# S4 DATASTRUCTURES(DS) Code: 4133
# MODULE 4 - GRAPH

## Syllabus

**MODULE IV GRAPH**

**4.1 Understanding Graph and its operations**

4.1.1 Explain graph and key terms related to graphs
4.1.2 Explain graph representations – adjacency matrix mehod and adjacency list method.
4.1.3 Describe graph traversals – DFS and BFS
4.1.4 Describe Graph ADT with dfs() and bfs() methods.
4.1.5 Describe and implement Warshall's algorithm for all-pairs shortest path

**4.2 Understanding Searching and Sorting**

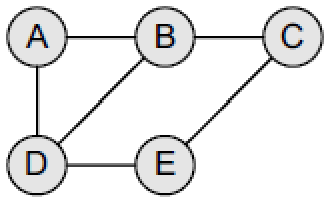
4.2.1 Explain and implement linear search and binary search algorithms
4.2.2 Explain and implement bubble sort and quick sort algorithms

## GRAPH

- Graph is an abstract non-linear data structure
- Graph is basically a collection of vertices (also called nodes) and edges that connect these vertices.
- Concept of graph is from the tree data structure where instead of having a purely parent-to-child relationship between tree nodes,any kind of complex relationship can exist.

### Definition:

- **A graph G is defined as an ordered set(V,E) where V(G) represents the set of vertices and E(G) represents the edges that connect these vertices**

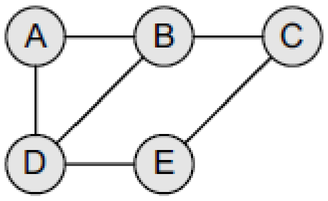


- Figure shows a graph with V(G) = {A, B, C, D and E} and
  E(G) = {(A, B), (B, C),(A, D), (B, D), (D, E), (C, E)}.
- Note that there are **five vertices or nodes** and **sixedges in the graph**.

### Undirected Graph

- A graph can be directed or undirected.
- In an **undirected graph**, edges do not have any direction associated with them.
- In an undirected graph it does not give any information about the direction of the edges.

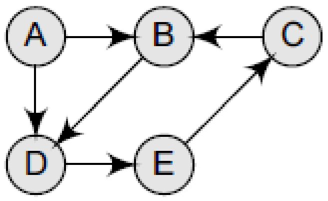
- If an edge is drawn between nodes A and B then the nodes can be traversed from A to B as well as from B to A.



- Figure shows a graph with V(G) = {A, B, C, D and E} and
  E(G) = {(A, B), (B, C),(A, D), (B, D), (D, E), (C, E)}.
- Note that there are five vertices or nodes and sixedges in the graph.

## Directed graph

- In a **directed graph** edges form an ordered pair.
- In a **directed graph,**edges have **direction** associated with them.
- If there is an edge from A to B, then there is a path from A to B but not from B to A
- The edge (A,B) is said to initiate (start) from node A (known as initial node) and terminate (end) at node B (terminal node).



- Figure shows a directed graph V(G) = {A, B, C, D and E} and
  E(G) = {(A, B), (C,B), (A, D), (B, D), (D, E), (C, E)}.
- In a directed graph, edgesform an ordered pair. If there is an edge from A to B, then there is a path from A to B but not fromB to A.
- The edge (A, B) is said to initiate from node A (also known as initial node) and terminateat node B (terminal node).

## Labelled graph or weighted graph

- A graph is said to be labelled if every **edge in the graph is assigned some data**.
- In a **weighted graph**, the **edges of the graph are assigned some weight or length**.
- The weight of an edge denoted by w(e) is a positive value which indicates the cost of traversing the edge.

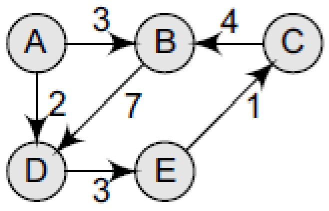


Figure shows weighted graph.

## Graph terminology

### Adjacent nodes or neighbours

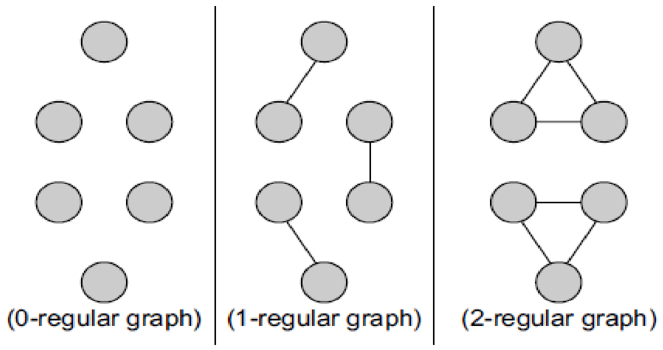
- For every edge, e=(a,b) that connects nodes a and b, the nodes a and bare the end points and are said to be the adjacent nodes or neighbours.

### Degree of a node

- Degree of  a node u, deg(u), is the **total number of edges** containing the node u.
- If deg(u)=0, it means that u does not belong to any edge and such a node is known as an isolated node.

### Regular graph

- It is a graph where each vertex has the same number of neighbours
- That is, every node has the same degree.
- A regular graph with vertices of degree k is called a k-regular graph or a regular graph of degree k.



### Path

- A path p written as p={v0,v1,v2…..vn} of length n from a node u to v is defined as a sequence of (n+1) nodes.

### Closed path

- A path p is known as a closed path if the edge has the same end-points

**Simple path**

- A path p is known as a simple path if all the nodes in the path are distinct with an exception that v0 may be equal to vn
- If v0=vn, then the path is called a closed simple path.

**Cycle**

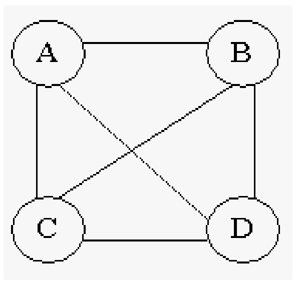
- A path in which the first and the last vertices are same.
- A simple cycle has no repeated edges or vertices(except the first and last vertices)

**Connected graph**

- A graph is said to be connected if for any two vertices(u,v) in V there is a path from u to v.
- There are no isolated nodes in a connected graph
- A connected graph that does not have any cycle is called a tree.
- Therefore, a tree is treated as a special graph.

**Complete graph**

- A graph G is said to be complete if **all its nodes are fully connected**.
- That is, there is a path from one node to every other node in the graph.
- A complete graph has n(n-1)/2 edges, where n is the number of nodes in G.



**Multiple edges**

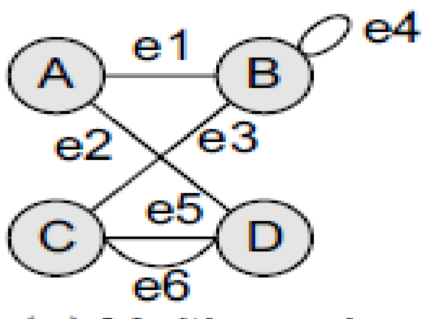
- Distinct edges which connect the same end-points are called multiple edges.

**Loop**

- An edge that has identical end-points is called a loop. Ie, e=(u,u)

**Multi-graph**

- A graph with a multiple edges and/or loops is called a multi-graph



(a) Multi-graph

**Clique**

- In an undirected graph G=(V,E), clique is a subset of the vertex set C subset of V, such that for every two vertices in C, there is an edge that connect two vertices.

**Size of a graph**

- The size of a graph is the total number of edges in it.

## Directed graphs

- A directed graph G also known as a digraph, is a graph in which every edge has a direction assigned to it.
- An edge of a directed graph is given as an ordered pair(u,v) of nodes in G.
- For an edge(u,v):-
    - The edge begins at u and ends at v.
    - U is known as the origin or initial point of e.
    - V is known as the destination or terminal point of e
    - U is the predecessor of v
    - V is the successor of u
    - Nodes u and v are adjacent to each other

## Terminology of a Directed Graph

**Out-degree of a node**

- The out-degree of a node u, written as outdeg(u), is the number of edges that originate(begins) at u.

**In-degree of a node**

- The in-degree of a node u, written as indeg(u), is the number of edges that terminate(ends) at u

**Degree of a node**

- The degree of a node, written as deg(u), is equal to the sum of in-degree and out-degree of that node.therefore, deg(u)=indeg(u)+outdeg(u)

**Isolated vertex**

- A vertex with degree zero. Such a vertex is not an end-point of any edge.

