MATS UNIVERSITY

NAAC GRADE A+ ACCREDITED UNIVERSITY

# MATS CENTRE FOR DISTANCE & ONLINE EDUCATION

## Mathematical Foundation of Computer Application

**Master of Computer Applications (MCA)**
**Semester - 1**


SELF LEARNING MATERIAL

MATS UNIVERSITY
www.matsuniversity.ac.in

NAAC GRADE A+ ACCREDITED UNIVERSITY

# Master of Computer Applications

## ODL MCA DSC-104

## Mathematical Foundation of Computer Application


| | |
|---|---|
| **Course Introduction** | **1** |
| **Block 1**<br>**Set theory, Mathematical Logic, Relation and Function** | **2-75** |
| **Unit 1:** Introduction to Set theory | |
| **Unit 2:** Tautology, Contradiction, Logical Equivalence | |
| **Unit 3:** Relation | |
| **Unit 4:** Function | |
| **Block 2**<br>**POSETS and Lattices** | **76-105** |
| **Unit 5:** Partial order relation | |
| **Unit 6:** Lattice | |
| **Unit 7:** Distributive and Complemented lattice | |
| **Block 3**<br>**Boolean Algebra** | **106-135** |
| **Unit 8:** Basic concepts of Boolean Algebra | |
| **Unit 9:** Karnaugh map | |
| **Unit 10:** Applications of Boolean Algebra in switching circuits Logic circuits | |
| **Block 4**<br>**Graph Theory** | **136-208** |
| **Unit 11:** Basic concepts of graph theory | |
| **Unit 12:** Matrix Representation of Graphs, Directed Graphs | |
| **Unit 13:** Tree and its properties | |
| **Block 5**<br>**Semi Groups and Monoids** | **209-272** |
| Unit 14: Algebraic Structure, Binary Operation, Properties, Semi Group, Monoid, Group Theory | |
| Unit 15: Abelian group, Cyclic group, Generators, Permutation group, Subgroup | |
| Unit 16: Homomorphism, Isomorphism and Automorphism. | |
| Unit 17: Cosets, Lagrange's Theorem, Normal Subgroup | |
| **Glossary** | **273-276** |

**COURSE DEVELOPMENT EXPERT COMMITTEE**

Prof. (Dr.) K. P. Yadav, Vice Chancellor, MATS University, Raipur, Chhattisgarh

Prof. (Dr.) Omprakash Chandrakar, Professor and Head, School of Information Technology, MATS University, Raipur, Chhattisgarh

Prof. (Dr.) Sanjay Kumar, Professor and Dean, Pt. Ravishankar Shukla University, Raipur, Chhattisgarh

Prof. (Dr.) Jatinder kumar R. Saini, Professor and Director, Symbiosis Institute of Computer Studies and Research, Pune

Dr. Ronak Panchal, Senior Data Scientist, Cognizant, Mumbai

Mr. Saurabh Chandrakar, Senior Software Engineer, Oracle Corporation, Hyderabad

**COURSE COORDINATOR**

Prof. (Dr.) A. J. Khan, Professor, School of Information Technology, MATS University, Raipur, Chhattisgarh

**COURSE PREPARATION**

Prof. (Dr.) A. J. Khan, Professor and Ms. Arifa Khan, Assistant Professor, School of Information Technology, MATS University, Raipur, Chhattisgarh

March, 2025

**ISBN: 978-93-49916-45-6**



Printed & Published on behalf of MATS University, Village-Gullu, Aarang, Raipur by Mr. Meghanadhudu Katabathuni, Facilities & Operations, MATS University, Raipur (C.G.)



Printed at: The Digital Press, Krishna Complex, Raipur-492001(Chhattisgarh)

# Acknowledgement

The material (pictures and passages) we have used is purely for educational purposes. Every effort has been made to trace the copyright holders of material reproduced in this book. Should any infringement have occurred, the publishers and editors apologize and will be pleased to make the necessary corrections in future editions of this book.

# COURSE INTRODUCTION

Mathematical foundation plays a crucial role in computer applications by providing a theoretical framework necessary for problem solving data structuring and algorithm design. This course equips student with fundamental mathematical concepts including set theory, logic, Boolean algebra, graph theory and group theory which are essential for understanding and developing computing solutions.

**Block 1: Set Theory, Mathematical Logic, Relation and Function**

This Block includes the fundamental concepts of a set theory relations and functions. It covers the principal of logical connectivity, logical equivalence and properties of function to develop a strong mathematical strong.

**Block 2: Poset and Lattice**

Understanding partial order relations and lattice structure is crucial in optimization problems and hierarchical data representation. This Block focuses on ordered sets and their applications.

**Block 3: Boolean Algebra**

Boolean algebra is the foundation of digital logic design and computational logic. this Block delves into Boolean expressions, simplifications techniques and circuit applications.

**Block 4: Graph Theory**

Graph theory provides a framework for modeling relationship and networks, widely used in computing and data structure designs

**Block 5: Semi Group and Monoids**

Algebraic structure such as groups and monoids form the basis of the cryptography, automata theory and database security.

# Block 1: Set theory, Mathematical Logic, Relation and Function

## Unit 1: Introduction to Set theory

**Structure**


1.1 Introduction
1.2 Learning Outcomes
1.3 Introduction to Set Theory, Cartesian product
1.4 Relations and Functions as Subsets of Cartesian Products
1.5 Advanced Topics: Power Sets and Cartesian Product
1.6 Statements and Notations, Logical Connectives
1.7 Conditional Statements: Implication and Its Logical Structure
1.8 Biconditional and Logical Equivalence:
1.9 Summary
1.10 Exercises
1.11 References and Suggested Readings


## 1.1: Introduction

Set theory serves as the cornerstone of modern mathematics and computer science, providing a systematic framework for representing and analyzing collections of objects. It introduces the concept of sets, subsets, and operations such as union, intersection, and complement, which are essential for logical reasoning and data organization. Understanding sets helps in defining relationships among data elements and enables the construction of structured databases, algorithms, and programming logic. Venn diagrams visually demonstrate set operations and enhance conceptual clarity. Through the study of set theory, learners develop critical analytical skills that form the basis for advanced topics like relations, functions, and discrete structures used in computing applications.

## 1.2: Learning Outcomes

- To understand the fundamental concepts of set theory and Cartesian products.
- To explore statements, logical connectives, and their applications.

- To analyze tautologies, contradictions, and logical equivalences.
- To study relations, types of binary relations, and equivalence relations.
- To understand functions, their properties, and composition.

## 1.3: Introduction to Set Theory, Cartesian product

**Fundamentals of Set Theory:**

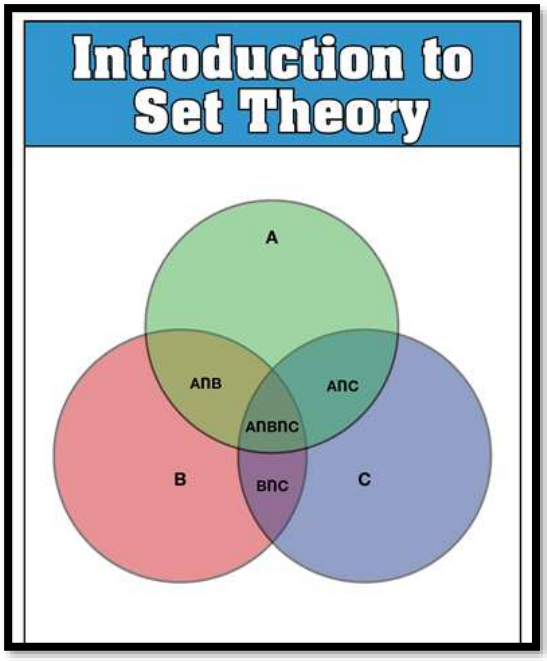



Fig: 1.1 Introduction to Set theory

Like so many concepts in modern mathematics, set theory is a language in which most everything else is written and read. A set, in its most basic sense, is a well-defined collection of distinct objects, called the elements or members of the set. The concept of a "well-defined" collection is essential—it means that we can take an object, and be able to tell if it is in the set or not without ambiguity. This little idea, formalized by Georg Cantor in the late nineteenth century, has changed the way mathematicians think and supplied a lingua franca Moduleing disparate branches of mathematics. Sets encapsulate the primitive idea of collection and containment — an idea so simple and natural, yet one that becomes deeply powerful when formalised. The development of set theory represented a paradigm shift in mathematics, transitioning mathematics from a

focus on the concrete to the abstract, and providing a common language that could cross mathematical domains. The universal applicability of set theory has led to it being designated the "foundation of mathematics," a "base" upon which arbitrary mathematical trees—relations, functions, algebraic structures—can be built accurately and rigorously. The set theoretic notation is clean, simple and expressive as a whole: a set is represented with a capital letter (A, B, X) and its elements are written in a lowercase letter (a, b, x). A relationship between a set and an element is denoted by writing $x \in A$, to indicate that x is an element of A; and $x \notin A$, to indicate that x is not an element of A. Sets can be defined either by listing the elements explicitly (the roster method) or by defining a property that only the elements of the set satisfy (the set-builder notation). For example, using the roster method, the set of all even natural numbers less than 10 can be expressed as {2, 4, 6, 8}, and using the set-builder notation, as $\{x \in \mathbb{N} \mid x \text{ is even and } x < 10\}$. In dealing with sets, there are a number of special sets that can be thought of as "reference" sets: 1. The empty set is denoted by the symbol ∅ (which contains no elements). 2. The universal set, U, is a set that contains all elements under consideration, as when the context involves a set of real numbers, a set of complex numbers, etc. 3. The set of natural numbers, $\mathbb{N}$ 4. The integers, $\mathbb{Z}$ 5. The rational numbers, $\mathbb{Q}$ 6. The real numbers, $\mathbb{R}$ 7. The complex numbers, $\mathbb{C}$ These and types of ultimately first lesions, and especially $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{C}$, form the basis upon which advanced mathematical structures and theories create. In set theory, the relationship between sets is commonly described in terms of containment: set A is a subset of set B (denoted as $A \subseteq B$) iff every element of A is an element of B. If, charged, there is at least one element in B that is not in A, then A is a proper subset of B (denoted as $A \subset B$). A set A is equal to a set B (in symbols, $A = B$) if and only if A has exactly the same elements that B has, or more formally $A \subseteq B$ and $B \subseteq A$. These basic membership, subset and equality relations are the grammatical rules of set theory language, allowing us to communicate mathematical logic. In our study of set theory, we get to the idea of the power set of a given set A (or P(A) or 2^A), which is the collection of all subsets of A, including the empty set and A itself. For example, if a set contains n elements, then its power set has exactly 2^n subsets, which shows the

exponential correspondence between a set and its family of subsets. This phenomenon revealing an intricate relationship between combinatory and set-theoretical aspects of nature portends the copious roles now played within the set-theoretically aligned worlds by the various members of the algebraic and analytic branches of mathematical thought. Now, mathematicians have used these concepts as a basis for a more general language — set theory — for describing and analyzing infinite collections, which has resulted in deep insights into the very nature of infinity. Cantor showed that not all infinities are the same; that the "size" or "cardinality" of the set of natural numbers ($\aleph_0$, or "aleph-naught") is different from that of the set of real numbers (c, the cardinality of the continuum). This realization of different "sizes" of infinity transformed mathematics and would become a fruitful topic of research within the areas of set theory, logic, and the foundations of mathematics.

**Set Operations and Their Properties**

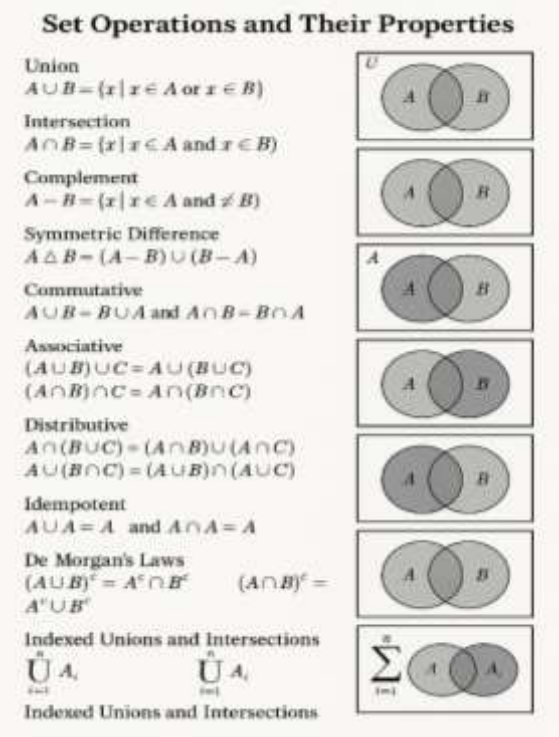



Fig: 1.2 Set operations and Their Properties

This powerful method of mathematical reasoning is because set operations give us the ability to create new sets through a combination of existing sets. Union, intersection, and complement are the three foundational set operations, and each of them has its place in describing relationships between sets. Let A and B be two sets, the

Notes

union of sets A and B, denoted A∪ B, is the set of elements that lie in A or in B (or in both). In formal notation, A ∪ B = {x | x ∈ A or x ∈ B}. From Venn diagrams-discussion point of view, the union is all the area of both sets. For instance, if A = {1, 2, 3} and B = {3, 4, 5}, we have A ∪ B = {1, 2, 3, 4, 5}. In the context of set operations, the intersection of two sets A and B (written A ∩ B) is defined as the set of elements that are in both A and B. More formally, A ∩ B = {x | x ∈ A and x ∈ B}. The intersection (∩) of the dataset A and B is represented in a Venn diagram as the overlapping region of two dataset. Let's continue with the example: A ∩ B = {3}. If a pair of sets has no elements in common, then their intersection will be the empty set, and they are called disjoint sets. If U is a universal set, then let A^c or U - A denote the complement of a set A: the elements of A^c are those in U not present in A. That is, A^c = {x ∈ U | x ∉ A}. We can also define the difference of two sets, so that A - B (also known as A \ B) contains all of the elements contained in A that aren't in B, that is, A - B = {x | x ∈ A and x ∉ B}. This is also expressed as A ∩ B^c, showing how the difference between two sets relates to intersection and complement. All elements that are in either set A or set B, but not in both, are included in the symmetric difference of the two sets, denoted A△ B. By Definition, A △ B = (A − B) ∪ (B − A) = (A ∪B) − (A ∩ B). Operations that satisfy many algebraic properties that correspond to those from other mathematical structures: the commutative property (A ∪ B = B ∪ A and A ∩ B = B ∩ A), the associative property ((A ∪ B) ∪ C = A ∪ (B ∪ C) and (A ∩ B) ∩ C = A ∩ (B ∩ C)), the distributive property (A ∩ (B ∪ C) = (A ∩ B) ∪ (A ∩ C) and A ∪ (B ∩ C) = (A ∪ B) ∩ (A ∪ C)), and the idempotent property (A ∪ A = A and A ∩ A = A). This can handle union and intersection operations in an elegant way, as given in De Morgan laws: (A ∪ B)c = Ac ∩ Bc and (A ∩ B)c = Ac ∪ Bc. Moreover, sets have identity: A ∪∅ = A (the empty set is the identity with respect to union) and A ∩ U = A (the universal set is the identity with respect to intersection). Complementary identities are A ∪ A^c = U and A ∩ A^c = ∅, which formalize the intuitive understanding that a set and its complement make up the entire universal set, and they do not overlap. NOTE: With multiple sets, you can justify operations based on indexing. For a collection of sets $\{A_1, A_2, \ldots, A_n\}$ we can define the union as $\bigcup_{i=1}^{n} A_i = A_1 \cup A_2 \cup \ldots \cup A_n$ and the

intersection as $\bigcap_{i=1}^{n} A_i = A_1 \cap A_2 \cap \ldots \cap A_n$. It is an indexing that enables the compact representation of operations over large collections of sets, and can be made more general to infinite collections. From this perspective, set operations not only serve up practical computational tools for reasoning about math and about our world, but they also uncover some beautiful and deep structural patterns. The language of the operations and properties of sets is equivalent to the operations and properties of a Boolean algebra. Set theory is a favorite topic because it exposes an overlap in the landscape of mathematics, linking disparate branches of the field in the process. Venn diagrams provide insight into the geometric interpretation of set operations. Sets are depicted as regions in a rectangle (the universal set) in these diagrams, while operations correspond to combinations of these regions. Venn diagrams of three or more sets may become complicated; however, they are very useful in visualizing the relationships between the different sets. Set operations find many practical applications in diverse areas. This is part of the query language of databases and its semantics — queries can be expressed through set operations on tables containing data. In probability theory, an event can be modeled as a set, and operations on sets correspond to logical relationships between events. In topology, open and closed sets and their operations essentially determine the basic structure of topological spaces. The operations and their properties that you are learning become essential tools for developing rigorous mathematical analysis and problem-solving as you study more advanced topics in set theory.

**Cartesian Product: Definition and Basic Properties**

The Cartesian product is a basic operation in set theory that lets us combine existing sets into new sets, creating ordered pairs of elements. A Cartesian product is a fundamental operation in set theory, named after the French mathematician and philosopher René Descartes who developed the systematic application of coordinates in geometry. That is, for any two sets A and B, their Cartesian product, written as $A \times B$, is the set of all possible ordered pairs (a, b), where $a \in A$ and $b \in B$: $A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$. The key idea is that we are working with an ordered pair: (a, b) is not the same as (b, a), unless a = b. This introduces a kind of asymmetry that sets the Cartesian product apart from set operations such as union and

intersection. The Cartesian product of two sets A and B is denoted by A × B and result is a set of all ordered pairs (x, y) such that x belongs to A and y belongs to B. As this set is the Cartesian product of 2 sets, so contains 6 elements: |A × B| = |A| × |B| = 2 × 3 = 6. This multiplicative character of cardinality is a fundamental aspect of the Cartesian product and why it's called that. While the union of sets merges their elements, the Cartesian product maintains the individual identity of each set and establishes a relation between their elements.

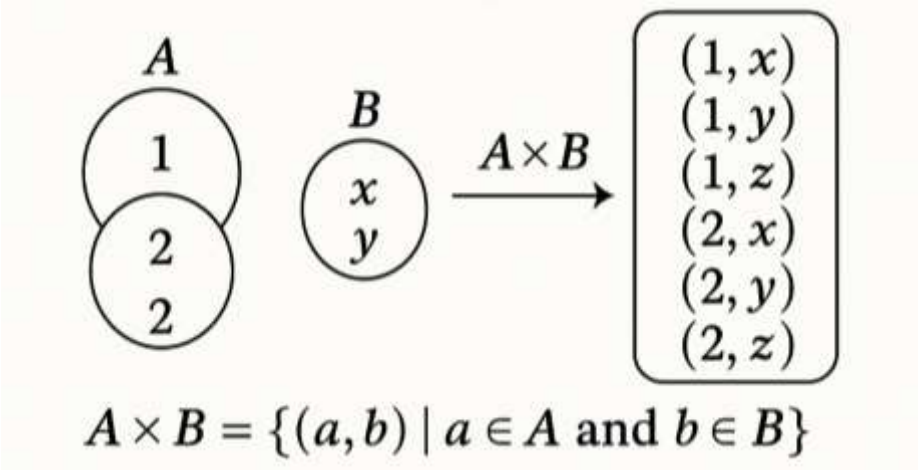



Fig: 1.3 Cartesian Product

The Cartesian product preserves structure, making it crucial when it comes to defining relations and functions, two concepts that are fundamental to mathematics. For multiple sets, the Cartesian product can similarly be extended. For three sets A, B, and C, we define A × B × C = {(a, b, c) | a ∈ A, b ∈ B, c ∈ C}, the ordered triples with the indicated components. More generally, for n sets $A_1$, $A_2$, …, $A_n$, their Cartesian product $A_1 \times A_2 \times \ldots \times A_n$ consists of all ordered n-tuples $(a_1, a_2, \ldots, a_n)$ such that $a_i \in A_i$ for i = 1, …, n. If all input sets are still the same, say A, we tend to denote the n-fold Cartesian product A × A × … × A by $A^n$; guiding this notation is the connection we will see between Cartesian products and exponentiation defined within the context of sets. The Cartesian product has a few important properties. In contrast to union and intersection, it is not generally commutative: A × B ≠ B × A unless A = B or at least one of the sets is empty. It is, however, associative in a certain way: the set of ordered pairs (A × B) × C is not the same as the set A × (B × C) of ordered pairs, but there is a natural bijection between (A × B) × C and A × (B × C). Now, this bijection maps ((a, b), c) to (a, (b, c)), preserving the ordering

information between different levels of parentheses. The Cartesian product distributes over union: $A \times (B \cup C) = (A \times B) \cup (A \times C)$ and $(B \cup C) \times A = (B \times A) \cup (C \times A)$. It becomes very useful for simplifying complex functions involving Cartesian products, due to this property. But distribution over intersection ($A \times (B \cap C) = (A \times B) \cap (A \times C)$ and $(B \cap C) \times A = (B \times A) \cap (C \times A)$) requires equality, not just containment. Notably, the interaction of Cartesian products with the empty set: that means if A or B is something like empty, then $A \times B$ is empty. Since you cannot form any ordered pairs if you do not have an element to pull from one of the two sets that make up that ordered pair. The Cartesian product has deep geometrical interpretations. $R \times R$ $R \times R$ is commonly called R2 R 2 and it is s well-known fact that R2 R 2 is the so-called plane (i.e. coordinate system for pairs of reals). For instance, $\mathbb{R}^3 = \mathbb{R} \times \mathbb{R} \times \mathbb{R}$ is a three-dimensional space. This approach, by Descartes, not only changed the face of geometry, but also allowed algebraic techniques to be used to solve geometric problems, and vice versa. The Cartesian product can also be used to define relations between sets. A relation from a set A to a set B is just a subset of the Cartesian product $A \times B$, where the elements (a, b) of the relation can be interpreted as a connection or correspondence between elements a of A and b of B. Special types of relations, equivalence relations and order relations, are very important in many branches of mathematics. Functions, the ubiquitous building blocks throughout mathematics, are special kinds of relations. It maps each $a \in A$ to a unique $b \in B$ and is defined as a subset of $A \times B$ where each element from A appears exactly once in the first component of the ordered pair. Consequently, the idea of a function is an immediate consequence of the Cartesian product. That is why you covered the Cartesian product in the same way, as it is fundamental in Computer Science topics: in database theory in particular where relations (which in DB table form) are combined and joins are defined The Cartesian product is a crucial operation in the study of cardinal arithmetic (especially for infinite sets) and leads to wonderful results about the "sizes" of different infinities. The Cartesian product is an important concept in set theory, as it allows us to combine sets in a way that captures the relationship between their elements.

**Applications of Cartesian Products in Mathematics**

The Cartesian product not only plays a fundamental role in set theory but also serves as a key concept with applications spanning across different mathematical landscapes, fostering connections between seemingly unrelated areas through its elegant and versatile framework. In linear algebra, we build vector spaces through Cartesian products. A vector space of dimension n over a field F is a Cartesian product of n copies of the set F F^n = F × F ×... × F (n times). This way of looking at things helps clarify the coordinate representation of vectors, in which each component relates to one dimension. One of the basic building operations — matrix multiplication — is done using dot products performed on components that come from the Cartesian Product structure. In three-dimensional space, the cross product, denoted a × b for two vectors a and b, is defined via the determinant of a matrix built from the components of these vectors—a property made available using the Cartesian product representation of the vectors themselves. The Cartesian product is used in abstract algebra to construct direct products of algebraic structures. For groups G, H, we define the Direct Product G × H to be a new group with such an operation: $(g_1, h_1) * (g_2, h_2) = (g_1 * g_2, h_1 * h_2)$ Similar constructions hold for rings, modules, and other algebraic structures, enabling mathematicians to compose complex structures from simpler ones. This is how finitely generated abelian groups are classified, because any finitely generated abelian group is a direct product of cyclic groups, a result exhibiting a remarkable degree of order between a huge class of algebraic objects. This agrees with the definition used in topology, where the Cartesian product X × Y of topological spaces X and Y carries the product topology such that the projection maps are continuous (that is, we take the coarsest topology on X × Y that makes the two projection maps continuous). This means that topologists can build new spaces and work with controlled properties, and this leads to some fundamental results in topology, e.g., the Tychon off theorem on compactness of products of arbitrary families of compact spaces. At an extremely high level, the definitions of topological manifold (the underpinnings of differential geometry and mathematical physics) are premised on the idea that every point has a neighborhood that is homeomorphic to an open subset of $\mathbb{R}^n$ (which itself is just a Cartesian product).

