:: UNIT – 1 ::

TOPIC ➔ ALGORITHM ANALYSIS

CONTENTS:

1.1 – The Analysis of Algorithm

Data Structure:

- A systematic way of accessing and organizing data is known as *Data Structure.*

Algorithm:

- An Algorithm is a step by step sequence of instruction to solve the computational problem in a finite amount of time in an English language.
- Algorithm can be in any natural language.
- In other words, an algorithm is logical representation of instructions which should be executed to perform a meaningful task.
- Every algorithm should have the five characteristics:

  Input: Any algorithm should take either zero or more inputs.

  Output: Any algorithm should produce one or more outputs.

  Definiteness: Each and every step of algorithm should be defined clearly.

  Effectiveness: Any person can be able to calculate the values involved in the procedure (process) of algorithm. (Ex: Dry Run)

  Termination: Any algorithm must be terminated with finite no. of steps.

- After designing an algorithm, the analysis of algorithm comes.

- The analysis of algorithm is used to check and predict the correctness of an algorithm.
- The analysis of algorithm means tracing the all steps of instructions, checking logical correctness and mathematical techniques (equations).
- The simple design of an algorithm makes easier to be implemented. That means, the simplicity of an algorithm makes us to easily analyze (dry run) the problem.

1.2 – Time and Space Complexities

- Complexity of an algorithm is also known as computational complexity.
- The analysis of algorithm means prediction of resources which includes memory, logic gates, time and etc.
- Computational complexity is a characterization of the time and space requirements for solving a problem by an algorithm.
- The algorithm mainly depends on performance analysis, measurement.
- The analysis of the program requires two main considerations: Space Complexity and Time Complexity
- The time complexities of an algorithm / program mean the amount of time that computer needs to run / execute the program.
- The space complexities of an algorithm / program mean the amount of memory (space) requires by the computer to run /execute the program.

➢ ***Time Complexity:***

✓ This means the amount of time requires by the computer to run any program / algorithm.

✓ For any program, the exact time will depend on the implementation of an algorithm, language used to execute that program, capabilities of CPU and CPU speed, and other hardware specification.

✓ To measure the time complexity accurately, we have to count all types of operations performed in an algorithm.

- ✓ That means, if we know the time taken by each instruction then we can easily compute the time taken by an algorithm to complete its execution.
- ✓ The time complexity depends on the amount of data inputted to an algorithm.
- ✓ This time is varying from machine to machine.
- ✓ For Example: If we have an algorithm with instruction: a = a + 1. As this is an independent statement (without any loop / condition), so the no. of time it will execute is 1. So, the frequency count for this algorithm is 1 only.
- ✓ Similarly if we have another algorithm like,

```
for(i=1; i<=n; i++)
        a= a+1;
```

As we can see that the algorithm contains a loop and the no. of times the statement (a = a + 1) executed is n. So, the frequency count for is n.

- ✓ Now, consider following algorithm,

```
for(i=1; i<=n; i++)
    for(j = 1;j<=n;j++)
        a= a+1;
```

As we can see that the algorithm contains loops and the no. of times the statement (a = a + 1) executed is n^2. So, the frequency count is n^2.

- ➢ ***Space Complexity:***

- ✓ The amount of memory is required to run and completion of an algorithm or program is known as Space Complexity.
- ✓ The space complexity is required to study when the program is run on multi-user system.
- ✓ The space needed by program consists of following components:
    - ✓ Fixed Space Requirement: This includes the instruction space for simple variables, fixed size structured variables and constants.
    - ✓ Variable Space Requirement: This consists of space needed by structured variables whose size depends on particular

instance of variables (Dynamic Allocation). It also includes additional space required when the function uses recursion.

- ✓ Except these components, a program also require data space, stack space, return address space and etc.

1.3 – Asymptotic Notation

- ❑ When we study algorithm, we are interested in characterizing them according to their efficiency.
- ❑ We are usually interesting in the order of growth of the running time of an algorithm, not an exact running time. This is also referred to as Asymptotic Running Time.
- ❑ The Asymptotic Notation gives us a method for classifying functions according to their rate of growth.
- ❑ The Asymptotic Notations are the “Big O”, “Big Omega”, “Big Theta” are used in Asymptotic Analysis.

1.4 – Classes of Algorithm

- ❑ Any Algorithm can be classified into two categories:
    - By Implementation Way: This category is classified in:
        - Recursion / Iteration:
            - In this the recursive algorithm invokes itself until a certain condition is true.
            - And iterative algorithm invokes repeatedly (like loops).
        - Logical:
            - This is used in the computation and used by the control component to determine the way (if condition).
            - Any change in this may change well-defined algorithm.

- Serial / Parallel / Distributed:
  - A computer which can execute one instruction of an algorithm at a time is known as *Serial Computers*. And this type of algorithm is known as *Serial Algorithm.*
  - *Parallel algorithms* take advantage of computer architectures where several processors can work on a problem at the same time.
  - *Distributed algorithms* utilize multiple machines connected with a network.
  - The parallel and distributed algorithms divide the problem into sub problems.
- Deterministic or Non-Deterministic:
  - Deterministic algorithms solve the problem with exact decision at every step of the algorithm.
  - Non-Deterministic algorithm solves problems via guessing, although typical guesses are made more accurate through the use of heuristics.
  - Exact or Approximate:
    - Many algorithms have exact solution, whereas some have approximate solution.
    - The approximation algorithm tries to find an approximation that is close to the true / final solution.

- By Design Paradigm: Followings are different types of paradigms which are different from each other:
  - Brute-force or Exhaustive Search:
    - This is the natural method for designing an algorithm of trying every possible solution to see which is best.
  - Divide and Conquer (Also known as *Decrease and Conquer* algorithm):

- This type of algorithm repeatedly divides the problems into smaller problems till we get solution. The example of this is Merge Sorting.

- Dynamic Programming & The Greedy Method:
  - Dynamic Programming avoids repeating calculation of results for previously processed inputs.
  - A greedy algorithm is similar to a dynamic programming, but the difference is that solution to the sub problems does not have to be known at each stage; instead a "greedy" choice can be made of what looks best at that time.
  - A greedy algorithm does not give accurate answer but it will be the fastest method.
  - The most popular greedy algorithm is finding the minimal spanning tree given by Huffman.

