Advantages of Algorithms:

- It is easy to understand.
- Algorithm is a step-wise representation of a solution to a given problem.
- In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

Disadvantages of Algorithms:

- Writing an algorithm takes a long time so it is time-consuming.
- Branching and Looping statements are difficult to show in Algorithms.

Advantages of Algorithms:

1. It is a step-wise representation of a solution to a given problem, which makes it easy to understand.
2. An algorithm uses a definite procedure.
3. It is not dependent on any programming language, so it is easy to understand for anyone even without programming knowledge.
4. Every step in an algorithm has its own logical sequence so it is easy to debug.
5. By using algorithm, the problem is broken down into smaller pieces or steps hence, it is easier for programmer to convert it into an actual program.

Disdvantages of Algorithms:

1. Alogorithms is Time consuming.
2. Difficult to show Branching and Looping in Algorithms.
3. Big tasks are difficult to put in Algorithms.

Characteristics of Algorithms:

Precision – the steps are precisely stated(defined).

Uniqueness – results of each step are uniquely definedand only depend on the input and the result of the precedingsteps.

Finiteness – the algorithm stops after a finite number ofinstructions are executed.

Input – the algorithm receives input.

Output – the algorithm produces output.

Generality – the algorithm applies to a set ofinputs.

Brute Force Algorithm:

This is the most basic and simplest type of algorithm. A Brute Force Algorithm is the straightforward approach to a problem i.e., the first approach that comes to our mind on seeing the problem. More technically it is just like iterating every possibility available to solve that problem.

For Example: If there is a lock of **4-digit** PIN. The digits to be chosen from **0-9** then the brute force will be trying all possible combinations one by one like **0001**, **0002**, **0003**, **0004**, and so on until we get the right PIN. In the worst case, it will take **10,000 tries** to find the right combination.

Recursive Algorithm:

This type of algorithm is based on recursion. In recursion, a problem is solved by breaking it into subproblems of the same type and calling own self again and again until the problem is solved with the help of a base condition.

Some common problem that is solved using recursive algorithms are Factorial of a Number, Fibonacci Series, Tower of Hanoi, DFS for Graph, etc.

Divide and Conquer Algorithm:

In Divide and Conquer algorithms, the idea is to solve the problem in two sections, the first section divides the problem into subproblems of the same type. The second section is to solve the smaller problem independently and then add the combined result to produce the final answer to the problem.

Some common problem that is solved using Divide and Conquers Algorithms are Binary Search, Merge Sort, Quick Sort, Strassen's Matrix Multiplication, etc.

Dynamic Programming Algorithms:

This type of algorithm is also known as the memoization technique because in this the idea is to store the previously calculated result to avoid calculating it again and again. In Dynamic Programming, divide the complex problem into smaller overlapping subproblems and storing the result for future use.

The following problems can be solved using Dynamic Programming algorithm Knapsack Problem, Weighted Job Scheduling, Floyd Warshall Algorithm, etc.

Greedy Algorithm:

In the Greedy Algorithm, the solution is built part by part. The decision to choose the next part is done on the basis that it gives the immediate benefit. It never considers the choices that had taken previously.

Some common problems that can be solved through the Greedy Algorithm are Dijkstra Shortest Path Algorithm, Prim's Algorithm, Kruskal's Algorithm, Huffman Coding, etc.

Introduction to Exponents

Definition

$a^m = a \times a \times a \times a \ldots m$ times.

The **exponent** of a number says how many times the number should be used in a multiplication. In above notation, 'm' is the exponent and the number (the **base**) is 'a' which is multiplied 'm' times

To evaluate an exponent,

- we write it in expanded form and multiply,

Example:

Evaluate 3^4

Solution

$3^4 = 3 \times 3 \times 3 \times 3 = 81$.

Exponentiation

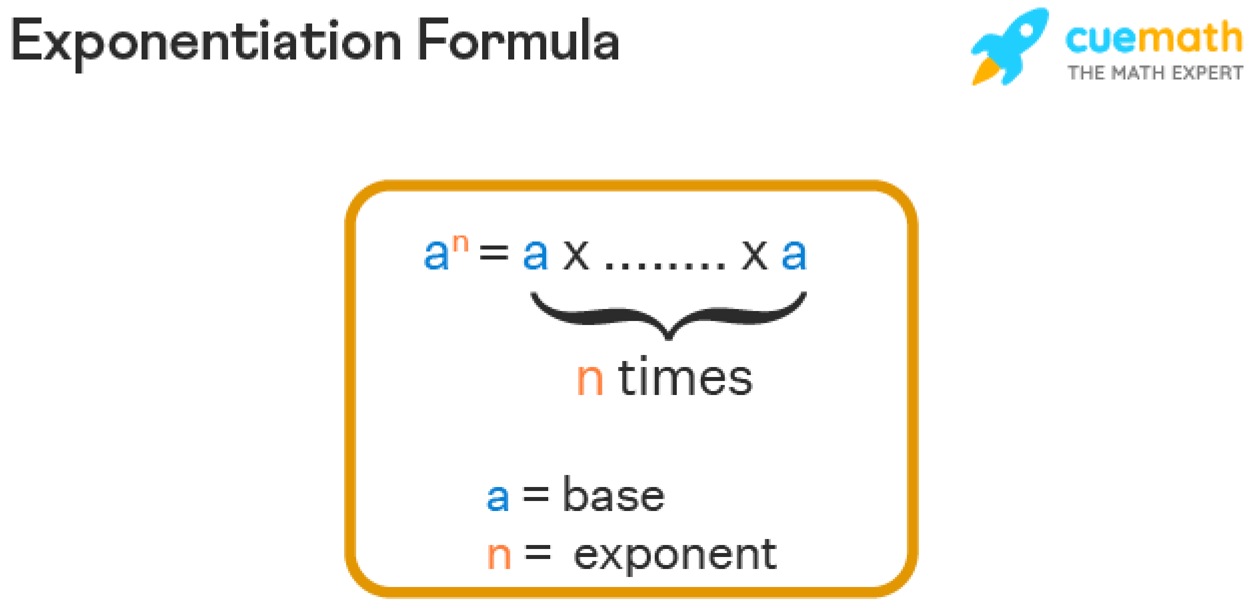
Exponentiation is a process or operation of taking the exponent of a number. If x is an integer raised to n which is a positive integer, then it can be expressed as x^n. This process is known as exponentiation in mathematics. Let us learn more about its rules and properties in this article.

What is Exponentiation?

Exponentiation is the process of doing repeated multiplication by writing the given expression in the form of a^b, where a is the base and b is the exponent. Suppose we are multiplying 5 ten times, so instead of writing it as a multiplication fact, we use the exponentiation algorithm to write it as 5^{10}.

Exponentiation Formula

There is a specific way to write expressions or numbers in the form of exponentiation. The exponentiation formula with 'a' as the base and 'n' as the exponent is given below:


Exponentiation Formula
cuemath
THE MATH EXPERT
$a^n = \underbrace{a \times \ldots\ldots \times a}_{n \text{ times}}$
a = base
n = exponent

By using this formula of exponentiation, we can easily write any repeated multiplication fact in the exponential form.