**Pendant vertex**

- A vertex with degree one. Also known as leaf vertex

**Source**

- A node u is known as a source if it has a positive out-degree but a zero in-degree

**Sink**

- A node u is known as a sink if it has a positive in-degree but a zero out-degree

**Reachability**

- A node v is said to be reachable from node u, if and only if there exists a(directed) path from node u to node v

**Strongly connected directed graph**

- A directed graph(digraph) is said to be strongly connected iff there exist a path between every pair of nodes in G. ie, if there is a path from node u to v, then there must be a path from node v to u

**Unilaterally connected graph**

- A directed graph(digraph) is said to be unilaterally connected if there exist a path between any pair of nodes u,v in G such that there is a path from u to v or a path from v to u, but not both

**Parallel/multiple edges**

- Distinct edges which connect the same end points are called multiple edges.

  ie, e=(u,v) and e'= (u,v) are known as multiple edges of graph G

**Simple directed graph**

- A directed graph G is said to be a simple directed graph iff it has no parallel edges. However, a simple directed graph may contain cycles with an exception that it cannot have more than one loop at a given node.

# Representation of Graph

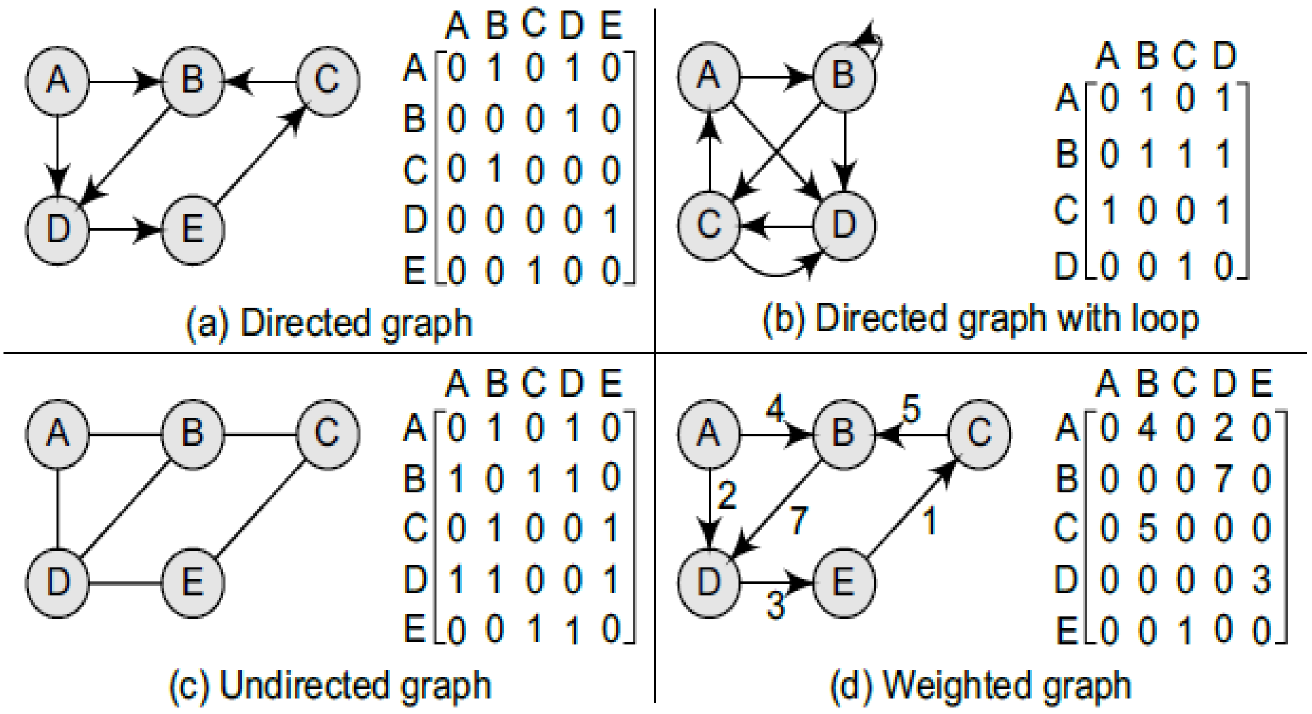
There are two common ways of **storing graphs in the computer's memory**.

They are:

- **Sequential representation** by using an **adjacency matrix.**
- **Linked representation** by using an **adjacency list**, that stores the neighbours of a node using a linked list

## Adjacency matrix representation

- An adjacency matrix is used to represent which nodes are adjacent to one another.
- Two nodes are said to be adjacent if there is an edge connecting them.
- In a directed graph G, if node v is adjacent to node u, then there is definitely an edge from u to v. ie, if v is adjacent to u, we can get from u to v by traversing one edge.



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 1 & 0 \\ B & 0 & 0 & 0 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 0 \\ D & 0 & 0 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 0 & 0 \end{array}$$

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 0 & 1 \\ B & 0 & 1 & 1 & 1 \\ C & 1 & 0 & 0 & 1 \\ D & 0 & 0 & 1 & 0 \end{array}$$

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 1 & 0 \\ B & 1 & 0 & 1 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 1 \\ D & 1 & 1 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 1 & 0 \end{array}$$

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 4 & 0 & 2 & 0 \\ B & 0 & 0 & 0 & 7 & 0 \\ C & 0 & 5 & 0 & 0 & 0 \\ D & 0 & 0 & 0 & 0 & 3 \\ E & 0 & 0 & 1 & 0 & 0 \end{array}$$

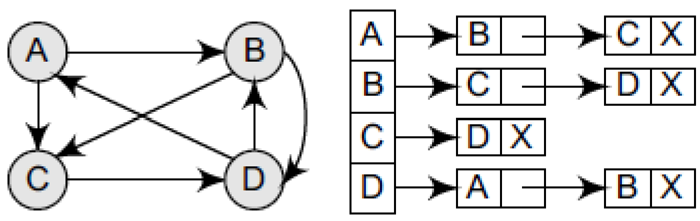
**Figure shows Graphs and their corresponding adjacency matrices.**

- For a simple graph (that has no loops), the adjacency matrix has 0s on the diagonal.
- The adjacency matrix of an undirected graph is symmetric.

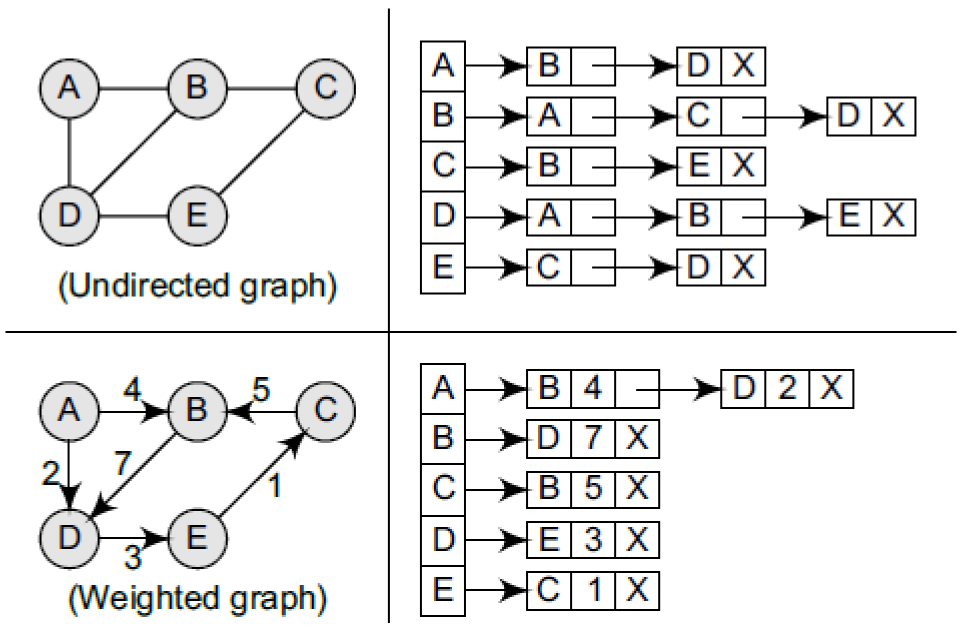
- Number of 1s (or non-zero entries) in an adjacency matrix is equal to the number of edgesin the graph.
- The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.

## Adjacency list Representation

- An adjacency list is another way in which **graphs can be represented in the computer's memory.**
- Adjacency list structure **consists of a list of all nodes in G.**
- Every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it.