1.5 – BIG – O or BIG – OH NOTATION

- There are some notations used to determine the time as well as space complexities of the algorithms. Those are Big-Oh, Big-Omega, Little-Oh, Big-Theta and etc.
- Big-O notation (A mathematical tool) was introduced in P. Bachmann's 1892 book Analytische Zahlentheorie.
- Using this notation, the time taken by the algorithm and the space required to run the algorithm can be determined.
- This information is useful to set the prerequisites of algorithms and to develop and design efficient algorithms.
- Developers use this notation to reach to best solution for the given problem.
- There are three different types of time complexities for analyzing algorithm:

  Best Case Time Complexity, Average Case Time Complexity, Worst Case Time Complexity

- The worst case is when an algorithm requires a maximum number of iterations or steps to search and find out the target value in the array. That is the target value will be found at nth position of the array. (i.e. f(n) = n)
- The best case is when the number of steps is as less as possible. That is, if the target value found at first position then only one iteration is executed. (i.e. f(n) = 1)
- The average case falls between best and worst cases. If the target value is found at n / 2nd position, on an average we need to compare the target value with only half of the element in the array. (i.e. f(n) = n / 2)
- For Example: For the quick sort the worst case complexity is O(n2) whereas for Bubble Sort the average case complexity is O(n2).
- Thus, quick sort can be graded as the better than bubble sort.
    - Based on time complexity representation of the Big-O notation, the algorithm can be categorized as:
        - Constant Time – O(1): That means the algorithm requires fixed number of steps regardless of the size of the task. For Example: Push and Pop operation for a stack.
        - Linear Time – O(n): That means the algorithm requires the number of steps relative to the size of the task. For Example: Traversal of a list with n elements.
        - Quadratic Time – O(n^2): The number of operations in relative to the size of the task squared. For Example: Comparing two 2D- Arrays.
        - Logarithmic Time – O(log n): For Example: Binary Search Operations.
        - O (n log n): For Example: Sorting algorithms like Quick and Merge.
    - Big O is asymptotic execution time of an algorithm.
    - In this, f(n) represents the computing time of some algorithm and g(n) represents a standard functions. That

means for BIG-O notation, it is also pronounced as f(n) is big-oh of g(n).

- So, g(n) is an asymptotic upper bound for f(n). The equation is: $|f(n)| \leq c\ |g(n)|$, for $n \geq n0$.

1.6 – BIG-OMEGA NOTATION (Ω)

- As we know, the Big-Oh Notation provides an asymptotic way of saying that the function is "less than or equal to" another function; likewise the Big-Omega notation provides an asymptotic way to denote the function is "greater than or equal to" another function.
- In this, f(n) represents the computing time of some algorithm and g(n) represents a standard functions. That means the Big-Omega notation (also pronounced as f(n)) is big-omega of g(n).
- So, g(n) is an asymptotic lower bound for f(n). The equation is: $|f(n)| \geq c\ |g(n)|$, for $n \geq n0$.

ASYMPTOTIC ANALYSIS

- This type of analysis is based on the idea that as the problem size grows, the complexity can be describe as a simply to some known function.
- This is incorporated in the notations for asymptotic performance.
- When the execution time of an algorithm varies and other factors which may differ from computer to computer then this type of analysis is used.
- The absolute growth depends on the machine used to execute the program, the compiler used to construct the program, and many other factors.
- We would like to have a way of describing the inherent complexity of a program (or piece of a program), independent of machine / compiler considerations.
- This means that we must not try to describe the absolute time or storage needed.
- We must instead concentrate on a "proportionality" approach, expressing the complexity in terms of its relationship to some known function. This type of analysis is known as asymptotic analysis.

TOPIC → ADVANCED CONCEPTS OF C

CONTENTS:

- ➔ **Dynamic allocation and de-allocation of memory (malloc(), calloc() and free())**
- ➔ **Dangling Pointer Problem**
- ➔ **Enumerated Constants**

Memory Allocation / Deallocation Functions

- ❑ To allocate memory dynamically (i.e. during program execution), C language provides memory allocations functions.
- ❑ The dynamic memory allocation means a compiler allocates memory to pointer variable at runtime.

  - **malloc():**
    - Description: Allocate memory in bytes and return first byte of address to the pointer.
    - Syntax: void *malloc(byte size);
    - Example:

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
void main()
{
        int *no1,*no2,*ans;
        clrscr();
        no1 =(int *)malloc(sizeof(int));
        no2 =(int *)malloc(sizeof(int));
        ans =(int *)malloc(sizeof(int));
        printf("Enter no 1 : ");
        scanf("%d",no1);
        printf("Enter no 2 : ");
        scanf("%d",no2);
        *ans = *no1 + *no2;
        printf("\nSum = %d",*ans);
        free(no1);
        free(no2);
        free(ans);
        getch();
}
```

Output:

```
Enter no 1 : 5
Enter no 2 : 10
Sum = 15
```

- **calloc():**
  - Description: Allocate memory of array elements and initialize to zero, return address of first element. Calloc() is used to allocate memory for array / structure type variables. The default value for variable allocated by calloc() is 0 whereas by malloc() is garbage.

  - Syntax: void *calloc(variable, size);
  - Example:

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
void main()
{
        int i,*p,*q;
        clrscr();
        p = (int *)malloc(5*sizeof(int));
        q = (int *)calloc(6,sizeof(int));
        for(i=0;i<5;i++)
                printf("%d\t%u\n",*p,p++);
        printf("\n\n");
        for(i=0;i<5;i++)
                printf("%d\t%u\n",*q,q++);
        getch();
}
```

Output:

```
2162        1454
-31954      1456
11528       1458
8222        1460
17          1462

0           1468
0           1470
0           1472
0           1474
0           1476
```

- **realloc():**
  - Description: Allocate again previously allocated memory by malloc() or calloc().
  - Syntax: void *realloc(variable, newsize);
  - Example:

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
void main()
{
        char *str;
        clrscr();
        printf("Enter Name : ");
        str = (char *)malloc(sizeof(5));
        gets(str);
        printf("Name is : %s",str);
        printf("\nEnter Name : ");
        str = (char *)realloc(str,sizeof(10));
        gets(str);
        printf("Name is :%s",str);
        getch();
}
```

Output:

```
Enter Name : rpbc
Name is : rpbc
Enter Name : rpbc Rajkot
rpbc rajkot
```

- **free():**
  - Description: Release allocates memory, allocated by malloc(), calloc() and realloc().
    Syntax: free(pointer);

Dangling Pointer Problem

- Dangling Pointer arise when an object is deleted or de-allocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the de-allocated memory.
- In Short, Pointer pointing to non-existing memory location is called Dangling Pointer Problem.
- There are different ways where Pointer acts as Dangling Pointer.

1)
```
#include<stdio.h>
void main()
{
        char *p1 = malloc(value);
        ….
        ….
        free(p1);       //p1 now becomes dangling pointer…
}                       //Solution: write p1 = NULL after free();
```

2)
```
#include<stdio.h>
void main()
{
        char *p1 = NULL;
        ….
        {
                char ch;
                p1 = &ch;
        }
  /*Here p1 becomes dangling as it points to ch which is in inner block.*/
}
```

Example-1:

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
void main()
{
        int *p;
        clrscr();
        p = (int *)malloc(2 * sizeof(int));
        p[0] = 10;
        p[1] = 20;
        printf("\nNumber – 1 = %d",p[0]);
        printf("\nNumber – 2 = %d",p[1]);
        free(p);
        //p=NULL;
        printf("\n%d",p[0]);
        getch();
}
```

Example-2:

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

int *call();
void main()
{
        int *p;
        clrscr();
        p = call();
        fflush(stdin);
        printf("%d",*p);
        getch();
}

int *call()
{
        int x = 25;         //Make this static and check…
        ++x;
        return &x;
}
```

Enumerated Constants

- Enumerated Constants can be defined with "enum" keyword.
- *enum* is user defined data type for integral type of constants.

1)

```
#include <stdio.h>
void main()
{
        enum {RED=5, YELLOW, GREEN=4, BLUE};
        printf("RED = %d\n", RED);
        printf("YELLOW = %d\n", YELLOW);
        printf("GREEN = %d\n", GREEN);
        printf("BLUE = %d\n", BLUE);
 }
```

2)