Exponentiation Properties

There are some exponentiation properties that will help you to simplify expressions with powers easily. They allow us to rewrite exponents in a simplified manner to make it easy for us to do calculations. The list of exponentiation properties or the basic rules of exponentiation are given below:

Exponentiation Property	**Formula**	**Example**
Zero Exponent	$a^0 = 1$	$2^0 = 1$
Identity Exponent	$a^1 = a$	$3^1 = 3$
Product of Powers	$a^m \times a^n = a^{m+n}$	$2^2 \times 2^3 = 2^{2+3} = 2^5$
Quotient Property	$a^m/a^n = a^{m-n}$	$5^5/5^3 = 5^{5-3} = 5^2$
Negative Exponent Property	$a^{-m} = 1/a^m$; $(a/b)^{-m} = (b/a)^m$	$7^{-2} = 1/7^2$
Power of a Power	$(a^m)^n = a^{mn}$	$(2^3)^4 = 2^{12}$
Power of a Product	$(ab)^m = a^m b^m$	$(2 \times 3)^6 = 2^6 \times 3^6$
Power of a Quotient	$(a/b)^m = a^m/b^m$	$(2/3)^4 = 2^4/3^4$

We can easily solve small exponents by using repeated multiplication. For example, to find the value of 2^3, we can multiply 2 three times. But, how to solve expressions with large exponents? We can simplify such numbers or expressions using binary exponentiation or exponentiation by squaring explained below.

Basis of comparison	Linear search	Binary search
Definition	The linear search starts searching from the first element and compares each element with a searched element till the element is not found.	It finds the position of the searched element by finding the middle element of the array.
Sorted data	In a linear search, the elements don't need to be arranged in sorted order.	The pre-condition for the binary search is that the elements must be arranged in a sorted order.
Implementation	The linear search can be implemented on any linear data structure such as an array, linked list, etc.	The implementation of binary search is limited as it can be implemented only on those data structures that have two-way traversal.
Approach	It is based on the sequential approach.	It is based on the divide and conquer approach.
Size	It is preferrable for the small-sized data sets.	It is preferrable for the large-size data sets.
Efficiency	It is less efficient in the case of large-size data sets.	It is more efficient in the case of large-size data sets.
Worst-case scenario	In a linear search, the worst- case scenario for finding the element is O(n).	In a binary search, the worst-case scenario for finding the element is $O(\log_2 n)$.
Best-case scenario	In a linear search, the best-case scenario for finding the first element in the list is O(1).	In a binary search, the best-case scenario for finding the first element in the list is O(1).
Dimensional array	It can be implemented on both a single and multidimensional array.	It can be implemented only on a multidimensional array.

Asymptotic Analysis

Using asymptotic analysis, we can get an idea about the performance of the algorithm based on the input size. We should not calculate the exact running time, but we should find the relation between the running time and the input size. We should follow the running time when the size of the input is increased.

For the space complexity, our goal is to get the relation or function that how much space in the main memory is occupied to complete the algorithm.

Asymptotic Behavior

For a function **f(n)** the asymptotic behavior is the growth of f(n) as n gets large. Small input values are not considered. Our task is to find how much time it will take for a large value of the input.

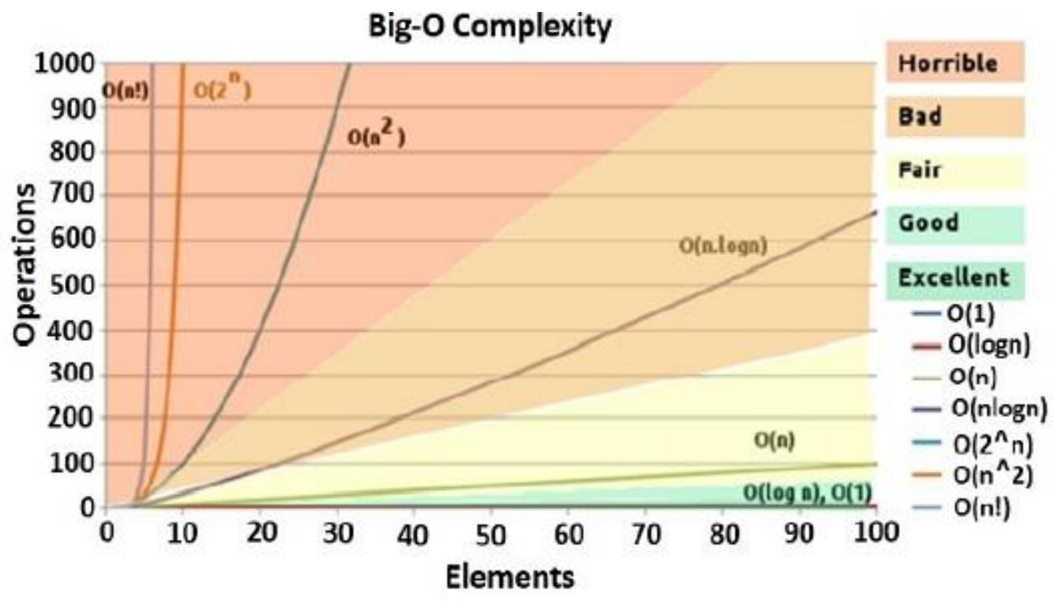
For example,f(n) = c * n + k as linear time complexity.f(n) = c *(n*n) + k is quadratic time complexity.

The analysis of algorithms can be divided into three different cases. The cases are as follows

Best Case − Here the lower bound of the running time is calculated. It describes the behavior of an algorithm under optimal conditions.

Average Case − In this case, we calculate the region between the upper and lower bound of the running time of algorithms. In this case, the number of executed operations are not minimum and not maximum.

Worst Case − In this case we calculate the upper bound of the running time of algorithms. In this case, a maximum number of operations are executed.



Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as *f*(n) and may be for another operation it is computed as *g*(n^2). This means the first operation running time will increase linearly with the increase in **n** and the running time of the second operation will increase exponentially when **n** increases. Similarly, the running time of both operations will be nearly the same if **n** is significantly small.

Usually, the time required by an algorithm falls under three types −

- **Best Case −** Minimum time required for program execution.
- **Average Case −** Average time required for program execution.
- **Worst Case −** Maximum time required for program execution.