Cartesian products appear frequently in analysis, particularly in multivariate calculus. A function f: $\mathbb{R}^n \to \mathbb{R}^m$ from several variables f: $\mathbb{R}^n \to \mathbb{R}^m$ maps between two Cartesians. The Cartesian product provides the coordinate structure from which partial derivatives, gradient vectors, and multiple integrals are defined. The chain rule for multivariate functions, a primary result of calculus, describes how derivatives behave under composition, using the product structure of the domain and co domain. In probability theory, the Cartesian product is useful for modeling experiments with more than one outcome. If ($\Omega_1$, $F_1$, $P_1$) and ($\Omega_2$, $F_2$, $P_2$) are probability spaces, the product space ($\Omega_1 \times \Omega_2$, $F_1 \times F_2$, $P_1 \times P_2$) is constructed to model the joint system. Formally the independence of events, or independent random variables is defined as how the probability measures act on the product structure. Independent random variables, a core concept in the formulation of statistical theory, are defined in the framework of the Cartesian product. This Cartesian product plays an important role in data modeling and algorithm design in computer science. In relational database theory, however, it is even more general, as tables are modeled as Cartesian products with constraints, and operations like joins are defined as selections from these products. The efficiency of algorithms on multidimensional data often shows a Cartesian product structure in the input space. Dynamic programming, a powerful algorithmic technique, often harnesses how problems can be separated along the lines of what makes up a Cartesian product of potential states. In graph theory, the Cartesian product of graphs G and H is a graph denoted G □ H, with vertex set as the Cartesian product of the vertex sets of G and H, where two vertices are adjacent if they are adjacent in G while being equal in H (or vice versa). This construction gives rise to important families of graphs like hypercube and grid graphs, which have applications in network design, coding theory, and distributed computing. The hypercube $Q_n$, the n-fold Cartesian product of the complete graph $K_2$, has particularly interesting applications in computer science — indeed as the topology of various parallel computing architectures. Breaking ground: Cartesian product to model game theory. The strategy space in an n-player game is typically represented as the Cartesian product of individual strategy sets. Nash equilibrium (a key concept in economic theory) is defined as a point in this product space at which no player

can resultantly gain by unilaterally changing its strategy. Many combinatorial optimization problems can be formulated with the objective of finding optimal points in highly structured Cartesian products subject to a variety of constraints. The tensor product of two vector spaces is the vector space of all bilinear functions on the Cartesian product. The primary nature of the product alongside the popularity of the product across mathematics. It acts as a robust tool for constructing more intricate mathematical objects from simpler constituents, offering a structured method to merge, while retaining their structural elements. By providing a way for mathematicians to approach problems in a lower-dimensional space, the constructive property of the Cartesian product can illuminate aspects of mathematical structures that may not be easily accessible in their original dimensions.

## 1.4: Relations and Functions as Subsets of Cartesian Products

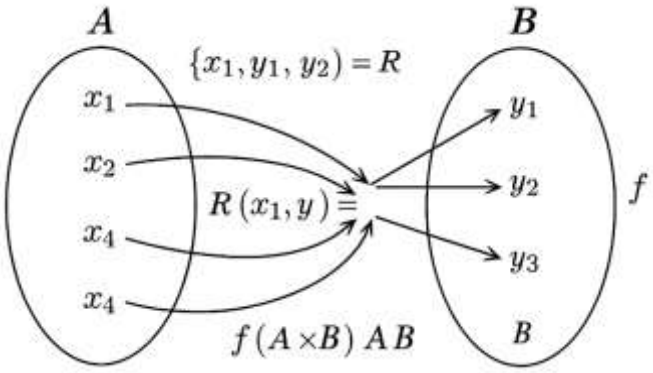



Fig: 1.4 Relations and Functions as Subsets of Cartesian Products

These two fundamental concepts are the relations and functions, concepts that underlie all branches of mathematics itself that can be exactly formulated through the language of Cartesian products. In formal set theory, a relation R from a set A to a set B is defined as a subset of the Cartesian product A × B: R ⊆ A × B. Each ordered pair (a, b) ∈ R establishes a connection or correspondence between an element a ∈ A and an element b ∈ B, capturing the fundamental concept of relating items from distinct sets while adhering to the mathematical structure afforded by the theory of sets. Often a R twice commutated b is written as (a, b) ∈ R, to highlight that they are related. As an example, let A denote a set of cities and B denote

temperatures in degrees Celsius, then a relation $R \subset A \times B$ could be pairs (city, temperature) representing the measured temperature for each city on a certain day. Based on the properties of relations, we can classify it into different types which have certain mathematical significance. A relation R on a set A (i.e., a relation from A to itself, or subset of $A \times A$) is reflexive if $(a, a) \in R$ for all $a \in A$, so every element is related to itself. The relation R is symmetric if whenever $(a, b) \in R$ then $(b, a) \in R$, which means that the relation works both ways. Transitive: for all (a,b),(b,c) in R implies (a,c) in R, that is, the relation "passes through" intermediate elements. An equivalence relation is a relation which is reflexive, symmetric and transitive at the same time, and is a basic idea used to create equivalence classes from sets. For example, in number theory, we speak of the congruence modulo n, while in geometry we first speak of the similarity of geometric figures, and later of isomorphism of algebraic structures. Another category of relations is order relations that satisfy different properties. A partial order equivalently is a relation that is reflexive, ant symmetric ($(a, b) \in R$ and $(b, a) \in R$ implies $a = b$), and transitive. A total order also imposes the requirement that for any two elements a and b, either (a, b) is in R or (b, a) is in R: order relations are very important in defining order within ordered sets, and has applications that range from the natural ordering of numbers to the subset relation on a power set. And many different tools can be used to model and represent relations (and perform analysis). Note that the graph for a relation R from A to B is the set of ordered pairs $\{(a, b) \mid (a, b) \in R\}$ which we can visualize when A and B are finite sets. Another way to represent relations is by means of matrices: Given two finite sets $A = \{a_1, a_2,..., a_m\}$ and $B = \{b_1, b_2,..., b_n\}$, a relation $R \subset A \times B$ can be expressed as an $m \times n$ matrix M in which $M(i, j) = 1$ if $(a_i, b_j) \in R$ and $M(i, j) = 0$ otherwise. This matrix representation allows relations to be analysed computationally, linking set-theoretic constructs to linear algebra. The operations on relations include composition, inverse, and the various set-theoretic operations on their graphs. Let $R \subset A \times B$ and $S \subset B \times C$ be relations, then the composition $S \circ R \subset A \times C$ is defined by $\{(a, c) \mid \exists\ b \in B$ such that $(a, b) \in R$ and $(b, c) \in S\}$. The inverse of a relation $R \subset A \times B$ is called R^(-1): $\{(b, a) \mid (a, b) \in R\} \subset B \times A$ (it reverses the relation). Union, intersection, and difference of relations (considered as sets of ordered pairs) are inherited directly

from the corresponding set operations. Functions are a special class of relations that have important properties in mathematics. We say a function from A to B, f: A → B, is a relation f ⊂ A × B with the property that for all a ∈ A, there exists exactly one b ∈ B such that (a, b) ∈ f, so the mapping of f passes the vertical line test (every input has one output and one only). Like so, f(a) = b, rather than (a, b) ∈ f — functions emphasize the (what goes in what comes out of) mapping over the (mapping) itself. In this case we use notation dom(f) as the domain of f, where f is a function from A to B; the set of f of all the function values arrived are defined as the co domain (or return values for a function): img(f) is the image of f, some subset of B such that: {b ∈ B : ∃ a ∈ A : f(a) = b}; Depending on the functions increase property on the values of the target set, we can categorize those functions. A function f: A → B is injective (one-to-one) if no two inputs provide the same outputs: ∀$a_1$, $a_2$ ∈ A, $a_1 \neq a_2 \Rightarrow f(a_1) \neq f(a_2)$. That is, f is injective if $f(a_1) = f(a_2)$ implies $a_1 = a_2$. If f is a function from a set A to a set B, we denote f: A to B. A function is subjective (onto) if for every b ∈ B there is at least one a ∈ A such that f(a) = b: for all b ∈ B, ∃ a ∈ A: f(a) = b. A function which is injective (one-to-one) and subjective (onto) is called bijective, i.e., we have a one-to-one correspondence between the elements of A and those of B. Bijections indicate when two sets have the same cardinality, a very important notion in the study of infinite sets. The operations on functions are composition, restriction and extension. We define the composition of f: A → B and g : B → C as g∘f : A → C such that (g∘f)(a) = g(f(a)) for all a ∈ A. It is an associative operation but not commutative in general. If f: A → B is a function, the restriction of f to a subset A' ⊂ A is the function f|A': A' → B defined by f|A'(a) = f(a) for all a ∈ A'. In contrast, a function g: A' → B can sometimes be extended to a function f: A → B where A' ⊂ A, such that f|A' = g. The idea of the image of a subset under functions and its inverse image (preimage) provide excellent tools for understanding how the function transforms sets. For a function f: A → B and subsets A' ⊂ A and B' ⊂ B, the followings are defined; image: f(A') := {f(a) | a ∈ A'} preimage: f^(-1)(B') := {a ∈ A | f(a) ∈ B'} These operations fulfill different set-theoretic properties: $f(A_1 \cup A_2) = f(A_1) \cup f(A_2)$, but equality for intersection holds only in special cases; for preimages, both distribution properties hold: $f^{-1}(B_1 \cup B_2) = f^{-1}(B_1) \cup f^{-}$

1)($B_2$) and f^(-1)($B_1 \cap B_2$) = f^(-1)($B_1$) ∩ f^(-1)($B_2$). Some special types of functions are injections, surjection's, bijections, and functions with other algebraic or analytical properties such as homomorphism, homeomorphism, and continuous functions. The identity function on the (potentially infinite) set of elements A, id_A: A → A by mapping each element a ∈ A to itself: id_A(a) = a, acts as the neutral element for function composition. Every bijection f: A → B has an inverse function f^(-1): B → A such that f^(-1) ∘ f = id_A and f ∘ f^(-1) = id_B, which gives us a way to "undo" the action of f. The study of relations and functions in terms of Cartesian products not only gives a rigorous foundation to these notions but also unveils deep connections between these concepts and other areas of mathematics. This set-theoretic treatment is made possible since the specific details of the objects being related and/or mapped of interest are effectively hidden behind this abstraction, and the relation can be treated as a distinct mathematical object in its own right, leading to a unified approach that results in being able to treat these common mathematical constructs in similar ways.

## 1.5: Advanced Topics: Power Sets and Cartesian Product

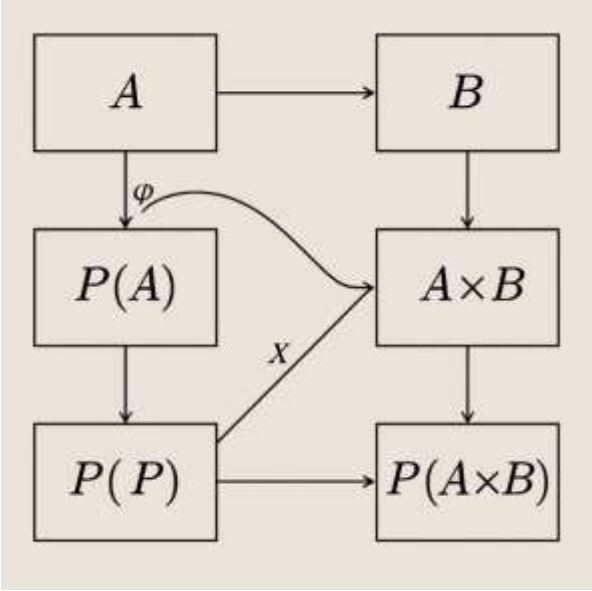



Fig: 1.5 Power Sets and Cartesian Product

The connections between power sets and Cartesian products are deeply revealing about the structure of sets and give rise to some sophisticated constructions throughout mathematics with crucial

Notes

consequences. Power set the power set of a set A, denoted P(A) or 2^A, is the set of all subsets of A, including the empty set ∅ and A itself. So for a finite set A with n elements, there are exactly 2^n subsets in the power set, hence the notation 2^A is in exponential terms. We can also see that the size of the power set grows exponentially in terms of A -- a growth level that is directly corresponding with binary sequences, as the number of subsets corresponds to the number of ways we can pick or not pick Module association of the individual elements that make up A (i.e. for A { x, y, z, w } = {{ }, {x}, {y}, {z}, {w}, {x,y}, {x,z}...) Now, we're going step by step into understanding why we are using Cartesian products in the first place. For two sets, A and B, the set of all functions f: A → B, sometimes denoted B^A, has cardinality |B|^|A| (where A and B are finite). In this sense, when we refer to the power set of A as P(A)=2A, we are defining it in the sense that we are identifying a set with all functions that can be defined taking the set A as an argument. This is the Cartesian product of the two sets of their power sets P(A) and P(B), which will yield all ordered pairs (X, Y) where X ⊆ A and Y ⊆ B, while not the same as the power set of the Cartesian product P(A × B), which includes all subsets of A × B, e.g., arbitrary collections of ordered pairs. The connection between these two constructions is subtle but instructive: there is an injection from P(A) × P(B) into P(A × B) that sends (X, Y) to X × Y, but this mapping is not subjective unless one of the two sets is empty or singleton. This confirms that power sets for Cartesian products are not distributive: P(A × B) ≠ P(A) × P(B) in general. This is due to the fact that P(A × B) also includes relations between A and B that cannot be captured as Cartesian products of subsets. Binary relations between two sets A and B (subsets of A × B) are an important concept in set theory and across mathematics. The power set P(A × B) is exactly the set of all binary relations from A to B. For finite sets |A| = m and |B| = n, we can construct 2mn different relations, which illustrates the combinatorial explosion that results when considering every possible connection between elements of the two sets. This viewpoint is a bridge — connecting the power set (along with products, both Cartesian and otherwise) and relational theory; revealing their inter-relatedness. Another connection between power sets and Cartesian products comes from characteristic functions. The mapping between

the set of real numbers and the set of real numbers, where the characteristic function defined as $\chi A:\{0,1\}\rightarrow\{0,1\}$ given by: $\chi\_X(a) = 1$ if $a \in X$, $\chi\_X(a) = 0$ if $a \notin X$ establishes a bijection $X \mapsto \chi A : P(A) \leftrightarrow$ where $a \subseteq A$ $X \in P(A)$ $P(A) \rightleftarrows \{0, 1\}^A$. Moreover, associated mapping for $\chi\_A$ $P(X) \leftrightarrow P(A)$ Find relations between the elements of P(X) and P(A): $X \in P(A) \leftrightarrow A \in P(X)$ and so forth. Measures that are characteristic functions are typically used in functional analysis, while functions that operate on Boolean algebra are used in digital logic. The connections between Cartesian products and power-sets have beautiful expression in category theory, a branch of mathematics that abstracts and generalizes large swathes of mathematics. The Cartesian product is the categorical product in the category of sets, and the power set operation corresponds to the categorical notion of an exponential object. Since one is only concerned with a category-theoretic perspective on things, one can witness how such constructions are examples of more abstract patterns in other mathematics and this provides insight into their properties. While talking about power sets and Cartesian products, it adds more complexity to ordinary sets. Cantor's theorem expresses that for any set A, the cardinality of P(A) (i.e. the powerset of A) is strictly greater than the cardinality of A, which constitutes an infinite hierarchy of ever growing infinite cardinalities. This result, together with properties of Cartesian products of infinite sets, underlies cardinal arithmetic — the study of operations on the "sizes" of infinite sets. For infinite cardinal numbers $\alpha\alpha$ and $\beta\beta$, the cardinality of their Cartesian product $\alpha\times\beta\alpha \times \beta$ equals.

## 1.6: Statements and Notations, Logical Connectives

**Introduction to Mathematical Logic and Statements**

All mathematical reasoning for which reason judgment based on logic takes place. The form of mathematics we are interested in, mathematical logic, is at its core about formally specifying the rules we can use to detect validity in reasoning, and the differences between forms of valid and invalid reasoning. The simplest entities in this logical system are mathematical statements: declarative sentences that are unequivocally true or false, but not both at the same time. This binary quality of mathematical statements, that they must have precisely one truth value, is the bedrock of classical logic, a system that has reigned over mathematical thought since the age of Aristotle. For instance, take the statement "7 is a prime number." This is a

proper mathematical statement because it is an assertion that can be proved true or false according to established mathematical methods. Likewise, the statement "For all real number x, x2 ≥ 0" is a mathematical statement, whose truth may be determined. On the contrary, phrases such as "Please solve this equation" or "x + 5" do not qualify as mathematical propositions; the first one is an imperative rather than a proposition, whereas the second one is just an algebraic expression that does not express anything true or false. This is important because there are statements whose truth values are theoretically determinate, but whose truth values would be practically unknowable with current mathematics. For example, the claim "The Riemann Hypothesis is true" is a well-formed mathematical statement, even though we know neither it nor its negation to be true.

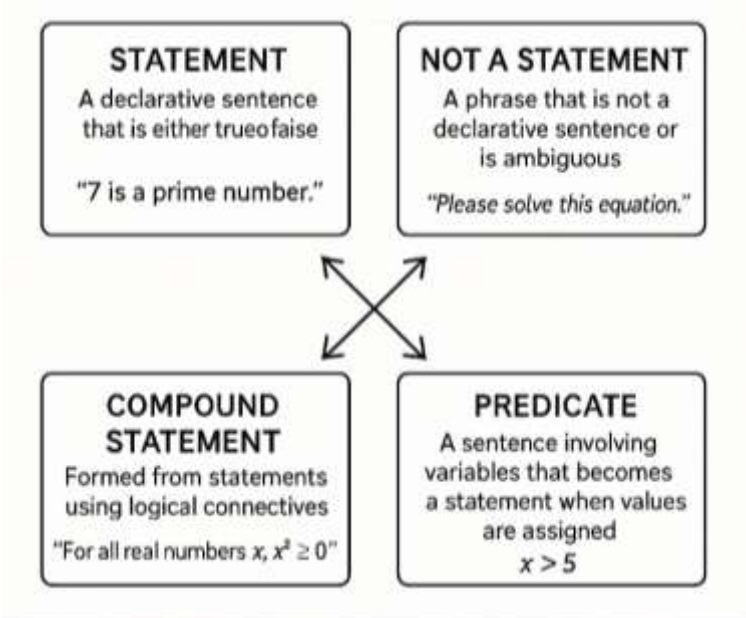



Fig: 1.6 Introduction to Mathematical Logic and Statements

This brings us to a critical difference between the semantic property of truth and falsity versus the epistemological issue of determining the truth-value of a statement. A language is formed where statements are represented by variables p, q, r, or capital letters P, Q, R, or any combination thereof so that we can examine logical relationships abstracted from any particular content. It allows us to analyze the composition of logical argumentation without getting blinded by the content of a specific proposition. Mathematicians add even more categories based upon the foundation. Atomic statements are statements without decomposed parts; they cannot be broken into more statements. Compound statements are statements made from the combination of local blocks of statements connected by logical connectives. Open statements with undecided truths are statements with variables defined such that the truth of the statement is defined by those variables. Quantified statements are statements that make

claims about every or some of the items within a domain. This classification gives us a taxonomy through which we can explore the richness of mathematical reasoning more systematically. The other key difference is between statements and predicates. A predicate is a statement form with one or more variables, which becomes a statement when replacements are made for the variables that give it a

| CONNECTIVE | SYMBOL | TRUTH TABLE | | | |
|---|---|---|---|---|---|
| Negation | ¬ | $p$ | | $\neg p$ | |
| | | T<br>F | | F<br>T | |
| Conjunction | ∧ | $p$ | $q$ | $p \wedge q$ | |
| | | T<br>F | T<br>F | T<br>F | T<br>F |
| Disjunction | ∨ | $p$ | $\vee$ | $p \vee q$ | |
| | | T<br>F | T<br>T | T<br>T | T<br>T |
| Conditional | → | $p \rightarrow q$ | | $p \rightarrow q$ | |
| | | T<br>F | T<br>F | T<br>T | T<br>T |
| Biconditional | ↔ | $p \leftarrow q$ | | $p \leftrightarrow q$ | |
| | | T<br>F | T<br>F | T<br>F | T<br>F |

Fig: 1.7 Mathematical Logic

definite truth value. For example, the expression “x > 5” is a predicate that becomes a statement with a specific truth value when a concrete value is provided for the variable x. Statements and their logical relations become the foundation for advanced topics like formal proof systems, model theory, and issues of completeness or consistency in the context of mathematical theories. **Logical Connectives: Foundations and Truth-Functional Analysis** Just as addition and multiplication come together to form sum-product structure to represent efficient computations, logical connectives are the fundamental operations that enable us to build complex statements from simpler ones, yielding a grand and sophisticated language with which we can communicate complex mathematics. The basic building blocks of logical syntax are the logical connectives: negation, conjunction, disjunction, conditional, and biconditional—and these provide the syntax for binding simple or atomic statements into more complex propositions to encapsulate mathematical relationships of increasing complexity. All connectives have truth-functional behavior: that is, the truth table for any compound statement is always completely determined by the truth values of its constituent statements and by the precise logical relation which is being asserted between them by means of the connective. ¬p is false, and if p is false, ¬p is true. This operation, also known as "logical NOT," enables us to express the denial or negation of a particular statement. For instance, if we have that p = "x = 5", then ¬p means "x ≠ 5". The logical

conjunction of two statements p and q, denoted p ∧ q (or sometimes p & q), states that both statements hold true at the same time. This connective, which is to propositional logic as the linguistic “AND” is to the human language, produces a true compound statement if and only if both of its constituent statements are true and is false otherwise. For example, let p be "x > 0" and q be "x 0 and x 0 or x < 10," which holds for all real numbers except x = 0. Data scientist =⇒ Machine Learning interpreter |Professional experience | Be sure to know your conditional statements — These statements are based on the conditional statement, If the antecedent (p) is true, then the consequent (q) will be true. This connect conjunctive is false only in the case where p is true and q is false, in all other cases the connective statement is true. This defies our intuition when p is false, yet it is the stipulation of the truth table of p implies q that is consistent regarding mathematical implications. Lastly, the biconditional (p ↔ q (or p ⇔ q)) indicates logical equivalence between two statements; it states that both statements have the same value in terms of truth (both true and both false). This “if and only if” relation combines two conditionals (p → q and p ← q) and is true exactly when the constituent statements match truth value. Truth tables are a systematic way to analyze such connectives and show all possible combinations of truth values for the component statements and the truth value of the compound statement. Whereas for conjunction (p ∧ q) we have a truth table with four rows representing the four possible combinations of truth values for p and q, but the conjunction is true only in the case when both p and q are true. They provide these truth-functional definitions to give logical connectives the mathematical (and hence unambiguous) meaning they lack in natural language. In addition to the five primitive connectives, we can define other operations such as the exclusive disjunction (XOR, a.k.a. p ⊕ q) and the Shaffer stroke (NAND, denoted p | q). Incidentally, some of these operations have the property of functional completeness, which means that these properties may be expressed, either individually or in any combination cast, any truth functional operation. In particular, the negation plus conjunction pair can express all other connectives as can the Sheffer stroke alone. This functional completeness illustrates how even small subsets of logical connectives can be highly expressive. More than that, the algebraic properties of the logical connectives (associatively, commutatively, distributive, identity, inverse) show us that they have very rich similarities with operations on other mathematical systems as well. Another example comes from conjunction and disjunction being duals of each other, as expressed by their relationship through De Morgan's laws: ¬ (p ∧ q) ≡ ¬p ∨ ¬q and ¬(p ∨ q) ≡ ¬p ∧ ¬q, which illustrate how negation behaves in relation to them, so it should behave with all of the other connectives, providing us with powerful tools for either

simplifying or reasoning all the valid instances of the system. Learning logical connectives and their properties might seem purely abstract, yet serves as the basis for proving techniques in mathematics, designing algorithms and encapsulating circuit design in computer engineering, querying databases in data science, etc.