```
#include<stdio.h>
enum week{Sun, Mon, Tue, Wed, Thu, Fri, Sat};
void main()
{
        enum week day;
        day = Mon;
        printf("%d",day);
}
```

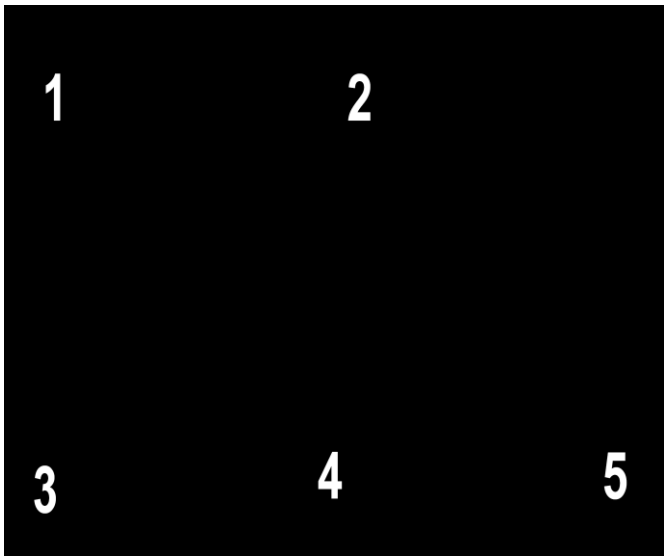
TOPIC → Graph

CONTENTS:

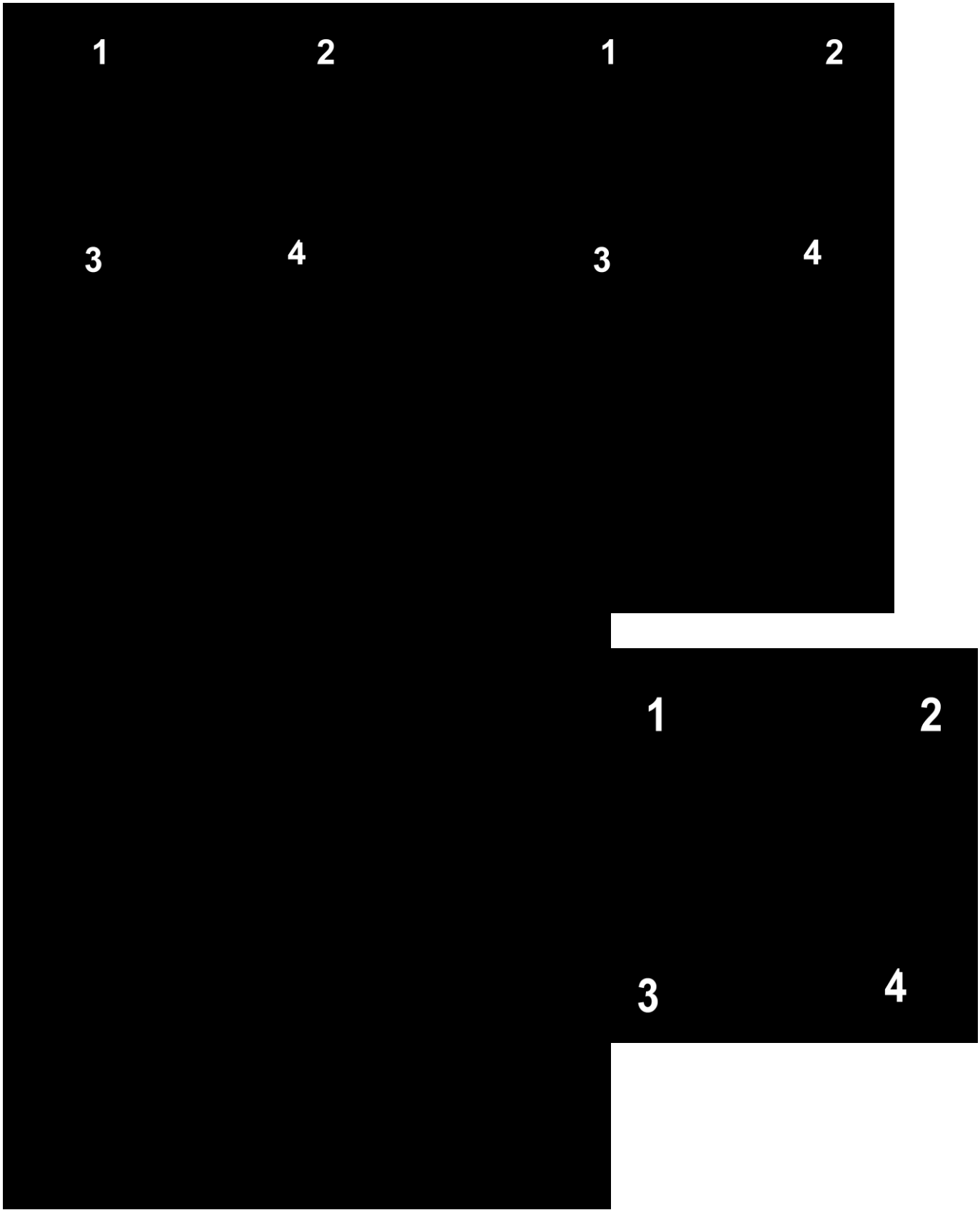
- ➔ **Adjacency Matrix and Adjacency List**
- ➔ **Graph Traversal – BFS, DFS**
- ➔ **Shortest Path Problem**
- ➔ **Minimal Spanning Tree**

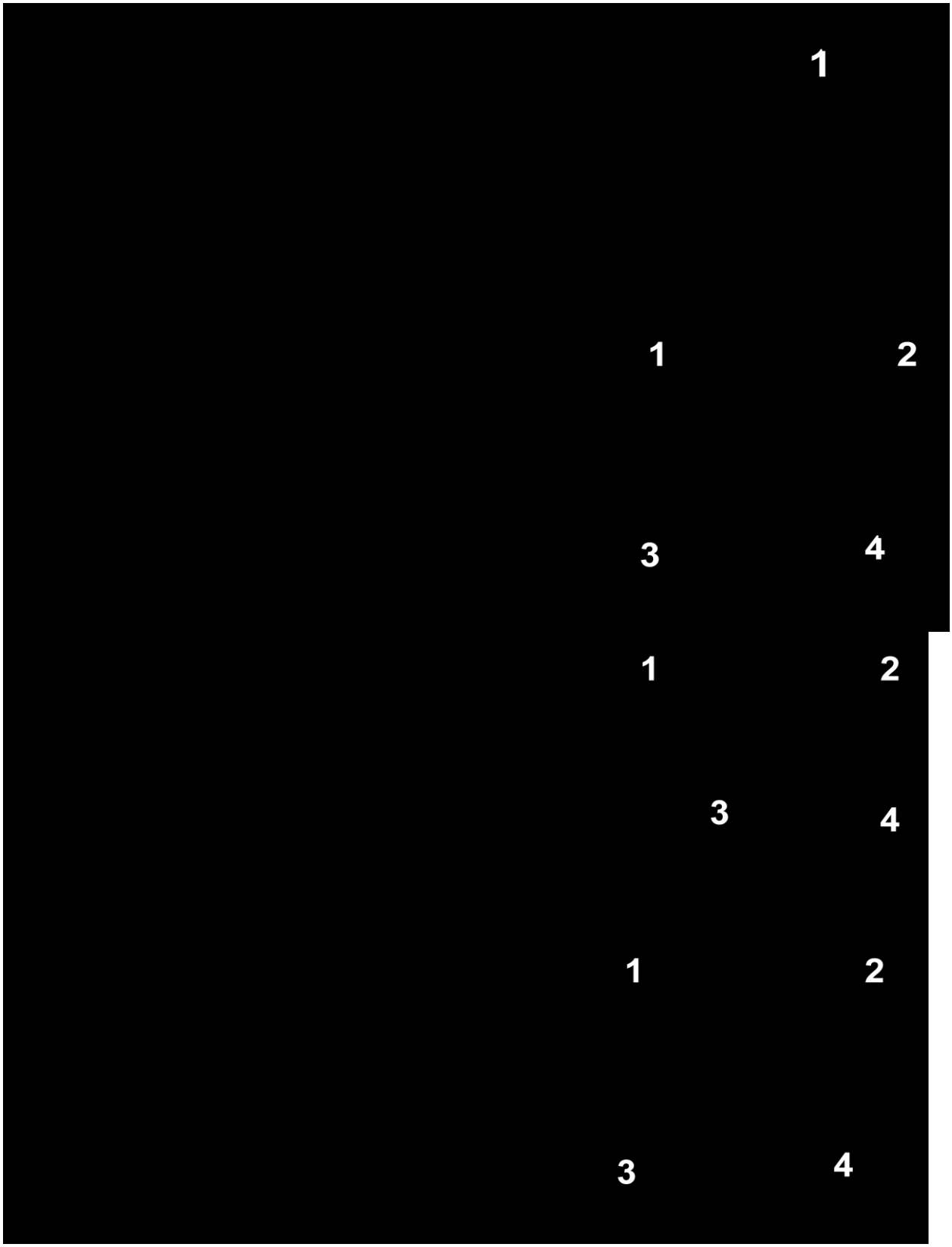
INTRODUCTION:

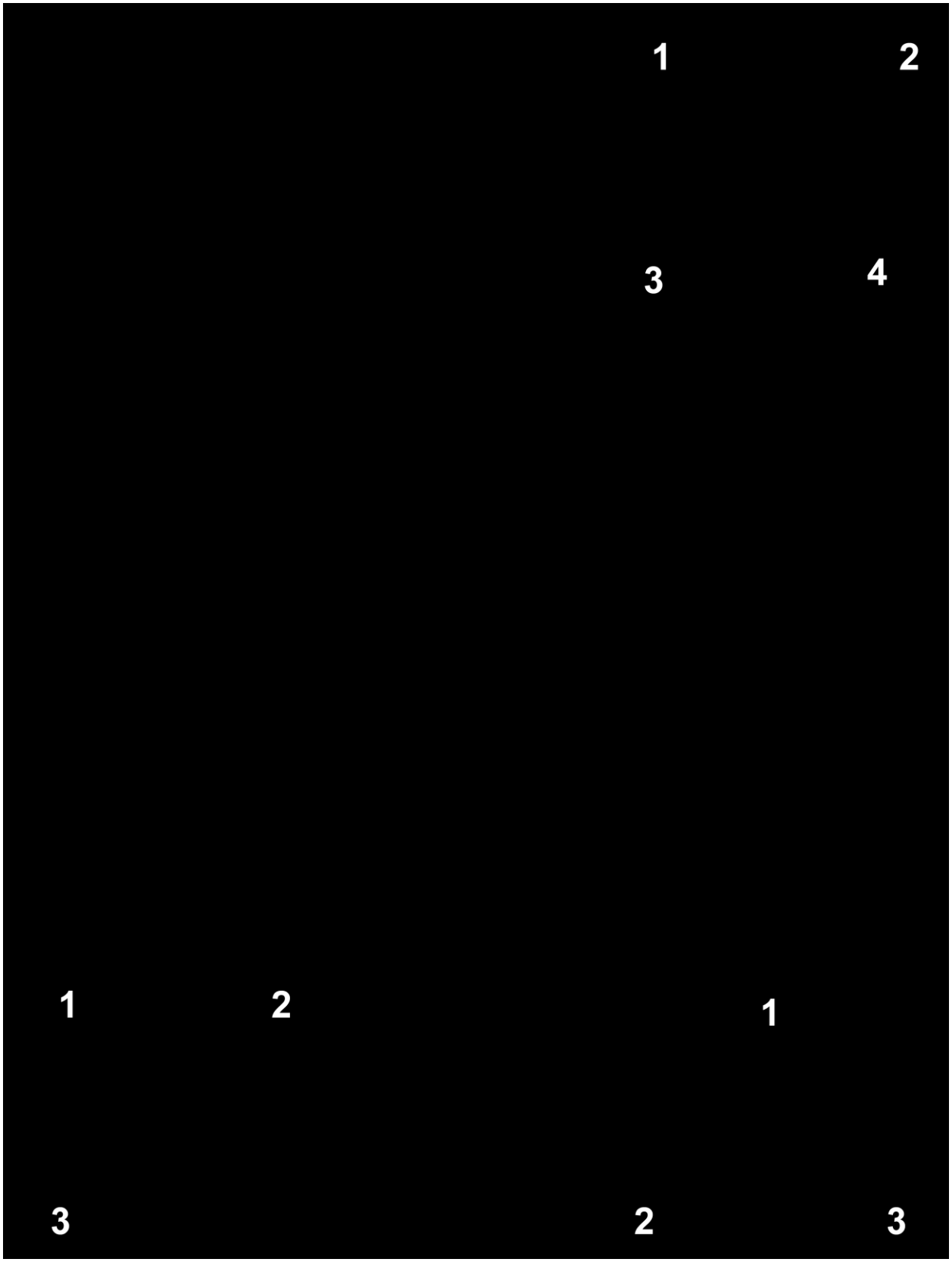
⇒ A graph is non-linear data structure.
⇒ It is a general tree with no parent-child relationship.
⇒ This data structure is useful in many fields related to science / computer science.
⇒ For Example: Graphs are used in mapping, games, puzzles, networks, engineering, transportation and etc.
⇒ In general, graphs represent a relatively less relationship between the data items.
⇒ It is a way of representing relationships that exists between pairs of objects.
⇒ *A graph G includes* set of Vertices V (called nodes) and set of Edges (E). It is represented as Graph G = (V, E).
⇒ *The Vertices (or v)* is a finite and non empty set and Edges (or E) denotes pair of connected vertices.
