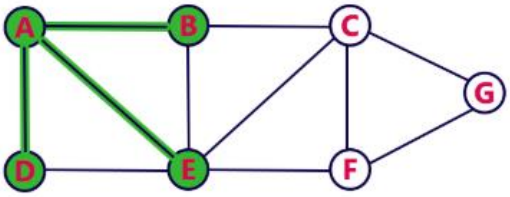


Queue

A						

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

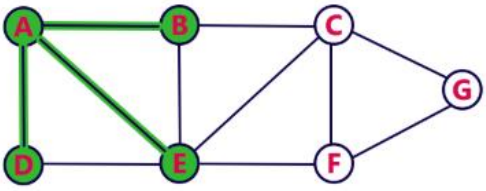


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

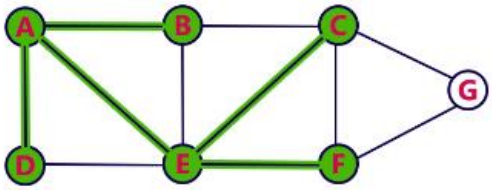


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

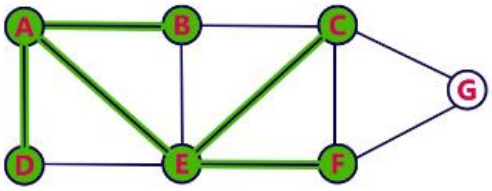


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

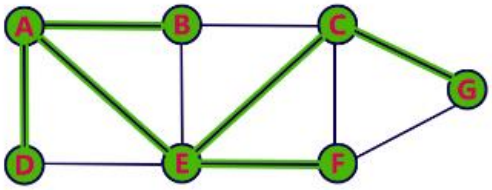


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

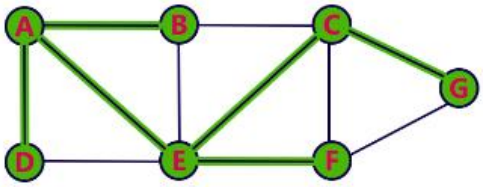


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

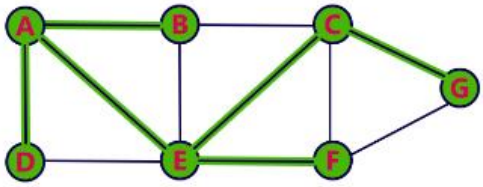


Queue

						G

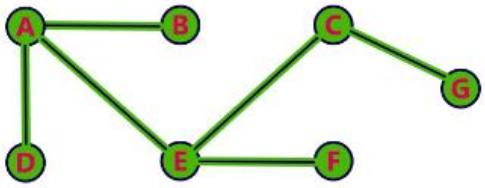
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Breadth First Search (BFS) Algorithm

create a queue Q

mark v as visited and put v into Q

while Q is non-empty

remove the head u of Q

mark and enqueue all (unvisited) neighbours of u

Breadth First Search (BFS) Program in C

```
#include<stdio.h>
int a[20][20],q[20],visited[20],n,i,j,f=0,r=-1;
void bfs(int v)
{
 for(i=1;i<=n;i++)
if(a[v][i] && !visited[i])
q[++r]=i;
 if(f<=r)
 {
 visited[q[f]]=1;

 bfs(q[f++]);
 }
}
void main()
{
 int v;
 printf("\n Enter the number of vertices:");
```

```
scanf("%d",&n);
for(i=1;i<=n;i++)
{
 q[i]=0;
 visited[i]=0;
}
printf("\n Enter graph data in matrix form:\n");
for(i=1;i<=n;i++)
 for(j=1;j<=n;j++)
        scanf("%d",&a[i][j]);
printf("\n Enter the starting vertex:");
scanf("%d",&v);
bfs(v);
printf("\n The node which are reachable are:\n");
for(i=1;i<=n;i++)
if(visited[i])
 printf("%d\t",i);
 else
printf("\n Bfs is not possible");
}
```

4.1.4 Applications of Graphs

Since they are powerful abstractions, graphs can be very important in modeling data. In fact, many problems can be reduced to known graph problems. Here we outline just some of the many applications of graphs.