(Directed graph)



## Advantages of adjacency list

- It is **easy to follow and show the adjacent nodes** of a particular node.
- It is **used to storing graphs** that have a small to moderate number of edges.
- **Adding new nodes in G is easy**. In adjacency matrix, adding nodes is difficult.

## Graph traversal algorithm

- Traversing means *accessing each node exactly once*.
- There are two standard methods of graph traversal algorithm.

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

- Search algorithms make use of a variable STATUS.
- During the execution of the algorithm every node in the graph will have the variable STATUS set to 1 or 2, depending on its current state.

| Status | State of the node | Description |
|---|---|---|
| 1 | Ready | The initial state of the node N |
| 2 | Waiting | Node N is placed on the queue or stack and waiting to be processed |
| 3 | Processed | Node N has been completely processed |

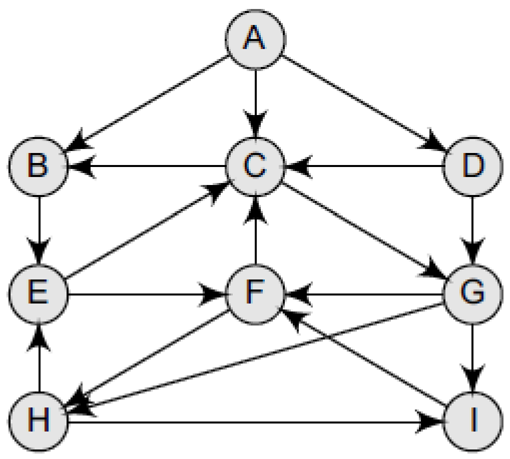
## BFS (Breadth First Search)

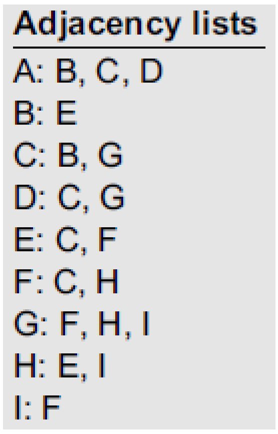
- Breadth First Search is a graph search algorithm that **begins at the root node and goes to all the neighbouring nodes. No node is processed more than once**. For this we are using a queue.
- First start examining the node A and then all the neighbours of **A** are examined.
- In the next step, examine the neighbours of neighbours of A and so on, until it finds the goal.

## Algorithm for BFS

```
Step 1: SET STATUS = 1 (ready state)for each node in G.
Step 2: Enqueue the starting node Aand set its STATUS = 2(waiting state).
Step 3: Repeat Steps 4 and 5 untilQUEUE is empty.
Step 4:         Dequeue a node N. Process itand set its STATUS = 3(processed state).
Step 5:         Enqueue all the neighbours ofN that are in the ready state(whose STATUS = 1)
                and set their STATUS = 2(waiting state).
        [END OF LOOP]
Step 6: EXIT
```

**Example:**





**Adjacency lists**

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

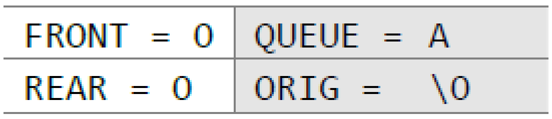
- The **minimum path P can be found by applying the breadth-first search algorithm** that begins at A and ends when I is encountered.

- During the execution of the algorithm, we **usetwo arrays: QUEUE and ORIG**.
  - **QUEUE** is used **to hold** the **nodes that have to be processed**.
  - **ORIG** is used **to keep track of the origin of each edge**.

- Initially, FRONT = REAR = –1.

**The algorithm for this is as follows:**

(a) Add A to QUEUE and add NULL to ORIG.



| | |
|---|---|
| FRONT = 0 | QUEUE = A |
| REAR = 0 | ORIG = \0 |

(b) Dequeue a node by setting FRONT = FRONT + 1 (remove the FRONT element of QUEUE) and enqueuer the neighbours of A. Also, add A as the ORIG of its neighbours.

| FRONT = 1 | QUEUE = A | B | C | D |
|---|---|---|---|---|
| REAR = 3 | ORIG = \0 | A | A | A |

(c) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of B. Also, add B asthe ORIG of its neighbours.

| FRONT = 2 | QUEUE = A | B | C | D | E |
|---|---|---|---|---|---|
| REAR = 4 | ORIG = \0 | A | A | A | B |

(d) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of C. Also, add C asthe ORIG of its neighbours. Note that C has two neighbours B and G. Since B has already beenadded to the queue and it is not in the Ready state, we will not add B and only add G.

| FRONT = 3 | QUEUE = A | B | C | D | E | G |
|---|---|---|---|---|---|---|
| REAR = 5 | ORIG = \0 | A | A | A | B | C |

(e) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of D. Also, add D asthe ORIG of its neighbours. Note that D has two neighbours C and G. Since both of them have already been added to the queue and they are not in the Ready state, we will not add themagain.

| FRONT = 4 | QUEUE = A | B | C | D | E | G |
|---|---|---|---|---|---|---|
| REAR = 5 | ORIG = \0 | A | A | A | B | C |

(f) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of E. Also, add E asthe ORIG of its neighbours. Note that E has two neighbours C and F. Since C has already been added to the queue and it is not in the Ready state, we will not add C and add only F.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| FRONT = 5 | QUEUE = A | B | C | D | E | G | F |
| REAR = 6 | ORIG = \0 | A | A | A | B | C | E |

(g)Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of G. Also, add G asthe ORIG of its neighbours. Note that G has three neighbours F, H, and I.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| FRONT = 6 | QUEUE = A | B | C | D | E | G | F | H | I |
| REAR = 9 | ORIG = \0 | A | A | A | B | C | E | G | G |

Since F has already been added to the queue, we will only add H and I. As I is our finaldestination, we stop the execution of this algorithm as soon as it is encountered and addedto the QUEUE.

**Backtrack from I using ORIG to find the minimum path P**.
Thus, minimum path P is **A -> C -> G -> I.**

## Application of BFS

1. Finding all connected components in a graph.
2. Finding all nodes within an individual connected component.
3. Finding the shortest path between two nodes u and v, of an unweighted graph.
4. Finding the shortest path between two nodes u and v of a weighted graph.

## DFS(Depth First Search)

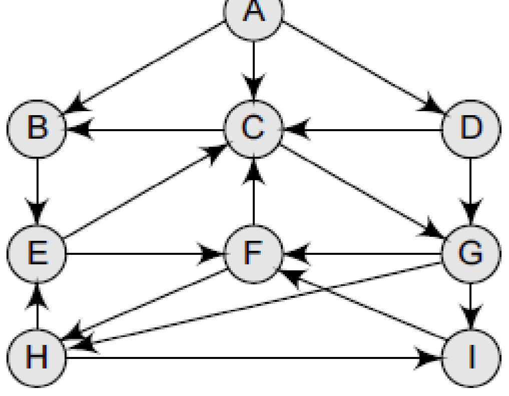
- The depth-first search algorithm progress by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered.
- When a dead-end is reached, the algorithm backtracks, returning to the most that has not been completely explored.

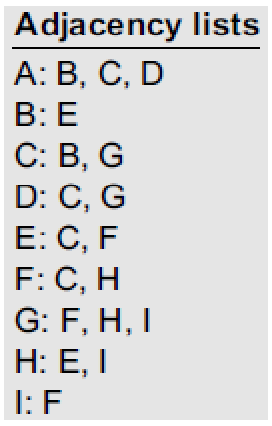
  - Depth-first search begins at a starting node A which becomes the current node.
  - Then, it examines each node N along a path P which begins at A. That is, we process a neighbourof A, then a neighbour of neighbour of A, and so on.
  - During the execution of the algorithm, if wereach a path that has a node N that has already been processed, then we backtrack to the currentnode.
  - Otherwise, the unvisited (unprocessed) node becomes the current node.

## Algorithm for depth-first search

```
Step 1: SET STATUS = 1 (ready state) for each node in G.
Step 2: Push the starting node A on the stack and setits STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until STACK is empty.
Step 4:         Pop the top node N. Process it and set itsSTATUS = 3 (processed state)
Step 5:         Push on the stack all the neighbours of N thatare in the ready state (whose
                  STATUS = 1) andset their STATUS = 2 (waiting state).
        [END OF LOOP]
Step 6: EXIT
```

**Example:**




Adjacency lists

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

(a) Push H onto the stack.