⇒ In other words, we can say that graph is a set of objects together with a collection of connected pair.
⇒ The Vertices also referred to as *nodes* whereas the edges also referred to as *arcs*.



- The single unit of Vertices is known as Vertex. For Ex: 1 in this graph.
- In this graph G, we have set of Vertices V = {1,2,3,4,5} and set of Edges E = {(1,2), (1,3), (2,3), (3,4), (4,5)}.
- The graph has very limited relationship between the vertices (nodes). As we can see there is no direct relationship between 1 and 4 although there are connected through 3.


1
2
1
2
3
4
3
4
1
2
3
4


1
1
2
3
4
1
2
3
4
1
2
3
4


1
2
3
4
1
2
1
3
2
3

Adjacency Matrix and Adjacency List:

There are different ways to represent any graph.

1) Adjacency Matrix Representation:

⇒ It is also known as Array Representation or Sequential representation of graph.

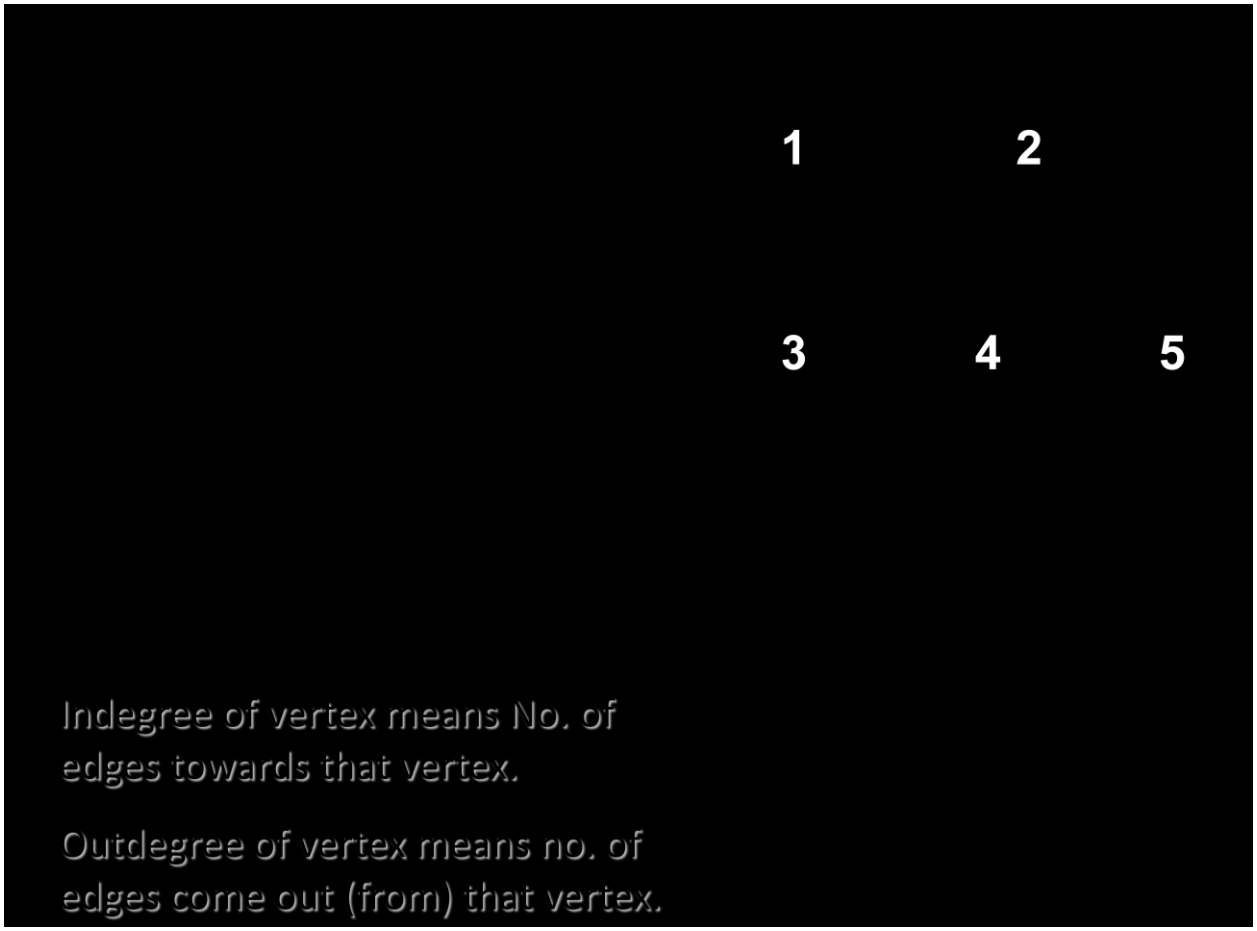
In this, Graph G=(V, E) is a matrix A(a_{ij}) such that

$$a_{ij} = \begin{cases} 1 \text{ if edge (i, j) belongs to E} \\ 0 \text{ otherwise} \end{cases}$$

=> That means, if there is an edge from Vertex 1 to Vertex 2 then a_{ij} has value 1 else a_{ij} has value 0.


1
2
3
4