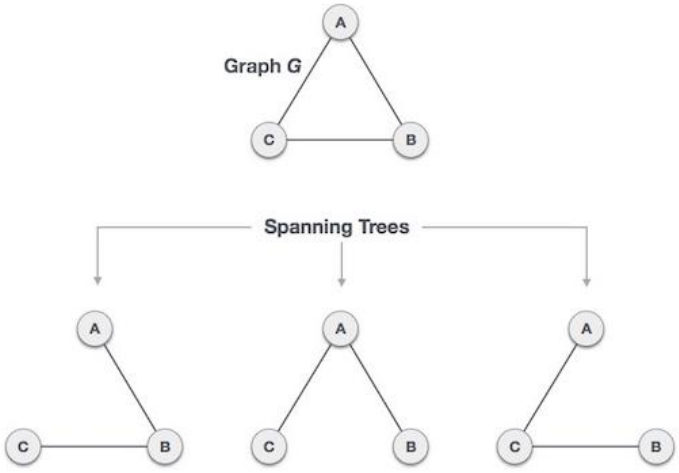
1. **Social network graphs**: To tweet or not to tweet. Graphs that represent who knows whom, who communicates with whom, who influences whom or other relationships in social structures. An example is the twitter graph of who follows whom. These can be used to determine how information flows, how topics become hot, how communities develop, or even who might be a good match for who, or is that whom.
2. **Transportation networks**: In road networks vertices are intersections and edges are the road segments between them, and for public transportation networks vertices are stops and edges are the links between them. Such networks are used by many map programs such as Google maps, Bing maps and now Apple IOS 6 maps (well perhaps without the public transport) to find the best routes between locations. They are also used for studying traffic patterns, traffic light timings, and many aspects of transportation.

3. **Utility graphs**: The power grid, the Internet, and the water network are all examples of graphs where vertices represent connection points, and edges the wires or pipes between them. Analyzing properties of these graphs is very important in understanding the reliability of such utilities under failure or attack, or in minimizing the costs to build infrastructure that matches required demands.
4. **Document link graphs**: The best known example is the link graph of the web, where each web page is a vertex, and each hyperlink a directed edge. Link graphs are used, for example, to analyze relevance of web pages, the best sources of information, and good link sites.
5. **Protein-protein interactions graphs**: Vertices represent proteins and edges represent interactions between them that carry out some biological function in the cell. These graphs can be used, for example, to study molecular pathways—chains of molecular interactions in a cellular process. Humans have over 120K proteins with millions of interactions among them.
6. **Network packet traffic graphs**: Vertices are IP (Internet protocol) addresses and edges are the packets that flow between them. Such graphs are used for analyzing network security, studying the spread of worms, and tracking criminal or non-criminal activity.
7. **Neural networks**: Vertices represent neurons and edges the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 1011 neurons and close to 1015 synapses.
8. **Semantic networks**: Vertices represent words or concepts and edges represent the relationships among the words or concepts. These have been used in various models of how humans organize their knowledge, and how machines might simulate such an organization.
9. **Graphs in epidemiology:** Vertices represent individuals and directed edges the transfer of an infectious disease from one individual to another. Analyzing such graphs has become an important component in understanding and controlling the spread of diseases.
10. **Dependence graphs**: Graphs can be used to represent dependences or precedence's among items. Such graphs are often used in large projects in laying out what components rely on other components and used to minimize the total time or cost to completion while abiding by the dependences.

4.1.5 Spanning Trees

Definition:

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.



A complete undirected graph can have maximum nn-2 number of spanning trees, where n is the number of nodes. In the above addressed example, 33−2 = 3 spanning trees are possible.

Minimum Spanning Tree(MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph.

A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

Types of Minimum Spanning-Tree

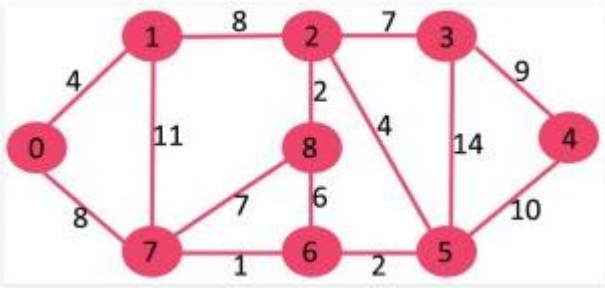
1. Kruskal's Algorithm
2. Prim's Algorithm

Kruskal's Algorithm

Algorithm Steps:

1. Sort the graph edges with respect to their weights.
2. Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
3. Only add edges which doesn't form a cycle, edges which connect only disconnected components.