**STACK: H**

(b) Pop and print the top element of the STACK, that is, H. Push all the neighbours of H onto the stack that are in the ready state. The STACK now becomes

**PRINT: H** **STACK: E, I**

(c) Pop and print the top element of the STACK, that is, I. Push all the neighbours of I onto the stack that are in the ready state. The STACK now becomes

**PRINT: I** **STACK: E, F**

(d) Pop and print the top element of the STACK, that is, F. Push all the neighbours of F onto the stack that are in the ready state. (Note F has two neighbours, C and H. But only C will be added, as H is not in the ready state.) The STACK now becomes

**PRINT: F** **STACK: E, C**

(e) Pop and print the top element of the STACK, that is, C. Push all the neighbours of C onto the stack that are in the ready state. The STACK now becomes

**PRINT: C** **STACK: E, B, G**

(f) Pop and print the top element of the STACK, that is, G. Push all the neighbours of G onto the stack that are in the ready state. Since there are no neighbours of G that are in the ready state, no push operation is performed. The STACK now becomes

**PRINT: G** **STACK: E, B**

(g) Pop and print the top element of the STACK, that is, B. Push all the neighbours of B onto the stack that are in the ready state. Since there are no neighbours of B that are in the ready state, no push operation is performed. The STACK now becomes

**PRINT: B** **STACK: E**

(h) Pop and print the top element of the STACK, that is, E. Push all the neighbours of E onto the stack that are in the ready state. Since there are no neighbours of E that are in the ready state, no push operation is performed. The STACK now becomes empty.

**PRINT: E** **STACK:**

Since the STACK is now empty, the depth-first search of G starting at node H is complete and the nodes which were printed are:

**H, I, F, C, G, B, E**

These are the nodes which are reachable from the node H.

## Applications of DFS algorithm

1. Finding a path between two specified nodes u,v of an unweighted graph.
2. Finding a path between two specified nodes u and v of a weighted graph.
3. Finding whether a graph is connected or not.
4. Computing the spanning tree of a connected graph.

# Warshall's Algorithm

- ➢ Warshall has given a very efficient algorithm to calculate the path matrix.
- ➢ Warshall's algorithm defines path matrices P0, P1, P2, …, Pn as

$$\begin{tikzcd} & 1 \text{ [if there is a path from } v_i \text{ to } v_j\text{. The path should not use any other nodes except } v_1, v_2, \ldots, v_k\text{]} \\ P_k[i][j] \arrow[ur] \arrow[dr] & \\ & 0 \text{ [otherwise]} \end{tikzcd}$$

## Modified Warshall's Algorithm

- ➢ Warshall's algorithm can be modified to obtain a matrix that gives the shortest paths between thenodes in a graph G.
- ➢ Use the adjacency matrix A of G and **replace all** the values of A which are **zero by infinity (∞).**

- **Infinity (∞) denotes** a very large number andindicates that **there is no path between the vertices.**

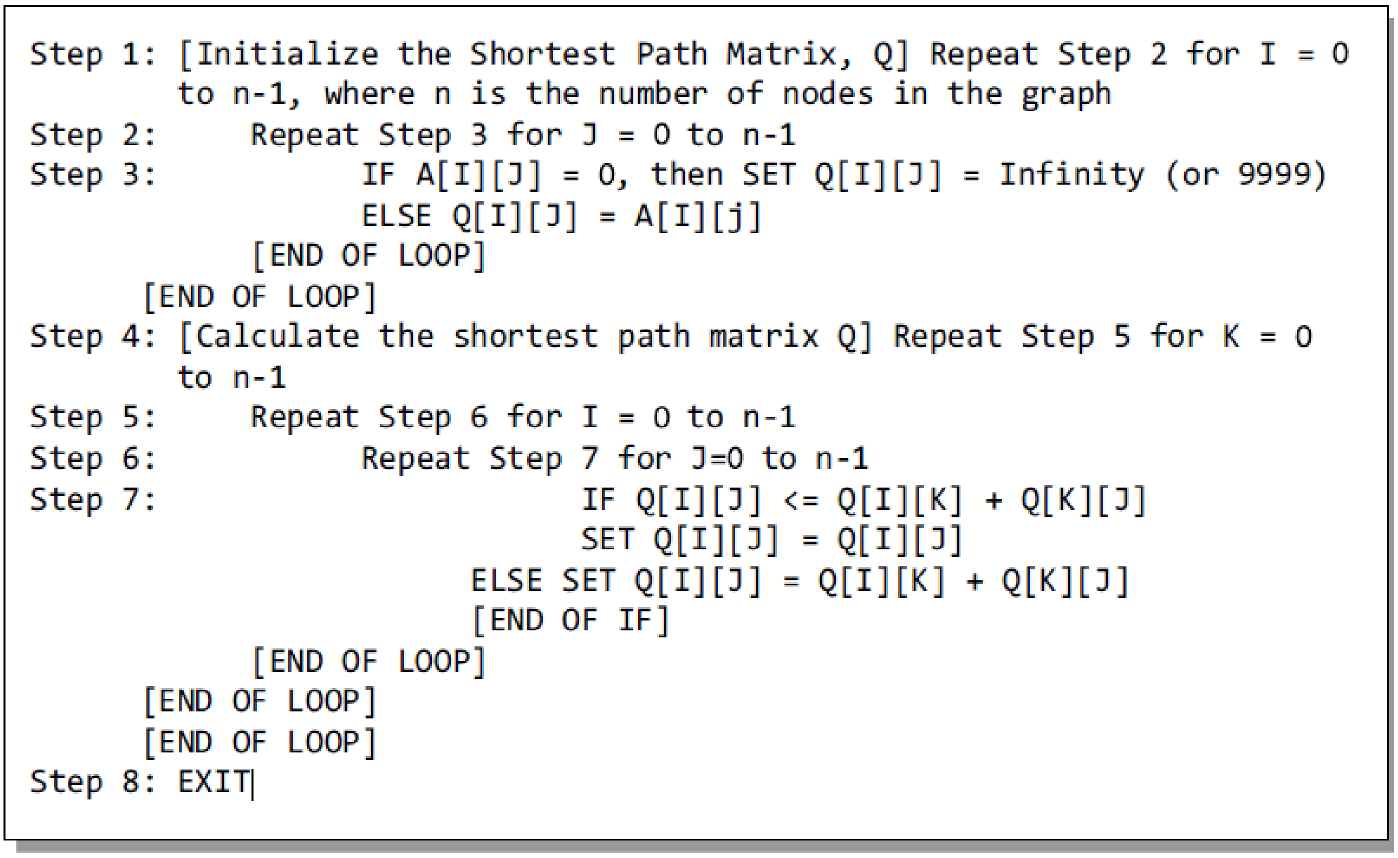
- In Warshall's modified algorithm, we obtaina set of matrices Q0, Q1, Q2, ..., Qm using the formula given below.

  - **Qk[i][j] = Minimum( Mk–1[i][j], Mk–1[i][k] + Mk–1[k][j])**

- Q0 is exactly the same as A with a little difference that every element having a zero value in A is replacedby (∞) in Q0.

- Using the given formula, the matrix Qn will give the path matrix that has the shortest pathbetween the vertices of the graph.

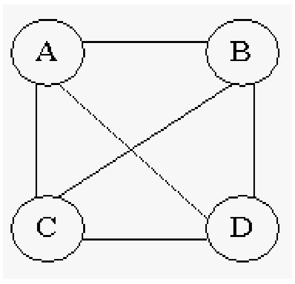
## Modified Warshall's algorithm



```
Step 1: [Initialize the Shortest Path Matrix, Q] Repeat Step 2 for I = 0
        to n-1, where n is the number of nodes in the graph
Step 2:     Repeat Step 3 for J = 0 to n-1
Step 3:           IF A[I][J] = 0, then SET Q[I][J] = Infinity (or 9999)
                  ELSE Q[I][J] = A[I][j]
            [END OF LOOP]
      [END OF LOOP]
Step 4: [Calculate the shortest path matrix Q] Repeat Step 5 for K = 0
        to n-1
Step 5:     Repeat Step 6 for I = 0 to n-1
Step 6:           Repeat Step 7 for J=0 to n-1
Step 7:                       IF Q[I][J] <= Q[I][K] + Q[K][J]
                              SET Q[I][J] = Q[I][J]
                        ELSE SET Q[I][J] = Q[I][K] + Q[K][J]
                        [END OF IF]
            [END OF LOOP]
      [END OF LOOP]
      [END OF LOOP]
Step 8: EXIT
```