5


1
2
3
4
5

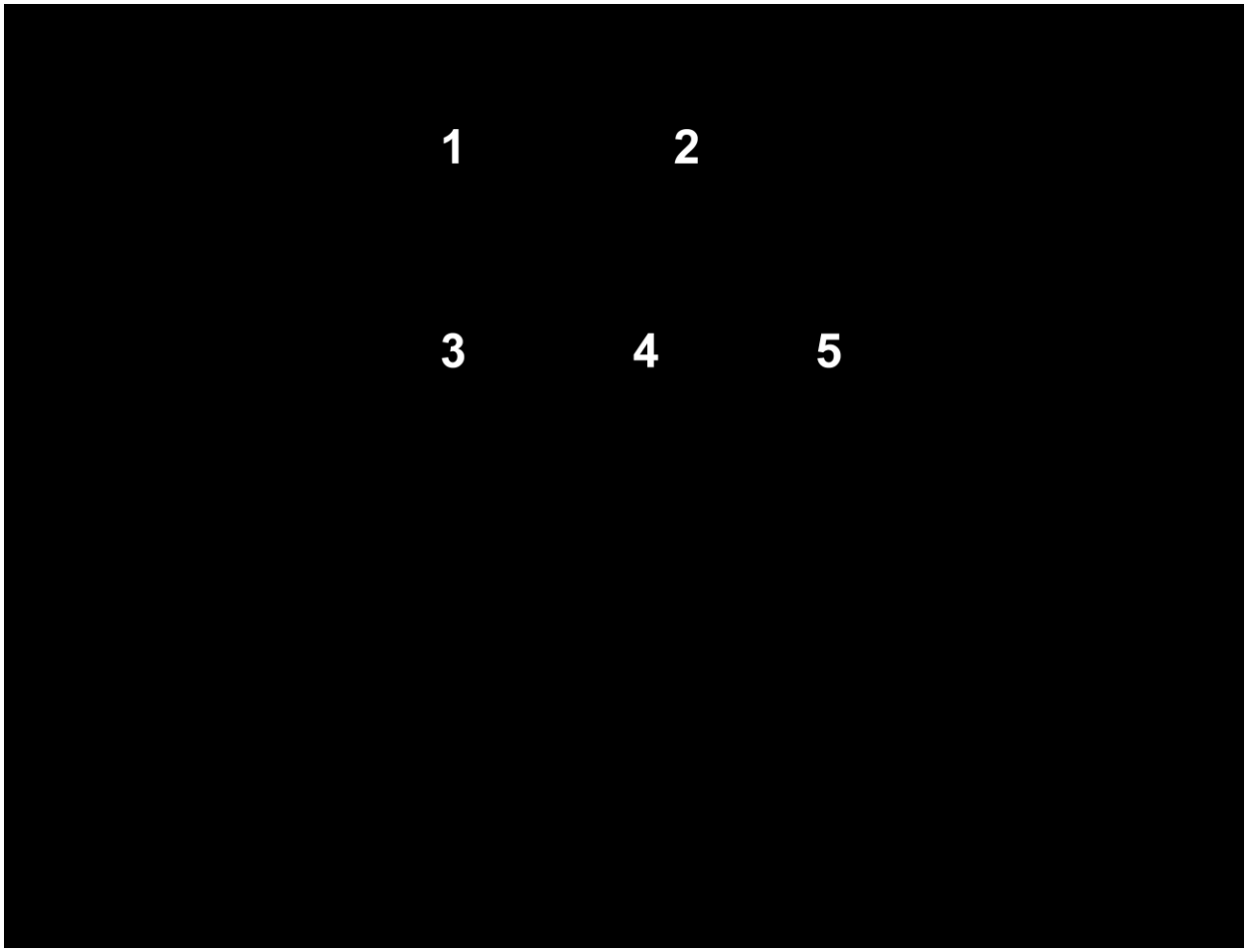

1
2
3
4
5
Indegree of vertex means No. of edges towards that vertex.
Outdegree of vertex means no. of edges come out (from) that vertex.

1 2

3 4

1 2

3 4 5


1
2
3
4
5

Graph Traversal:

⇒ Traversal of graph means visiting all the nodes.

⇒ There are two methods for traversing graph:

- ⇒ (1) Depth First Search (DFS)
- ⇒ (2) Breadth First Search (BFS)

⇒ When a graph is traversed by visiting nodes in the forward (deeper) direction as long as possible, the traversal is called DFS.

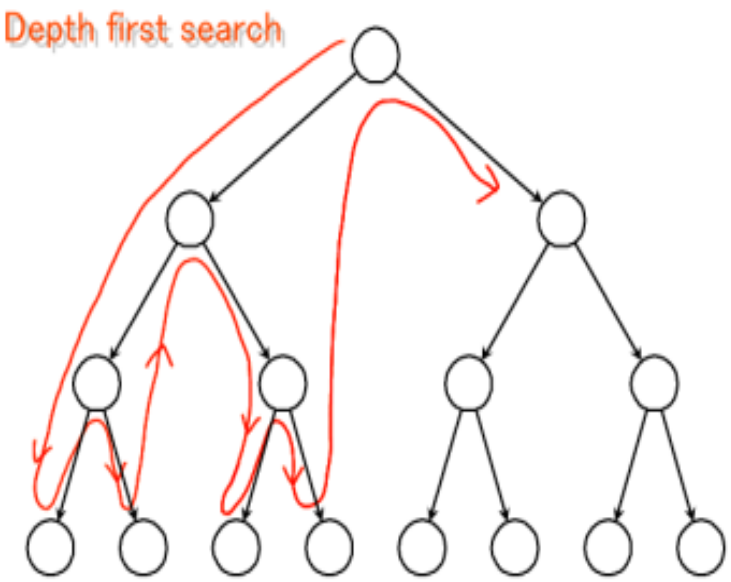
⇒ When a graph is traversed by visiting all the adjacent nodes / vertices, the traversal is called BFS.

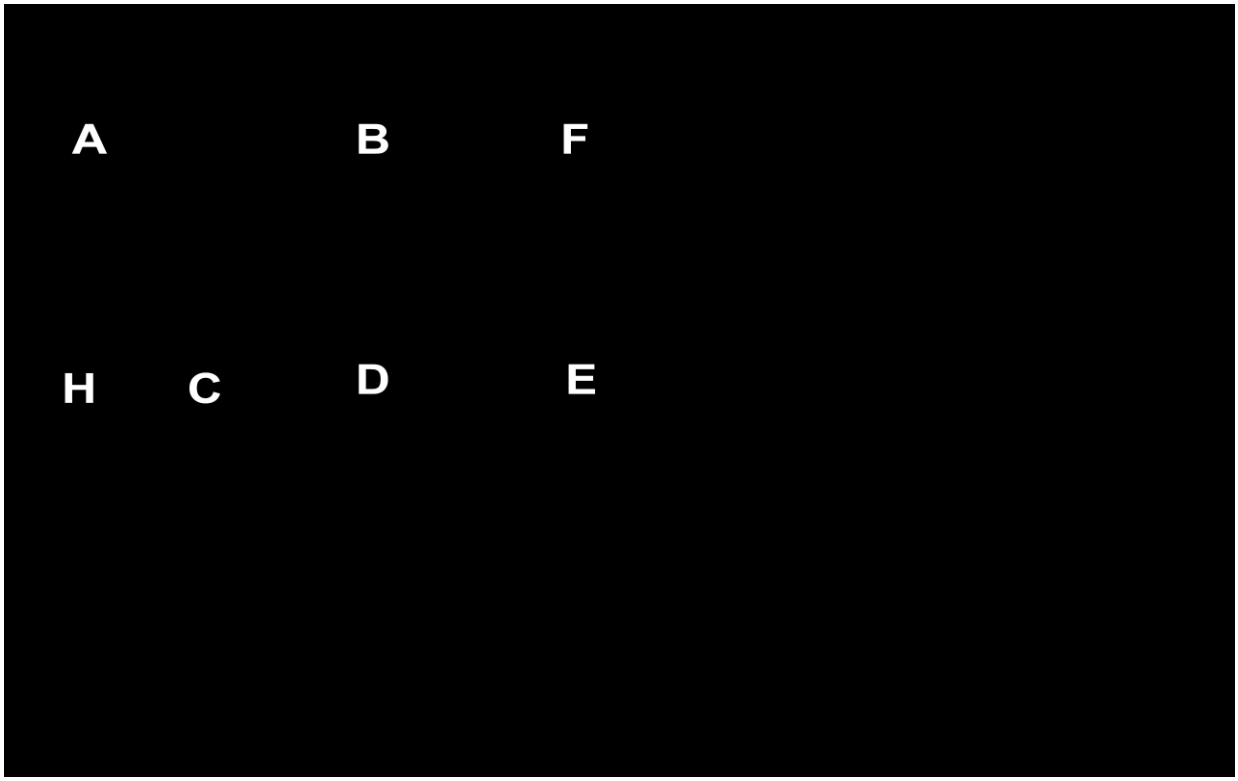
⇒ Depth First Search (DFS):

- ⇒ The strategy adopted in depth first search is to search deeper whenever possible. This algorithm repeatedly searches deeper by visiting unvisited vertices and whenever an unvisited vertex is not found, it backtracks to previous vertex to find out whether there are still unvisited vertices.

⇒ Depth First Search (DFS) - ALGORITHM:

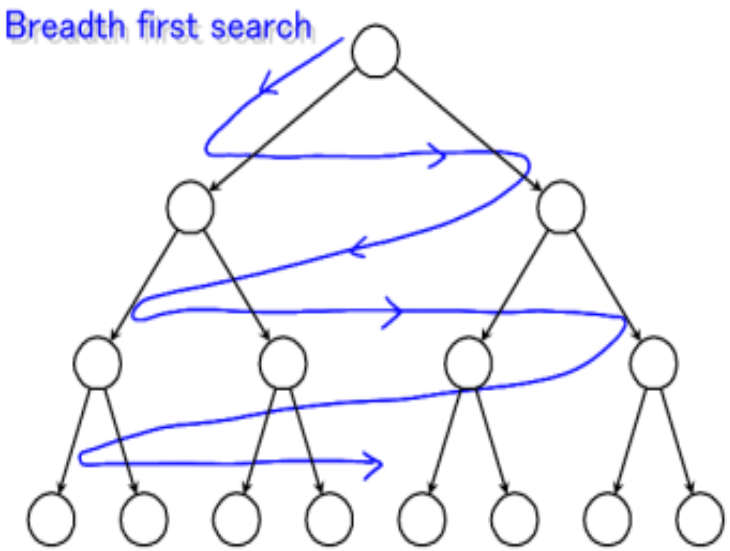
- ⇒ Start from any vertex (source) in the graph and mark it visited.