Consider the below input graph.



The graph contains 9 vertices and 14 edges. So, the MST formed will be having (9 – 1) = 8 edges.

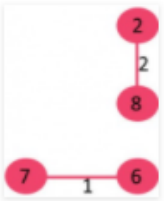
After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

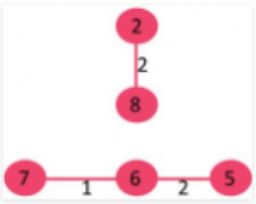
Now pick all edges one by one from sorted list of edges

1. *Pick edge 7-6:* No cycle is formed, include it.

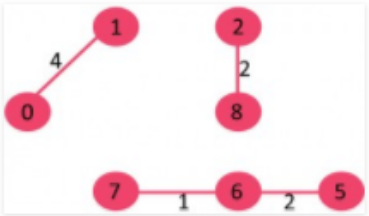
2. *Pick edge 8-2:* No cycle is formed, include it.



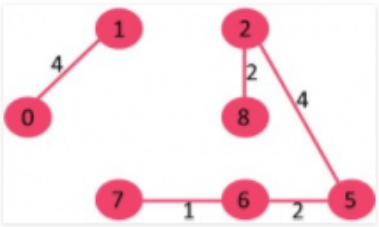
3. *Pick edge 6-5:* No cycle is formed, include it.



4. *Pick edge 0-1:* No cycle is formed, include it.

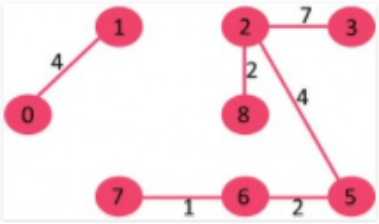


5. *Pick edge 2-5:* No cycle is formed, include it.



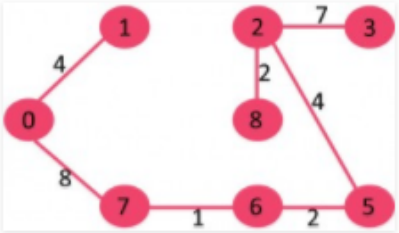
6. *Pick edge 8-6:* Since including this edge results in cycle, discard it.

7. *Pick edge 2-3:* No cycle is formed, include it.



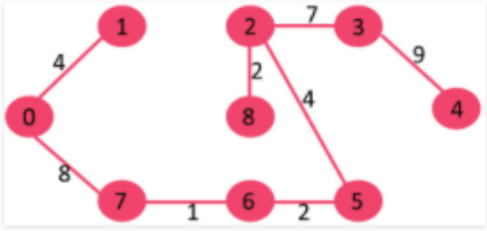
8. *Pick edge 7-8:* Since including this edge results in cycle, discard it.

9. *Pick edge 0-7:* No cycle is formed, include it.



10. *Pick edge 1-2:* Since including this edge results in cycle, discard it.

11. *Pick edge 3-4:* No cycle is formed, include it.



Since the number of edges included equals (V – 1), the algorithm stops here.

C Program for Kruskal's Algorithm

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,cost[9][9],parent[9];
int find(int);
int uni(int,int);
void main()
{
        clrscr();
        printf("\n\tImplementation of Kruskal's algorithm\n");
        printf("\nEnter the no. of vertices:");
        scanf("%d",&n);
        printf("\nEnter the cost adjacency matrix:\n");
        for(i=1;i<=n;i++)
        {
                for(j=1;j<=n;j++)
                {
```

```
                scanf("%d",&cost[i][j]);
                if(cost[i][j]==0)
                        cost[i][j]=999;
        }
    }
    printf("The edges of Minimum Cost Spanning Tree are\n");
    while(ne < n)
    {
        for(i=1,min=999;i<=n;i++)
        {
            for(j=1;j <= n;j++)
            {
                if(cost[i][j] < min)
                {
                    min=cost[i][j];
                    a=u=i;
                    b=v=j;
                }
            }
        }
        u=find(u);
        v=find(v);
        if(uni(u,v))
        {
            printf("%d edge (%d,%d) =%d\n",ne++,a,b,min);
            mincost +=min;
        }
        cost[a][b]=cost[b][a]=999;
    }
    printf("\n\tMinimum cost = %d\n",mincost);
    getch();
}
int find(int i)
{
    while(parent[i])
    i=parent[i];
    return i;
```