**Example:**

**Undirected graph**



**Adjacency matrix A** =
$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 1 & 1 \\ B & 1 & 0 & 1 & 1 \\ C & 1 & 1 & 0 & 1 \\ D & 1 & 1 & 1 & 0 \end{array}$$

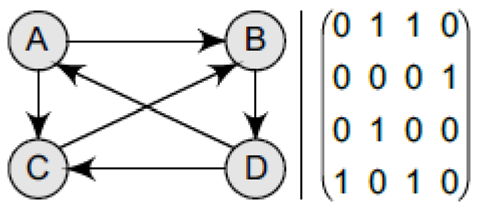
$Q_0$ =
$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & 1 & 1 & 1 \\ B & 1 & \infty & 1 & 1 \\ C & 1 & 1 & \infty & 1 \\ D & 1 & 1 & 1 & \infty \end{array}$$

$Q_1$ =
$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & 1 & 1 & 1 \\ B & 1 & \infty & 1 & 1 \\ C & 1 & 1 & \infty & 1 \\ D & 1 & 1 & 1 & \infty \end{array}$$

$Q_2$ =
$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 3 & 1 & 1 & 1 \\ B & 1 & 3 & 1 & 1 \\ C & 1 & 1 & 3 & 1 \\ D & 1 & 1 & 1 & 3 \end{array}$$

$Q_3$ =
$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 3 & 1 & 1 & 1 \\ B & 1 & 3 & 1 & 1 \\ C & 1 & 1 & 3 & 1 \\ D & 1 & 1 & 1 & 3 \end{array}$$

**Directed graph**



$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

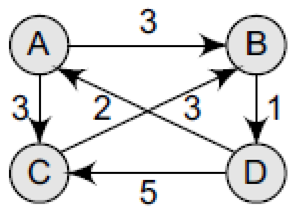
$Q_0$ =
$$\boxed{\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & 1 & 1 & \infty \\ B & \infty & \infty & \infty & 1 \\ C & \infty & 1 & \infty & \infty \\ D & 1 & \infty & 1 & \infty \end{array}}$$

$Q_1$ =
$$\boxed{\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & 1 & 1 & 2 \\ B & 2 & \infty & 2 & 1 \\ C & \infty & 1 & \infty & 2 \\ D & 1 & 2 & 1 & \infty \end{array}}$$

$Q_2$=
$$\boxed{\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 3 & 1 & 1 & 2 \\ B & 2 & 3 & 2 & 1 \\ C & 3 & 1 & 3 & 2 \\ D & 1 & 2 & 1 & 3 \end{array}}$$

$Q_3$=
$$\boxed{\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 3 & 1 & 1 & 2 \\ B & 2 & 3 & 2 & 1 \\ C & 3 & 1 & 3 & 2 \\ D & 1 & 2 & 1 & 3 \end{array}}$$

**Weighted graph**



$$\begin{pmatrix} 0 & 3 & 3 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 3 & 0 & 0 \\ 2 & 0 & 5 & 0 \end{pmatrix}$$

$Q_0$ ==

| | A | B | C | D |
|---|---|---|---|---|
| A | ∞ | 3 | 3 | ∞ |
| B | ∞ | ∞ | ∞ | 1 |
| C | ∞ | 3 | ∞ | ∞ |
| D | 2 | ∞ | 5 | ∞ |

$Q_1$ =

| | A | B | C | D |
|---|---|---|---|---|
| A | ∞ | 3 | 3 | 4 |
| B | 3 | ∞ | 6 | 1 |
| C | ∞ | 3 | ∞ | 4 |
| D | 2 | 5 | 5 | ∞ |

$Q_2$ =

| | A | B | C | D |
|---|---|---|---|---|
| A | 6 | 3 | 3 | 4 |
| B | 3 | 6 | 6 | 1 |
| C | 6 | 3 | 9 | 4 |
| D | 2 | 5 | 5 | 6 |

$Q_3$ =

| | A | B | C | D |
|---|---|---|---|---|
| A | 6 | 3 | 3 | 4 |
| B | 3 | 6 | 6 | 1 |
| C | 6 | 3 | 9 | 4 |
| D | 2 | 5 | 5 | 6 |

## The graph ADT

Abstract datatype Graph
{
**Instances**
A set of vertices V and set of edges E.
}

{
**Operations**
Number of vertices() : return the number of vertices in the graph.
Number of edges() : return the number of edges in the graph.
existsEdge(I,j): return TRUE if edge(I,j)exists,FALSE otherwise.
insertEdge(): insert the edge into the graph.
eraseEdge():delete the edge.
Indegree():return the in-degree of vertex I.
Outdegree():return the out-degree of vertex I.
Degree():return the degree of vertex I.
bfs() : return the nodes in bfs order.
dfs() : return the nodes in dfs order.
}

## Applications of graphs

1. In **circuit networks** where points of connection are drawn as vertices and component wires become the edges of the graph.
2. In **transport networks** where stations are drawn as vertices and routes become the edges of the graph.
3. In **maps** that draw cities/states/regions/ as vertices and routes become the edges of the graph
4. In **program flow analysis** where procedures or modules are treated as vertices and calls to these procedures are drawn as edges of the graph.
5. Graph can be used for finding shortest paths, project planning etc.
6. In **flowcharts or control-flow graphs**, the statements and conditions in a program are represented as nodes and the flow of control is represented by the edges.

## Searching

- **Searching means to find whether a particular value is present in an array or not**.
- There are two popular methods for searching the array elements:

  1) **Linear search**
  2) **Binary search**

The algorithm that should be used depends entirely on how the values are organized in the array.

- If the elements of the array are **arranged sorted order**, then **binary search** should be used as it is more efficient for sorted lists in terms of complexity
- If the elements of the array are **not in sorted order**, then **linear search** should be used.

## Linear Search

- Linear search also called as **sequential search**, is a very simple method used for searching an array for a particular value.
- It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found.
- **Linear search is mostly used to search an unordered list of elements.**

## Algorithm for linear search

```
LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3: Repeat Step 4 while I<=N
Step 4:        IF A[I] = VAL
                    SET POS = I
                    PRINT POS
                    Go to Step 6
               ELSE
                    SET I = I + 1
               [END OF IF]
          [END OF LOOP]
Step 5: IF POS = –1
                  PRINT VALUE IS NOT PRESENTIN THE ARRAY
          [END OF IF]
Step 6: EXIT
```

**Example:**

Consider an array **A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5};**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **10** | **8** | **2** | **7** | **3** | **4** | **9** | **1** | **6** | **5** |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

- The value to be searched is VAL = 7,
- Searching means to find whether the value ‘7’ is present in the array or not. If yes, then it returns the position of its occurrence.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **10** | **8** | **2** | **7** | **3** | **4** | **9** | **1** | **6** | **5** |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

**10!=7**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **10** | **8** | **2** | **7** | **3** | **4** | **9** | **1** | **6** | **5** |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

**8!=7**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **10** | **8** | **2** | **7** | **3** | **4** | **9** | **1** | **6** | **5** |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

**2!=7**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **10** | **8** | **2** | **7** | **3** | **4** | **9** | **1** | **6** | **5** |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

**7=7**

- Here, POS = 3 (index starting from 0).

## Binary Search

- Binary search is a searching algorithm that works efficiently with **a sorted list.**

- In binary search, BEG and END are the beginning and ending positions of the array.
- BEG = lower_bound
- END = upper_bound
- MID is calculated as (BEG + END)/2
- The algorithm will end when A[MID] = VAL

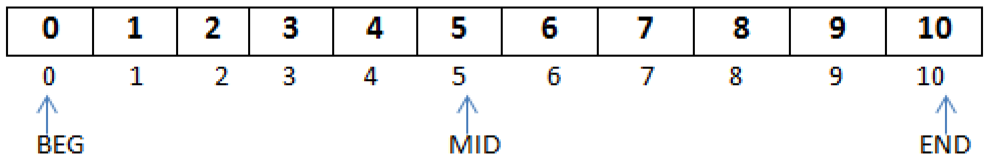
## Algorithm for binary search

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_boundEND = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3:       SET MID = (BEG + END)/2
Step 4:       IF A[MID] = VAL
                  SET POS = MID
                  PRINT POS
                  Go to Step 6
              ELSE IF A[MID] > VAL
                  SET END = MID - 1
              ELSE
                  SET BEG = MID + 1
```