- ⇒ Find vertex that is adjacent to the source and not previously visited using adjacency matrix and mark it visited.
- ⇒ Repeat this process for all vertices that is not visited, if a vertex is found visited in this process, then return to the previous step and start the same procedure from there. Do this process recursively.

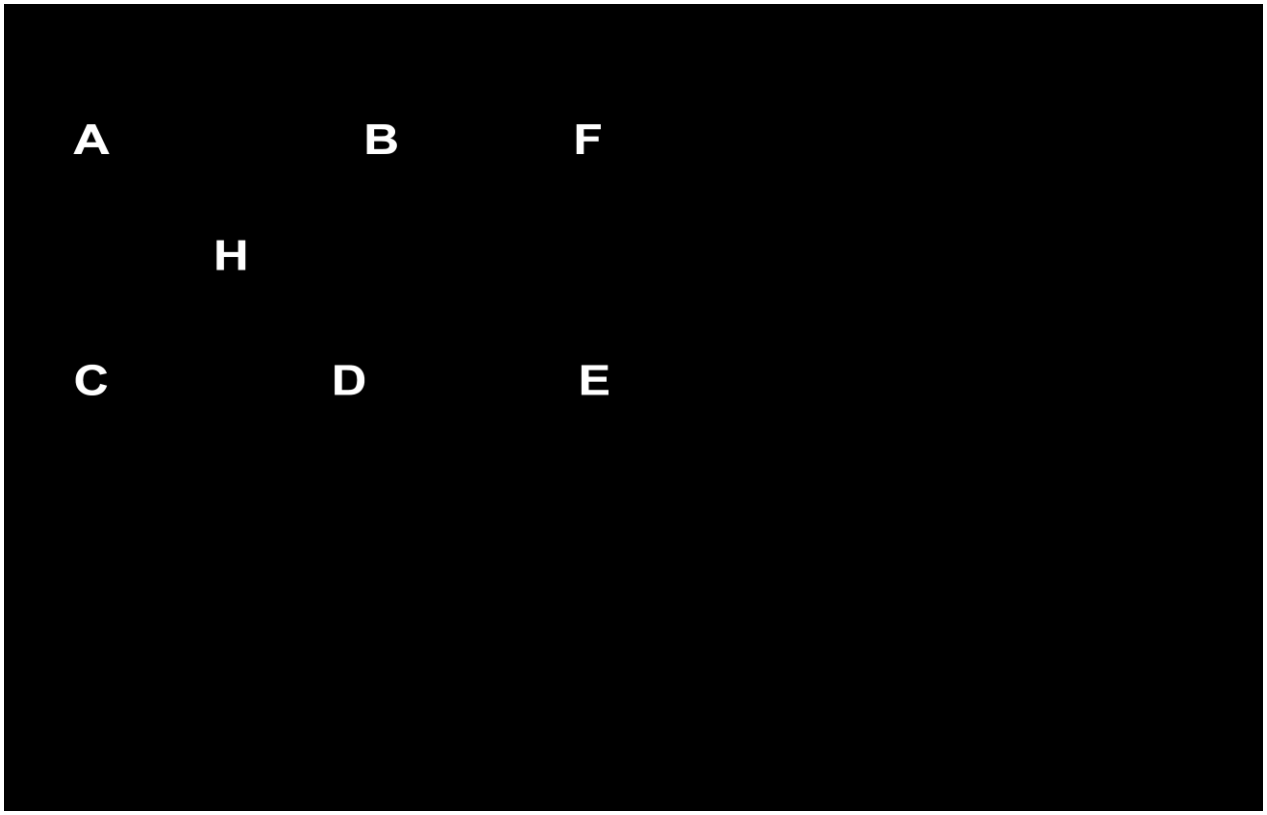




⇒ Breadth First Search (BFS) - ALGORITHM:

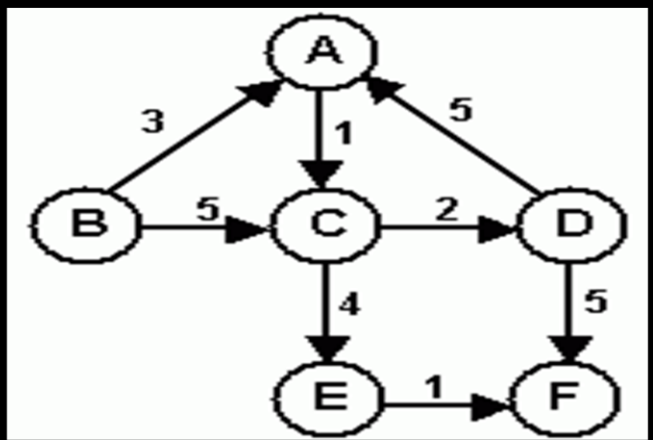
- ⇒ The strategy adopted in BFS is to start search at a vertex (source).
- ⇒ Once you started at source, the number of vertices that are visited as part of the search is 1 and all the remaining vertices need to be visited. Then, search the vertices which are adjacent to the visited vertex from left to order.
- ⇒ In this way, all the vertices of the graph are searched.

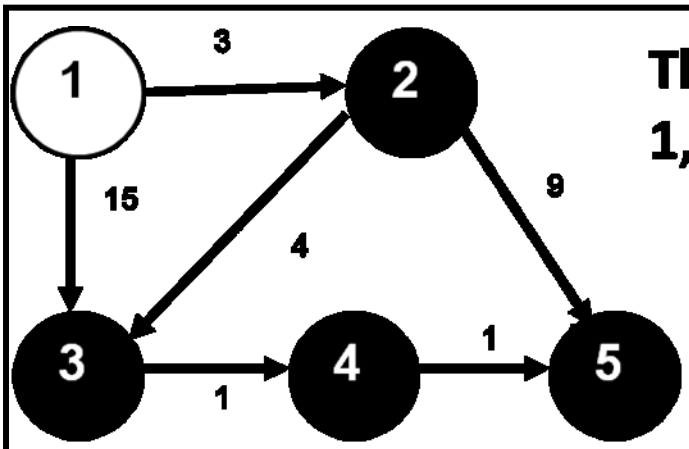




Shortest Path Problem:

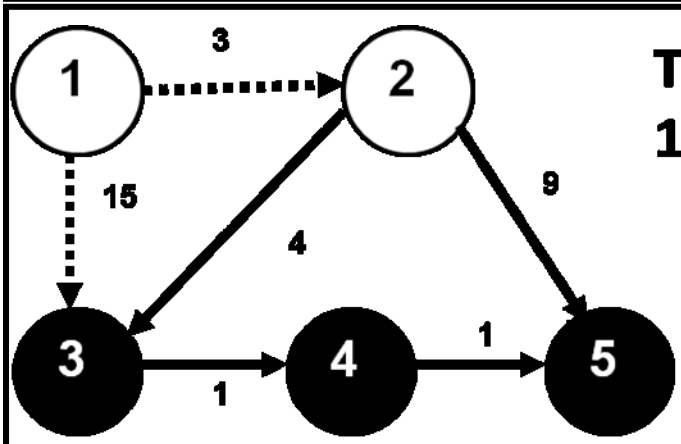
- ⇒ For travelling from one place to another, we can have following questions?
  - ⇒ What is fastest way to get from X to Y?
  - ⇒ Which route is the least expensive?
  - ⇒ What is the shortest distance from X to Y?