```
}
int uni(int i,int j)
{
        if(i!=j)
        {
                parent[j]=i;
                return 1;
        }
        return 0;
}
```

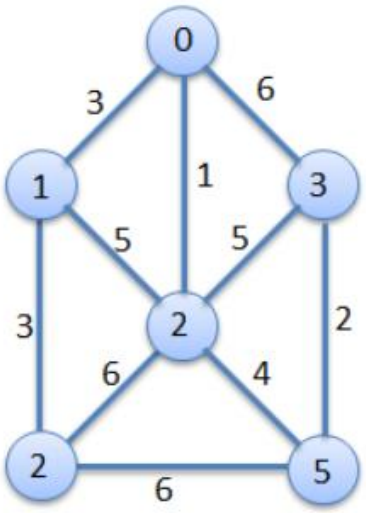
Prim's Algorithm

In Prim's Algorithm we grow the spanning tree from a starting position. Unlike an edge in Kruskal's, we add vertex to the growing spanning tree in Prim's.

Algorithm Steps

1. Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
2. Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.
3. Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

Consider the following graph



Step1

No. of Nodes	0	1	2	3	4	5
Distance	0	3	**1**	6	∞	∞
Distance From		0	0	0		

Step2

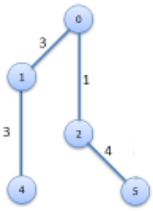
No. of Nodes	0	1	2	3	4	5
Distance	0	**3**	0	5	6	4
Distance From		0		2	2	2

Step3

No. of Nodes	0	1	2	3	4	5
Distance	0	0	0	5	**3**	4
Distance From				2	1	2

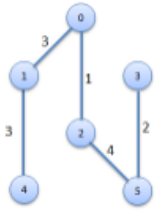
Step4

No. of Nodes	0	1	2	3	4	5
Distance	0	0	0	5	0	**4**
Distance From				2		2



Step5

No. of Nodes	0	1	2	3	4	5
Distance	0	0	0	**3**	0	0
Distance From				2		2



Minimum Cost = 1+2+3+3+4 = 13

C program for Prim's Algorithm

```
#include<stdio.h>
#include<conio.h>
```

```
int a,b,u,v,n,i,j,ne=1;
 int visited[10]={0},min,mincost=0,cost[10][10];

void main()
{

        clrscr();

        printf("\nEnter the number of nodes:");

        scanf("%d",&n);

        printf("\nEnter the adjacency matrix:\n");
        for(i=1;i<=n;i++)

        for(j=1;j<=n;j++)

        {
                scanf("%d",&cost[i][j]);

                if(cost[i][j]==0)

                        cost[i][j]=999;
        }
        visited[1]=1;
        printf("\n");
        while(ne < n)
        {
                for(i=1,min=999;i<=n;i++)
                for(j=1;j<=n;j++)

                if(cost[i][j]< min)

                if(visited[i]!=0)

                {

                        min=cost[i][j];

                        a=u=i;

                        b=v=j;

                }

                if(visited[u]==0 || visited[v]==0)