Asymptotic Notations

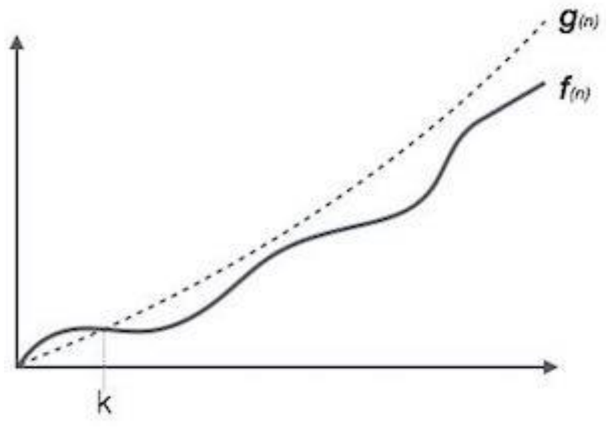
Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation

- Ω Notation
- θ Notation

Big Oh Notation, O

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

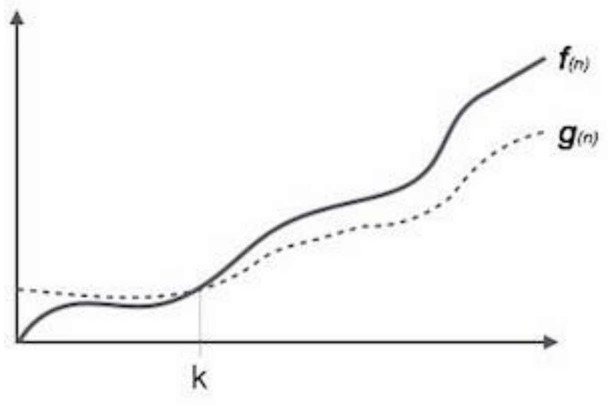


For example, for a function ***f*(n)**

```
O(f(n)) = { g(n) : there exists c > 0 and n₀ such that f(n)
≤ c.g(n) for all n > n₀. }
```

Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

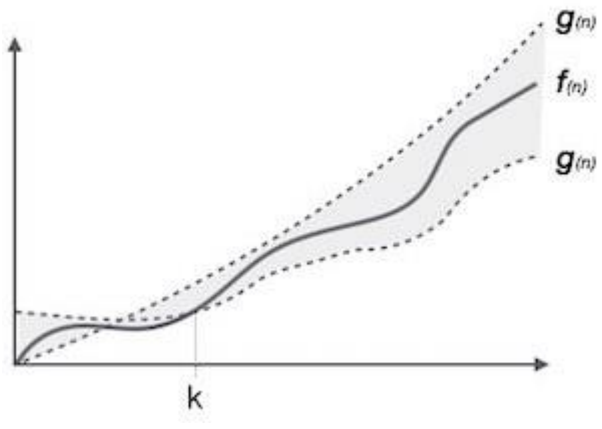


For example, for a function ***f*(n)**

$\Omega(f(n)) \geq \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } g(n) \leq c.f(n) \text{ for all } n > n_0. \}$

Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows −



$\theta(f(n)) = \{ g(n) \text{ if and only if } g(n) = O(f(n)) \text{ and } g(n) = \Omega(f(n)) \text{ for all } n > n_0. \}$

Common Asymptotic Notations

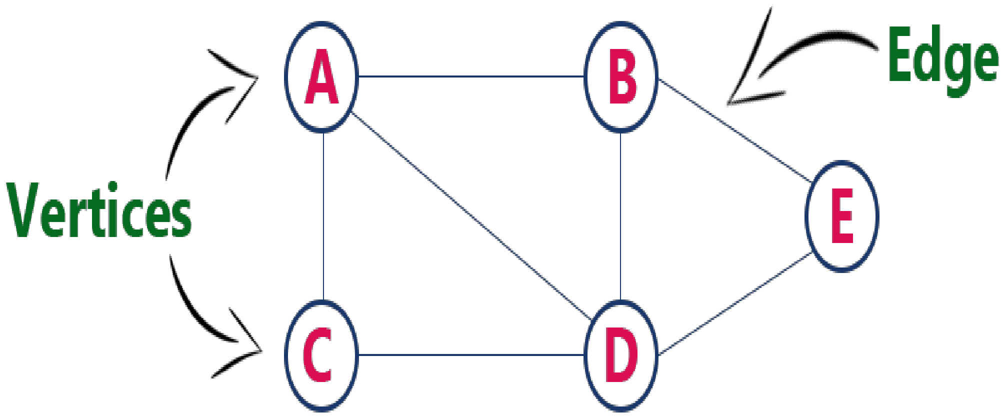
Following is a list of some common asymptotic notations −

Constant	−	O(1)
Logarithmic	−	O(log n)
Linear	−	O(n)
n log n	−	O(n log n)
Quadratic	−	$O(n^2)$
Cubic	−	$O(n^3)$

polynomial	–	$n^{O(1)}$
exponential	–	$2^{O(n)}$

Introduction to Graphs
A graph 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 Graph is a collection of vertices and arcs which connects vertices in the graph. A graph G is
represented as G = (V , E), where V is set of vertices and E is set of edges.
Example: 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)}. This is a graph with 5 vertices and 6 edges.



Graph Terminology

1.Vertex : An 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.

2.Edge : An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as

(starting Vertex, ending Vertex).

In above graph, the link between vertices A and B is represented as (A,B).

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

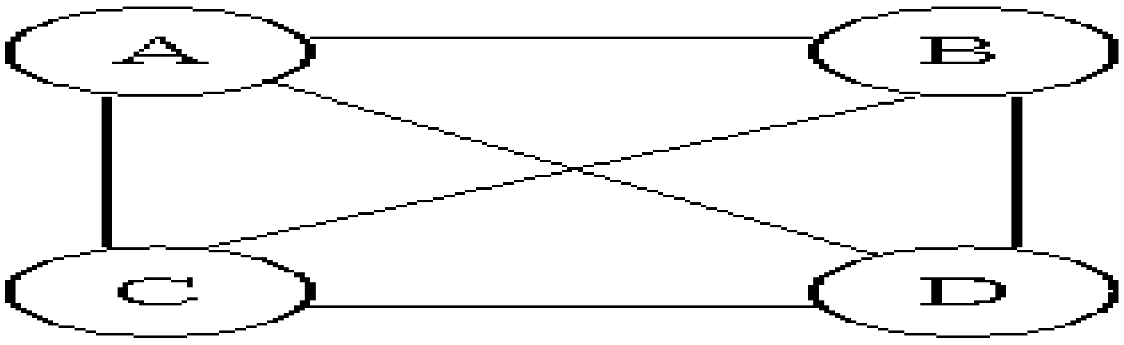
then edge (A , B) is not equal to edge (B , A).2

3.Weighted Edge - A weighted edge is an edge with cost on it.

Types of Graphs

1.Undirected Graph

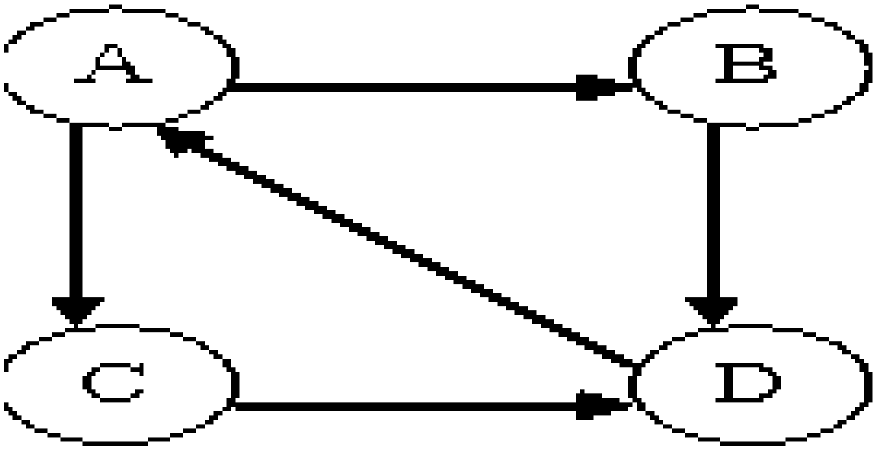
A graph with only undirected edges is said to be undirected graph.