**Negation: The Fundamental Unary Operator**

In classical propositional logic, negation is the only unary logical operator, meaning an operator that takes a single statement and constructs its logical negation. Represented with ¬ (or ~!, or ¬), negation changes a proposition p into its negation ¬p, stating exactly the opposite of the original statement. The ease of this operation hides its deep centrality in mathematical argument and its widespread occurring across branches of mathematics. Negation is truth-functional because it obeys a truth table: negate a true statement to get a false statement, negate a false statement to get a true statement. This behavior exemplifies logical contradiction—¬p is true if and only if p is false, and vice versa. In natural language negation can be expressed as "it is not the case that," "not," or "it is false that," although in mathematical notation the more succinct symbolic representation is used. Negation has its own formal properties that are making it different from other logical connectives. Most unwieldy of all, negation is involutive: ¬ (¬p) ≡ p. This property, known as the law of double negation, distinguishes classical logic from intuitionist and other non-classical logical systems where double negation elimination does not universally hold. Moreover, negation is neither idempotent (¬p ≠ p), commutative (it is a unary operator, so the question does not apply), nor associative (likewise). Negation is central to some of the most basic logical principles. Since it is not logically possible for a proposition and its negation to be true at the same time, this principle is formally stated as ¬ (p ∧ ¬p) (meaning the negation of p and not p). This rule, derived from Aristotle's metaphysical writings, is a foundational tenet of classical logical inference. The law of excluded middle, which states that either something is true or not: p ∨ ¬p, also rejects all intermediate possibilities. These principles seem intuitively obvious, but they have been challenged in several non-classical logics (many-valued logics and fuzzy logic including quantum logic), where propositions might take intermediate truth values or where the bivalence principle (that propositions are either true or false) is relaxed. In mathematical

practice, negation is more than the mere opposite. It allows drafts of proof by contradiction (reduction ad absurdum), where one concludes some proposition p, because assume ¬p leads to some logical contradiction. This powerful technique has produced many important results in mathematics, from the irrationality of $\sqrt{2}$ to existence and uniqueness theorems. Negation can help you express mathematical concepts clearly: when x does not equal y (that is, $x \neq y$) is the same thing as x = y is false; x is not an element of S (that is, $x \notin S$) is the same thing as x is an element of S is false; the complementation operation in set theory (the notation $A^c$ or $\bar{A}$ representing the elements not in set A, where A is a set). Negation interacts with other logical connectives in ways that yield important logical equivalences, such as De Morgan's laws. These laws state that negation of conjunction is disjunction of the negations ($\neg(p \wedge q) \equiv \neg p \vee \neg q$) and vice versa negation of disjunction is conjunction of the negations ($\neg(p \vee q) \equiv \neg p \wedge \neg q$). These equivalences are building blocks for logical simplifications and proof methods. Furthermore, negation converts various other connectives to their duals: negation of an implication ($\neg(p \rightarrow q) \longleftrightarrow p \wedge \neg q$), and negation of a biconditional ($\neg(p \leftrightarrow q) \longleftrightarrow p \oplus q$). Negation interacts with quantifiers to give rise to especially important logical forms. If you see a universal statement $(\forall x)\ P(x)$, then its negation is equivalent to the existential statement $(\exists x)\ \neg P(x)$; similarly, if you see an existential, $(\exists x)\ P(x)$, then its negation is the universal $(\forall x)\neg P(x)$. It is important to note that these relationships, known also as the quantifier negation laws, guide us through understanding the meaning of mathematical statements as well as the development of proofs on expressions presented in the quantifier form. Negation has applications beyond the theoretical, finding its usefulness in computer science and digital electronics as it has a real-world counterpart in the form of the NOT gate, a fundamental component of digital circuit design. Negation, commonly used in programming languages as an operator (for instance,! or NOT critical for conditional control structures and logical tests. Negation in database query languages disallows certain records, while negation plays an integral role in knowledge representations and reasoning systems in AI. Negation can also affect how one thinks about what can be known, which in turn touches upon some common themes in epistemology, and even the philosophy of mathematics. Then,

exploring the consequences of this perspective leads to various alternatives that become prominent in specific areas of mathematics, such as constructivist mathematics or intuitionist logic, in which negation is used in a more limited way, especially in existence proofs, preferring constructive techniques involving finites or constructivist methods over applied logical ones involving negation. These concerns reveal how this apparently straightforward calculation relates to deep issues regarding the underpinnings of mathematical reasoning and the character of mathematical reality.

**Conjunction and Disjunction: The Core Binary Connectives**

As the basis for expressing complicated relationships between mathematical statements, conjunction and disjunction make up a complementary pair of binary logical connectives. In this way, mathematicians gain the ability to express compound statements that incorporate atomic statements through logical operations that work roughly speaking like the "and" and "or" of natural language. The conjunction of two statements p and q (note the symbol $p \wedge q$) states that both constituent statements are true at the same time. Its truth functional behavior is simple: the conjunction is true if and only if both p and q are true; otherwise it is false. This definition matches the intuitive meaning of “and” in ordinary reasoning, in which the assertion that “A and B” necessarily implies the truth of both A and B, since the mathematical statement “$x > 0$ and x 0) $\wedge$ (x 1 (in formal notation: (x 1)) is actually true for all x excluding those x representing a number from the closed interval [0,1]. This gives rise to some fundamental algebraic properties of conjunction and disjunction which make logical reasoning and proof-writing easier. Both operations are commutative ($p \wedge q \equiv q \wedge p$ and $p \vee q \equiv q \vee p$), indicating the order of the constituent statements does not change the truth value of the compound statement. They are also associative (($p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$ and $(p \vee q) \vee r \equiv p \vee (q \vee r)$), so we can write expressions like $p \wedge q \wedge r$ or $p \vee q \vee r$ unambiguously. Both operations are idempotent, which invites us to eliminate unnecessary repetitions in conjunctions and disjunctions of statements ($p \wedge p \equiv p$; $p \vee p \equiv p$). Also, conjunction and disjunction distribute over one another, but in different ways: $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$ and $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$. They are what allows logical expressions to be transformed from one form to another, often making complex

statements simpler, or allowing a hidden logical structure to emerge. If we observe the identity elements, true (T) is the identity element for conjunction, and false (F) is the identity element for disjunction: therefore, $p \wedge T \equiv p$, and $p \vee F \equiv p$, and if there is an an nihilators which results otherwise, we have that F is the annihilator for conjunction ($p \wedge F \equiv F$), and T is the annihilator for disjunction ($p \vee T \equiv T$). These properties have analogues in other mathematical systems, which are particularly set theory, where conjunction corresponds to intersection ($\cap$) and disjunction corresponds to union ($\cup$). The formal similarities between propositional logic and set theory come out through the isomorphism between Boolean algebra and the algebra of sets, specifically with conjunction and disjunction being the central structural features in this correspondence. For example, the relationship between conjunction and disjunction is dual; this means that you can turn conjunction into disjunction by negating the expression (De Morgan's laws): $\neg(p \wedge q) \equiv \neg p \vee \neg q$, and $\neg(p \vee q) \equiv \neg p \wedge \neg q$. There's a duality to the rest of the properties: the identity element for one is the annihilator for the other (since the annihilator for either is itself an iterate, and behaves as an inertial property), and the distributive laws for each over the other mirror this. In many areas of mathematics, conjunction and disjunction take center stage. And in Set theory, $x \in A$ and $x \in B$ and ($x \in A$ or $x \in B$) are used to dentate the intersection and union of the two sets A and B. In number theory, certain important classes of numbers tend to be characterized by conjunctions of primarily tests or divisibility conditions. In analysis, the requirement for continuity at a point amounts to a conjunction of conditions on the behavior of the function at that point, while discontinuity is a property that can be expressed as a disjunction of ways in which continuity fails. Conjunction and disjunction are also utilized as basic elements for more sophisticated logical constructions in mathematical logic. The exclusive disjunction (XOR, $p \oplus q$) can, for example, be defined using the raw connectives as $(p \vee q) \wedge \neg(p \wedge q)$, which is true if exactly one of the statements is true. Likewise, all other derived connectives and logical constructs can be represented with combinations of conjunction, disjunction, and negation, so this tiny set of operators has expressive completeness. In computer science, for example, their application extends to Boolean operations within programming languages, circuit design, database theory

(specifically query conditions), and artificial intelligence (in knowledge representation and automated reasoning systems). The embodiment of these operations in electronic circuits — via AND and OR gates — underpins the foundations of digital computing, while their use in database query languages allows for the creation of complex search parameters. The complement of the multiplication law for independent events in probability theory ($P(A \cup B) = P(A) + P(B) - P(A \cap B)$) shows the interplay between conjunction and disjunction and how those logical concepts carry over to randomness and probability. So far as the conjunction and disjunction are concerned, the mechanics of the philosophical implications touch base with some of the most basic questions in mathematics and logic about composition and choices. Thus, one finds that the definitions of these operations in classical logic are much more precise than they are however you might use them in a sentence or in other non-classical logic models. The duality of conjunction and disjunction, along with their complementary natures, demonstrates some of the elegant symmetries that exist within mathematical logic and provide a rich framework to express and analyze the logical structures of mathematical statements.

## 1.7: Conditional Statements: Implication and Its Logical Structure

In mathematical reasoning, perhaps the most fundamental (and yet subtle) logical connective is the conditional statement, usually denoted $p \rightarrow q$ or $p \Rightarrow q$. The conditional states an "if-then" relationship between two statements: the antecedent p and the consequent q; it represents the most basic form of logical implication: that if p is true, then q must also be true. This connective is all over mathematics statements of theorems ("If a triangle is equilateral, then it is equiangular"), definitions ("A function f is continuous at a point c if for every $\varepsilon > 0$, there exists a $\delta > 0$ such that...") and across the cloth of mathematical argumentation. At first glance, the conditional is a simple and familiar construction, but it contains subtleties that have led to much logical and philosophical study. The truth-functional definition of the conditional may at first seem counterintuitive: $p \rightarrow q$ is false only when the antecedent p is true and the consequent q is false; in all other cases—including when the

antecedent is false—the conditional is true. This definition, sometimes referred to as the material conditional, differs from many natural language interpretations of "if-then" statements, in which they often imply or entail, causally or inferentially that the antecedent and consequent are connected. The truth table for the conditional indicates this — when p is true and q is true, the conditional is true (as it should be), when p is true and q is false, the conditional is false (as it should be), but when p is false, the conditional is true no matter the truth assignment of q. This last point—that any conditional with a false antecedent follows as a matter of course—underpins the principle of explosion or ex false quodlibet ("from falsehood, all things follow"), a principle that has led to alternative treatments of conditionals in non-classical logics. There are a number of equivalent formulations of the conditional that shed light on its meaning. The statement $p \rightarrow q$ is logically equivalent to $\neg p \vee q$ — that is, either the antecedent does not happen or the consequent does happen. And this equivalence also explains why the antecedent being false yields a true conditional: if p is false, $\neg p$ is true, so the disjunction $\neg p \vee q$ is true no matter whether q is true or false. At the same time, there is another formulation that represents the conditional as the negation of a certain conjunction: $p \rightarrow q \Leftrightarrow \neg (p \wedge \neg q)$, which affirms it is not the case that p holds, but q does not, that is: if p, then q. This view highlights that the conditional, in its essence, excludes precisely one of four combinations of truth values for p and q, leading to a number of derived logical forms. It has the form $p \rightarrow q$ whose contra positive (and logically equivalent statement) $\neg q \rightarrow \neg p$ states that if the consequent is false, then the antecedent must also be false. One of the basic rules of classical logic is that a conditional and its contra positive are equivalent $p \rightarrow q \equiv \neg q \rightarrow \neg p$. This equivalence lays the groundwork for many proof strategies, where proving a statement may be easier if we instead prove its contra positive. The converse of $p \rightarrow q$ is $q \rightarrow p$ and simply switches the antecedent and consequent. The converse is not logically equivalent to the original conditional; given $p \rightarrow q$, we cannot conclude $q \rightarrow p$ at all. Confusing a conditional with its converse is a common fallacy of mathematical reasoning (the "affirming the consequent" fallacy). The next "most general" form is $\neg p \rightarrow \neg q$, known as the inverse of $p \rightarrow q$, i.e., you negate both of the parts of the original conditional. Similar to the converse, the inverse is not

equivalent to the original conditional (although the inverse and converse are equivalent to one another, as they are the contra positive of each other). A conditional, its converse, inverse, and contra positive together form a logically interconnected system, that sheds light on the relationship between p and q from different perspectives. In mathematics, chains of conditionals often occur in sequence of logical deductions. Hypothetical syllogism (or chain rule) given $p \rightarrow q$ and $q \rightarrow r$ we obtain $p \rightarrow r$ formalizes the transitive property between logical premises. In this spirit, the law is the basis for the structure of mathematical proofs, which logically pull conclusions through chains of implications that link the premises to what we normally call theorems. The correspondence between conditionals and logical equivalence is of particular importance. Let p and q be two propositions such that both p and q are true; $p \rightarrow q$ is true; and $q \rightarrow p$ is true; then p and q are logically equivalent symbolized as $p \leftrightarrow q$. This relationship, both ways, which can be expressed as "if and only if" (often referred to as "iff"), becomes very important in mathematical definitions, and characterizations, where the conditions that must hold are necessary and sufficient. Conditionals are found throughout mathematical arguments, in many different forms and contexts. In hypothetical reasoning, we anticipate consequences based on an antecedent we presume. Proof by contradiction involves assuming that the conclusion is false, derive a contradiction, thus proving that $\neg p \rightarrow (q \wedge \neg q)$ and finally conclude that p must be true by the laws of classical logic. Conditional statements also form the basis for the principle of mathematical induction: to show that some property P (n) holds for all natural numbers, one shows P(1) (the base case) and then shows that P(k) implies P(k+1) for all natural numbers k (the inductive step). The conditional structure of this statement reflects the reasoning behind the inductive argument: given the truth of one case, you can infer the truth of the next. Besides classical propositional logic, there are different alternative formulations of conditionals that have emerged to avoid issues considered troublesome with the material conditional. Conditionals in relevant real logic are demanded to have a legitimate relationship between their antecedent and consequent. Intuitionist logic interprets conditionals constructively: it doesn't just need to say if for an antecedent there is an antecedent, but requires methods to convert

evidence for the antecedent into that of the consequent. In probabilistic reasoning, for example, the most natural counterpart of conditionals is conditional probabilities, where P (q|p) measures the probability of q when p is true. These alternative approaches reveal the multifaceted nature of conditional reasoning compared with the truth-functional definition offered by classical logic. The conditional appears in particular mathematical contexts as well. An example from set theory shows how to use "if … then …" statements: "if $x \in A$, then $x \in B$" characterizes subset $A \subseteq B$. Another example: "if $||x - y|| < \delta$, then $|f(x) - f(y)| < \varepsilon$" in functional analysis expresses continuity of a function f; and in number theory "if n is prime, then n is either 2 or odd" characterizes such property. As these examples show, the conditional is one of the threads that shapes the logical relationships among concepts and properties in mathematics. Object of interest: Generalization and its connection between logic and mathematical practice. The difference between material implication (the truth-functional conditional of classical logic) and stronger notions of implication in terms of relevance, causality, or necessity has led to a lot of philosophical logic work. These are also considerations that highlight the pivotal nature of conditional reasoning at the heart of human thought in general and mathematical thought in the specific, showing that the conditional has long been characterized as one of the building blocks of logical inference and mathematical demonstration.

## 1.8: Biconditional and Logical Equivalence: The Foundation of Mathematical Definitions

You can find detailed information about when two statements are equivalent with the biconditional statement that is written using $p \leftrightarrow q$ (or $p \Leftrightarrow q$ ). The biconditional is one of the most powerful tools we have in mathematical logic, as it is used to establish logical equivalence between two statements p and q. The implication in the biconditional $p \rightarrow q$ is unidirectional and can simply be read as p implies q but biconditional says more; it is the simultaneous claim that p and q are both true at the same time, or equivalently, that p implies q and q implies p at the same time. This is the natural language phrase "if and only if" (abbreviated as "iff" in the jargon of mathematical writing) which is used to pin down precise definitions and

characterizations as well as the necessary and sufficient conditions all over mathematics. Only with a truth-functional interpretation will the biconditional $p \leftrightarrow q$ be true exactly when p and q have the same truth value (either both true or both false) and false when p and q have different truth values. This is an example of the behavior captured in its truth table, where the biconditional is only "true" in two of the four cases where p and q have truth values: if both are "true" or both are "false." In more formal logic terms, the biconditional is the conjunction of two implications saying both that p implies q and that q implies p, or equivalently in terms of disjunction either both statements are true or both are false. The negation of a biconditional gives an exclusive disjunction (XOR): $\neg(p \leftrightarrow q) \equiv (p \vee q) \wedge \neg(p \wedge q) \equiv p \oplus q$, stating that one — and not both — of the statements is true. These relationships illustrate the key role of the biconditional in capturing different kinds of logical relationships between statements. The biconditional has some important algebraic properties. Like conjunction and disjunction it is commutative ($p \leftrightarrow q \equiv q \leftrightarrow p$), as one would expect from the symmetrical nature of logical equivalence. It is also associative ($(p \leftrightarrow q) \leftrightarrow r \equiv p \leftrightarrow (q \leftrightarrow r)$), thus allowing for unambiguous expressions of mutual equivalence among multiple statements. The biconditional is reflexive ($p \leftrightarrow p \equiv T$), in other words, every statement is equivalent to itself, and it has a variant of the transitive property: if $p \leftrightarrow q$ and $q \leftrightarrow r$, then also $p \leftrightarrow r$, thus forming an equivalence relation in the logical sense and partitioning the space of statements into equivalence classes of propositions that are interchangeable in a logical context. In mathematical practice, the biconditional plays its most important role in definitions, where it interconnects the definientia (the concept being defined) with the definiens (the conditions which characterize the concept). For example, the biconditional relation "A triangle is equilateral if and only if all sides of the triangle have equal length" defines a biconditional property of being an equilateral triangle. This also allows us to create statements that in a sense compel the definition to serve as both necessary and sufficient conditions, which speaks to how we use definitions in our logic. Mathematical definitions are highly logical, and this structure nearly removes ambiguity because it specifically dictates a condition for when a concept does apply. In addition to formal definitions, biconditionals also arise in

mathematical theorems that say that different conditions or characterizations are equivalent. Theorems of the type "The following statements are equivalent... presents a list of conditions, claiming that each follows logically from every other. These results are especially dear in mathematics because they provide various insights into the same object, linking its intuitive description with its more formal or abstract formulations. One example is the following considered one of the central statements in linear algebra “The following are equivalent for a square matrix A: (1) A invertible; (2) det(A) $\neq$ 0; (3) Ax = 0 has only the trivial solution; (4) The rows of A form a linearly independent set...” gives distinct but logically equivalent perspectives on matrix invertibility. In proving such equivalence theorems, it is common to show a series of implications (usually in a circular fashion), i.e., condition (1) implies condition (2) implies condition (3).. implies condition (1). In proofs in mathematics, biconditionals may sometimes appear as the strengthened form of conditionals originally conjectured. However, not only the result $p \rightarrow q \rightarrow p \leftrightarrow q$ is stronger than the previous ones, but also the progression from $p \dashrightarrow q$ to the equivalence gives a precise and good characterization of the mathematical result. The biconditional is also fundamental to logical equivalence and has a central role in the logical equivalency of formulas or expressions. Two logical formulas $\varphi$ and $\psi$ are said to be logically equivalent, denoted $\varphi \equiv \psi$, if and only if $\varphi \leftrightarrow \psi$ is a tautology (that is, true under every possible assignment of truth values to the atomic statements they contain). The concept of logical equivalence allows us to transform and simplify complex logical expressions in a manner analogous to how algebraic equivalence allows us to manipulate mathematical equations. Some important examples of logical equivalence include De Morgan's laws ($\neg(p \wedge q) \equiv \neg p \vee \neg q$ and $\neg(p \vee q) \equiv \neg p \wedge \neg q$), the law of contraposition ($p \rightarrow q \equiv \neg q \rightarrow \neg p$) and distributive properties ($(p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$). The formal study of such equivalences is a rather large field of mathematical logic, and provides some tools for logical simplification, and analysis. In higher-level logical contexts, biconditionals relate to quantifiers to represent uniqueness statements (as in defining the meaning of a function). To claim that there exists a unique element satisfying property P(x), we say $(\exists x)\,(P(x) \wedge (\forall y)(P(y) \leftrightarrow y = x))$ for

example. In a like manner, setting f(x)=y if and only if XYZ allows us to specify exactly what makes f return the value y for the input x. This can be interpreted in set theory, where biconditionals define criteria for membership in a set: $x \in \{y \mid P(y)\}$ if and only if P(x). These instances, along with others, show that the biconditional is ubiquitous in the formal framework of mathematics — it furnishes the logical basis for sharp definitions, characterizations, and unique specifications. Biconditionals have applications in many fields including mathematics and science. In computer science, biconditionals show up in logical equivalence checking in circuit design and in specification languages in software verification. In mathematical modeling, they assist in formulating precise conditions that define phenomena of interest. This reduces the demand for visualizable intuitions and concepts to make sense of mathematics, particularly in formal verification and theorem proving where optimally exhaustively, reasoning with logical equivalences makes simple dynamic programming of formula and transforming complex logical formulas valid. The idea of necessary and sufficient conditions, expressed through biconditionals, organizes scientific explanation, whether the discipline is physics, economics, or medicine. Biconditionals have philosophical implications relating to the nature of definition, mathematical truth, and conceptual analysis.

## Check Your Progress

1. What is a set? Explain with an example.
   .....
   .....
   .....
   .....
   .....
   .....

2. Differentiate between finite and infinite sets with suitable examples.
   .....
   .....
   .....
   .....

.....

.....

## 1.9: Summary

This unit introduced the fundamental concept of sets, which form the basis of modern mathematics and computer science. It explained various types of sets such as finite, infinite, null, and universal sets, along with methods of representing them. The unit discussed key operations like union, intersection, difference, and complement, supported by Venn diagrams to illustrate relationships between sets. Through the study of subsets, power sets, and laws of set operations, learners gained a deeper understanding of how sets structure data and logical relationships. The concepts learned here provide a strong foundation for advanced topics like relations, functions, and discrete mathematical models used in computer applications

## 1.10: Exercises

**Multiple Choice Questions**

1. Which of the following represents a *null set*?
   a) {0}
   b) { }
   c) {1, 2, 3}
   d) {a, b}
   **Answer:** b) { }
2. If A = {1, 2, 3} and B = {3, 4, 5}, then A ∪ B =
   a) {1, 2, 3, 4, 5}
   b) {3}
   c) {1, 2, 4, 5}
   d) {1, 2, 3, 5}
   **Answer:** a) {1, 2, 3, 4, 5}
3. The universal set is the set that
   a) Contains all subsets of a given set
   b) Contains all possible elements under consideration
   c) Has no elements
   d) Contains only one element
   **Answer:** b) Contains all possible elements under consideration
4. Which of the following statements is *true*?
   a) Every set is a subset of itself

b) The null set is not a subset of any set

c) $A \subset A$ for all sets A

d) The universal set has no subsets

**Answer:** a) Every set is a subset of itself

5. The Venn diagram is mainly used to represent

a) Numbers and equations

b) Set operations and relationships

c) Geometric shapes

d) Algebraic expressions

**Answer:** b) Set operations and relationships

**Descriptive Questions**

1. Define a set and explain different ways to represent it with examples.
2. Describe various types of sets and give one example for each.
3. Explain basic set operations like union, intersection, and complement.
4. What are subsets and proper subsets? Give suitable examples.
5. Discuss the applications of set theory in computer science.