                {

                        printf("\n Edge %d:(%d %d) cost:%d",ne++,a,b,min);
```

```
                mincost+=min;
                visited[b]=1;
            }
            cost[a][b]=cost[b][a]=999;
        }
        printf("\n Minimun cost=%d",mincost);
        getch();
}
```

4.2 Introduction to Sorting

4.2.1Insertion Sort

Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

Step-by-Step Process

- Step 1: Assume that first element in the list is in sorted portion of the list and remaining all elements are in unsorted portion.
- Step 2: Consider first element from the unsorted list and insert that element into the sorted list in order specified.
- Step 3: Repeat the above process until all the elements from the unsorted list are moved into the sorted list.

Consider the following unsorted list of elements...

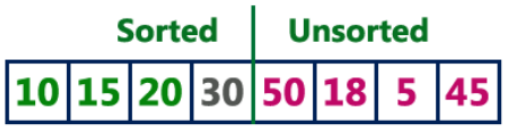
Asume that sorted portion of the list empty and all elements in the list are in unsorted portion of the list as shown in the figure below...

Move the first element 15 from unsorted portion to sorted portion of the list.

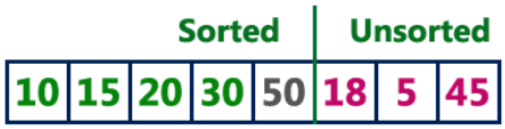
To move element 20 from unsorted to sorted portion, Compare 20 with 15 and insert it at correct position

To move element 10 from unsorted to sorted portion, Compare 10 with 20 and it is smaller so swap. Then compare 10 with 15 again smaller swap. And 10 is insert at its correct position in sorted portion of the list.

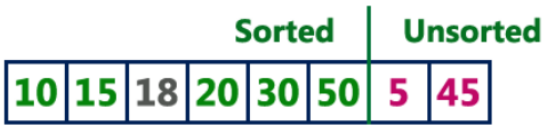
To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.



To move element 50 from unsorted to sorted portion, Compare 50 with 30, 20, 15 and 10. And it is larger than all these so 50 is directly inserted at last position in sorted portion of the list.

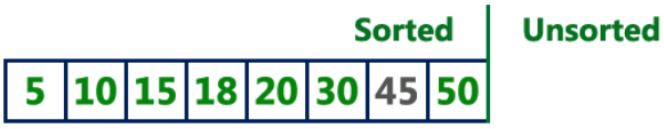


To move element 18 from unsorted to sorted portion, Compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion.



To move element 5 from unsorted to sorted portion, Compare 5 with 50, 30, 20, 18, 15 and 10. Since 5 is smaller than all these element, move 10, 15, 18, 20, 30 and 50 one position to the right in the list and insert 5 at first position in the sorted list.

To move element 45 from unsorted to sorted portion, Compare 45 with 50 and 30. Since 45 is larger than 30, move 50 one position to the right in the list and insert 45 after 30 in the sorted list.



Unsorted portion of the list has became empty. So we stop the process. And the final sorted list of elements is as follows...

Time Complexity:

Worst Case : $O(n^2)$

Best Case : $\Omega(n)$

Average Case : $\Theta(n^2)$

C Program for Insertion Sort