Undirected Graph.

2.Directed Graph

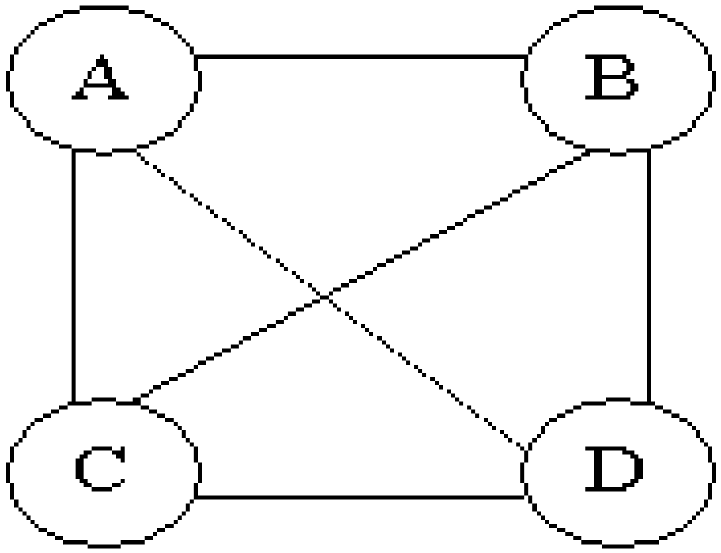
A graph with only directed edges is said to be directed graph.



Directed Graph.

3.Complete Graph

A graph in which any V node is adjacent to all other nodes present in the graph is known as a complete graph. An
undirected graph contains the edges that are equal to edges = n(n-1)/2 where n is the number of vertices present in
the graph. The following figure shows a complete graph.

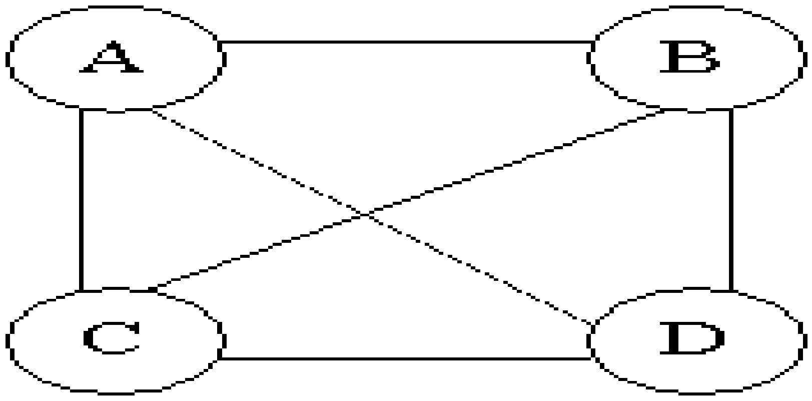


A complete graph.

4.Regular Graph

Regular graph is the graph in which nodes are adjacent to each other, i.e., each node

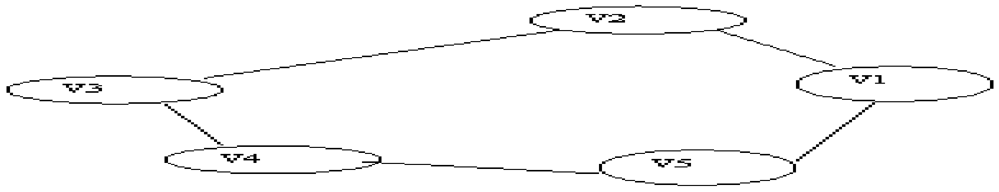
is accessible from any other
node.



A regular graph

5.Cycle Graph

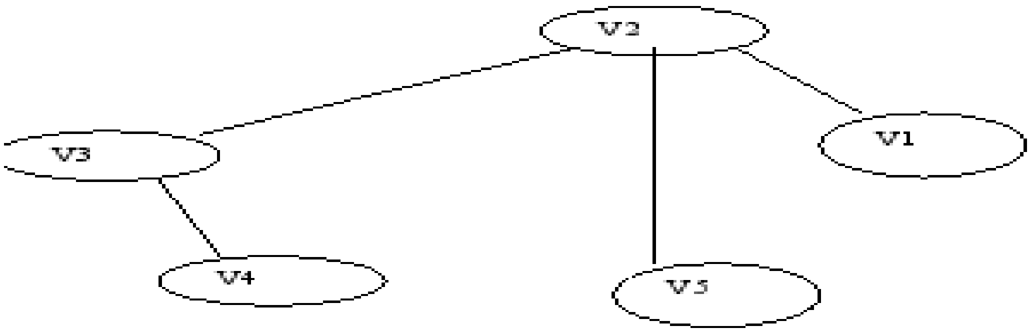
A graph having cycle is called cycle graph. In this case the first and last nodes are the same. A closed simple path
is a cycle.



A cycle graph

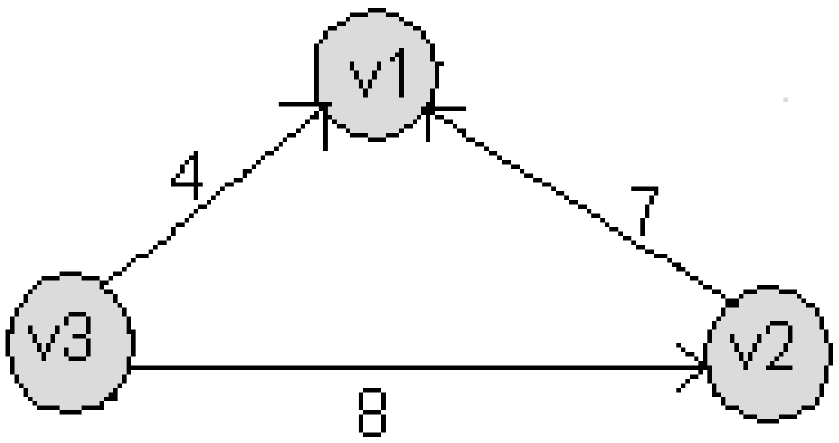
6.Acyclic Graph

A graph without cycle is called acyclic graphs.



A acyclic graph

7. Weighted Graph
A graph is said to be weighted if there are some non negative value assigned to each edges of the graph. The
value is equal to the length between two vertices. Weighted graph is also called a network.