## 1.11: References and Suggested Readings


- Rosen, Kenneth H. *Discrete Mathematics and Its Applications*, 8th Edition, McGraw-Hill Education, 2019.
- Lipschutz, Seymour, and Marc Lipson. *Schaum's Outline of Discrete Mathematics*, 4th Edition, McGraw-Hill Education, 2017.
- Kolman, Bernard, Robert C. Busby, and Sharon Ross. *Discrete Mathematical Structures*, 6th Edition, Pearson Education, 2018.
- Epp, Susanna S. *Discrete Mathematics with Applications*, 5th Edition, Cengage Learning, 2019.
- Tremblay, J. P., and R. Manohar. *Discrete Mathematical Structures with Applications to Computer Science*, Tata McGraw-Hill, 2001.
- Grimaldi, Ralph P. *Discrete and Combinatorial Mathematics: An Applied Introduction*, 5th Edition, Pearson, 2003.

## Block 1: Set theory, Mathematical Logic, Relation and Function

## Unit 2: Tautology, Contradiction, Logical Equivalence

**Structure**

2.1 Introduction
2.2 Learning Outcomes
2.3 Tautology, Contradiction, Logical Equivalence
2.4 Formal Definitions and Truth Table Analysis
2.5 Properties and Theoretical Foundations
2.6 Extensions to Predicate Logic and Modal Logic
2.7 Philosophical Implications and Foundational Issues
2.8 Advanced Topics and Emerging Directions
2.9 Summary
2.10 Exercises
2.11 References and Suggested Readings

## 2.1: Introduction

Mathematical logic forms the foundation of reasoning and problem-solving in computer science and mathematics. It provides formal techniques to represent, analyze, and validate logical statements systematically. This unit introduces the essential concepts of propositions, logical connectives, truth tables, and methods of inference. Learners will explore how logical reasoning enables the construction of valid arguments, proofs, and algorithms. The study of logic helps in understanding conditional and compound statements, tautologies, contradictions, and equivalences that are vital for programming and software verification. Through this unit, students gain the analytical skills required to design and evaluate logical models used in computer applications, databases, and artificial intelligence.

## 2.2: Learning Outcomes

- Define and identify propositions, logical connectives, and compound statements.
- Construct and analyze truth tables to determine the validity of logical expressions.

- Differentiate between tautology, contradiction, and contingency in propositional logic.
- Apply rules of inference and logical equivalence to derive valid conclusions.
- Represent logical statements using symbolic notation and translate them between natural and formal language.
- Demonstrate the application of mathematical logic in algorithm design, decision-making, and problem-solving in computer science.
- Evaluate the soundness and validity of logical arguments through formal reasoning.

## 2.3: Tautology, Contradiction, Logical Equivalence

**Introduction to Fundamental Logical Concepts**

Formal logic provides the language of mathematics. (Note: This special language is such that it permits mathematicians to express statements with clarity, analyze the meaning behind those statements in a systematic way, and derive conclusions with certainty.) Three key concepts in mathematical logic are the tautology, the contradiction, and the logical equivalence. These principles are the foundation of higher-level logical reasoning and underpin the abstractions behind math proofs. A tautology is a logical formula that is always true, no matter what values are assigned to the variables in the formula. A formula is tautological when it evaluates to true regardless of the truth values assigned to the variables within it. Because tautologies maintain their truth regardless of context it makes tautologies a super powerful part of mathematical reasoning. Take, for example, the law of excluded middle: "P ∨ ¬P" (either a proposition holds or its negation holds). Regardless of what we plug in for P, this formula is always true. On the other hand, a contradiction is a logical formula which is false under all interpretations. A contradiction always evaluates to false regardless of what truth values we use for its component propositions. The simplest example is the formula "P ∧ ¬P" (something and not something are true at the same time), which can never be satisfied. This is why, although it sounds contradictory, contradictions is my best friend when making logical proofs, especially when we do proof by contradiction (reduction ad absurdum) and prove something by

showing that its negation leads to an impossibility. Logical equivalence is a relation between logical formulas. For two formulas to be logically-equivalent means that if you have two formulas and under every interpretation that assigns truth values to its propositions they have the same truth values then the two formulas are logically-equivalent. Not only do logical equivalences give mathematicians other ways to write the same logical content, they also can inform structural insights that can be buried in the original formulation. For instance, the logical equivalence of "P → Q" (if P, then Q) and "¬P ∨ Q" (not-P or Q) shows that implication can be meaningfully expressed in terms of negation and disjunction alone. These three ideas — tautology, contradictory, and logical equivalence — are the foundation for propositional and predicate logic. Reasoning is the basis for the exact formulation of mathematical theories, the construction of valid arguments, verifying whether systems of axioms are consistent. We will cover formal definitions, methods of representing and exploring properties of operations (truth tables, Laws of logic), mathematical implications, and philosophical considerations as we undertake a deeper exploration of these concepts. We will also identify relationships between these logical ideas and sections of math like set theory and algebra.

## 2.4: Formal Definitions and Truth Table Analysis

**Propositional Logic:**

Let's begin our tour of the world of tautologies and contradictions and logical equivalences, with some basic building blocks of propositional logic. In propositional logic, we deal with propositional functions, which are statements that can be true or false, along with logical connectives that can combine the propositional functions into compound propositions. The basic logical connectives are negation (¬), conjunction (∧), disjunction (∨), implication (→), and biconditional (↔). In propositional logic, a well-formed formula (WFF) is formed according to specific syntactical rules. Propositional variables (P, Q, R, etc.) are the simplest WFFs, representing atomic propositions. Logical connectives are applied to simpler formulas to create more complex WFFs. For instance, if P and Q are WFFs, so are ¬P, P ∧ Q, P ∨ Q, P → Q, and P ↔ Q. In a propositional logic, the truth values of propositional variables are the building blocks for

meaning, so the semantics of a WFF is described by a truth assignment, which assigns true or false to every propositional variable. Using the truth tables for the logical connectives, the truth value of a compound formula can then be calculated recursively. E.g., P ∧ Q is true iff (↔) if both P and Q are true; P ∨ Q is true iff at least one (or in logic speak 1 of P or Q is true and so on.

**Tautologies: Universal Truths**

Technically, a tautology is a well-formed formula (WFF) that is true under every possible assignment of truth values. There are 2n such truth assignments if a formula has n different propositional variables. A formula is a tautology if and only if it is true for each of 2^n of these assignments. Take the formula P ∨ ¬P. This formula has propositional variable P, so we have 2^1 = 2 different truth assignments. P can be true or false. If P is true then ¬P is false; thus P ∨ ¬P is true. If P is false, then ¬P is true, and P ∨ ¬P is again true. This means that P ∨ ¬P is a tautology. Truth tables can be used to analyze more complex tautologies. For instance, look at the formula (P → Q) ↔ (¬P ∨ Q):

**Table 2.1: Truth Table of (P → Q) ↔ (¬P ∨ Q)**

| P | Q | P → Q | ¬P | ¬P ∨ Q | (P → Q) ↔ (¬P ∨ Q) |
|---|---|---|---|---|---|
| T | T | T | F | T | T |
| T | F | F | F | F | T |
| F | T | T | T | T | T |
| F | F | T | T | T | T |

Since the formula is true for all truth assignments, it is a tautology. This tautology captures a fundamental logical equivalence between implication and disjunction.

**Example**

Consider the proposition:

(p∨¬p)

No matter what p is (true or false), this whole statement is **always**

This is called the Law of the Excluded Middle in logic:

p∨¬p



*Fig 2.1 Tautology*

**Table 2.2: Truth Table of p∨¬p**

| $p$ | $\neg p$ | $p \vee \neg p$ |
|---|---|---|
| T | F | T |
| F | T | T |
| P | T | T |

It states: Either a statement is true, or its negation is true. Always true, hence a tautology.

**Contradictions: Universal Falsehoods**

A contradiction is a WFF, which is defined as evaluating to false under all assignment of truth values. A tautology and a contradiction are the negations of each other. The simplest contradictory is P ∧ ¬P: when P is true, ¬P is false → P ∧ ¬P is false. If P is false, then ¬P is true, but in that case, P ∧ ¬P will still be false. Therefore, P ∧ ¬P is false under any truth assignment, and it is contradiction. The Contradictory example is (P ∨ Q) ∧ ¬P ∧ ¬Q; again we will analyze truth table in this formula, it was found that this formula will always be false:

**Table 2.3: Truth Table of (P ∨ Q) ∧¬P ∧¬Q**

| **P** | **Q** | **P ∨ Q** | **¬P** | **¬Q** | **¬P ∧¬Q** | **(P ∨ Q) ∧¬P ∧¬Q** |
|---|---|---|---|---|---|---|
| T | T | T | F | F | F | F |
| T | F | T | F | T | F | F |
| F | T | T | T | F | F | F |
| F | F | F | T | T | T | F |

It can be confirmed that this formula is a contradiction because it evaluates to false under every possible truth assignment. Thus this contradiction shows that a disjunction cannot hold true when both of its disjuncts are false.

**Logical Equivalence: Structural Insights**

A WFF A is logically equivalent to a WFF B (notated A ≡ B) if two WFFs are equivalent if they have the same truth under all assignments (i.e. all interpretation). A ≡ B if and only if A ↔ B is a tautology. To

prove ¬(P ∧ Q) ≡ ¬P ∨ ¬Q (De Morgan's laws), for example, we can use a truth table:

**Table 2.4: Truth Table of ¬(P ∧ Q) ≡ ¬P ∨ ¬Q**

| P | Q | P ∧ Q | ¬(P ∧ Q) | ¬P | ¬Q | ¬P ∨¬Q |
|---|---|---|---|---|---|---|
| T | T | T | F | F | F | F |
| T | F | F | T | F | T | T |
| F | T | F | T | T | F | T |
| F | F | F | T | T | T | T |

Since all possible assignments of truth values yield the same result for ¬(P ∧ Q) and ¬P ∨ ¬Q, they are logically equivalent. Other important logical equivalences include:

- Double Negation: ¬¬P ≡ P
- Commutatively: P ∧ Q ≡ Q ∧ P; P ∨ Q ≡ Q ∨ P
- Associatively: (P ∧ Q) ∧ R ≡ P ∧ (Q ∧ R); (P ∨ Q) ∨ R ≡ P ∨ (Q ∨ R)
- Distributive: P ∧ (Q ∨ R) ≡ (P ∧ Q) ∨ (P ∧ R); P ∨ (Q ∧ R) ≡ (P ∨ Q) ∧ (P ∨ R)
- De Morgan's Laws: ¬(P ∧ Q) ≡¬P ∨¬Q; ¬(P ∨ Q) ≡¬P ∧¬Q
- Implication: P → Q ≡ ¬P ∨ Q
- Contra positive: P → Q ≡ ¬Q → ¬P

This guild like explanations really helps to reason about how logical equivalences works in mid you.

## 2.5: Properties and Theoretical Foundations

**Algebraic Structure of Propositional Logic**

Tautology, contradiction, and logical equivalence are closely associated with the algebraic structure of propositional logic. Sure enough: the collection of all WFFs of propositional logic, considered up to logical equivalence, is a Boolean algebra, a mathematical structure that lies at the heart of connections from set theory and topology, to digital theater set design. In this algebraic setting, tautologies correspond to the top element (denoted 1), contradictions correspond to the bottom element (denoted 0), and logical equivalence corresponds to equality. Complements, interactions and unions in logical connectives correspond respectively to ¬, ∧ and ∨. This abstract view gives a useful machinery for understanding and transforming logical formulas. For instance, one of the rules in

Boolean algebra is the duality principle, which says that if we replace ∧ with ∨ and tautologies with contradictions in any theorem, we get another one which is also correct. De Morgan's laws and numerous other logical equivalences echo this duality.

**Normal Forms and Canonical Representations**

In propositional logic, every WFF emerges logically equivalent to formulas in different forms of normal forms which served standardized forms helpful for analytical and practical usages. A formula is in conjunctive normal form (CNF) if it is the conjunction of disjunctions of literals, and a literal is either an propositional variable or its negation. E.g. (P ∨ ¬Q) ∧ (¬P ∨ R) is in CNF. All of these logical equivalences are in fact based on the fact that every WFF can be converted to a logically equivalent formula in CNF. In a parallel fashion, a formula is in disjunctive normal form (DNF) if it consists of a disjunction over conjunctions over literals. E.g. (P ∧ ¬Q) ∨ (¬P ∧ R) is in DNF. Once more, every WFF is equivalent to a logically equivalent formula in DNF. These normal forms have significant theoretical and practical consequences:

• A formula is a tautology if and only if its disjunctive normal form (DNF) has its clauses span for all possible conjunctive clauses (or, equivalently, that its conjunctive normal form (CNF) reduces to a single literal).

• A formula is unsatisfiable if and only if its DNF is integral empty disjunctive clauses (or equivalently if its CNF is unsatisfiable).

• Two formulas are logically equivalent iff they have the same canonical DNF or CNF forms.

**Decision Procedures and Computational Complexity**

This is known as the satisfiability problem (SAT): a formula is satisfiable (SAT) if at least one truth assignment makes it true and unsatisfiable (UNSAT) if no truth assignment makes it true (i.e. a contradiction). A formula is tautologous iff its negation is unsatisfiable. For propositional logic, the problem of satisfiability is decidable: that is, there is an algorithm that can determine, for an arbitrary WFF, whether it is satisfiable. The simplest method is to build a truth table and test all truth assignments. But this means we need to evaluate 2^n assignments where n is the number of propositional variables in the formula, which is computationally infeasible for large n. Since then, more efficient algorithms, e.g., the

Davis-Putnam Logemann Loveland (DPLL) algorithm and modern SAT solvers based on conflict-driven clause learning have been devised. However, the SAT problem is NP-complete, which means that no polynomial time algorithm for solving it is known (or believed to exist) in the general case. The implications of this computational complexity are far-reaching, impacting fields such as artificial intelligence, formal verification, and cryptography. Research on efficient decision procedures for identifying tautologies, contradictions, and logical equivalences is still ongoing.

**Logical Consequence and Entailment**

Tautology, contradiction, and logical equivalence are related to the concept of logical consequence or entailment, indicated by the symbol ⊨. We say that a set of formulas Γ logically entails a formula A (Γ ⊨ A) if and only if every truth assignment making all formulas in Γ true also makes A true. The following are some of the major relationships between these concepts:

• ⟦ A is a tautology⟧ iff ⊨ A (i.e., A is entailed by the empty set of premises).

• A formula A is a contradiction iff A ⊨ B for any formula B (from a contradiction follows anything).

• A and B are logically equivalent (A ⇔ B) if and only if A ⊨ B and B ⊨ A Analyzing the second point we can see that our statement (2) is derivation from (1) and simply means that if we know that A implies B we can actually think about ¬A as A does not hold we can say I have Derived ¬A (1) implies (2)

They form the semantic underpinning of more formal deductive systems, in which a proof (typically noted with the symbol ⊢) corresponds to a syntactic derivation with respect to a (possibly infinite) set of rules. One of the fundamental results of mathematical logic is the soundness and completeness theorem, which states that for a certain deductive system, we have A⊢B if and only if A ⊨ B, bridging the semantic notion of entailment with the syntactic notion of proof.

**Applications in Mathematical Reasoning and Proof Techniques**

**Tautologies as Logical Laws**

In mathematics, tautologies function as the laws or principles (§ 10.) of logic that dictate what valid reasoning is. They are statements that

will be true regardless of the situation and in which context you use it, without destroying the validity of an argument. Examples of such basic tautologies, which are logical laws, are:

- Law of Excluded Middle: $P \vee \neg P$
- Law of Non-Contradiction: $\neg(P \wedge \neg P)$
- Law of Double Negation: $P \leftrightarrow \neg\neg P$
- Modus Ponens: $(P \wedge (P \rightarrow Q)) \rightarrow Q$
- Modus Tollens: $(\neg Q \wedge (P \rightarrow Q)) \rightarrow \neg P$
- Hypothetical Syllogism: $((P \rightarrow Q) \wedge (Q \rightarrow R)) \rightarrow (P \rightarrow R)$
- Disjunctive Syllogism: $((P \vee Q) \wedge \neg P) \rightarrow Q$

The basis of formal deductive systems consists of these tautologies which serve as the elements upon which valid mathematical proof can be built. This is also known as a "formal proof" in which you can justify every step either by applying a logical law (tautology) or citing a theorem or axiom that you've proven before. For instance, let's prove the statement "If n is odd, then $n^2$ is odd." Let P be the statement "n is odd" and Q the statement "$n^2$ is odd." We need to show $P \rightarrow Q$ and we might do so as follows:

1. If n is odd, then $n = 2k + 1$ for some integer k. (Definition)
2. If $n = 2k + 1$, then $n^2 = (2k + 1)^2 = 4k^2 + 4k + 1 = 2(2k^2 + 2k) + 1$. (Algebraic manipulation)
3. If $n^2 = 2m + 1$ for some integer m, then $n^2$ is odd. (Definition)
4. Therefore, if n is odd, then $n^2$ is odd. (Hypothetical Syllogism)

Each step in this proof relies on logical laws (tautologies) that ensure the validity of the reasoning.

**Contradictions and Proof by Contradiction**

A common method in mathematics is proof by contradiction (reductio ad absurdum). The basic idea is to start by assuming the negation of whatever you want to prove and then show how that leads to a contradiction. Which leads us to the conclusion that since contradictions are universally false, the original assumption must be wrong, and subsequently the statement to be proved must therefore be true? The logic behind proof by contradiction derives from the tautology $((P \rightarrow Q) \wedge (P \rightarrow \neg Q)) \rightarrow \neg P$; that is, if we can demonstrate that some hypothesis P both leads to a Q and also to $\neg Q$ then P must be false (because this combination cannot be true). For example, consider proving that $\sqrt{2}$ is irrational. We proceed by contradiction:

1. Assume $\sqrt{2}$ is rational. (Assumption for contradiction)

2. If $\sqrt{2}$ is rational, then $\sqrt{2} = a/b$ for some co prime integers a and b, where $b \neq 0$. (Definition of rational number)
3. If $\sqrt{2} = a/b$, then $2 = a^2/b^2$, thus $a^2 = 2b^2$. (Algebraic manipulation)
4. If $a^2 = 2b^2$, then $a^2$ is even, and therefore a is even. (Properties of even numbers)
5. If a is even, then $a = 2c$ for some integer c. (Definition of even number)
6. If $a = 2c$, then $a^2 = 4c^2$, thus $2b^2 = 4c^2$, and therefore $b^2 = 2c^2$. (Substitution)
7. If $b^2 = 2c^2$, then $b^2$ is even, and therefore b is even. (Properties of even numbers)
8. If both a and b are even, then they have a common factor of 2. (Definition of evenness)
9. But this contradicts the assumption that a and b are coprime. (Contradiction)
10. Therefore, $\sqrt{2}$ is irrational. (Proof by contradiction)

This shows that by identifying a contradiction we can eliminate an assumption and we can thus prove the veracity of its opposite.

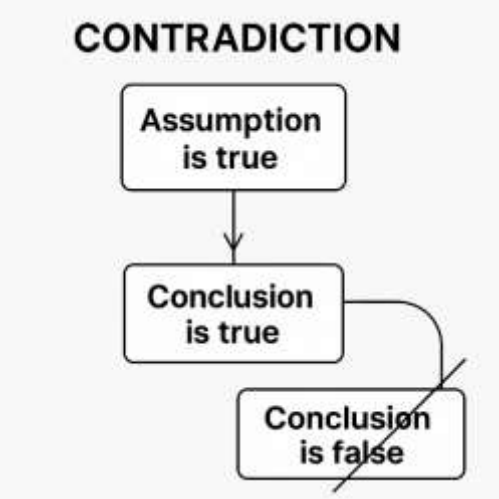



Fig: 2.2 Contradiction

**Logical Equivalence in Simplification and Transformation**

Logical equivalences help mathematicians to provide the most simplified/meaningful representation of complex validities. This kind of substitution of sub expressions for logically equivalent alternatives can serve to clarify the logical structure of a statement, or expose connections that were not immediately obvious. For instance, a logical

statement in the form of "If it is not raining then I will go for a walk, but if I do not go for a walk then it is raining." Let P be “It is raining” and Q be “I will go for a walk.” Formally speaking, this can be expressed as $(\neg P \rightarrow Q) \land (\neg Q \rightarrow P)$. We can condense this expression using logical equivalences: Using $P \rightarrow Q \equiv \neg P \lor Q(\neg P \rightarrow Q) \land (\neg Q \rightarrow P) \equiv (P \lor Q) \land (Q \lor P)$ (Commutatively of $\lor$) $\equiv (P \lor Q) \land (P \lor Q)$ (Idempotence of $\land$) $\equiv P \lor Q$ Likewise, logical equivalences are of utmost importance in mathematics as we need to manipulate theorems or definitions to some other formulations. For example, the epsilon-delta, limits, and sequence definitions of continuity of a function at a point all say the same thing. All three formulations are logically equivalent, and mathematicians are free to use whichever form suits a particular context.

**Propositional Logic in Computer Science and Engineering**

In computer science and engineering, the concepts of tautology, contradiction and logical equivalence are useful in formal verification, digital circuit design, and automated theorem proving, among other things. In formal verification, one wants to show through purely mathematical means that a system (like a computer program or an electronic circuit) satisfies certain properties or specifications. This is a logic problem: Given that S captures the behavior of the system and that P is the property we desire, we need to check that $S \rightarrow P$ is a tautology. Automated approaches for tackling such verification problem are model checking and satisfiability solvers. It makes perfect sense to describe digital circuitry in logical expressions (for example, cannot be more naturally expressed). A tautology corresponds to a circuit that is always 1 (true), regardless of the inputs; a contradiction corresponds to a circuit that is always 0 (false); and logical equivalence corresponds to two circuits that has the same input-output behavior. It is from this Boolean algebra and logical equivalences that many tools can be used to such ends to minimize and maximize circuits logic which conforms to their logic types. In automated theorem proving, automated computer programs attempt to discover proofs of instances of mathematical theorems automatically. If the negation of target theorem is converted into CNF, the resolution rule can be applied to derive a contradiction, and this is the approach of resolution-based theorem proves. If a contradiction is derived, then the initial theorem therefore is proven

true. This automating of logical reasoning has transformed entire fields as diverse as mathematics and artificial intelligence.

## 2.6: Extensions to Predicate Logic and Modal Logic

**Quantifiers and Validity in Predicate Logic**

Propositional logic gives the background on things like tautologies, contradictions, and logical equivalents, but many mathematical statements need the extra power of predicate logic. Predicate logic adds quantifiers (∀ for "for all" and ∃ for "there exists") and predicates (relations between objects) to propositional logic. In predicate logic, these notions generalize to validity, satisfiability and logical equivalence with respect to all interpretations (not just truth-assignments). An interpretation in predicate logic assigns a domain of discourse, interprets predicate symbols as relations on the domain, interprets function symbols as functions on the domain, and assigns domain elements to constant symbols. In predicate logic, a formula is valid (like a tautology) if all interpretations satisfy it. For example, $\forall x\ (P(x) \rightarrow P(x))$ is valid because it is true for any set of objects in the domain and for any interpretation of the predicate P. If it is false under all interpretations, then a formula is unsatisfiable (similarly, a contradiction). As an example, $\exists x(P(x) \wedge \neg P(x))$ is unsatisfiable since there is no element that satisfies P(x) and ¬P(x) at the same time. Two formulas are said to be logically equivalent if they have the same truth value for all possible interpretations. In particular, $\neg\forall x\ P(x) \sim \exists x\ \neg P(x)$ (one of the De Morgan laws for quantifiers). In predicate logic, the analysis of validity is exponentially more complicated than propositional logic. In propositional logic, determining whether a given formula is a tautology is decidable (i.e., there is an algorithm that can answer this question), but for predicate logic, the validity problem is in general undividable (as shown by Church's theorem).