```
#include<stdio.h>
#include<conio.h>
void main(){
  int size, i, j, temp, list[100];
   printf("Enter the size of the list: ");
  scanf("%d", &size);
   printf("Enter %d integer values: ", size);
  for (i = 0; i < size; i++)
    scanf("%d", &list[i]);
      //Insertion sort logic
  for (i = 1; i < size; i++) {
    temp = list[i];
    j = i - 1;
    while ((temp < list[j]) && (j >= 0)) {
      list[j + 1] = list[j];
      j = j - 1;
    }
    list[j + 1] = temp;
  }
   printf("List after Sorting is: ");
  for (i = 0; i < size; i++)
    printf(" %d", list[i]);
 getch();
}
```

4.2.2 Selection Sort

Selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending). In selection sort, the first element in the list is selected and it is compared repeatedly with remaining all the elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped. Then we select the element at second position in the list and it is compared with remaining all elements in the list. If any element is smaller than the selected element, then both are swapped. This procedure is repeated till the entire list is sorted.

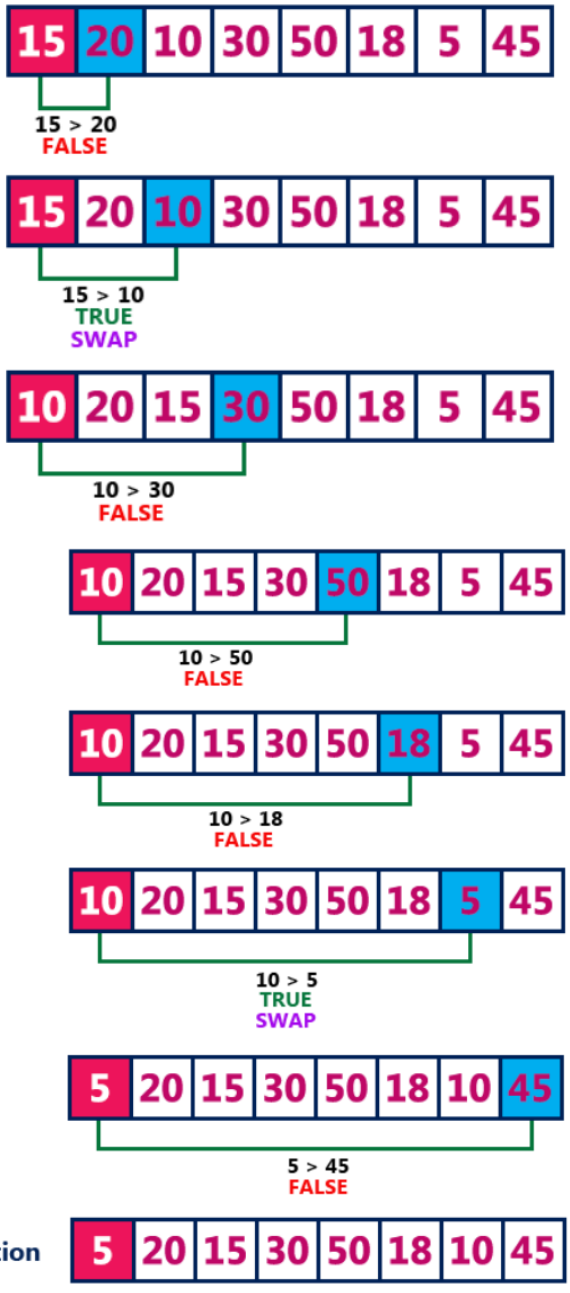
Step by Step Process

- Step 1: Select the first element of the list (i.e., Element at first position in the list).
- Step 2: Compare the selected element with all other elements in the list.
- Step 3: For every comparison, if any element is smaller than selected element (for Ascending order), then these two are swapped.
- Step 4: Repeat the same procedure with next position in the list till the entire list is sorted.

Consider the following unsorted list of elements...

Iteration #1

Select the first position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.



List after 1st iteration

Iteration #2

Select the second position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

Iteration #3

Select the third position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

Iteration #4

Select the fourth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

Iteration #5

Select the fifth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

Iteration #6

Select the sixth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

Iteration #7

Select the seventh position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

Time Complexity:

Worst Case : $O(n^2)$

Best Case : $\Omega(n^2)$

Average Case : $\Theta(n^2)$

C Program for Selection Sort

```
#include<stdio.h>
#include<conio.h>

void main(){

 int size,i,j,temp,list[100];
 clrscr();
```

```
printf("Enter the size of the List: ");
scanf("%d",&size);

printf("Enter %d integer values: ",size);
for(i=0; i<size; i++)
   scanf("%d",&list[i]);

//Selection sort logic

for(i=0; i<size; i++){
   for(j=i+1; j<size; j++){
      if(list[i] > list[j])
   {
         temp=list[i];
         list[i]=list[j];
         list[j]=temp;
      }
   }
}

printf("List after sorting is: ");
for(i=0; i<size; i++)
   printf(" %d",list[i]);

getch();
}
```

4.2.3 Bubble Sort

Bubble sort algorithm starts by comparing the first two elements of an array and swapping if necessary, i.e., if you want to sort the elements of array in ascending order and if the first element is greater than second then, you need to swap the elements but, if the first element is smaller than second, you mustn't swap the element. Then, again second and third elements are compared and swapped if it is necessary and this process go on until last and second last element is compared and swapped. This completes the first step of bubble sort.

If there are n elements to be sorted then, the process mentioned above should be repeated n-1 times to get required result.

But, for better performance, in second step, last and second last elements are not compared becuase, the proper element is automatically placed at last after first step.

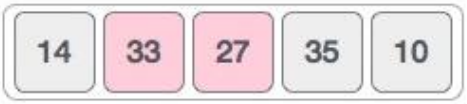
Similarly, in third step, last and second last and second last and third last elements are not compared and so on.

Consider the following list of elements

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.

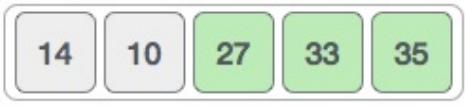
Then we move to the next two values, 35 and 10.

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

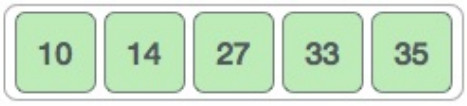
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this –

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.



C program for Bubble Sort