A weighted graph

Outgoing Edge
A directed edge is said to be outgoing edge on its orign vertex.
Incoming Edge
A directed edge is said to be incoming edge on its destination vertex.
Degree
Total number of edges connected to a vertex is said to be degree of that vertex.
Indegree
Total number of incoming edges connected to a vertex is said to be indegree of that vertex.
Outdegree
Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex

Parallel edges or Multiple edges
If there are two undirected edges to have the same end vertices, and for two directed edges to have the same
origin and the same destination. Such edges are called parallel edges or multiple edges.
Self-loop

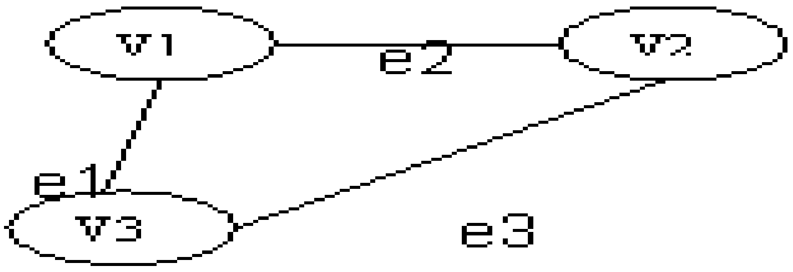
An edge (undirected or directed) is a self-loop if its two endpoints coincide.

Simple Graph

Circuit

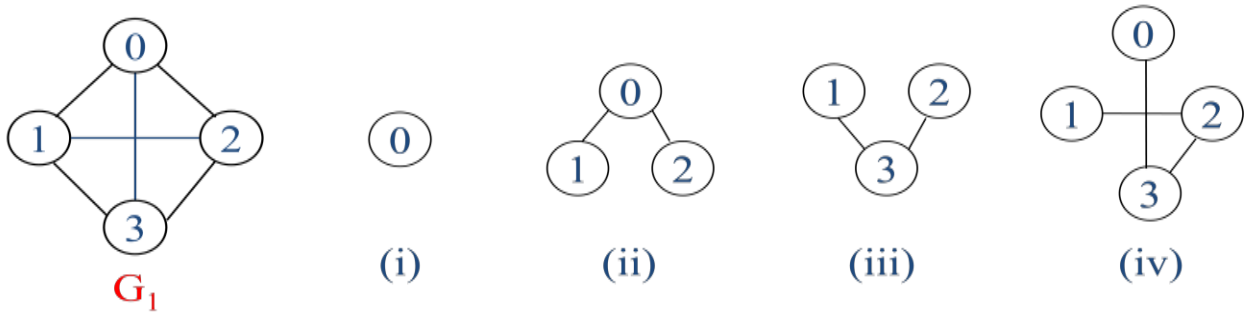
A closed walk in which no vertex (except the initial and the final vertex) appears more than once is called a
circuit.

A circuit having three vertices and three edges.



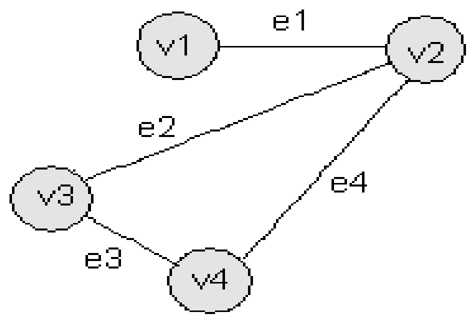
Sub Graph

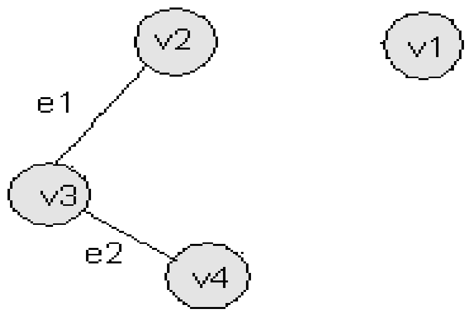
A graph S is said to be a sub graph of a graph G if all the vertices and all the edges of S are in G, and each edge of
S has the same end vertices in S as in G. A subgraph of G is a graph G' such that V(G') Í V(G) and E(G') Í
E(G)



Connected Graph

A graph G is said to be connected if there is at least one path between every pair of vertices in G. Otherwise,G is
disconnected





A connected graph G A disconnected graph G

Degree

In an undirected graph, the number of edges connected to a node is called the degree of that node or the degree of a node is the number of edges incident on it.

In the above graph, degree of vertex v1 is 1, degree of vertex v2 is 3, degree of v3 and v4 is 2 in a connected graph.

Isomorphism

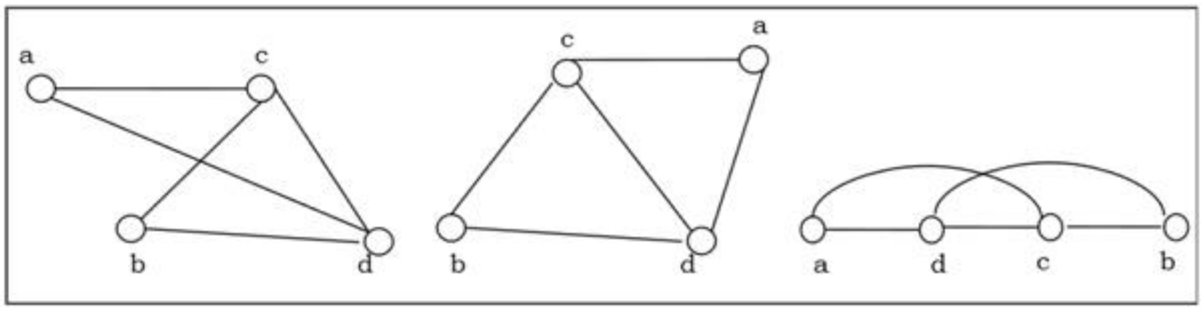
If two graphs G and H contain the same number of vertices connected in the same way, they are called isomorphic graphs (denoted by G ≅ H).

It is easier to check non-isomorphism than isomorphism. If any of these following conditions occurs, then two graphs are non-isomorphic −

- The number of connected components are different
- Vertex-set cardinalities are different
- Edge-set cardinalities are different
- Degree sequences are different

Example

The following graphs are isomorphic −



Homomorphism

A homomorphism from a graph ***G*** to a graph ***H*** is a mapping (May not be a bijective mapping) h: G → H such that − (x, y) ∈ E(G) → (h(x), h(y)) ∈ E(H). It maps adjacent vertices of graph G to the adjacent vertices of the graph ***H***.

Properties of Homomorphisms

- A homomorphism is an isomorphism if it is a bijective mapping.
- Homomorphism always preserves edges and connectedness of a graph.
- The compositions of homomorphisms are also homomorphisms.
- To find out if there exists any homomorphic graph of another graph is a NPcomplete problem.