**Modal Logic and Possible Worlds Semantics**

Modal logic is an extension of classical logic it includes operators that qualify the truth of a proposition relative to a certain modality (e.g. necessity,□ or possibility,◇). These modalities enable a finer logical treatment, especially of statements about what must be true, might be true, is known, is believed, or should be the case. The definitions of tautology, contradiction, and logical equivalence that we have seen in propositional logic are generalized in modal logic,

taking these additional operators into account. A common semantic framework is the powerful possible worlds semantics (or Kripke semantics), where, rather than evaluating formulas in isolation in a single world, one considers a collection of related possible worlds. In modal logic, a formula is a logical necessity (the modal analogue of a tautology) if there is no possible world in which it is false in all possible models. For example, $\Box(P \vee \neg P)$, which claims that the law of excluded middle is necessarily true, is a logical necessity. A formula is logically impossible (akin to an inconsistency) if it is false in every possible world in all possible models. In modal logic, two formulas are said to be logically equivalent when they have the same truth value in all possible worlds in all possible models. For instance, it is logically equivalent that $\Box P$ iff $\neg\Diamond\neg P$ (P is necessary only if it is not possible that $\neg P$). Different systems of modal logic (such as S4, S5, or K) are distinguished by their axioms governing the behavior of the modal operators, reflecting different ideas about the meaning of necessity and possibility.

## 2.7: Philosophical Implications and Foundational Issues

**Logical Truths and the Nature of Mathematics**

This remarkable property raises deep philosophical concerns regarding the mind and mathematical truth; the nature of such statements, called tautologies, states that they can only be true because of their logical form, a unique quality. Are the truths of mathematics just ornate tautologies, as some logicist philosophers have claimed? Or do they have some alternative epistemological status? The logicist program originated with Got lob Frege and Bertrand Russell, who sought to reduce mathematics to logic by proving that all mathematical concepts could be defined in terms of logical concepts, and that all mathematical theorems could be derived from logical axioms. In this view, mathematical truths would, indeed, be complex tautologies. But it was not without significant challenges. Kurt Gödel's incompleteness theorems showed that any consistent formal system capable of expressing elementary arithmetic contains statements that are true but cannot be proved in the system. This means that there are true statements in mathematics that cannot be proved via a formal approach, this implies that "mathematical truth" is more than what can be described by formal logical systems. There is a

philosophical debate here too: Platonism (mathematical objects exist independent of human minds), formalism (mathematics amounts to manipulating meaningless symbols according to formal rules), and intuitionism (mathematical objects are mental constructions), structuralism (mathematics studies abstract structures). In addition to this technical comedy, tautology in a sense answers the question by establishing the constraint of what is effectively mathematical knowledge, the line separating the logical and extra-logical components of our formal system (David Farber and Steven G. Krantz, "Tautology").

**The Law of Non-Contradiction and Para consistent Logics**

The law of non-contradiction (formally ¬ (P ∧ ¬P)) has been a fundamental tenet of Western philosophy since Aristotle. It states that a statement cannot be true and false at the same time. This is the principle behind the idea of non-contradiction in classical logic and justification for proof by contradiction in mathematics. Nevertheless, certain philosophical and mathematical situations have inspired the formation of par consistent logics, which relax the explosive principle of classical logic (that a contradiction implies any arbitrary statement). Such logics admit contradictions and avoid early trivialization. Non-classical logic is an umbrella term for a slew of alternatives to classical propositional and predicate logic, including temporal and modal logic, deontic logic, relevance logic and intuitions logic. For instance, the Liar Paradox ("This statement is false") appears to yield the result that the statement is simultaneously true and false, a contradiction. Para consistent logics offer systems for reasoning about these sorts of paradoxes without descending into total inconsistency. Para consistent logics raise some baffling considerations about what logical laws are and whether the law of non-contradiction applies universally. Most mathematicians still work within classical logic, but studying other logical systems deepens our comprehension of the nature of mathematics and the prospects for logical reasoning.

**Logical Equivalence and the Problem of Translation**

Logical equivalence also leads to philosophical questions about the nature of meaning, identity, and translation. Do two logically equivalent formulas express the same proposition? Or do they have different propositions that just happen to have the same truth

conditions? Applying W.V. Quince's idea of the indeterminacy of translation, you could tell your friend that as there are multiple equally valid translations of a statement from one language to another, there exists no fact of the matter about any statements which translation is "correct." This indeterminacy raises questions about the assumption that logically equivalent statements express the same meaning. In practice, logically equivalent statements of theorems or definitions give different perspectives or suggest different generalizations. The Jordan curve theorem, for example, can be stated in terms of point-set topology or in terms of homology theory. These formulations are logically equivalent, but emphasize different aspects of the result and point to different mathematical traditions. Logical equivalence bears philosophical implications for the individuation of propositions, the nature of syntax-semantics connections, and the nature of mathematical grasp. It highlights the intricate relationship between formal logical form and the substance of mathematical theories.

## 2.8: Advanced Topics and Emerging Directions

**Many-Valued Logics and Fuzzy Logic**

Classical logic is bivalent: it allows only two truth values, true and false. Under this definition, tautology, contradiction, and logical equivalence are straightforward. Nonetheless, not only applications in mathematics, computer science, and linguistics motivated the development of many-valued logics, which permit the introduction of other truth values, besides true and false. In a multi-valued logic a tautology is generalised to one formula that always gets the designated truth value (typically the maximum value), irrespective of the truth values of the proposition forming her. A formula is a contradiction if and only if it always has the non-designated truth value (commonly the minimal value). It should hang on the definition of logical equivalence, having the same truth values under all relevant assignments. Fuzzy logic takes this one step further by specifying that truth values may be any real number in [0, 1], indicating the degrees of truth. In fuzzy logic, a tautology is a formula that has degree of truth 1 under all interpretations, a contradiction has degree 0 under all interpretations Logical equivalence means the having the same degree of truth under all interpretations. Applications

include artificial intelligence, control theory, and semantics of vague natural language statements. They're mathematical structures that serve as general tools for reasoning with uncertainty and incomplete truth, generalizing logical systems beyond the rigidities of classical true/false.

**Quantum Logic and Non-Classical Logics**

Many intuitions about physical reality and many more found in classical logic were put on bushwhacked notice with the advent of quantum mechanics in the early 20th century. However, quantum logic was developed as a logical framework that better accommodates our reasoning about quantum phenomena. In quantum logic, the distributive law of classical logic ($P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$) does not in general hold. This breakdown illustrates the non-classical behavior in quantum systems, which mainly represents complementarily: some properties of quantum systems cannot be measured simultaneously with arbitrary precision. Tautology, contradiction, and logical-equivalence need to be rethought in this non-classical setting. A quantum tautology means a formula that evaluates to "true" under any quantum-mechanical interpretation. In parallel, quantum logical equivalence would ensnare the equal conduct of logically correlated quantum propositions. Various other non-classical logics have been devised for different purposes, such as intuitionist logic (which does not accept the law of excluded middle), relevance logic (which requires some relevant connection between the premises and the conclusion in an argument), and linear logic (which treats logical formulas as resources that can be consumed when they are used). We have built a whole diversity of logical systems, each with unique views on tautology, contradiction, and logical equivalence.

**Computational Aspects and Automated Reasoning**

Finding tautologies, contradictions, and logical equivalences is a fundamental task in automated reasoning systems, such as theorem provers, symbolic computation software, and artificial intelligence systems. By using efficient algorithms for these tasks computers can contribute to mathematical research, the verification of software and hardware, and knowledge representation. PAR(allelism, across architectures and domains)Recent advances in SAT solving (i.e.,

Notes

deciding the satisfiability of propositional formulas) have resulted in significant progress for the solution of large-scale problems.

## Check Your Progress

1. Define propositional logic. Explain the role of logical connectives in forming compound propositions.

   .....
   .....
   .....
   .....
   .....
   .....

2. What are tautology and contradiction? Illustrate both with examples using truth tables.

   .....
   .....
   .....
   .....
   .....
   .....

## 2.9: Summary

This unit explored the fundamental principles of mathematical logic, which serve as the backbone of analytical reasoning and computation. It covered the structure of propositions, logical connectives, and the construction of truth tables to evaluate the validity of statements. Learners studied different forms of reasoning such as implication, equivalence, tautology, and contradiction, along with methods of proof and inference. The unit emphasized how logical rules form the basis for decision-making processes in programming, database queries, and algorithm design. By mastering these logical techniques, students develop the capability to represent complex problems formally and verify the correctness of solutions in computer-based systems.

## 2.10: Exercises

**Multiple Choice Questions**

1. Which of the following is a *proposition*?
   a) What time is it?
   b) 5 + 3 = 8
   c) Come here!
   d) How are you?
   Answer: b) 5 + 3 = 8
2. The negation of the statement "It is raining" is:
   a) It is not raining
   b) It will rain
   c) It has rained
   d) None of these
   Answer: a) It is not raining
3. If $p$ and $q$ are propositions, then $p \wedge q$ is true when:
   a) Either $p$ or $q$ is true
   b) Both $p$ and $q$ are true
   c) Both $p$ and $q$ are false
   d) Only $p$ is true
   Answer: b) Both $p$ and $q$ are true
4. A statement that is always true, regardless of the truth values of its variables, is called a:
   a) Contradiction
   b) Contingency
   c) Tautology
   d) Proposition
   Answer: c) Tautology
5. The logical connective used to represent "if...then..." is:
   a) $\wedge$
   b) $\vee$
   c) $\rightarrow$
   d) $\leftrightarrow$
   Answer: c) $\rightarrow$

**Descriptive Questions**

1. Define a proposition and explain different types of propositions with examples.
2. Discuss the role of logical connectives in forming compound statements. Illustrate using truth tables.

3. Explain the concept of tautology, contradiction, and contingency with suitable examples.
4. What are rules of inference? Describe their importance in logical reasoning and proofs.
5. Discuss the significance of mathematical logic in computer science and programming.

## 2.11 References and Suggested Readings


- Rosen, Kenneth H. *Discrete Mathematics and Its Applications*, 8th Edition, McGraw-Hill Education, 2019.
- Epp, Susanna S. *Discrete Mathematics with Applications*, 5th Edition, Cengage Learning, 2019.
- Enderton, Herbert B. *A Mathematical Introduction to Logic*, 2nd Edition, Academic Press, 2001.
- Grimaldi, Ralph P. *Discrete and Combinatorial Mathematics: An Applied Introduction*, 5th Edition, Pearson, 2003.
- Mendelson, Elliott. *Introduction to Mathematical Logic*, 6th Edition, CRC Press, 2015.

## Block 1: Set theory, Mathematical Logic, Relation and Function

## Unit 3: Relation

**Structure**

3.1 Introduction
3.2 Learning Outcomes
3.3 Relation, Types of Binary Relation, Equivalence Relation
3.4 Types of Binary Relations and Their Applications
3.5 Equivalence Relations and Partitions
3.6 Operations on Relations and Relation Algebras
3.7 Advanced Topics and Modern Developments in Relation Theory
3.8 Summary
3.9 Exercises
3.10 References and Suggested Readings

## 3.1: Introduction

Relations and functions are fundamental mathematical concepts that establish connections between elements of different sets. This unit focuses on understanding how ordered pairs, relations, and functions describe the association between data elements — an idea central to computer science and database design. Learners will explore various types of relations such as reflexive, symmetric, transitive, and equivalence relations, as well as representations using matrices and digraphs. The unit also introduces the concept of functions, their types, and composition, which are essential for modeling computational processes and algorithmic mappings. By studying relations and functions, students develop the ability to represent data dependencies, build logical structures, and apply mathematical reasoning in diverse computing contexts.

## 3.2: Learning Outcome

- Define and represent relations between sets using ordered pairs, matrices, and digraphs.
- Classify relations as reflexive, symmetric, transitive, and equivalence relations.

- Understand and illustrate functions as special types of relations.
- Differentiate between one-to-one, onto, and many-to-one functions.
- Apply composition and inverse of functions to problem-solving.
- Represent relationships in real-world computing applications, such as database systems and mapping structures.
- Develop mathematical reasoning to model relationships in data structures and algorithms.

## 3.3: Relation, Types of Binary Relation, Equivalence Relation

### Introduction to Relations

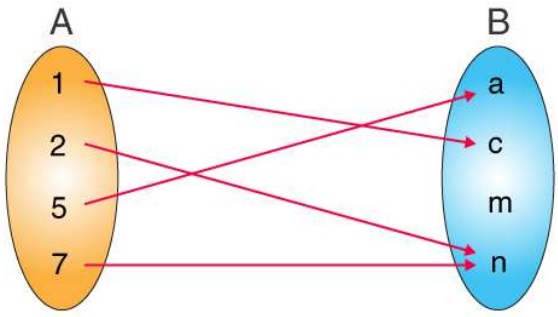



Fig: 3.1 Relation

Based are sets, the relations are the basic structures of the mathematics, formalizing the connections between the objects of the sets. They feature in the mix across disparate branches of mathematics and computer science, from abstract algebra to database theory. In formal terms, a relation R from a set A to a set B is a subset of the set of ordered pairs $A \times B$, which is the set of pairs (a, b) where $a \in A$ and $b \in B$; if a relation R contains (a, b), we also say that a is R related to b, and we commonly write aRb to denote it. Relations are a fundamental tool for abstraction and generalization of mathematical concepts and are used to describe, in a more general way than through functions, orderings, equivalences, and entities that combine various previous relations. Today, the study of binary relations on one set ($A = B$) is the foundation for many of the algebraic and structural properties investigated by the field of discrete mathematics. This relation can be represented as a directed graph, a matrix, or a set of ordered pairs, each of which has its uses in different cases. SUPER

(Relations) — The study of relations enables mathematicians to abstract phenomena at hand and free the core focuses of the problem from unnecessary details, leading to the development of concepts that may not arise otherwise. We will discuss different types of relations, their properties, and how they can be used to model real-world scenarios and solve mathematical problems. Relations play a major role in the intuitive go between connection and the brains of structures of formal mathematics and are therefore essential for building a solid groundwork in higher mathematics. Relations; In computer science, relations are the theoretical basis for relational databases (which store data in tabular format). The connection is an instance of how the mathematical theory of relations has practical significance beyond its theoretical interest. Relation theory was developed in relation to early set theory dating back to the work of mathematicians like Georg Cantor, Richard Dedekind and Ernst Schroder from 1874 to the late 19th century. But it was over the course of the 20th century that the formal theory of relations developed significantly, largely credited to Alfred Tarski, Alfred North Whitehead and Bertrand Russell and their magnum opus "Principia Mathematical." relation theory is an area of active research, and it is used in several modern fields such as artificial intelligence, machine learning, and network theory.

**Formal Definition and Representation of Relations**

A binary relation R from set A to set B is defined as a subset of the Cartesian product A × B, and if A = B the term binary relation can refer to a binary relation on A. There are multiple ways of representing relations, each of which has its own advantages in a given context. The simplest form is a collection of ordered pairs of the form {(a, b) : a ∈ A, b ∈ B, aRb} Alternatively, a relation on a finite set can be represented using a matrix, where the rows of the matrix are the elements in the first set, columns are the elements in a second set, and the entries indicate whether the elements are related (1) or not (0). For relations defined over a single set, digraphs (directed graphs) provide an intuitive pictorial representation: vertices can be thought of as elements from the set and directed edges represent (ordered) pairs that are related. Different aspects of the relation are emphasized and different types of operations and analyses facilitated by the respective representation. Matrix representations

allow common computational operations (e.g., relation composition) to be performed via matrix multipli- cation, while direct visualizations (like graphs) give instant insights about properties of interest (e.g., connectivity, cycles, etc.). In addition, the relations can also be expressed as predicates or formulas of a logic that identifies under what conditions elements are in a relation. P(x, y) is an example of a predicate since the "less than" relation on integers can be defined by this predicate: it would be true if $x < y$. This connects relation theory with logic and allows logical machinery to help us analyze relations. Representation depends on the application and studied properties. Generation of this description will naturally depend on the relation in question: for small relations, it might be most natural to enumerate the ordered pairs, whereas computation-intensive enumerations for much larger relations may be better expressed as algebraic expressions or logical formulas defined by membership. Adjacency lists or sparse matrices may be used to store relations in computer implementations to save space and improve time complexity. More advanced data structures like binary decision diagrams (BDDs) can compress relationships into a format that enables fast operations. Set theory underlies a more formal approach to relations that provides a basis for many useful structures in mathematics, including functions (special cases of relations), partial orders and equivalence relations, not to mention algebraic structures such as groups and rings. What this means is that by abstracting away the details, mathematicians can find similarities between mathematical objects that may have looked completely unrelated, and then create a unified theory to describe all of these once separate objects. Relations exercise a theory direction that fits nicely into both practical and theoretical bridges; for example, design guides for the relational model of databases stem from direct mapping of relations as an entity-relationship diagram into relational algebra, while problems in computational complexity often revolve around relations over input and output.

**Properties of Binary Relations**

All binary relations have some important properties that describe their behavior and structure. For all $a \in A$, a relation R on A is reflexive iff $(a, a) \in R$. A relation R is reflexive if no element is related to itself, so $(a, a) \notin R$ for all $a \in A$, and it is symmetric if,

whenever (a, b) ∈ R, we also have (b, a) ∈ R (so if a is related to b, then b is related to a): and, conversely, a relation is ant symmetric if, whenever (a, b) ∈ R and (b, a) ∈ R, then a = b (so no two distinct elements can be related in both directions). A relation is asymmetric if it is both reflexive and ant symmetric, which means that (if (a, b) ∈ R) => (b, a) ∉ R; Transitivity, is another key property: R is transitive if if (a, b) ∈ R ∧ (b, c) ∈ R then (a, c) ∈ R and means the relation "carries through" chains of related elements) All these properties are not mutually exclusive and relations can have different combinations of them. For example, a relation could be reflexive and symmetric but not transitive. A relation has certain properties which influence the behaviour of the relation mathematically, and what kind of structures it can model. Relations that are reflexive, symmetric, and transitive are called equivalence relations, and they partition sets into disjoint equivalence classes. Reflexive, ant symmetric, and transitive relations form partial orders that can represent hierarchical structures. As relations can be represented in various ways, each property could be verified using different methods. Reflexivity thus equates to having all 1s on the main diagonal in matrix representation; and symmetry in the main diagonal of the matrix. Transitivity can be checked via matrix multiplication: let M be the relation matrix; then R is transitive iff $M^2 \subseteq M$ (where $M^2$ is obtained via Boolean matrix multiplication). Reflexivity, symmetry, and transitivity can also be interpreted in terms of the representation of relations as graphs. Reflexivity corresponds to a self-loop on every vertex, symmetry to edges being bidirectional, and transitivity to the condition that if there exists a path from a to b and a path from b to c, then an edge from a to c also exists. Graphs also have operations such as complementation, inversion, and composition over relations that create new relations, impacting these properties. Order theory, which focuses on the structure of partial and total orders, and algebraic structures such as lattices and Boolean algebras defined in relation to certain types of relations centers around relation properties. In the field of computer science, knowledge of relation properties plays an important role in database design (functional dependencies), algorithm analysis (recurrence relations), and formal verification (transition relations in state machines).

## 3.4: Types of Binary Relations and Their Applications

Binary Relations: Binary relations can be categorized in many types according to their properties, these categories are having various applications in all over the mathematics and computer science. Reflexive, symmetric, and transitive relations are known as equivalence relations, one of the most important relations in terms of abstract algebra, topology, and number theory. They divide sets into non-overlapping equivalence classes, allowing the construction of quotient structures and abstract models. An example of equivalence relation is the congruence modulo n, which is a equivalence relation that is the basis of modular arithmetic, which is important in fields like coding theory and cryptography. A partial order is an ordering that is reflexive, ant symmetric, and transitive; partial orders are useful for modeling hierarchical orders, such as taxonomies, organizational charts, and dependencies between tasks in project management. If every pair of elements is comparable, we instead have a total order, such as the standard ordering of real numbers. An example of a preorder (both reflexive and transitive relation) is used in preference modeling and category theory; An example of tolerance relations (both reflexive and symmetric) is used in approximate reasoning and fuzzy logic." Functional relations are mathematical functions — every item within the domain corresponds to exactly one item within the co domain. Functions can be categorized as injective (one-to-one), subjective (onto), or bijective (both), and play an important role in solving equations and showing size equivalence between sets. Circular relations display cyclical patterns, where elements can be arranged into cycles — think rotate groups or periodic phenomena in time series analysis. Cyclic ordering relations extend "between's" to ternary relations over elements: they are used for example when representing points on a circle, or for scheduling cyclic events. Proximity relations measure the closeness or similarity between two objects and play an integral role in both cluster analysis, pattern recognition, as well as in many machine learning algorithms. Dominance relations compare objects with respect to multiple criteria and have been widely used in multicriteria decision making and game theory. In graph theory, binary relations form their foundation too: the edges are the relationships between the vertices. Specific relations and hierarchies (file systems, company structure, etc) are represented

by special graph structures, such as trees (acyclic connected graphs). Relations in a relational database mean tables and foreign keys create relationships between entities. The operations of relational algebra (such as selection, projection, join) are performed upon these relations to query and transform data. Relations can be used to represent connections between people (or nodes) in network models in sociology and communications, which allow for the analysis of information flow, the patterns of influence within a social network, and commModuley structures. Causal relations in statistics and empirical sciences describe cause-effect relationships that are core to prediction and explanation. Understanding these different patterns of relations helps mathematicians and computer scientists find models that suit the phenomena and exploit relation properties to solve various problems efficiently. The classification of relations is a powerful framework for recognizing similarities across domains and applying known mathematical techniques to situations they were not designed for.

## 3.5: Equivalence Relations and Partitions

Equivalence relations have a particularly nice place in mathematics, because they are intimately related to set partitions. A relation R on a set A is an equivalence relation if and only if it is reflexive (aRa for all $a \in A$), symmetric (if aRb, then bRa) and transitive (if aRb and bRc, then aRc). These properties guarantee that an equivalence relation partitions the set into disjoint subsets, known as equivalence classes, each class containing elements related to each other. For any element $a \in A$, its equivalence class [a] is the set of all elements equivalent to a, given by $[a] \equiv \{x \in A \mid xRa\}$. A foundational theorem of set theory shows that, for any set a, an equivalence relation on that set induces a partition of that set into disjoint equivalence classes, with a unique equivalence relation deriving from a partition. The one-to-one correspondence between equivalence relations on a set and partitions of that set is particularly useful in abstract mathematics and its applications. By grouping elements according to their relevant properties and ignoring their irrelevant differences, equivalence relations allow for abstraction. Equivalence relations allow us to "mod out," leading to structures called quotients that can provide insight and streamline many complex problems. For example,

congruence modulo n (where a ≡ b (mod n) iff n | (a − b)) decomposes integers into residue classes 0, 1,..., n-1 mod n, and is used in modular arithmetic with applications in computer algorithms and cryptography and number theory. Equivalence relations arise in abstract algebra as congruence's in groups, rings, and other algebraic structures, enabling the formation of quotient structures such as quotient groups and quotient rings. In this framework, quotient structures allow for the retention of key algebraic characteristics while simplifying the structure, enabling the classification and analysis of various algebraic systems. In topology, we use equivalence relations to obtain quotient spaces like the torus that is formed by identifying opposite edges of a rectangle. Translations, rotations, and reflections between sets of geometrical figures yield equivalence relations in which equivalent figures are considered the same under the transformation. In computer science, equivalence relations represent state equivalence in the minimization of finite automata, streamlining computational resources and preserving the recognized language. Hash tables are data structures that utilize equivalence relations to organize data efficiently by using hash functions. However this last definition leaves us with some computational aspects, namely: canonical elements representatives for equivalence classes and fast decision if two elements belong to the same class. Union-find data structures solve this problem using near-constant time operations, making them useful to network connected components, minimum spanning trees, and some computational geometry algorithms. Fuzzy equivalence relations and tolerance relations, which generalize the notion of strict equivalence to capture similarity or approximate equality, are also covered by equivalence relation theory. Where these generalizations can be applied are pattern recognition, clustering algorithms and artificial intelligence. This is essentially what equivalence theory represents a bridge between all extremes of mathematics, from the most esoteric theoretical constructs, down to the most concrete application oriented computational problem, all relating through a fundamental power of relation theory that allows to highlight essential structures across heterogeneous contexts.