  - ⇒ Which route reaches us as quick as possible with minimum cost?
- ⇒ Each of these questions is an instance of only one problem: SHORTEST PATH PROBLEM
- ⇒ In a graph – non linear data structure, to find the shortest path is the most important problem.
- ⇒ In an edge-weighted graph, the weight of an edge measures the cost of traveling that edge.
- ⇒ For example, in a graph representing a network of rails, the weights could represent: distance, cost or time.
- ⇒ Is the shortest path problem well defined?
- ⇒ If all the edges in a graph have non-negative weights, then it is possible to find the shortest path from any two vertices.


A
B
C
D
E
F
3
1
5
5
2
4
5
1



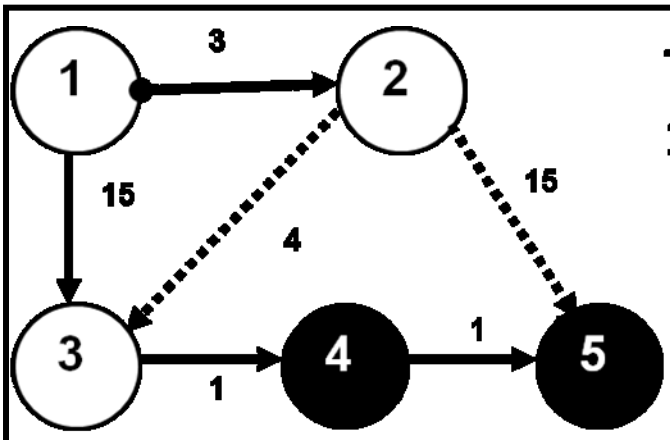
The Shortest Path is:
1,

Finding the shortest path from vertex 1 to vertex 5.



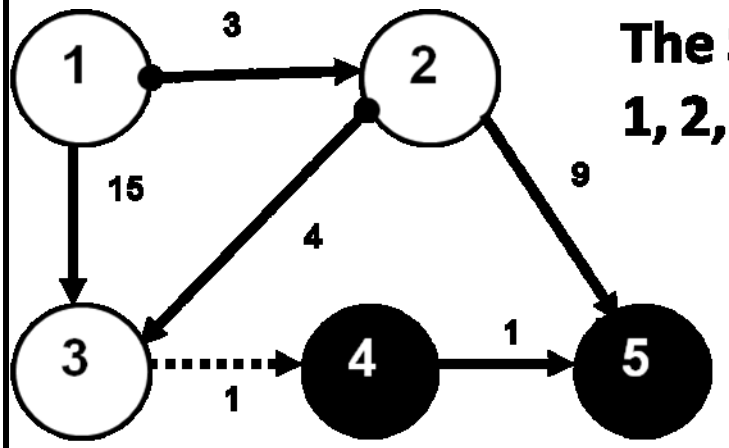
The Shortest Path is:
1

Checking the weighted edges (for shortest path) connected to vertex 1. As we can see that the edge with weight 3 is less than another edge(s), so move towards that direction.



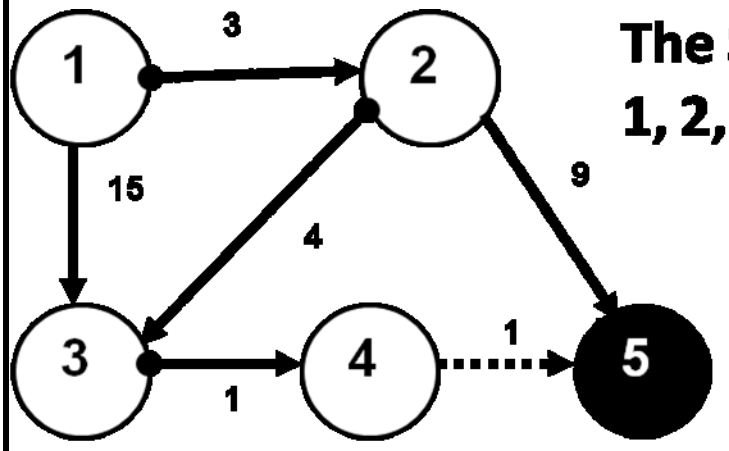
The Shortest Path is:
1, 2