(a)Representation of the Undirected Graph:

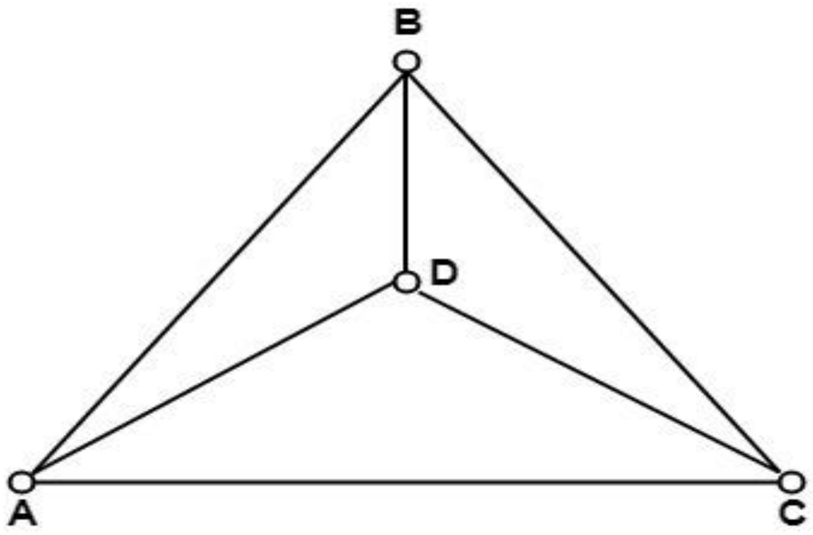
1. Adjacency Matrix Representation: If an Undirected Graph G consists of n vertices then the adjacency matrix of a graph is an n x n matrix A = [a_{ij}] and defined by

$$a_{ij} = \begin{bmatrix} 1, \text{if } \{V_i, V_j\} \text{is an edge i.e., } v_i \text{is adjacent to } v_j \\ 0, \text{if there is no edge between } v_i \text{ and } v_j \end{bmatrix}$$

If there exists an edge between vertex v_i and v_j, where i is a row and j is a column then the value of a_{ij}=1.

If there is no edge between vertex v_i and v_j, then value of a_{ij}=0.

Example: Find the adjacency matrix M_A of graph G shown in Fig:



Solution: Since graph G consist of four vertices. Therefore, the adjacency matrix wills a 4 x 4 matrix. The adjacency matrix is as follows in fig:

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

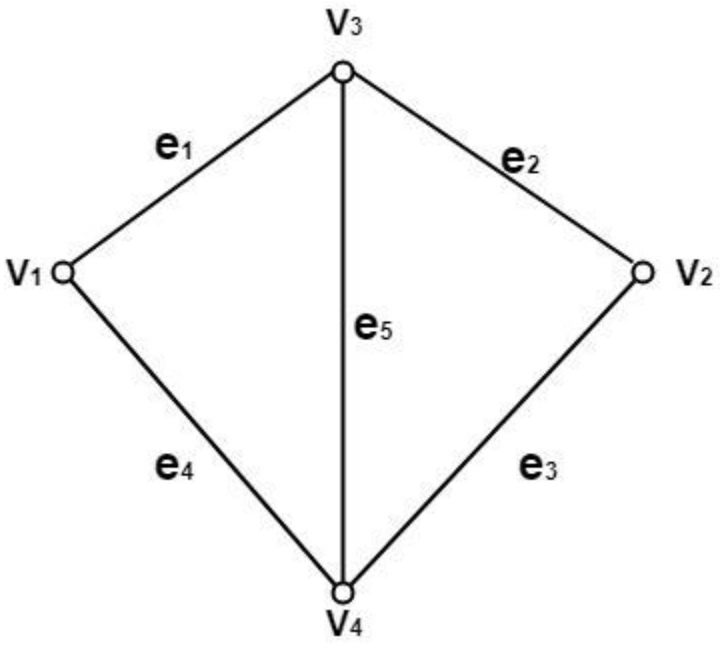
2. Incidence Matrix Representation: If an Undirected Graph G consists of n vertices and m edges, then the incidence matrix is an n x m matrix C = [c_{ij}] and defined by

$$c_{ij} = \begin{bmatrix} 1, \text{if the vertex } V_i \text{incident by edge } e_j \\ 0, \text{otherwise} \end{bmatrix}$$

There is a row for every vertex and a column for every edge in the incident matrix.

The number of ones in an incidence matrix of the undirected graph (without loops) is equal to the sum of the degrees of all the vertices in a graph.

Example: Consider the undirected graph G as shown in fig. Find its incidence matrix M_I.



Solution: The undirected graph consists of four vertices and five edges. Therefore, the incidence matrix is an 4 x 5 matrix, which is shown in Fig:

$$M_I = \begin{array}{c|ccccc} & e_1 & e_2 & e_3 & e_4 & e_5 \\ \hline V_1 & 1 & 0 & 0 & 1 & 0 \\ V_2 & 0 & 1 & 1 & 0 & 0 \\ V_3 & 1 & 1 & 0 & 0 & 1 \\ V_4 & 0 & 0 & 1 & 1 & 1 \end{array}$$

(b)Representation of Directed Graph:

1. Adjacency Matrix Representation: If a directed graph G consists of n vertices then the adjacency matrix of a graph is an n x n matrix A = $[a_{ij}]$ and defined by

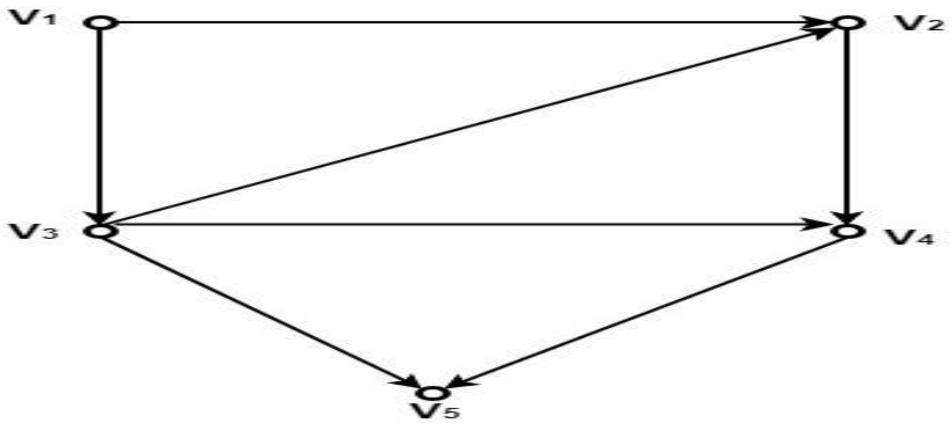
$$a_{ij} = \begin{bmatrix} 1, \text{if } \{V_i, V_j\} \text{is an edge i. e., } V_i \text{is initial vertex and } V_j \text{ is the final vertex} \\ 0, \text{if there is no edge between } V_i \text{ and } V_j \end{bmatrix}$$

If there exists an edge between vertex V_i and V_j, with V_i as initial vertex and V_j as a final vertex, then the value of $a_{ij}=1$.