## 3.6: Operations on Relations and Relation Algebras

Relations can be combined and transformed through various operations, forming algebraic structures called relation algebras. These operations allow relations to be manipulated and analyzed systematically. The basic operations are unions ($R \cup S$), intersections ($R \cap S$), and complements ($\bar{R}$) of relations, which obey the same principles known as the theory of sets as relations are set of tulles. If R is a relation, its inverse (or converse) is denoted $R^{-1}$, and consists of all the pairs (b, a) such that $(a, b) \in R$ — this simply reverses the direction of R. If R and S are both relations, the composition of R and S, denoted $R \circ S$, is the set of all pairs (a, c) such that there exists (b) where $(a, b) \in R$ and $(b, c) \in S$, which generalizes function composition to arbitrary relations. Those operations obey a number of algebraic laws, including associatively of composition and distributive of composition over union, and they serve as the foundation of relation algebra. Rhetorical question marks reward any good child who put all of one letter before the next: Not only do you can compute on pure sets, but there are additional operations like the relative product (R|S), where you multiply set R by the inverse of S, the domain restriction ($E \lhd R$) of pulling the first components of R into those in a set E, or the range restriction ($R \rhd F$), which limits the second components to F. Relation algebras, which were formalized by Alfred Tarski and his students, gives an axiomatic framework with which to study relations abstractly. These algebras become Boolean algebras and have additional operations finding a reflection of the relation theory, with a connection to logic, to set theory and to abstract algebra. In the late 19th century, Augustus De Morgan and Charles Sanders Peirce developed a calculus of relations, which was later further developed by Ernst Schroder. One can solve relational equations and do reasoning with respect to relations with this symbolic calculus. This calculus can be used in automated theorem proving, program verification and artificial intelligence. Transitive closure $R^+$ creates the least transitive relation whose projection contains R, and reflexive transitive closure R* simply includes if R has the reflexivity as well. Performing these operations is essential in many applications including graph theory, parsing algorithms, and reach ability analysis in networks. In particular, relations are a central feature of database management systems, and relational algebra

(selection, projection, join, etc.) serves as the theoretical underpinnings for query languages (e.g. SQL). The performance of the database is affected by the efficiency of these operations, leading to research in query optimization and algorithm design. For instance, binary decision diagrams (BDDs) and other symbolic representations in the area of computer science make it possible to efficiently manage and manipulate enormous relations and play a primary role in applications related to model checking, constraint satisfaction problems, and in the formal verification of hardware and software systems. Dynamic relations: such relations vary in time or are modified as new information becomes available, and as such requites are special operations and representations. Temporal relation algebras are generalizations of classical relation algebra employed to describe relations involving time, which are commonly used in temporal databases, scheduling and event processing systems. Relation algebras are both expressive and syntactically rich, making them useful for a wide range of applications across mathematics and computer science, including theoretical work in category theory, as well as practical implementations such as graph analysis and query processing in databases. Relations are a powerful tool that has applications in diverse fields of mathematics; a perspective that ties the operations and entities in relation algebra to classically algebraic structures open up exciting new paradigm for interpreting the world around us.

## 3.7: Advanced Topics and Modern Developments in Relation Theory

Until present, Relation Theory is widely studied and many new extensions, generalizations, applications are being introduced across different fields. Fuzzy relations generalize classical relations by permitting intermediate degrees of relatedness between elements, usually represented by values in the interval [0,1]. These relations represent non-specific or imprecise relationships and have applications in fuzzy logic, approximate reasoning and soft computing. A fuzzy equivalence relation (known as a similarity relation) extends classical equivalence relations to express degrees of similarity, and is used in areas such as pattern recognition, clustering and image processing Formal concept analysis is a theory founded by Rudolf Wille in the 1980s and proposes a method for discovering

conceptual structures in data through binary relations between objects and attributes. These structures act as complete lattices that uncover hierarchical knowledge structures which are beneficial in knowledge discovery, data mining, and ontology engineering. Introduced by Zdzisław Pawlak, rough set theory deals with approximations over sets, defined via equivalence relations, dealing with knowledge of different accuracy. Rough set is a tool for uncertainty handling which has been applied in decision support systems, machine learning and data analytics. They generalize the mathematical notions of classical relations to relations between quantum systems for which the certainty of a relationship follows the laws of quantum probability, an example of which cannot be captured with Boolean logic. This type of relation is very important in quantum computing and quantum information theory, as it establishes a platform for the studies of entanglement, quantum measurements, and so on, i.e. quantum algorithms, and so on. However, categorical approaches to relation theory locate relations in the context of category theory, making relationships with other mathematical constructions apparent and allowing for powerful abstract reasoning. In the semantics of computer languages and type systems, this view has enabled new understandings in theoretical computer science. Inductive logic programming, relational reinforcement learning, and statistical relational learning are relation-based approaches to machine learning implemented in the area of computational intelligence. These methods utilize relational representations, allowing for the capture of complex structures in data that access patterns often fail to express in attribute-value representations. Relation theory is thus side at itself at neither in much of the universe of the habitual. Two notable examples of such formalisms are Allen's interval algebra, which formalizes qualitative relationships between time intervals and is used for temporal reasoning, and region connection calculus which formalizes qualitative relationships between spatial regions and is used for spatial reasoning (e.g., in geographic information systems, robotics, cognitive science). (Antimatroids are a closely related concept.) Infinite relations on infinite sets lead to complications that use up the machinery of advanced set theory, such as cardinals, ordinals, and the axiom of choice. These relations appear in functional analysis, topology, and the foundations of mathematics, linking relation theory

with profound issues about mathematical infinity. Relation theory's focus on algorithmic elements has risen to significance with the emergence of big data and network science. Algorithms for computing relation properties, closures, and decompositions can be useful for analyzing large-scale networks like social networks, biological interaction networks, and the World Wide Web. Continuation of the convergence of relation theory with other areas of mathematics has produced non-trivial theoretical results and real-life applications. Parts of graph theory, order theory, algebraic topology, and model theory contribute to both fields and provide new methods for the solution of difficult problems. Description: As computing systems move towards being more distributed, concurrent, and interconnected, relation theory is providing fundamental concepts for reasoning about such systems and building them thereby. Whether you take block chain technology, distributed databases, or federated learning systems, relations provide the mathematical language of constraints, dependencies, and interactions. Relation theory's enduring impact is a consequence of its capacity to extract crucial forms from disparate domains and consequently furnish explicit forms of computation.

## Check Your Progress

1. Define a relation. Explain the types of relations such as reflexive, symmetric, and transitive with examples.

   .....
   .....
   .....
   .....
   .....
   .....

2. What is an equivalence relation? Give an example and explain its properties.

   .....
   .....
   .....
   .....
   .....

## 3.8: Summary

This unit introduced the concept of relations and functions as tools to describe connections and mappings between sets. It discussed how relations can be represented through ordered pairs, matrices, and digraphs, and how properties such as reflexivity, symmetry, and transitivity define their nature. The unit also explored equivalence relations and partitions of sets. Functions were presented as a specific form of relation where each element of the domain corresponds to exactly one element of the codomain. Students learned about different types of functions, including one-to-one, onto, and inverse functions, which are essential for modeling computational operations. Understanding relations and functions builds a foundation for advanced topics like graph theory, automata, and database management systems.

## 3.9: Exercises

**Multiple Choice Questions**

1. A relation $R$ on a set $A$ is called reflexive if:
   a) Every element of $A$ is related to itself
   b) No element of $A$ is related to itself
   c) Every pair of elements is related
   d) None of these
   **Answer:** a) Every element of $A$ is related to itself
2. The composition of two functions $f$ and $g$ is denoted by:
   a) f + g
   b) f * g
   c) f ∘ g
   d) f / g
   **Answer:** c) f ∘ g
3. A function that maps every element of the domain to a unique element of the codomain is called:
   a) Onto function
   b) One-to-one function
   c) Many-to-one function
   d) Constant function
   **Answer:** b) One-to-one function

4. The inverse of a function exists only if the function is:
   a) Onto only
   b) One-to-one only
   c) One-to-one and onto
   d) None of these
   **Answer:** c) One-to-one and onto
5. Relations can be represented using:
   a) Matrices
   b) Truth tables
   c) Flowcharts
   d) Algorithms
   **Answer:** a) Matrices

**Descriptive Questions**

1. Define a relation and explain different ways of representing it with examples.
2. Distinguish between reflexive, symmetric, and transitive relations.
3. Explain equivalence relations and show how they partition a set.
4. Define a function and explain the difference between one-to-one and onto functions.
5. Discuss the importance of relations and functions in computer science applications.

## 3.10: References and Suggested Readings


- Rosen, Kenneth H. *Discrete Mathematics and Its Applications*, 8th Edition, McGraw-Hill Education, 2019.
- Kolman, Bernard, Robert C. Busby, and Sharon Ross. *Discrete Mathematical Structures*, 6th Edition, Pearson Education, 2018.
- Epp, Susanna S. *Discrete Mathematics with Applications*, 5th Edition, Cengage Learning, 2019.
- Tremblay, J. P., and R. Manohar. *Discrete Mathematical Structures with Applications to Computer Science*, Tata McGraw-Hill, 2001.
- Grimaldi, Ralph P. *Discrete and Combinatorial Mathematics: An Applied Introduction*, 5th Edition, Pearson, 2003.

## Block 1: Set theory, Mathematical Logic, Relation and Function

# Unit 4: Function

**Structure**

4.1 Introduction
4.2 Learning Outcomes
4.3 Function, Types of function
4.4 Applications of Logical Equivalence
4.5 Summary
4.6 Exercises
4.7 References and Suggested Readings

## 4.1 Introduction

This unit introduces the concept of functions as a special type of relation that defines a unique mapping between elements of two sets. Functions play a vital role in mathematical modeling, programming, and computational processes. Learners will understand how a function associates each element of a domain with exactly one element of a codomain. The unit covers various types of functions such as one-to-one, onto, into, and many-to-one functions. It also explains composite and inverse functions that are commonly used in problem-solving and data transformations. Understanding functions helps students to describe and analyze real-world relationships in computer applications such as algorithms, data structures, and databases.

## 4.2: Learning Outcome

- Define a function and describe how it differs from a general relation.
- Identify the domain, codomain, and range of a function.
- Distinguish between one-to-one, onto, into, and many-to-one functions.
- Explain the concepts of composite and inverse functions with examples.
- Represent functions using algebraic expressions, tables, and graphs.
- Apply functions to model and solve computational and logical problems.

- Relate the concept of functions to programming, algorithm design, and database systems.

## 4.3: Function, Types of Function

**Introduction to Function**

A function is a special relation between two sets such that each element of the first set (Domain) is related to exactly one element of the second set (Co-domain).

Properties

- Each element of Domain must be mapped.
- No element of Domain has more than one image.
- Different elements of Domain may map to the same element in Co-domain.

Types of Functions

- One–one (Injective): Different inputs give different outputs.
- Onto (Surjective): Every element of Co-domain is an image of some element.
- Bijective: Both One–one and Onto.

**1. Domain, Co-domain, and Range**

**Definition:**

- **Domain**: Set of all possible inputs.
- **Co-domain**: Set of all possible outputs allowed.
- **Range**: Actual outputs that come from the function.

**Example:**

f:A→B where A={1,2,3} B={a,b,c,d}

f(1)=a, f(2)=c, f(3)=b

**Domain**={1,2,3}
**Co-domain**={a,b,c,d}
**Range** = {a,c,b}

**◈ 2. One–One (Injective) Function**

**Definition:**

Every element of the domain maps to a **unique** element of the co-domain.

**Example:**

f(x)=2xfromR→R

Different x values give different outputs.
f(1)=2 =4 – no repeats.

### ◈ 3. Onto (Surjective) Function

**Definition:**

Every element of the co-domain has at least one pre-image in the domain.

**Example:**

f:{1,2,3}→{a,b,c}with
f(1)=a, f(2)=b, f(3)=c
All elements a,b,c are covered ⇒ Onto.

### ◈ 4. Bijective Function

**Definition:**

A function that is **both one–one and onto**.

**Example:**

f:{1,2,3}→{a,b,c}with
f(1)=a, f(2)=b, f(3)=c
Every input has a unique output and covers all outputs.

### ◈ 5. Identity Function

**Definition:**

Maps every element to itself.

**Example:**

f(x)=xforallxindomain.
f(1)=1, f(2)=2, f(3)=3

### ◈ 6. Constant Function

**Definition:**

Maps every element of the domain to a **single fixed value** in the co-domain.

**Example:**

f(x)=5f(x)forallxindomain.
f(1)=5, f(2)=5, f(3)=5

### ◈ 7. Composition of Functions

**Definition:**

(f∘g)(x)=f(g(x))

**Example:**

f(x)=x+1 and g(x)=2

(f∘g)(x)=f(g(x))=f(2x)=2x+1

### ◈ 8. Inverse Function

**Definition:**

Reverses the effect of a bijective function. If f:A→B is bijective, then f−1:B→A

**Example:**

f(x)=x+3 ⇒ f−1(x)=x−3

### ◈ 9. Even and Odd Functions

**Even:**f(−x)=f(x)

Example: f(x)=x2

**Odd:**f(−x)=−f(x)

Example: f(x)=x3

### 10. Real-life Function Example

**Temperature conversion:**

f(C)=95C+32

Maps Celsius to Fahrenheit.

Diagram of a Function

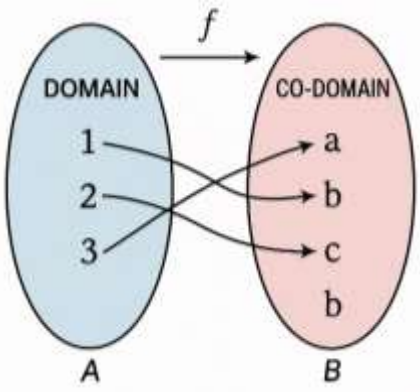



Fig: 4.1 Function

**Example 1:**

Let A= {1,2,3} and B={a,b,c,d}

Define a function f:A→B as:

F (1) = a, f (2) = c, f (3) =b

Each element of A(domain) is mapped to exactly one element of B (co-domain).
Range of f is {a,b,c} (only those elements of B that are actually used).

Therefore, f is a valid function.

**Example 2:**

**Set A:** {Apple, Banana, Mango}
**Set B:** {Red, Yellow, Green}
**Define:** A function mapping each fruit to its color:

f(Apple)=Red, f(Banana)=Yellow, f(Mango)=Green

**Range:** {Red, Yellow, Green}

**Example 3:**

**Set A:** {1,2,3,4}
**Set B:** {Odd, even}
**Define:** A function mapping each number to its type:

$F(1) = Odd, f(2) = Even, f(3) = Odd, f(4) = Even$

**Range:** {Odd, even}

**Example 4:**

**Set A (Domain):** {2,4,6}
**Set B (Co-domain):** {1,4,9,16,25,36}
**Define:** f(x)=x2

$F(2) = 4, f(4) = 16, f(6) = 36$
**Range:** {4,16,36}

## 4.4: Applications of Logical Equivalence

Logical equivalence is not only a theoretical construct but also an important practical tool. In mathematics, science, and computer applications, simplifying logical expressions allows us to reduce complex situations into manageable forms. For example, when analyzing a long conditional statement, one may replace it with an equivalent biconditional form to reveal hidden relationships.
One area where logical equivalence is particularly useful is in set theory. As we know, statements about membership in sets can be expressed in terms of logic. For instance, the identity
$(A \cup B)c = Ac \cap Bc$

is directly derived from the equivalence $\neg(p \vee q) \equiv (\neg p \wedge \neg q)$. Similarly,

$(A \cap B)c = Ac \cup Bc$

is based on $\neg(p \wedge q) \equiv (\neg p \vee \neg q)$. These transformations make it easier to prove theorems about sets and functions.

**Proof Techniques and Logical Equivalence**

Logical equivalence provides the foundation for methods of proof such as proof by contradiction and proof by contrapositive.

**Example:** Prove that if $n^2$ is even, then n is even.

**Contrapositive:** If n is odd, then $n^2$ is odd.

Let $n = 2k+1$. Then $n^2 = (2k+1)(2k+1) = 4k^2 + 4k + 1 = 2(2k^2+2k) + 1$. This is odd. Therefore, the contrapositive is true and hence the original statement is also true.

This shows how logical equivalence often allows shorter and clearer proofs.

**Applications in Computer Science**

Logical reasoning underpins many areas of computer science. Digital Circuits: Logical expressions correspond directly to circuits. The conjunction (AND), disjunction (OR), and negation (NOT) map to logic gates. More complex connectives such as exclusive OR (XOR) can be expressed using combinations of simpler ones.

**Example:** The XOR of two inputs A and B can be expressed as

$(A \vee B) \wedge \neg(A \wedge B)$.

This expression is directly translated into a digital circuit.

Database Queries: In SQL and other query languages, logical operators are used to filter data. For example:

SELECT * FROM Students WHERE Age > 18 AND Course = 'MCA';

The WHERE clause here is a direct representation of conjunction in propositional logic.

Artificial Intelligence: Many expert systems represent knowledge in the form of logical rules. Logical equivalence helps simplify the rules and improve reasoning efficiency.

**Worked Example: Simplification**

Simplify $(p \vee q) \wedge (\neg p \vee r)$.

Step 1: Apply distributive law:

$= (p \wedge \neg p) \vee (p \wedge r) \vee (q \wedge \neg p) \vee (q \wedge r)$

Step 2: Eliminate contradictions:

$p \wedge \neg p$ = False, so expression becomes:

$= (p \wedge r) \vee (q \wedge \neg p) \vee (q \wedge r)$

This is the simplified form.

**Logic in Algorithms**

In algorithm design, decision-making processes are expressed using logical statements.

**Example: Binary Search**

If the key is less than the middle element, search in the left half; otherwise, search in the right half.

Logically, this is expressed as:

(key < mid) → search left

(key > mid) → search right

By framing conditions in logical form, algorithms can be reasoned about more clearly, and their correctness can be proved systematically.

**Importance of Logical Equivalence**

Logical equivalence not only simplifies proofs and expressions but also provides the bridge between mathematics and computation. It ensures that transformations in logic preserve meaning, and therefore, results obtained after simplification remain valid. This makes it an essential tool in every branch of computer science — from digital systems to programming, and from artificial intelligence to database management.

## Check Your Progress

1. Define a function. Explain the difference between one-to-one, onto, and many-to-one functions with examples.

   .....
   .....
   .....
   .....
   .....
   .....

2. What is the inverse of a function? State the condition under which a function has an inverse.

.....
.....
.....
.....
.....
.....

## 4.5: Summary

This unit discussed functions as a special form of relation where each element of a domain corresponds to exactly one element in the codomain. It explained different types of functions including one-to-one, onto, into, and many-to-one, along with their properties and graphical representation. Learners also studied the concepts of composition and inverse of functions, which are used to combine or reverse mappings between sets. The understanding of functions forms a bridge between pure mathematics and its applications in computer science, especially in data processing, program logic, and modeling relationships between inputs and outputs in algorithms.

## 4.6: Exercises

**Multiple Choice Questions**

1. A function is said to be one-to-one if:
   a) Each element of the domain is mapped to one or more elements of the codomain
   b) Each element of the domain is mapped to a unique element of the codomain
   c) Every element of the codomain is mapped to more than one element of the domain
   d) None of the above
   Answer: b) Each element of the domain is mapped to a unique element of the codomain
2. The composition of two functions f and g is represented as:
   a) f + g
   b) f * g
   c) f ∘ g
   d) f – g
   Answer: c) f ∘ g
3. The inverse of a function exists only if the function is:
   a) Onto only
   b) One-to-one only
   c) One-to-one and onto
   d) Constant
   Answer: c) One-to-one and onto

4. The range of a function is:
   a) The set of all possible inputs
   b) The set of all possible outputs
   c) The set of all ordered pairs
   d) The union of domain and codomain
   Answer: b) The set of all possible outputs
5. If $f(x) = x^2$, then $f(3)$ equals:
   a) 3
   b) 6
   c) 9
   d) 12
   Answer: c) 9

**Descriptive Questions**

1. Define a function. Explain how it differs from a general relation.
2. Describe the types of functions with suitable examples.
3. What are composite and inverse functions? Explain their properties with examples.
4. Discuss the applications of functions in computer science and algorithm design.
5. Represent a function using a table and graph, and explain the relation between domain and range.

## 4.7: References and Suggested Reading


- Rosen, Kenneth H. Discrete Mathematics and Its Applications, 8th Edition, McGraw-Hill Education, 2019.
- Epp, Susanna S. Discrete Mathematics with Applications, 5th Edition, Cengage Learning, 2019.
- Kolman, Bernard, Busby, Robert C., and Ross, Sharon. Discrete Mathematical Structures, 6th Edition, Pearson Education, 2018.
- Grimaldi, Ralph P. Discrete and Combinatorial Mathematics: An Applied Introduction, 5th Edition, Pearson, 2003.
- Tremblay, J. P., and Manohar, R. Discrete Mathematical Structures with Applications to Computer Science, Tata McGraw-Hill, 2001.

## Block 2: Poset and Lattices

# Unit 5: Partial Order Relation

**Structure**

5.1 Introduction
5.2 Learning Outcome
5.3 Introduction to Partial Order Relations
5.4 Partial Ordered Set and Hasse Diagram Summary
5.5 Exercises
5.6 References and Suggested Reading

## 5.1: Introduction

This unit introduces the concept of partially ordered sets and lattices, which form an important part of discrete mathematics and computer science. A partially ordered set, or poset, is a set equipped with a binary relation that describes how elements can be compared in a hierarchical manner. Learners will explore how ordering relations such as reflexivity, antisymmetry, and transitivity define a poset. The unit also covers the concept of Hasse diagrams used to represent posets graphically. Lattices are special kinds of posets in which every pair of elements has a unique least upper bound and greatest lower bound. Understanding posets and lattices helps in organizing data, reasoning about hierarchies, and designing efficient algorithms in computer science applications.

## 5.2: Learning Outcome

- Identify and illustrate relations that form a poset.
- Construct and interpret Hasse diagrams for a given poset.
- Explain the concepts of least upper bound and greatest lower bound.
- Define lattices and differentiate them from general posets.
- Apply lattice properties to solve problems in algebraic structures and logic.
- Relate the concepts of posets and lattices to data organization and hierarchy modeling in computer science.