```
                [END OF IF]
            [END OF LOOP]
Step 5: IF POS = -1
                PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
            [END OF IF]
Step 6: EXIT
```

**Example:**

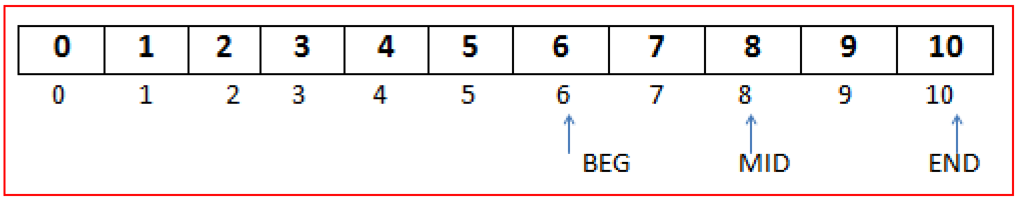
Consider an array **A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};**



The value to be searched is **VAL = 9.**

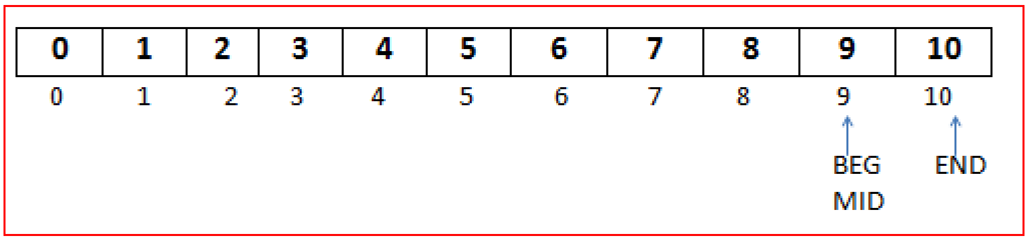
- The algorithm will proceed in the following manner.
  BEG = 0, END = 10, MID = (0 + 10)/2 = 5.
  Now, VAL = 9 and A[MID] = A[5] = 5.
  **A[5] is less than VAL,**
  therefore, we now search for the value in **the second half of the array**.

- So,we change the values of BEG and MID.
  Now, BEG = MID + 1 = 6, END = 10, MID = (6 + 10)/2 =16/2 = 8
  VAL = 9 and A [MID] = A[8] = 8



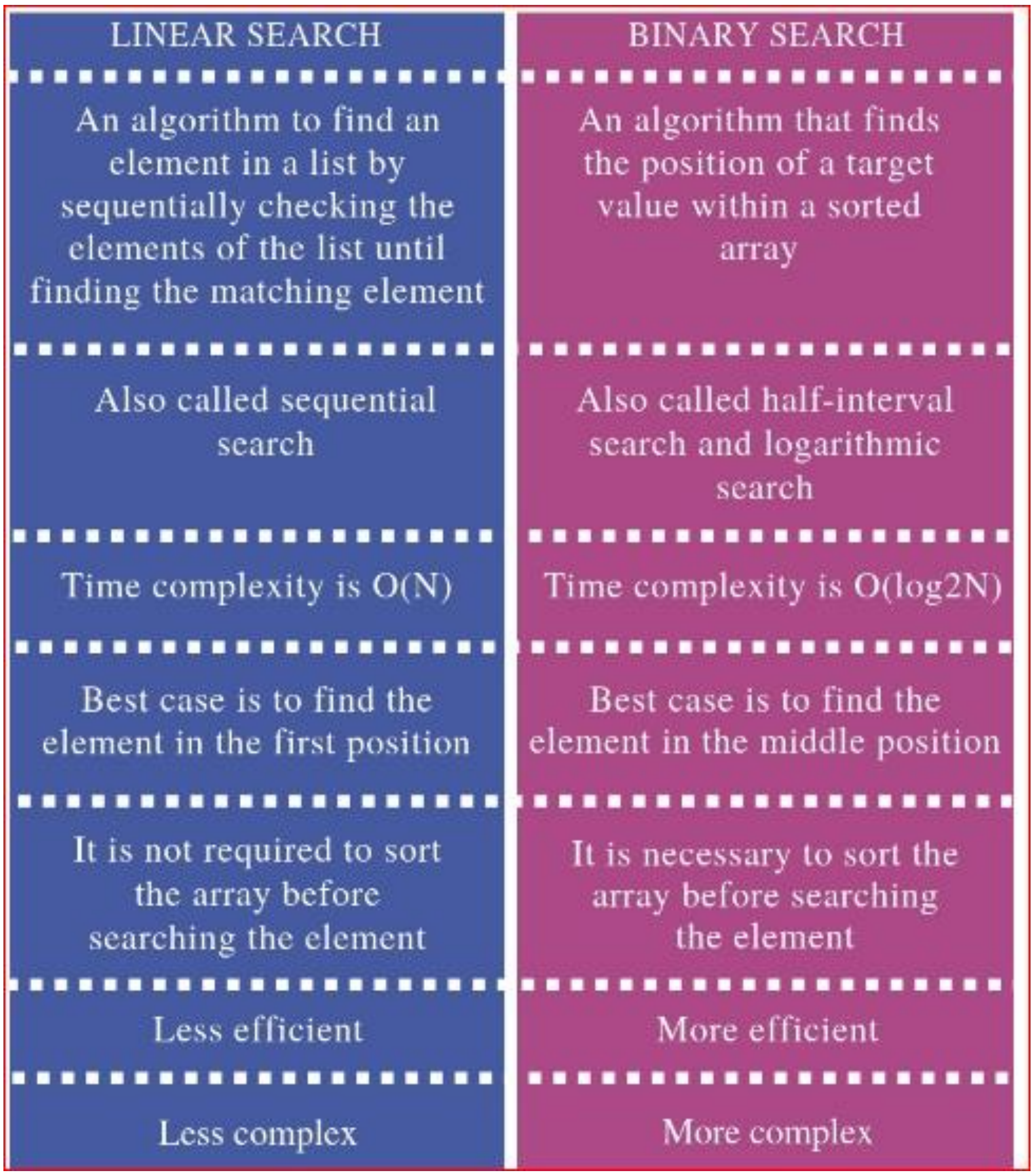
**A[8] is less than VAL,**
therefore, we now search for the value in the **second half of the segment.**

- So, again we change the values of BEG and MID.
  Now, BEG = MID + 1 = 9, END = 10, MID = (9 + 10)/2 = 9



Now, VAL = 9 and A[MID] = 9.

## <u>Comparison of linear search and binary search</u>



| LINEAR SEARCH | BINARY SEARCH |
|---|---|
| An algorithm to find an element in a list by sequentially checking the elements of the list until finding the matching element | An algorithm that finds the position of a target value within a sorted array |
| Also called sequential search | Also called half-interval search and logarithmic search |
| Time complexity is O(N) | Time complexity is O(log2N) |
| Best case is to find the element in the first position | Best case is to find the element in the middle position |
| It is not required to sort the array before searching the element | It is necessary to sort the array before searching the element |
| Less efficient | More efficient |
| Less complex | More complex |

## Sorting

- **Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending.**
- There are two types of sorting
    - **internal sorting ( sorting the data in the computers memory)**
        - ✓ Bubble sort
        - ✓ Quick sort
    - **External sorting (sorting the data stored in files)**

## Internal sorting methods

## 1)Bubble sort

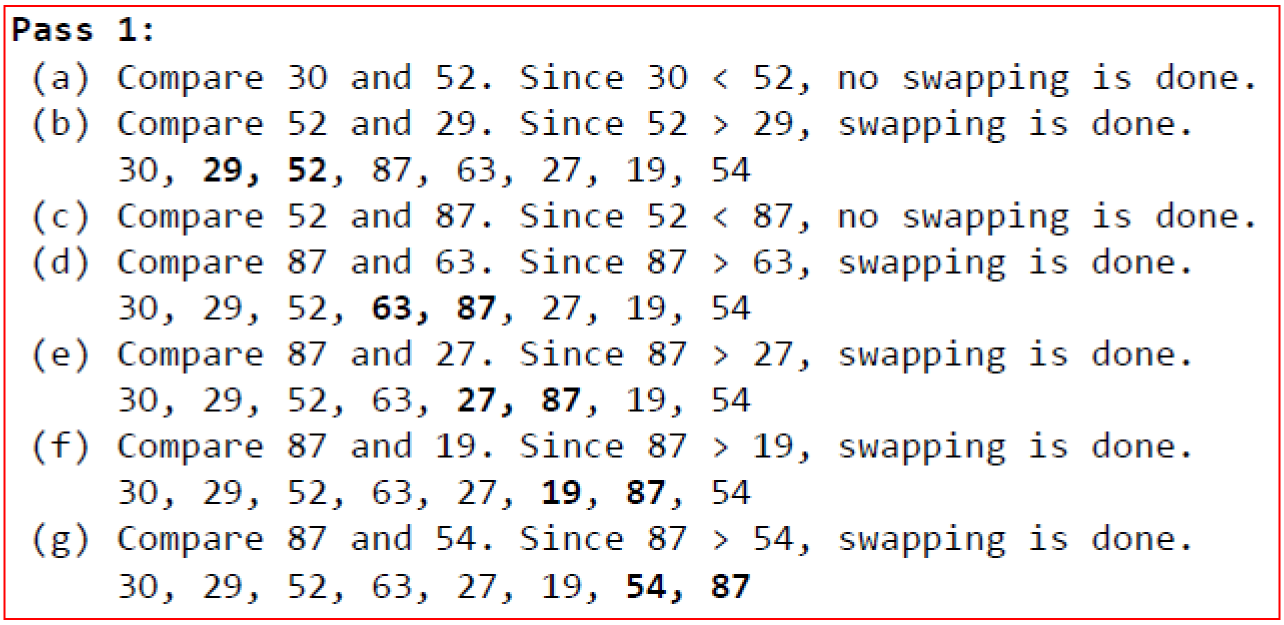
- This is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment.
- In bubble sorting cosecutive adjacent pairs of elements in the array are compared with each other.
- If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one.
- This process will continue till the list of unsorted elements exhausts.
- This procedure of sorting is called bubble sorting because elements 'bubble' to the top of the list.
- Note that at the end of the first pass, the largest element in the list will be placed at its proper position (i.e., at the end of the list).

### Algorithm for bubble sort