```
/*C Program To Sort data in ascending order using bubble sort.*/
#include <stdio.h>
int main()
{
   int data[100],i,n,step,temp;
   printf("Enter the number of elements to be sorted: ");
   scanf("%d",&n);
   for(i=0;i<n;++i)
   {
      printf("%d. Enter element: ",i+1);
      scanf("%d",&data[i]);
   }
   for(step=0;step<n-1;++step)
   for(i=0;i<n-step-1;++i)
   {
      if(data[i]>data[i+1])   /* To sort in descending order, change > to < in this line. */
      {
         temp=data[i];
         data[i]=data[i+1];
         data[i+1]=temp;
      }
```

```
    }
    printf("In ascending order: ");
    for(i=0;i<n;++i)
        printf("%d  ",data[i]);
    return 0;
}
```

Time complexity

Worst Case : $O(n^2)$

Best Case : $\Omega(n)$

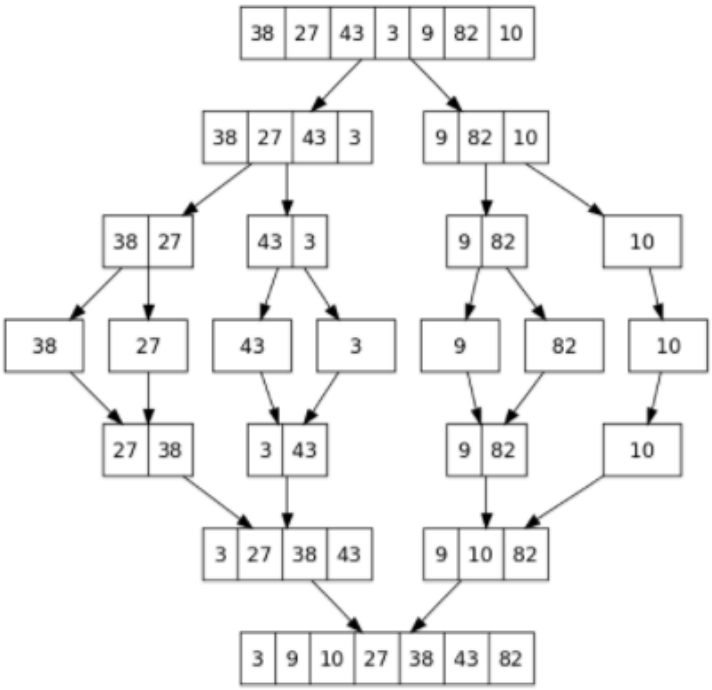
Average Case : $\Theta(n^2)$

4.2.4 Merge Sort

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

Step-by-Step Process

- First divide the list into the smallest unit (1 element),
- Then compare each element with the adjacent list to sort and merge the two adjacent lists.
- Finally, all the elements are sorted and merged.



Time Complexity

Worst Case : O(nlogn)

Best Case : Ω(nlogn)

Average Case : Θ(nlogn)

C program for Merge Sort

```
#include<stdio.h>
void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);
 int main()
{
   int a[30],n,i;
   printf("Enter no of elements:");
   scanf("%d",&n);
   printf("Enter array elements:");

   for(i=0;i<n;i++)
      scanf("%d",&a[i]);

   mergesort(a,0,n-1);

   printf("\nSorted array is :");
   for(i=0;i<n;i++)
      printf("%d ",a[i]);

   return 0;
}
 void mergesort(int a[],int i,int j)
{
   int mid;