## 5.3: Introduction to Partial Order Relations

For example, mathematics helps us organize and understand relationships among elements in various sets. Partial Order Relations are one of the basic concepts in discrete mathematics and set theory and are used to define a certain order between elements. A relation is a concept to associate between the elements of one set with another set. In certain mathematical models, we need to know in what way one element/sequence is connected with another element/sequence in hierarchical Oder sequential way. Q- Partially ordered relations can be used for systematic study of such structure. These terminologies are especially important in disciplines such as computer science, decision theory, and data organization, where the ordering and hierarchy of items are critical. Definition and Properties of Partial Order Relations A partial order relation is a binary relation R on a set **S** that satisfies the following three properties:

1. **Reflexivity**: Every element is related to itself. That is, for every a∈Sa \in S, aRaaRa holds.
2. **Antisymmetry**: If aRbaRb and bRabRa, then a=ba = b.
3. **Transitivity**: If aRbaRb and bRcbRc, then aRcaRc.

The pair of a set S and a partial order relation R is called a partially ordered set (or poset), denoted as (S, R). In a total order, all elements must be comparable with each other, but unlike that a partial order allows that some elements of the set may be incomparable. Examples of Partial Order Relations

1. **Subset Inclusion (⊆\subseteq)**: Consider the power set of a set AA, which consists of all subsets of AA. The relation "subset of" (⊆\subseteq) is a partial order because it satisfies reflexivity (A⊆AA \subseteq A), ant symmetry (A⊆BA \subseteq B and B⊆AB \subseteq A implies A=BA = B), and transitivity (A⊆BA \subseteq B and B⊆CB \subseteq C implies A⊆CA \subseteq C).
2. **Divisibility (|)**: The relation "divides" on the set of natural numbers N is a partial order. If a|ba | b (i.e., a divides bb exactly), it follows the properties of a partial order.
3. **Hierarchical Structures**: In organizational charts or file directory structures, elements are arranged in a hierarchical order, which is an example of a partial order relation.

**Solved Examples**

**Example 1:** Verify if the relation R on the set S = {1, 2, 3, 4}**,** defined as aRb if and only if a divides b, forms a partial order.

**Solution:**

1. **Reflexivity**: Every number divides itself (e.g., 1∣11 | 1, 2∣22 | 2, etc.), so the relation is reflexive.
2. **Antisymmetry**: If a∣ba | b and b∣ab | a, then a=ba = b, satisfying ant symmetry.
3. **Transitivity**: If a∣ba | b and b∣cb | c, then a∣ca | c. For example, 2∣42 | 4 and 4∣84 | 8 implies 2∣82 | 8. Thus, the given relation is **a** partial order**.**

**Unsolved Problems**

1. Consider the set A = {1, 2, 3, 4, 5} with relation R defined as aRbaRb if a≤ba \leq b. Show that R is a partial order.
2. Prove that the relation of inclusion on the power set of S={a,b,c}S = \{a, b, c\} is a partial order.
3. Let S=ZS = \mathbb{Z} and define aRbaRb if and only if aa divides bb. Show that this is a partial order.

Partial order relations are a central concept in mathematical modeling and programming, as well as in the structuring of databases in which elements must be ordered or ranked. Through relations like ant symmetry, nodes of digraphs, relations, etc., we can have a better sense to solve problems in set theory, lattice theory and advanced mathematics. Studying real world problem using partial order relations will give you a rigorous framework building block of discrete mathematics and friends.

## 5.4: Partial Ordered Set and Hasse Diagram

**Partially Ordered Sets (Posets)**

A partially ordered set (poset) is a set PP together with a binary relation ≤\leq that satisfies the three properties:

1. Reflexivity: For all a∈Pa \in P, a≤aa \leq a.
2. Antisymmetry: if a \leq b and b \leq a, then a = b.
3. Transitivity: If a≤ba \leq b and b≤cb \leq c, then a≤ca \leq c.

These properties account for a partial order, as not all pairs of elements in PP are required to be comparable.

**Examples of Partially Ordered Sets**

- **Divisibility Relation**: The set {1,2,3,4,6,12}\{1, 2, 3, 4, 6, 12\} with the relation a≤ba \leq b if aa divides bb.
- **Subset Relation**: The power set of {1,2,3}\{1,2,3\} ordered by set inclusion.
- **Integer Ordering**: The set of natural numbers N with the usual ≤\leq relation.

**Hasse Diagram**

Connected Hasse diagrams are widely used in the theory of partially ordered sets (posets), where a Hasse diagram is a visual representation of a finite poset. It is a directed graph (in a simple way) that:

• O edges represent the partial order relation.

• Transitive edges are left out for clarity.

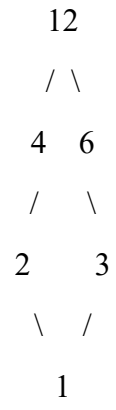
**Example Poset: Divisors of 12**

Consider the set:

S= {1,2,3,4,6,12}

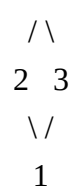
We define a partial order ≤ by divisibility:

A ≤ b if and only if a divides b.




***Example:***

Consider the posit {1, 2, 3, 6}\ {1, 2, 3, 6\} with the divisibility relation:




In this diagram:

- 11is minimal as it divides all elements.

- Since 11 divides both 22 and 33 it is placed above 11.
- 66 is at the peak as it is divisible by both 22 and 33

**4. Properties of Hasse Diagrams**

- Relation of Cover: An element aa covers bb if a>ba > b and there is no element cc such that bba > b and no element cc satisfies b
- Chains and Antichains: A chain is a totally ordered subset; an antichain consists of mutually incomparable elements.

**5. Solved Example**

**Problem:** Draw the Hasse diagram for {1,2,4,8,16}\{1, 2, 4, 8, 16\} with divisibility relation.

**Solution:**

1. Write out the divisibility relations: 1≤2≤4≤8≤161 \leq 2 \leq 4 \leq 8 \leq 16.
2. Since each element only divides the next one, the Hasse diagram is a linear chain**:**

16
|
8
|
4
|
2
|
1

**6. Unsolved Problems**

1. Construct the Hasse diagram for the power set of {a,b}\{a, b\} ordered by set inclusion.
2. Find the number of chains in the Hasse diagram of {1,2,5,10}\{1, 2, 5, 10\} under divisibility.
3. Prove that the divisibility relation on {1,3,9,27,81}\{1, 3, 9, 27, 81\} is a poset.

Another common use of partial order sets is in Hasse diagrams, which can be used to visualize the structure of hierarchies (remember lattice theory, Boolean algebra, and database design) these concepts are essential for all higher level topics in discrete mathematics.

## 5.5: Summary

This unit explained partially ordered sets as mathematical structures that describe order relations among elements of a set. It emphasized the properties of reflexivity, antisymmetry, and transitivity that characterize a poset. Learners studied how Hasse diagrams provide a clear graphical representation of the ordering relation. The unit also introduced lattices as a special form of poset where every pair of elements has a meet and join, known as the greatest lower bound and least upper bound. These ideas are widely used in logic design, database theory, information hierarchy, and formal concept analysis. Mastery of posets and lattices provides a theoretical foundation for reasoning about order, structure, and dependency in computer applications.

## Check Your Progress

1. Define a partial order relation. Explain the properties of reflexivity, antisymmetry, and transitivity with examples.

   .....
   .....
   .....
   .....
   .....
   .....

2. What is a poset? Illustrate with an example and explain its use in representing ordered structures.

   .....
   .....
   .....
   .....
   .....
   .....

## 5.6: Exercises

**Multiple Choice Questions**

1. A relation R on set A is a partial order if it is:
   a) Reflexive, symmetric, transitive
   b) Reflexive, antisymmetric, transitive

c) Irreflexive, antisymmetric, transitive
d) None of these
Answer: b) Reflexive, antisymmetric, transitive

2. The graphical representation of a poset is called:
   a) Tree diagram
   b) Venn diagram
   c) Hasse diagram
   d) Flowchart
   Answer: c) Hasse diagram
3. In a lattice, every pair of elements has:
   a) Only a least upper bound
   b) Only a greatest lower bound
   c) Both a least upper bound and a greatest lower bound
   d) Neither upper nor lower bound
   Answer: c) Both a least upper bound and a greatest lower bound
4. Which of the following is true for all lattices?
   a) They must be finite
   b) They must be total orders
   c) Meet and join exist for every pair of elements
   d) Every element is comparable to every other
   Answer: c) Meet and join exist for every pair of elements
5. Posets and lattices are widely used in:
   a) Numerical analysis
   b) Computer graphics
   c) Information ordering and logic design
   d) File compression
   Answer: c) Information ordering and logic design

**Descriptive Questions**

1. Define a partially ordered set and explain its properties with examples.
2. What is a Hasse diagram? Construct one for a given finite poset.
3. Differentiate between a poset and a lattice with suitable examples.
4. Explain the concepts of least upper bound and greatest lower bound in lattices.

5. Discuss the applications of posets and lattices in computer science and data organization

## 5.7: References and Suggested Reading


- Rosen, Kenneth H. Discrete Mathematics and Its Applications, 8th Edition, McGraw-Hill Education, 2019.
- Kolman, Bernard, Busby, Robert C., and Ross, Sharon. Discrete Mathematical Structures, 6th Edition, Pearson Education, 2018.
- Epp, Susanna S. Discrete Mathematics with Applications, 5th Edition, Cengage Learning, 2019.
- Grimaldi, Ralph P. Discrete and Combinatorial Mathematics: An Applied Introduction, 5th Edition, Pearson, 2003.

## Block 2: Posets and Lattices

# Unit 6: Lattice

**Structure**

6.1 Introduction
6.2 Learning Outcome
6.3 Lattice, Sub-Lattices, Well Ordered Set, Complete Lattice
6.4 Sub-Lattices and Their Properties
6.5: Complete Lattices and Their Properties
6.6 Advanced Concepts
6.7 Summary
6.8 Exercises
6.9 References and Suggested Readings

## 6.1: Introduction

This unit explains the concept of lattices as special kinds of partially ordered sets that play a fundamental role in discrete mathematics and computer science. A lattice is a poset in which every pair of elements has a least upper bound (join) and a greatest lower bound (meet). Learners will understand the structural properties of lattices and how they can be used to model hierarchies, data classifications, and logical relationships. The unit also discusses algebraic lattices, distributive and complemented lattices, and their applications in logic design, database theory, and knowledge representation. Understanding lattices helps students develop mathematical reasoning for organizing information and analyzing relationships within structured systems.

## 6.2: Learning Outcome

After studying this unit, learners will be able to:

- Define a lattice and identify its key properties.
- Differentiate between general posets and lattices.
- Explain the operations of meet and join with suitable examples.
- Describe distributive and complemented lattices and their characteristics.
- Construct and interpret lattices using Hasse diagrams.
- Apply lattice concepts in logic circuits, algebraic systems, and computer science applications.

- Analyze real-world hierarchical structures using lattice theory.

## 6.3: Lattice, Sub-Lattices, Well Ordered Set, Complete Lattice

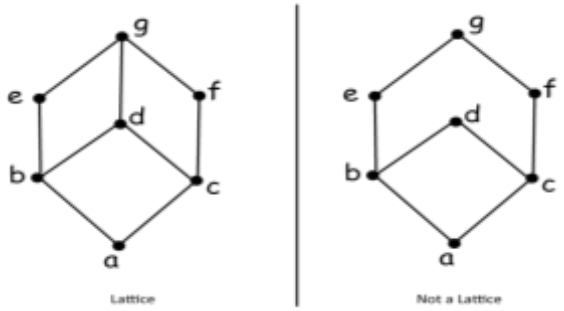



Fig: 6.1 Lattice

It offers a framework for examining the relationships between elements of a poset. A poset (P, ≤) is a set P with a binary relation ≤ that is reflexive (a ≤ a), ant symmetric (a ≤ b and b ≤ a −⟶ a = b) and transitive (a ≤ b and b ≤ c −⟶ a ≤ c). A lattice is a poset where every two elements have a least upper bound (supremum or join) and greatest lower bound (infimum or meet) ∧. The join of two elements a and b, written a ∨ b, is the least upper bound of a and b. The meet of a and b, written a ∧ b, is the greatest lower bound of a and b. A lattice can be represented visually with Hasse diagrams, which are a graphical representation of posets. We usually draw a Hasse diagram as dots for the elements, and lines for the relation ≤. When a ≤ b, b is above a, and we connect them with a line. The set of positive integers ordered by divisibility is a simple lattice. The meet of two integers a and b is their LCM, and the join is their GCD. For example, an order you might consider is the power set of a set (lifting using ordered inclusion). The join of two subsets A and B you have is their union (A ∪ B), while the meet you have is their intersection (A ∩ B). Lattices fulfill multiple essential characteristics, such as idempotent (a ∨ a = a and a ∧ a = a), commutative (a ∨ b = b ∨ a and a ∧ b = b ∧ a), associative (a ∨ (b ∨ c) = (a ∨ b) ∨ c and a ∧ (b ∧ c) = (a ∧ b) ∧ c), and absorption (a ∨ (a ∧ b) = a and a ∧ (a ∨ b) = a). This means that lattices are useful for studying ordered structures and how they relate to each other.

## 6.4: Sub-Lattices and Their Properties

A sub-lattice of a lattice (L, ∨, ∧) is a subset S of L that is also a lattice with respect to the same operations ∨ and ∧ as those of L. Namely, S is a sub-lattice if whenever a, b are in S, we also have a∨b, a∧b∈S. This condition ensures closure under the join and meets operations of L so that S is a lattice on its own. However, that closing property is not necessarily hold, thus a subset of a lattice is a POS but not a sub-lattice. For example, the positive integers ordered by divisibility are a lattice. For this lattice, the set {2, 3, 4, 5, 6} is a subset. This is not a sub-lattice because the join of 2 and 3, which is 6, is in the set, but the meet of 4 and 6, which is 2, is also in the set, but the join of 4 and 5 which is 20, is not in the set. Idempotence, commutatively, associatively, and absorption are many properties that a sub-lattice inherits from its parent lattice. These will not, however, save all properties. As another example, if L is a distributive lattice (satisfying a ∧ (b ∨ c) = (a ∧ b) ∨ (a ∧ c) and a ∨ (b ∧ c) = (a ∨ b) ∧ (a ∨ c)), then any sub-lattice of L is also distributive. But a sub-lattice of L is also modular if L is a modular lattice (and a ∧ (b ∨ (a ∧ c)) = (a ∧ b) ∨ (a ∧ c) if a ≤ b). Sub-lattices are a fundamental concept in lattice theory, and are used to break down complex lattices into simpler ones. The use of sub-lattices can be for the purposes of identifying patterns, symmetries, or relationships between different parts of the lattice. As an example, in the lattice of subsets of a set, an sub-lattice can be all the subsets that contain a particular element, or all the subsets that have a specific cardinality. They also have a role in lattice homeomorphisms, mappings between lattices that preserve the join and meet operations. A homomorphism between two lattices L and M takes a sub-lattice of L to a sub-lattice of M.

**Well-Ordered Sets and Their Significance**

A well-ordered set is a totally ordered set (i.e. of any two elements it can be determined whether one is greater than the other) such that every non-empty subset has a least element. Ri is a well-ordering principle and this property is known as the well-ordering principle. Well-orders are foundational in set theory and have wide application in mathematics, primarily in transfinite induction and ordinal arithmetic. An example of a well-ordering is the set of natural numbers (ℕ) with its standard ordering, often denoted with the symbol ≤. Each nonempty subset of ℕ has a least element. But the normal ordering of the integers (ℤ) is not a well-ordering since the subset of

negative integers has no least element. In the same vein, the real numbers ($\mathbb{R}$) with the standard ordering is not well-ordered since the interval (0, 1) has no least element. - Every set can be well-ordered (but this requires the axiom of choice, and that is a theorem in set theory, called the well-ordering theorem). Ordinal numbers which measure the "size"/"length" of well-ordered sets are strongly related to well-ordered sets Any well-ordered set is order-preserving equivalent to exactly one ordinal number. Ordinal numbers, too, have a well ordering, and they are a transfinite ordering that continues after the naturals. Transfinite induction extends the concept of mathematical induction to well-ordered sets. It enables us to show that some statement is true for all elements of a well-ordered set by demonstrating the statement for the least element and that if the statement holds for all elements less than some element then it holds for that element as well. Well-ordered sets find applications in many branches of mathematics, including topology, analysis, and computer science. Well-ordered sets are used in topology to form transfinite sequences of open sets (or closed sets). In analysis they are used to characterize transfinite sequences of functions or sets. Well-ordered sets are used in algorithms and design of data structures in computer science, specifically in termination proofs.

## 6.5: Complete Lattices and Their Properties

A complete lattice is a special type of poset in which all subsets, finite or infinite, have a supermom (least upper bound) and an infimum (greatest lower bound). This is a stronger condition than a lattice, which only requires that finite pairs of elements have a join and meet. Complete lattices play a central role in several fields of mathematics, such as topology, analysis, and domain theory. Let’s state a result on complete lattices: The power set of any set, partially ordered by inclusion, is a complete lattice. The join of any collection of subsets is their union, and the meet their intersection. Another example is the set of real numbers with standard ordering (augmented with positive and negative infinity), which is also a complete lattice. The join of any collection of reals is their supermom, and the meet is their infimum. All of those properties still hold for complete lattices: idempotence, commutatively, associatively, and absorption. They also satisfy the infinite distributive laws $a \wedge (\vee S) = \vee\{a \wedge s : s \in S\}$ and a

$\vee (\wedge S) = \wedge\{a \vee s : s \in S\}$ for any subset S of the lattice. Topologically complete lattices are used, for example, to define closure operators on sets as well as to study the lattice of open sets or closed sets. In analysis, they are used to define monotone functions and study fixed points of functions. If a function is monotone on a complete lattice, then it is also guaranteed to have a least fixed point and a greatest defined fixed point in that lattice, a result known as the Knaster Tarski theorem. Literally hundreds of papers have been published on complete lattices, while the ordering of the initial segments of a certain kind of directed set - known as a domain - has formed the basis of domain theory, an entire branch of mathematics. Complete lattices are used in domain theory to describe the spaces of computations and to analyse programming languages semantics.

**Solved Examples and Applications**

**Example 1:** Power Set Lattice X = {a, b, c}. The power set P(X) = {∅, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}} is a lattice under inclusion.

• Join: {a, b} ∨ {b, c}

Of course, we will continue with the rest of the paragraphs while adding solved examples, applications of concepts and unsolved questions as well.

**Solved Examples and Applications (Continued)**

**Example 1**: Power Set Lattice We take X = {a, b, c}. So the power set P(X) = {∅, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}} is a lattice ordered by inclusion.

• Join: {a, b} ∨ {b, c} = {a, b, c} (Union of the sets)

Meet: {a, b} ∧ {b, c} = {b} (Intersection of the sets)

• P(X) is also a complete lattice as every subset of P(X) has a join and meet.

**Example 2:** Divisibility Lattice Let L = {1, 2, 3, 4, 6, 12} with divisibility.

• Join: 3 ∨ 4 = 12 (LCM of 3 and 4)

• Meet: 6 ∧ 4 = 2 (GCD of 6 and 4)

• Make the Hasse-diagram of this lattice.

**Left: Set Proposition 3: (**Well-Ordered Set) Prove that every finite totally ordered set is well-ordered.

S is a finite totally ordered set.

S is countable whenever S is finite, and I-S is also finite.

• Induction, any finite subset has a least element.

• Thus, S is well-ordered.

**Applications:**

- **Computer Science:** Lattices play a role in data analysis, formal concept analysis and in the design of data structures. Domain theory, which serves as the mathematical basis for programming language semantics, uses complete lattices.
- **Database Theory:** In database theory, lattices can be used to reason about data dependencies and integrity constraints.
- **Formal Logic:** Lattices can be used to model the logical connectives and study the structure of logical theories.
- **Topology:** Closure operators are defined in complete lattices and the lattice of open sets or closed sets is studied.

**Unsolved Problems and Advanced Concepts**

**Unsolved Problems:**

1. **Congruence Lattice Problem:** Given a finite lattice L, decide if L is isomorphic to the congruence lattice of some algebra. This issue is solvable in general, yet undividable.
2. **Free Lattice Problem:** Give the structure of free lattices. The word problem is known to be solvable for free lattices, though the structure of such lattices is an active research area.
3. **Sub-lattice Embedding Problem:** For two finite lattices L and M, decide whether L is embeddable into M as a sub-lattice: Embedding of finite lattices is proven as NP-Complete. Here the problem is to create a substructure.
4. **Well-quasi-ordering Problem**: Investigate well-quasi-orderings, which generalize well-orderings. Study the relationship between well-quasi-orderings and termination arguments of algorithms.
5. **I remember an Intro** to Something course which had a section on Fixed-point theorems reduced to imply maximum and minimum elements in a complete lattice, which was applicable to stuff like program semantics, so If you can find something similar throw it at them.

## 6.6: Advanced Concepts:

• **Distributive Lattices:** Lattices where the distributive laws hold ($a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$ and $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$). Full

linear lattices are distributive : a lattice is a full linear lattice if and only if it is a distributive lattice and {{0}, {1}} is its only linear ideal.

• **Modular lattices:** lattices satisfying the modular law (a ∧ (b ∨ (a ∧ c)) = (a ∧ b) ∨ (a ∧ c), whenever a ≤ b). Distributive lattices are modular.

• **Heyting Algebras**: Lattices with an implication operation which generalizes intuitionist logic.

• **Scott Domains:** Domains are from domain theory, but Scott domains are a complete lattices. They can be algebraic functions or continuous.

• **Galois Connections:** Pairs of monotone functions between postsets adjoint to each other. They are used develop studied relationships between other ordered structures.

**Further Explorations and Conclusion**

Lattices, sub-lattices, well-ordered sets, complete lattices ultimate analysis Such numerical concepts are crucial in disciplines ranging from pure math to computer science, and provide us with powerful tools for exploring achieved order and its interactions. The real-world impact of these ideas can be observed in database systems, formal logic, and computer programming, to name a few. Mathematical advancements often lead to new applications and discoveries in these fields. This also implies the existence of different classes of lattices, such as: distributive, modular, Boolean, etc. The well-ordered set itself and the concept that as the most general form of the ordered set definitely can go for research. Complete lattices can also be studied in accordance to their domains and their role inside the domain theory, topology, and fix-point theory, and their various applications in computer science and others. The ideas on this section serve a good basis on lattice introduction and other topics related. Understanding lattices through solved examples, unsolved problems, and advanced topics can help students to appreciate the elegance and power of this area of study and its applications. As their mathematical adventure moves forward, the ideas and tools presented in this Module will become invaluable assets when analyzing or solving countless problems in not only mathematics but in a number of scientific fields as well.

**Introduction to Lattices and Distributive**

Lattices are a kind of algebraic structure on a set that introduce a structure to study partially ordered sets (posets) through the existence of least upper bounds (joins) and greatest lower bounds (meets) for every pair of elements. In many applications, we are interested in such lattices with extra properties that make them very useful. For example, distributive is one of these properties. Lattice (L, ∨, ∧) is distributive if for all a,b,c∈L: a∧(b∨c)=(a∧b)∨(a∧c) and a∨(b∧c)=(a∨b)∧(a∨c). These laws are similar to the distributive laws in plain algebra. Nevertheless, not all lattices are distributive. An example of a distributive lattice can be given by the power set of any set, ordered by inclusion. The join ∨ is replaced with the union operation ( ∪ ) and the meet ∧ is replaced with intersection operation ( ∩ ). They are easy to check because the distributive laws hold for set operations. As a second example, consider the (semi-)lattice of positive integers ordered by divisibility: here, join is given by taking LCM, whereas meet takes GCD. This is also a distributive lattice. Distributive lattices and their Hasse diagrams if a lattice contains a sub-lattice isomorphic to the pentagon lattice (N5) or the diamond lattice (M3), it is not distributive (in terms of Hasse diagram). In such a scenario, the element above 0 becomes less than that of the element above 1 (as it must be by construction), thus simulating a upside-down lattice. Distributive lattices have many nice properties, leading to their importance in various fields of mathematics and computer science. They are used, e.g., in Boolean algebra, formal logic, and data dependency theory in databases.