If there is no edge between vertex V_i and V_j, then the value of $a_{ij}=0$.

The number of ones in the adjacency matrix of a directed graph is equal to the number of edges.

Example: Consider the directed graph shown in fig. Determine its adjacency matrix M_A.



Solution: Since the directed graph G consists of five vertices. Therefore, the adjacency matrix will be a 5 x 5 matrix. The adjacency matrix of the directed graphs is as follows:

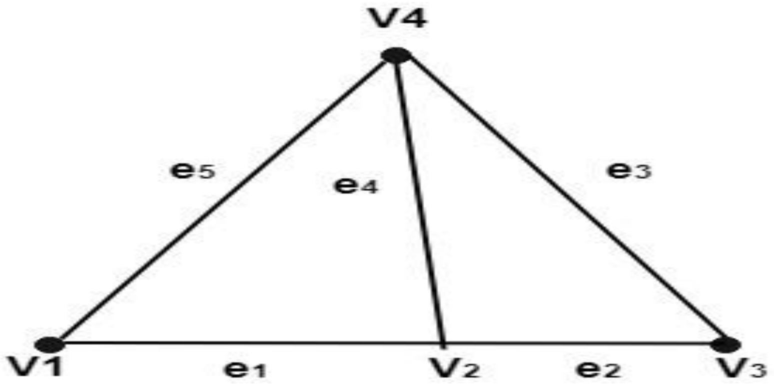
$$M_A = \begin{array}{c|ccccc} & V_1 & V_2 & V_3 & V_4 & V_5 \\ V_1 & 0 & 1 & 1 & 0 & 0 \\ V_2 & 0 & 0 & 0 & 1 & 0 \\ V_3 & 0 & 1 & 0 & 1 & 1 \\ V_4 & 0 & 0 & 0 & 0 & 1 \\ V_5 & 0 & 0 & 0 & 0 & 0 \end{array}$$

2. Incidence Matrix Representation: If a directed graph G consists of n vertices and m edges, then the incidence matrix is an n x m matrix $C = [c_{ij}]$ and defined by

$$c_{ij} = \begin{bmatrix} 1, \text{if } V_i \text{is the initial vertex of edge } e_j \\ -1, \text{if } V_i \text{is the final vertex of edge } e_j \\ 0, V_i \text{is not incident on edge } e_j \end{bmatrix}$$

The number of ones in an incidence matrix is equal to the number of edges in the graph.

Example: Consider the directed graph G as shown in fig. Find its incidence matrix M_I.



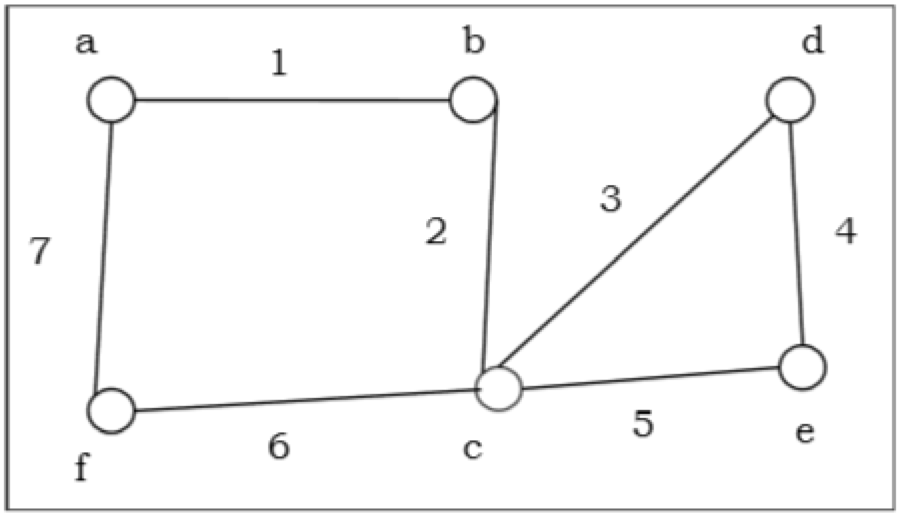
Solution: The directed graph consists of four vertices and five edges. Therefore, the incidence matrix is a 4 x 5 matrix which is show in fig:

$$|M_I = \begin{array}{cccccc} & e_1 & e_2 & e_3 & e_4 & e_5 \\ V_1 & -1 & 0 & 0 & 0 & -1 \\ V_2 & 1 & 1 & 0 & -1 & 0 \\ V_3 & 0 & -1 & 1 & 0 & 0 \\ V_4 & 0 & 0 & -1 & 1 & 1 \end{array}$$

Euler Graph - A connected graph G is called an Euler graph, if there is a closed trail which includes every edge of the graph G.

Euler Path - An Euler path is a path that uses every edge of a graph exactly once. An Euler path starts and ends at different vertices.

Euler Circuit - An Euler circuit is a circuit that uses every edge of a graph exactly once. An Euler circuit always starts and ends at the same vertex. A connected graph G is an Euler graph if and only if all vertices of G are of even degree, and a connected graph G is Eulerian if and only if its edge set can be decomposed into cycles.

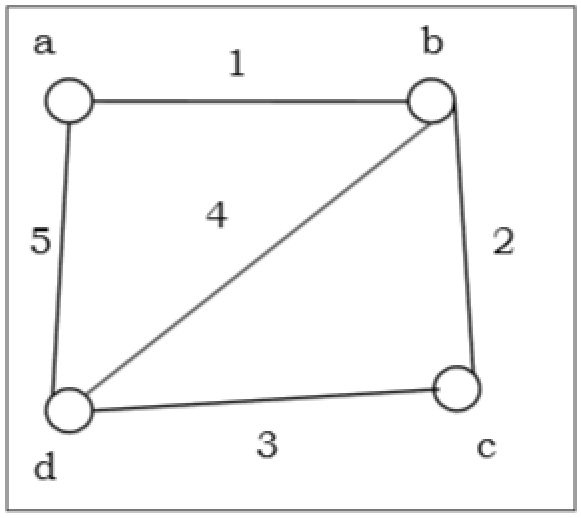


The above graph is an Euler graph as a 1 b 2 c 3 d 4 e 5 c 6 f 7 g covers all the edges of the graph.

Hamiltonian graph - A connected graph G is called Hamiltonian graph if there is a cycle which includes every vertex of G and the cycle is called Hamiltonian cycle. Hamiltonian walk in graph G is a walk that passes through each vertex exactly once.

In above example, sum of degree of a and c vertices is 6 and is greater than total vertices, 5 using Ore's theorem, it is an Hamiltonian Graph.

The above graph is an Euler graph as “a1b2c3d4e5c6f7g”“a1b2c3d4e5c6f7g” covers all the edges of the graph.