Checking the weighted edges (for shortest path) connected to vertex 2. As we can see that the edge with weight 4 is less than another edge(s), so move towards that direction.



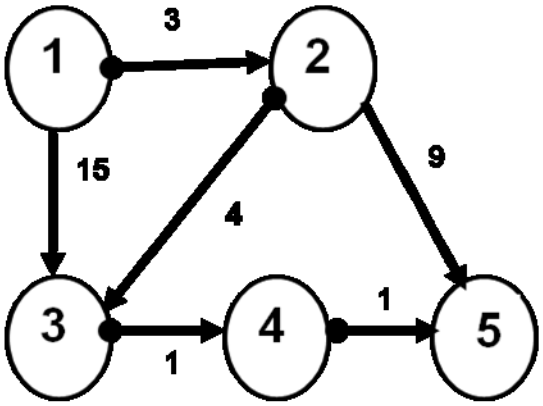
The Shortest Path is:
1, 2, 3

Checking the weighted edges (for shortest path) connected to vertex 3. As we can see that the edge with weight 1 is less than another edge(s), so move towards that direction.



The Shortest Path is:
1, 2, 3, 4

Checking the weighted edges (for shortest path) connected to vertex 4. As we can see that the edge with weight 1 is less than another edge(s), so move towards that direction.



Path Weight is:
3 + 4 + 1 + 1 = 9

The Shortest Path is:
1, 2, 3, 4, 5

The Shortest Path can also be determined by Kruskal's or Prim's Algorithm.

Minimal Spanning Tree:

⇒ A spanning tree of a graph is a sub graph that contains all the vertices and is a tree (with no cycle).

⇒ A graphs may have many spanning trees.

⇒ A MST for a weighted graph is spanning tree with minimum weight.

⇒ That is all vertices in the weighted graph will be connected with minimum edge with minimum weights.

⇒ The MST can be achieved by different algorithms:

- ⇒ Krushkal's Algorithm
- ⇒ Prim's Algorithm