   if(i<j)
   {
      mid=(i+j)/2;
      mergesort(a,i,mid);        //left recursion
      mergesort(a,mid+1,j);    //right recursion
      merge(a,i,mid,mid+1,j);    //merging of two sorted sub-arrays
   }
}
 void merge(int a[],int i1,int j1,int i2,int j2)
{
   int temp[50];    //array used for merging
   int i,j,k;
   i=i1;    //beginning of the first list
   j=i2;    //beginning of the second list
   k=0;
      while(i<=j1 && j<=j2)    //while elements in both lists
   {
```

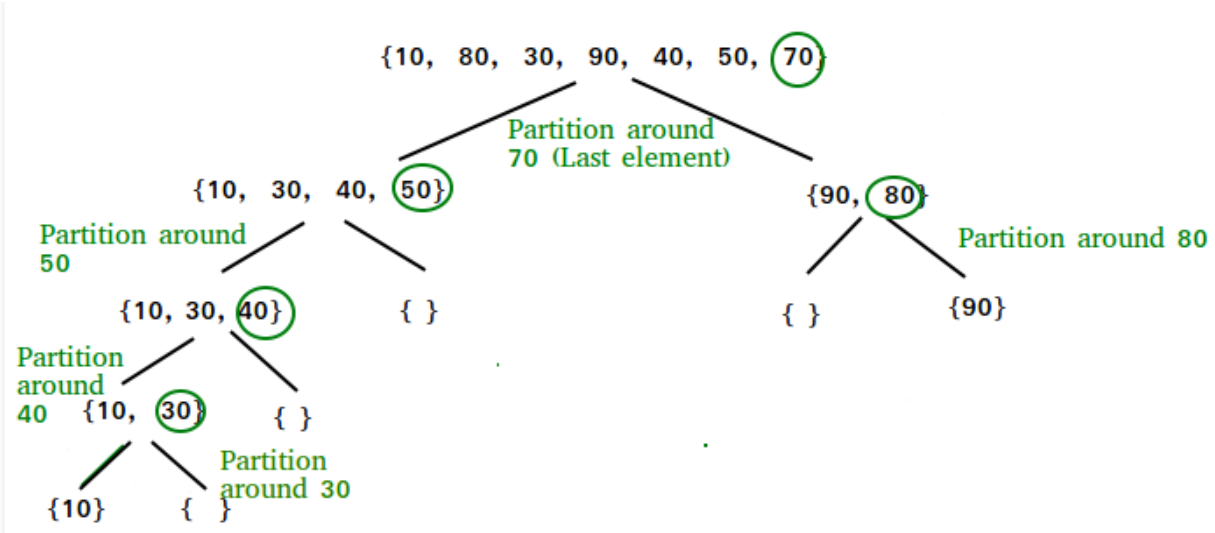
```
    if(a[i]<a[j])
      temp[k++]=a[i++];
    else
      temp[k++]=a[j++];
  }
    while(i<=j1)   //copy remaining elements of the first list
    temp[k++]=a[i++];
      while(j<=j2)   //copy remaining elements of the second list
    temp[k++]=a[j++];
      //Transfer elements from temp[] back to a[]
  for(i=i1,j=0;i<=j2;i++,j++)
    a[i]=temp[j];
}
```

4.2.5 Quick Sort

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of QuickSort that pick pivot in different ways.

Step-by-Step Process

1) Pick an element from the array, this element is called as pivot element.
2) Divide the unsorted array of elements in two arrays with values less than the pivot come in the first sub array, while all elements with values greater than the pivot come in the second sub-array (equal values can go either way). This step is called the partition operation.
3) Recursively repeat the step 2(until the sub-arrays are sorted) to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.



C 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, count, number[25];

   printf("How many elements are u going to enter?: ");
   scanf("%d",&count);

   printf("Enter %d elements: ", count);
   for(i=0;i<count;i++)
      scanf("%d",&number[i]);

   quicksort(number,0,count-1);

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

   return 0;
}
```

Time Complexity

Worst Case : $O(n^2)$

Best Case : $\Omega(n)$

Average Case : $\Theta(nlogn)$

4.2.6 Comparison of various sorting and search techniques

Sorting	Best Case	Average case	Worst Case	Comparison with others
Insertion	O(n)	O(n2)	O(n2)	Requires more comparisons than Quick sort but easy
Selection	O(n2)	O(n2)	O(n2)	Requires more comparisons than Quick sort but easy
Bubble	O(n)	O(n2)	O(n2)	Easy but requires more comparisons
Quick	O(nlogn)	O(nlogn)	O(n2)	Fast but difficult to implement
Merge	O(nlogn)	O(nlogn)	O(nlogn)	
Heap	O(nlogn)	O(nlogn)	O(nlogn)	
Linear Search	O(1)	O(n)	O(n)	Number of comparisons are more to search an element.
Binary Search	O(1)	O(logn)	O(logn)	Lists are to be in sorted order and number of comparisons are less.