```
BUBBLE_SORT(A, N)
Step 1: Repeat Step 2 For I = 0  to N-1
Step 2: Repeat step 3 For J = 0 to N - I
Step 3:    IF A[J] > A[J + 1]
                SWAP A[J] and A[J+1]
            [END OF INNER LOOP]
         [END OF OUTER LOOP]
 Step 4: EXIT
```

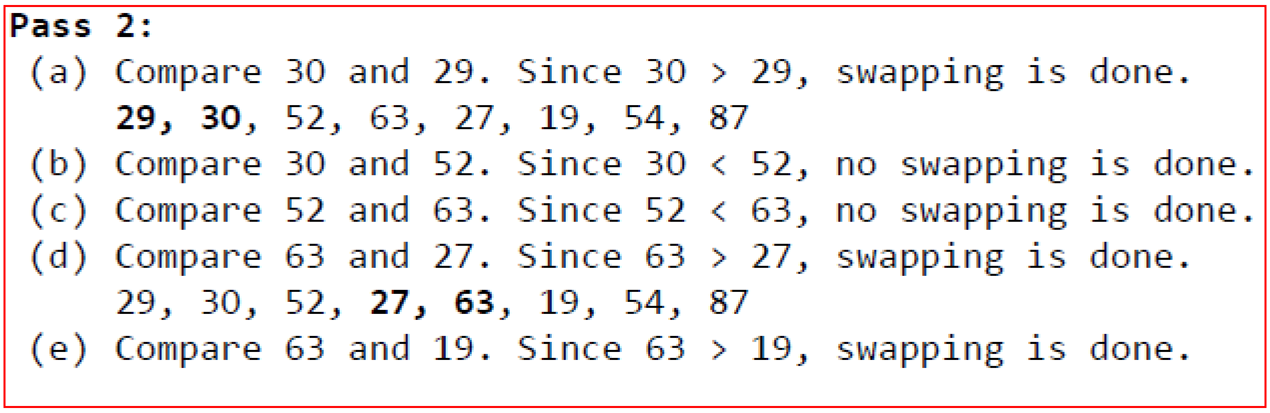
**Example :**

Consider an array A[] that has the following elements:
**A[] = {30, 52, 29, 87, 63, 27, 19, 54}**



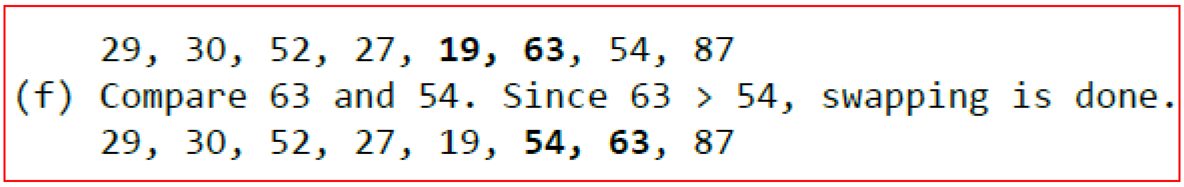
**Pass 1:**

(a) Compare 30 and 52. Since 30 < 52, no swapping is done.
(b) Compare 52 and 29. Since 52 > 29, swapping is done.
30, **29, 52**, 87, 63, 27, 19, 54
(c) Compare 52 and 87. Since 52 < 87, no swapping is done.
(d) Compare 87 and 63. Since 87 > 63, swapping is done.
30, 29, 52, **63, 87**, 27, 19, 54
(e) Compare 87 and 27. Since 87 > 27, swapping is done.
30, 29, 52, 63, **27, 87**, 19, 54
(f) Compare 87 and 19. Since 87 > 19, swapping is done.
30, 29, 52, 63, 27, **19, 87**, 54
(g) Compare 87 and 54. Since 87 > 54, swapping is done.
30, 29, 52, 63, 27, 19, **54, 87**



**Pass 2:**

(a) Compare 30 and 29. Since 30 > 29, swapping is done.
**29, 30**, 52, 63, 27, 19, 54, 87
(b) Compare 30 and 52. Since 30 < 52, no swapping is done.
(c) Compare 52 and 63. Since 52 < 63, no swapping is done.
(d) Compare 63 and 27. Since 63 > 27, swapping is done.
29, 30, 52, **27, 63**, 19, 54, 87
(e) Compare 63 and 19. Since 63 > 19, swapping is done.



29, 30, 52, 27, **19, 63**, 54, 87
(f) Compare 63 and 54. Since 63 > 54, swapping is done.
29, 30, 52, 27, 19, **54, 63**, 87

**Pass 3:**

(a) Compare 29 and 30. Since 29 < 30, no swapping is done.
(b) Compare 30 and 52. Since 30 < 52, no swapping is done.
(c) Compare 52 and 27. Since 52 > 27, swapping is done.
29, 30, **27, 52**, 19, 54, 63, 87
(d) Compare 52 and 19. Since 52 > 19, swapping is done.
29, 30, 27, **19, 52**, 54, 63, 87
(e) Compare 52 and 54. Since 52 < 54, no swapping is done.

**Pass 4:**

(a) Compare 29 and 30. Since 29 < 30, no swapping is done.
(b) Compare 30 and 27. Since 30 > 27, swapping is done.
29, **27, 30**, 19, 52, 54, 63, 87
(c) Compare 30 and 19. Since 30 > 19, swapping is done.
29, 27, **19, 30**, 52, 54, 63, 87
(d) Compare 30 and 52. Since 30 < 52, no swapping is done.

**Pass 5:**

(a) Compare 29 and 27. Since 29 > 27, swapping is done.
**27, 29**, 19, 30, 52, 54, 63, 87
(b) Compare 29 and 19. Since 29 > 19, swapping is done.
27, **19, 29**, 30, 52, 54, 63, 87
(c) Compare 29 and 30. Since 29 < 30, no swapping is done.

**Pass 6:**

(a) Compare 27 and 19. Since 27 > 19, swapping is done.
**19, 27**, 29, 30, 52, 54, 63, 87
(b) Compare 27 and 29. Since 27 < 29, no swapping is done.

**Pass 7:**

(a) Compare 19 and 27. Since 19 < 27, no swapping is done.

**Observe that the entire list is sorted now.**

## Quick sort algorithm

- Quick sort is **also known as partition exchange sort**.
- This algorithm **works by using a divide-and-conquer strategy** to divide a single unsorted array into two smaller sub-arrays.

- The quick sort algorithm works as follows:

  1. Select an element pivot from the array elements.
  2. Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it (equal values can go either way). After such a partitioning, the pivot is placed in its final position. This is called the ***partition*** operation.
  3. Recursively sort the two sub-arrays thus obtained. (One with sub-list of values smaller than that of the pivot element and the other having higher value elements.)

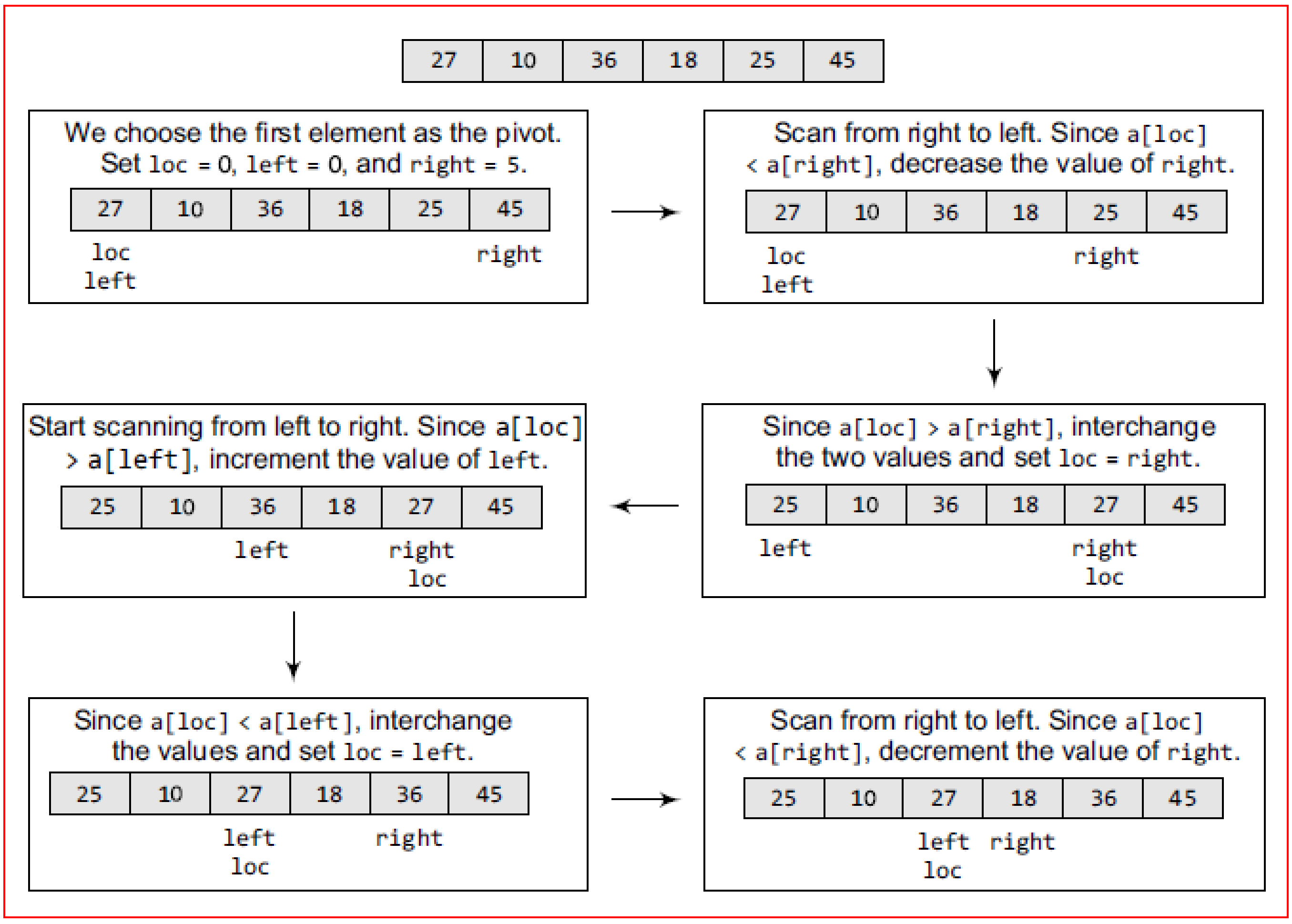
### Algorithm for Quick sort