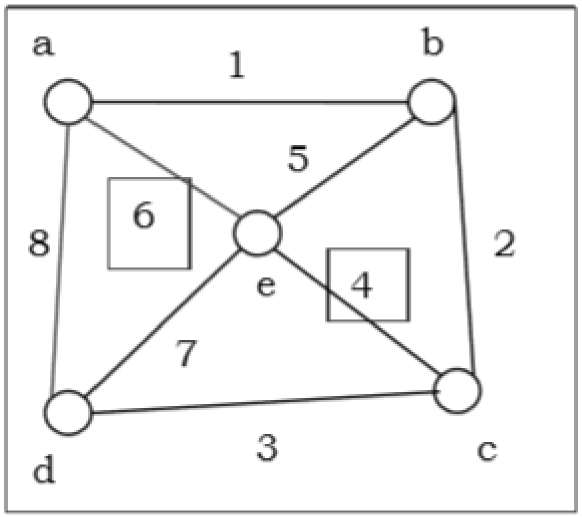


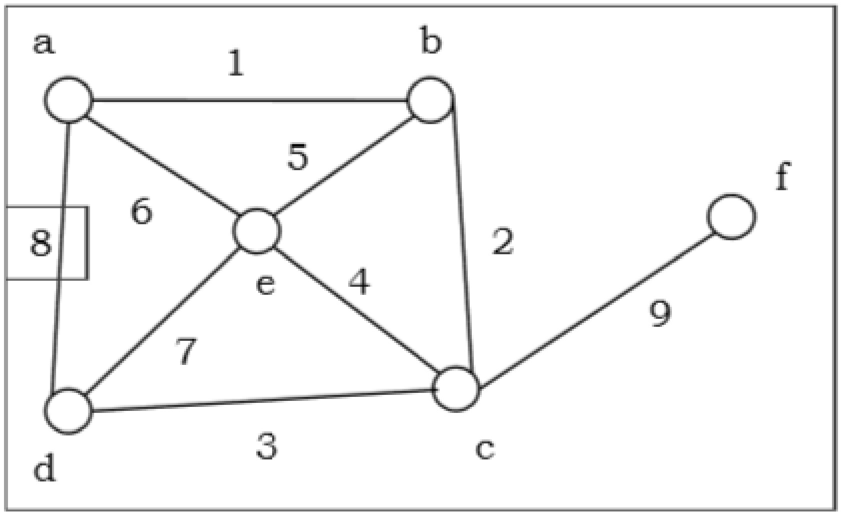
Hamiltonian Graphs

A connected graph GG is called Hamiltonian graph if there is a cycle which includes every vertex of GG and the cycle is called Hamiltonian cycle. Hamiltonian walk in graph GG is a walk that passes through each vertex exactly once.

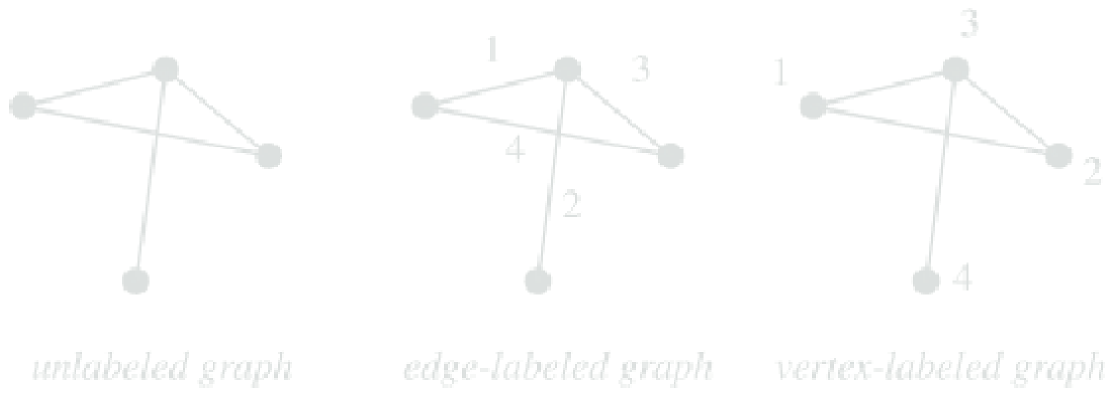
If GG is a simple graph with n vertices, where $n\geq3$n≥3 If $\deg(v)\geq\frac{n}{2}$deg(v)≥n2 for each vertex vv, then the graph GG is Hamiltonian graph. This is called **Dirac's Theorem**.

If GG is a simple graph with nn vertices, where $n\geq2$n≥2 if $\deg(x)+\deg(y)\geq n$deg(x)+deg(y)≥n for each pair of non-adjacent vertices x and y, then the graph GG is Hamiltonian graph. This is called **Ore's theorem**.

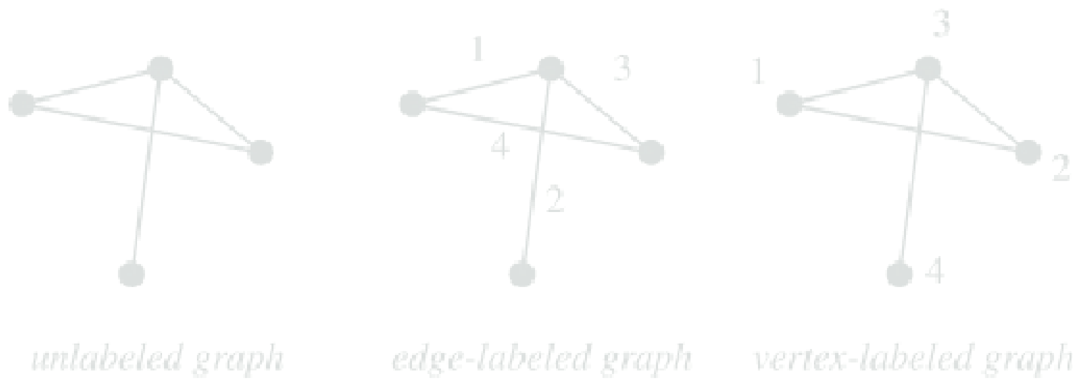




Labeled Graph



A labeled graph is a finite series of graph vertices with a set of graph edges of 2-subsets of . Given a graph vertex set , the number of vertex-labeled graphs is given by . Two graphs and with graph vertices are said to be isomorphic if there is a permutation of such that is in the set of graph edges iff is in the set of graph edges .



The term "labeled graph" when used without qualification means a graph with each node labeled differently (but arbitrarily), so that all nodes are considered distinct for purposes of enumeration. The *total* number of (not necessarily connected) labeled -node graphs for , 2, ... is given by 1, 2, 8, 64, 1024, 32768, ... (OEIS A006125; illustrated above), and the numbers of connected labeled graphs on -nodes are given by the logarithmic transform of the preceding sequence, 1, 1, 4, 38, 728, 26704, ... (OEIS A001187; Sloane and Plouffe 1995, p. 19).

The numbers of graph vertices in all labeled graphs of orders , 2, ... are 1, 4, 24, 256, 5120, 196608, ... (OEIS A095340), which the numbers of edges are 0, 1, 12, 192, 5120, 245760, ... (OEIS A095351), the latter of which has closed-form