```
PARTITION (ARR, BEG, END, LOC)

Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG = 0
Step 2: Repeat Steps 3 to 6 while FLAG = 0
Step 3: Repeat while ARR[LOC] <= ARR[RIGHT] AND LOC != RIGHT
                SET RIGHT = RIGHT - 1
        [END OF LOOP]
Step 4: IF LOC = RIGHT
                SET FLAG = 1
        ELSE IF ARR[LOC] > ARR[RIGHT]
                SWAP ARR[LOC] with  ARR[RIGHT]
                SET LOC = RIGHT
        [END OF IF]
Step 5: IF FLAG = 0
                Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
                SET LEFT = LEFT + 1
                [END OF LOOP]
Step 6:         IF LOC = LEFT
                        SET FLAG = 1
                ELSE IF ARR[LOC] < ARR[LEFT]
                        SWAP ARR[LOC] with  ARR[LEFT]
                        SET LOC = LEFT
                [END OF IF]
        [END OF IF]
Step 7: [END OF LOOP]
Step 8: END
```

```
QUICK_SORT (ARR, BEG, END)

Step 1: IF (BEG < END)
              CALL PARTITION (ARR, BEG, END, LOC)
              CALL QUICKSORT(ARR, BEG, LOC - 1)
              CALL QUICKSORT(ARR, LOC + 1, END)
        [END OF IF]
Step 2: END
```

**Example:**

Sort the elements given in the following array using quick sort algorithm



| | | | | | |
|---|---|---|---|---|---|
| 27 | 10 | 36 | 18 | 25 | 45 |

We choose the first element as the pivot. Set loc = 0, left = 0, and right = 5.

| | | | | | |
|---|---|---|---|---|---|
| 27 | 10 | 36 | 18 | 25 | 45 |

Scan from right to left. Since a[loc] < a[right], decrease the value of right.

| | | | | | |
|---|---|---|---|---|---|
| 27 | 10 | 36 | 18 | 25 | 45 |

Since a[loc] > a[right], interchange the two values and set loc = right.

| | | | | | |
|---|---|---|---|---|---|
| 25 | 10 | 36 | 18 | 27 | 45 |

Start scanning from left to right. Since a[loc] > a[left], increment the value of left.

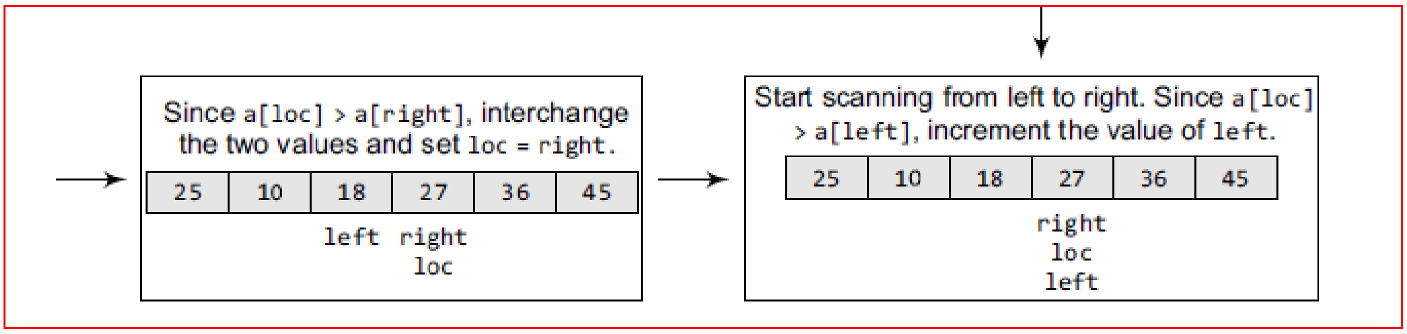
| | | | | | |
|---|---|---|---|---|---|
| 25 | 10 | 36 | 18 | 27 | 45 |

Since a[loc] < a[left], interchange the values and set loc = left.

| | | | | | |
|---|---|---|---|---|---|
| 25 | 10 | 27 | 18 | 36 | 45 |

Scan from right to left. Since a[loc] < a[right], decrement the value of right.

| | | | | | |
|---|---|---|---|---|---|
| 25 | 10 | 27 | 18 | 36 | 45 |



Since a[loc] > a[right], interchange the two values and set loc = right.

| | | | | | |
|---|---|---|---|---|---|
| 25 | 10 | 18 | 27 | 36 | 45 |

Start scanning from left to right. Since a[loc] > a[left], increment the value of left.

| | | | | | |
|---|---|---|---|---|---|
| 25 | 10 | 18 | 27 | 36 | 45 |