**Complemented Lattices and Their Properties**

A complement is another property a lattice can have. A lattice (L, ∨, ∧) with least element 0 and a greatest element 1 is called complemented if for all elements a in L there exists an element a' in L such that a ∨ a' = 1 and a ∧ a' = 0. The element a’ is called the complement of a In the complemented lattice every element has at least one complement. Nonetheless, complements need not be unique unless the lattice is also distributive. An example of a complemented lattice is the power set of a set ordered by inclusion. Complement of a Set: For the subset A, the complement of A is denoted by A', A ∪ A' = U (Universal set) and A ∩ A' = ∅ The lattice of divisors of a square-free integer ordered by divisibility is another

example. The divisor d's complement is the n/d quotient, with n being the square-free integer. In a complemented lattice, the bottom least element 0 is the complement of the top greatest element 1, and vice versa. Complementation is an involution, (a)' = (a')"{"n"=1, "t"=1} but, in the general case, we have (a ∨ b)' = a' ∧ b' and (a ∧ b)' = a' ∨ b' only when the lattice is also distributive. They appear in many branches such as Boolean algebra, switching circuit, quantum logic, etc. They offer a way of CNS representing and CNS manipulating logical operations CNS and CNS studying the structure CNS of complementary systems.

## Check Your Progress

1. Define a lattice. Explain how a lattice can be represented as a partially ordered set.

   .....
   .....
   .....
   .....
   .....

2. Discuss the properties of lattices such as commutativity, associativity, and absorption laws with examples.

   .....
   .....
   .....
   .....
   .....
   .....

## 6.7: Summary

This unit explored lattices as an extension of partially ordered sets, focusing on their structural and algebraic properties. It discussed how every pair of elements in a lattice possesses a meet and join, forming a complete and well-structured order. Learners studied the differences between distributive and complemented lattices and their role in simplifying logical expressions. The unit emphasized how lattices provide a theoretical foundation for Boolean algebra, decision-making systems, and data organization in computing. By understanding

lattices, students gain analytical tools to model order relations and hierarchies that occur in programming, logic design, and database theory.

## 6.8: Exercises

**Multiple Choice Questions**

1. A lattice is a poset in which every pair of elements has:
   a) Only a least upper bound
   b) Only a greatest lower bound
   c) Both a least upper bound and a greatest lower bound
   d) Neither upper nor lower bound
   Answer: c) Both a least upper bound and a greatest lower bound
2. The meet and join operations in a lattice are represented by:
   a) Addition and subtraction
   b) Union and intersection
   c) ∧ and ∨
   d) + and –
   Answer: c) ∧ and ∨
3. A lattice is distributive if:
   a) Meet and join satisfy the distributive laws
   b) Each element is comparable
   c) It is both bounded and complemented
   d) It has no upper bound
   Answer: a) Meet and join satisfy the distributive laws
4. In a complemented lattice, every element has:
   a) No complement
   b) One complement only
   c) At least one complement
   d) Exactly two complements
   Answer: c) At least one complement
5. Lattices are widely used in:
   a) Image processing
   b) Logical circuit design and database theory
   c) Random number generation
   d) Probability and statistics
   Answer: b) Logical circuit design and database theory

**Descriptive Questions**

1. Define lattice and explain how it differs from a general poset.
2. Describe the meet and join operations in a lattice with examples.
3. Explain the properties of distributive and complemented lattices.
4. Construct a lattice for a given set using the divisibility relation.
5. Discuss the importance of lattice theory in logic design and computer applications.

## 6.9: References and Suggested Readings


- Kolman, Bernard, Busby, Robert C., and Ross, Sharon. Discrete Mathematical Structures, 6th Edition, Pearson Education, 2018.
- Grimaldi, Ralph P. Discrete and Combinatorial Mathematics: An Applied Introduction, 5th Edition, Pearson, 2003.
- Epp, Susanna S. Discrete Mathematics with Applications, 5th Edition, Cengage Learning, 2019.
- Tremblay, J. P., and Manohar, R. Discrete Mathematical Structures with Applications to Computer Science, Tata McGraw-Hill, 2001.

# Unit 7: Distributive and Complemented lattice

**Structure**

7.1 Introduction
7.2 Learning Outcome
7.3 Distributive and Complemented Lattices
7.4 Advanced Concepts and Further Explorations
7.5 Summary
7.6 Exercises
7.7 References and Suggested Readings

## 7.1: Introduction

This unit discusses distributive and complemented lattices, which are special types of lattices having additional structural properties. A distributive lattice satisfies the distributive laws of meet and join operations, ensuring that the order of combining elements does not affect the result. A complemented lattice, on the other hand, provides each element with a complement that helps in simplifying logical expressions. These concepts are closely related to Boolean algebra and have direct applications in computer science, particularly in digital logic, switching circuits, and database theory. By studying distributive and complemented lattices, learners gain a deeper understanding of algebraic structures that support logical reasoning and data manipulation in computational systems.

## 7.2: Learning Outcomes

After studying this unit, learners will be able to:

- Define distributive and complemented lattices and describe their properties.
- Explain the distributive laws for meet and join operations with examples.
- Identify complemented elements in a lattice and determine their complements.
- Differentiate between general lattices, distributive lattices, and Boolean lattices.
- Apply lattice properties to simplify logical and algebraic expressions.

- Relate distributive and complemented lattices to Boolean algebra and logic circuit design.
- Use lattice theory concepts in analyzing order relations and data organization.

## 7.3: Distributive and Complemented Lattices

A distributive complemented lattice is referred to as a Boolean algebra. {Boolean algebras are one of the core algebraic structures that underpin logic, computer science, and much else besides.} A Boolean algebra (B, ∨, ∧, ', 0, 1) is a lattice that satisfies the following:

1. It distributive: a ∧ (b ∨ c) = (a ∧ b) ∨ (a ∧ c) and a ∨ (b ∧ c) = (a ∨ b) ∧ (a ∨ c).
2. It is complemented: For each a, there exists a complement a' with a ∨ a' = 1 and a ∧ a' = 0.
3. It has a minimum element, 0, and a maximum element, 1.

Complements are unique in a boolean algebra, and the De Morgan's laws hold in a boolean algebra: (a ∨ b)' = a' ∧ b' and (a ∧ b)' = a' ∨ b'. The collection of all subsets of any set, with operations of union, intersection, and complement, is a Boolean algebra. Boolean algebra: A Boolean algebra (also spelled Boolean algebra) is a mathematical structure composed of a set with two operations and their relations, the most common being the set of logical propositions with logical OR, AND NOT operations. It is also used for designing logic circuits, simplifying solutions, representing and manipulating logical expressions. They offer a strong foundation for reasoning about systems based on binary states and for solving problems related to logical constraints. Stone's representation theorem says every Boolean algebra is isomorphic to a field of sets. ZFC is the widely accepted formalism for set theory, while the setup for the theorem utilizes the algebra of logic..

**Solved Examples and Applications**

**Example 1:** Power Set Lattice X = {a, b}. Therefore the 100082 20:52:53,941 --> 20:52:56,529 the power set P(X) = {∅, {a}, {b}, {a, b}} 20:52:56,531 --> 20:52:59,614 is a Boolean algebra.

• Join: {a} ∨ {b} = {a, b} (Union)

• Meete: {a} ∧ {b} = ∅ (Intersection)

• Complement: {a}' = {b}

• Validate the distributive laws and De Morgan's laws.

**Example 2**: Divisors of 30Let f be the lattice of the divisors of 30 ordered by divisibility : { 1, 2, 3, 5, 6, 10, 15, 30 }.

• Join: 6 ∨ 10 = 30 (LCM)

• Meet: 6 ∧ 10 = 2 (GCD)

• Complement: 6 ′ = 5 (where 6 × 5 = 30 and GCD(6, 5) = 1)

• This lattice is a Boolean algebra.

**Example 3:** Logical Propositions Let us consider the Boolean algebra of logical propositions, with operations OR, AND NOT.

• (P ∧ Q) ∨ R = (P ∨ R) ∧ (Q ∨ R) (Distributive law)

• ¬(P ∨ Q) ≡ ¬P ∧ ¬Q (De Morgan's law)

**2.3.2 Applications:**

• **Digital Circuits:** Boolean algebra is utilized to create and evaluate digital circuits like logic gates and flip-flops.

• **Database Queries:** Boolean algebra is utilized to structure and refine database queries that involve logical conditions.

• **Formal verification:** The correctness of hardware and software systems is verified using Boolean algebra.

• **Descriptions of sets:** The term Boolean algebras structures provide the basis for set theory and the study of collection of sets.

**Unsolved Problems in Distributive and Complemented Lattices**

**Unsolved Problems:**

1. **Congruence Lattice Problem for Distributive Lattices:** If L is a finite distributive lattice, is L isomorphic to the congruence lattice of some algebra? This is easier than a full congruence lattice computation problem but still not easy to do.
2. **Free Distributive Lattices:** We have introduced modules and distributive lattices, now we will see how they relate when the lattices are free a.k.a. distributive lattices generated by a set of elements with no further relations. The word problem for free distributive lattices is known to be solvable and yet the structure of these lattices is not fully understood and is an active (open) area of research..
3. **Sub-lattice Embedding Problem for Distributive Lattices:** After which, given two finite distributive lattices L and M, decide whether L can be embedded as a sub-lattice of M. It is

NP-complete in general, and stills a low-level open problem in terms of efficient algorithms for particular cases..

4. **Characterization of Complemented Modular Lattices:** Explore arbitrary complemented modular lattices. Necessary and sufficient conditions are determined for a modular lattice to be complemented.
5. **Applications of Non-Boolean Complemented Lattices:** Studied complemented lattices that are not Boolean algebras and their applications in quantum logic, where the distributive law fails.
6. **Stone's Representation Theorem Extensions:** Use Stone's representation to learn work that generalizes it to other classes of lattices, for example, distributive lattices with more operations or complemented modular lattices.

## 7.4: Advanced Concepts and Further Explorations

**Advanced Concepts:**

- **Heyting Algebras:** generalizations of Boolean algebras used in intuitionistic logic. They are distributive lattices having an implication (→) such that both $a \wedge (a \to b) \leq b$ and $a \leq (b \to c) \Leftrightarrow a \wedge b \leq c$ hold which helps us think about constructive proofs and in computing programs.
- **De Morgan algebras**: This is a generalization of Boolean algebras relaxing the requirement that complements be unique. They are distributive lattices equipped with a unary operation (¬) satisfying $\neg\neg a = a$ and $\neg(a \vee b) = \neg a \wedge \neg b$. De Morgan algebras find applications in the study of non-classical logics, as well as in the design of fault-tolerant systems.
- **Orthomodular lattices:** and their generalization which are used in quantum logic. Covered lattices satisfying the orthomodular law are complemented lattices: if $a \leq b$ then $b = a \vee (b \wedge a0)$. Orthomodular lattices are a generalization of Boolean algebra that captures the essential features of quantum systems.
- **Stone Duality:** It is a general theory that connects topological spaces and lattices. It allows the translation to read between the geometric and algebraic structures and is employed on the subject of Boolean algebras, Heyting algebras, and several classes of lattices.

- **Lattice-ordered groups:** groups that, as lattices, have compatible group and lattice operations. They arise in the study of ordered algebraic structures and in the theory of partially ordered linear spaces..

**Further Explorations:**

• Explore the relationships between lattice theory and other branches of mathematics, including topology, algebra, and logic.

• Lattice Theory in Computer Science: Applications, Parts 1 and 2.

• Sequentially explore the generalizations of Boolean algebras (e.g. Heyting algebras, De Morgan algebras, orthomodular lattices, etc.) and their features.

• Investigate how lattice theory is applied in other areas such as physics, economics, and social sciences.

**Solved and Unsolved Problems**

**Solved Problems:**

1. **Prove that every Boolean algebra is distributive.**
    - Use the complement laws and the absorption laws to show that the distributive laws hold.
2. **Show that the power set of any set is a Boolean algebra.**
    - Verify the distributive laws, complement laws, and the existence of 0 and 1.
3. **Find the complement of 10 in the lattice of divisors of 30.**
    - 10' = 3, since 10 × 3 = 30 and GCD(10, 3) = 1.
4. **Prove De Morgan's laws in a Boolean algebra.**
    - Use the complement laws and the distributive laws.
5. **Show that every sub-lattice of a distributive lattice is distributive.**
    - Show that the distributive laws hold for any three elements of the sub-lattice.
6. **Determine if the lattice consisting of the divisors of 12 is a Boolean algebra.**
    - The divisors of 12 are {1,2,3,4,6,and 12}. Show that 3 does not have a compliment, therefore it is not a Boolean algebra.

7. **Draw the Hasse diagram for the lattice of subsets of {a,b,c} and verify it is distributive.**
    - Draw the diagram, then show that it does not contain N5 or M3.
8. **Prove that in a Boolean algebra, if a<=b then b'<=a'.**
    - Use the meet and join properties with compliments.
9. **Given a boolean algebra B, show that a ^ b = 0 and a v b = 1 implies b=a'.**
    - Use the compliment definition.
10. **Show that the lattice of all ideals of a ring is a complete lattice.**
    - Show that the intersection of ideals is an ideal, and that the union generates an ideal.
11. **Show that if a lattice is distributive, then if a^x=a^y and a v x = a v y, then x=y.**
    - Use the distributive properties.
12. **Show that if a lattice is a Boolean algebra, then (a v b) ^ (a v b') = a.**
    - Use distributive and complement properties.
13. **Show that if a lattice is a Boolean algebra, then (a ^ b) v (a ^ b') = a.**
    - Use distributive and complement properties.
14. **Show that if a,b are in a Boolean algebra, then (a ^ b') v (a' ^ b) = (a v b) ^ (a' v b').**
    - Use distributive and complement properties.
15. **Show that in a Boolean algebra, if a<=b, then a^b'=0.**
    - Use the definition of <= and compliments.

**Unsolved Problems:**

1. Characterize the lattices that can be embedded as sub-lattices of Boolean algebras.
2. Investigate the properties of lattices that are close to being Boolean algebras but do not satisfy all the axioms.
3. Explore the applications of non-distributive lattices in areas such as quantum information theory and artificial intelligence.
4. Develop efficient algorithms for solving problems involving Boolean algebras and related structures.

5. Investigate the connections between lattice theory and category theory, and explore the applications of these connections.
6. Find new applications for Heyting algebras in areas such as program verification and knowledge representation.
7. Explore the connections between lattice theory and formal concept analysis, and develop new methods for data analysis and knowledge discovery.
8. Investigate the properties of free lattices and free distributive lattices, and explore their applications in algebra and logic.
9. Find new applications for orthomodular lattices in areas such as quantum computing and quantum cryptography.
10. Explore new extensions and generalizations of Stone's representation theorem.

**Applications of Partially Ordered Sets**

Partially ordered sets (posets) appear frequently in computer science, mathematics, and real life. Unlike total orders, a poset allows some elements to remain incomparable, which makes it suitable for modeling many real situations.

**Task Scheduling:** When some tasks must be performed before others, but some can be done independently, the dependencies form a poset. For example, in software development, coding may depend on design, while documentation may proceed independently.

**Version Control:** In a project with multiple versions of code, the "is ancestor of" relation forms a partial order. Not all versions can be compared, but the relation still defines a structured order.

**Prerequisite Courses:** In education, some courses must be completed before others. The prerequisite relation is a partial order among all courses offered.

These examples show how posets naturally capture dependency relationships.

**Lattices in Computer Science**

A lattice is a poset in which every pair of elements has a greatest lower bound (meet) and a least upper bound (join). Lattices are especially useful in theoretical computer science.

**File Systems:** The directory structure of a computer can be viewed as a lattice, where the meet corresponds to the lowest common ancestor folder and the join represents the combined path.

**Security Levels:** In computer security, access levels such as "Confidential < Secret < Top Secret" form a lattice. Combining two security clearances requires finding their least upper bound.

**Data Mining:** Concept lattices are applied in formal concept analysis to organize large amounts of data into hierarchical structures.

**Boolean Algebra as a Lattice**

Boolean algebra is an important example of a distributive complemented lattice. In Boolean algebra, every element has a complement, and distributive laws always hold.

**Example:**

Consider the set {0,1} with the operations AND and OR.
The meet (∧) is AND, and the join (∨) is OR.
0 is the least element, and 1 is the greatest element.
The complement of 0 is 1, and the complement of 1 is 0.
This structure is both distributive and complemented, hence it forms a Boolean algebra.

**Worked Example**

Let us consider the set {1, 2, 3, 6} ordered by divisibility.
The meet of 2 and 3 is 1, since 1 divides both.
The join of 2 and 3 is 6, since 6 is divisible by both.
The meet of 2 and 6 is 2, while the join is 6.
Thus this set forms a lattice under the divisibility relation. This example shows how number theory provides natural lattices.

**Applications of Distributive and Complemented Lattices**

Distributive and complemented lattices are widely applied because they allow efficient simplification of expressions.

**Logic Circuits:** Every digital circuit is designed using Boolean algebra, which is a distributive complemented lattice. This allows systematic reduction of complex logical expressions into minimal circuits.

**Compiler Design:** When compilers optimize code, they rely on distributive properties of expressions. Boolean algebra laws are applied to simplify conditions and control flow.

**Database Theory:** Query optimization often uses lattice concepts. Operations such as union and intersection follow lattice properties that guide efficient execution.

**Information Flow:** In security models, lattices are used to describe how information flows between levels. Complementation represents access restrictions.

**Importance of Lattice Theory**

Lattice theory connects discrete mathematics with practical computer applications. It provides a unified language to describe order, hierarchy, and structure. Whether it is simplifying logical circuits, organizing data, or proving mathematical properties, lattices and posets form a fundamental part of computer science foundations.

## Check Your Progress

1. What is a distributive lattice? Explain the distributive laws that define it with an example.

   .....
   .....
   .....

2. Define a complemented lattice. Explain the conditions required for a lattice to be complemented.

   .....
   .....
   .....

## 7.5: Summary

This unit explained distributive and complemented lattices as important extensions of general lattice theory. Learners understood that in a distributive lattice, the operations of meet and join follow distributive laws, making it easier to manipulate algebraic and logical structures. In a complemented lattice, every element has at least one complement, allowing expressions to be reduced systematically. The unit also highlighted how Boolean lattices combine both distributive and complemented properties, forming the basis of Boolean algebra. These ideas are widely used in the design of digital circuits, data classification, and logical decision-making processes in computer science.

## 7.6: Exercises

**Multiple Choice Questions**

1. A lattice is said to be distributive if:
   a) Every element has a complement
   b) Meet and join satisfy distributive laws
   c) All elements are comparable
   d) It is both bounded and complemented
   Answer: b) Meet and join satisfy distributive laws
2. A complemented lattice is one in which:
   a) Each element has a unique complement
   b) Each element has at least one complement
   c) No element has a complement
   d) Every pair of elements is comparable
   Answer: b) Each element has at least one complement
3. A Boolean lattice is both:
   a) Distributive and complemented
   b) Modular and partial
   c) Complete and bounded
   d) Symmetric and transitive
   Answer: a) Distributive and complemented
4. The complement of an element a in a lattice satisfies:
   a) $a \vee a' = 1$ and $a \wedge a' = 0$
   b) $a \vee a' = 0$ and $a \wedge a' = 1$
   c) $a \vee a' = a$ and $a \wedge a' = a'$

d) None of these

Answer: a) $a \vee a' = 1$ and $a \wedge a' = 0$

5. Distributive and complemented lattices are primarily applied in:

   a) Database normalization and web design

   b) Logic circuit design and Boolean algebra

   c) Cryptography and data compression

   d) Probability and statistics

   Answer: b) Logic circuit design and Boolean algebra

**Descriptive Questions**

1. Define a distributive lattice and explain the distributive laws with examples.
2. What is a complemented lattice? Describe the conditions for an element to have a complement.
3. Differentiate between distributive, complemented, and Boolean lattices with suitable examples.
4. Explain the importance of complemented lattices in logic simplification and Boolean algebra.
5. Discuss the applications of distributive and complemented lattices in computer science and data organization.

## 7.7: References and Suggested Readings


- Kolman, Bernard, Busby, Robert C., and Ross, Sharon. Discrete Mathematical Structures, 6th Edition, Pearson Education, 2018.
- Grimaldi, Ralph P. Discrete and Combinatorial Mathematics: An Applied Introduction, 5th Edition, Pearson, 2003.
- Epp, Susanna S. Discrete Mathematics with Applications, 5th Edition, Cengage Learning, 2019.
- Tremblay, J. P., and Manohar, R. Discrete Mathematical Structures with Applications to Computer Science, Tata McGraw-Hill, 2001.

## Block 3: Boolean Algebra

# Unit 8: Basic concepts of Boolean Algebra

**Structure**


8.1 Introduction
8.2 Learning Outcome
8.3 Basic Concepts of Boolean algebra, Boolean Lattice, Boolean algebra
8.4 Boolean algebra and Its Theorems
8.5 Solved Examples
8.6 Boolean Functions
8.7 Disjunctive Normal Form (DNF)
8.8 Conjunctive Normal Form (CNF)
8.9 Bool's Expansion Theorem
8.10 Summary
8.11 Exercises
8.12 References and Suggested Readings


## 8.1: Introduction

This unit introduces the fundamental ideas of Boolean algebra, a mathematical structure that forms the basis of digital logic and computer system design. Boolean algebra deals with binary variables that take values 0 and 1 and with operations such as AND, OR, and NOT. Learners will study the axioms, laws, and properties that govern these operations and understand how they differ from ordinary arithmetic operations. The unit also explains truth tables and the concept of duality, which simplifies logical expressions. Boolean algebra provides a systematic method for analyzing and designing logic circuits, data processing units, and decision-making systems in computing.

## 8.2: Learning Outcome

- To understand the fundamental concepts of Boolean algebra and Boolean lattices.
- To analyze Boolean functions, disjunctive and conjunctive normal forms.
- To study Karnaugh maps for Boolean function simplification.

- To explore the applications of Boolean algebra in switching circuits and logic circuits.

## 8.3: Basic Concepts of Boolean algebra, Boolean Lattice, Boolean algebra

**Basic Concepts of Boolean algebra**

Boolean algebra is a mathematical structure that consists of binary variables and logical operations. Developed in the mid-19th century by George Boole, it underlies digital logic and computer science. Unlike classical algebra working on the real numbers, Boolean algebra works on binary values (0 and 1) according to a defined set of logical rules. The basic operations in Boolean algebra are AND, OR, and NOT; they are denoted respectively by multiplication, addition, and negation. These operations are used to construct Boolean expressions and simplified using laws and theorems of boolean algebra namely De Morgan's Theorems, Idempotent Laws, Absorption Law, Distributive Law. It forms the basis of digital circuit design, logic gates, and computer programming. It helps create circuits more efficiently by minimizing Boolean expressions (which in turn helps create circuits using the least number of logic gates). Its applications range from database management systems to artificial intelligence to network security. Most of the students of engineering, computer science, and mathematics need the most basic concepts of boolean algebra.

**Boolean Lattice**

A Boolean lattice is a special case of an algebraic structure that is both a lattice and satisfies the axioms of Boolean algebra. A (mathematics) lattice is a partially ordered set P, for which a pair x and y in P always has a specific least upper bound and greatest lower bound. When we include complementation and the distributive laws, we move into a Boolean lattice.

1. A Boolean lattice is precisely defined by a tuple (B, ∧, ∨, ¬, 0, 1), such that:
2. (B, ∧, ∨) forms a lattice.
3. There are two especially distinguished elements: 0 (minimum element) and 1 (maximum element).

4. Each element has a unique complement, such that x ∨ x' = 1 and x ∧ x' = 0.
5. The lattice obeys the associative, commutative, absorption, and distributive laws.

Boolean lattices are widely used in logic design, set theory, and mathematical modeling. And in doing so, they produce a model that expresses the logical relations and minimizes the logic expression. B) Also, in the analysis of switching circuits, Boolean lattices are useful to discuss about truth tables, Karnaugh maps and minimization methods.

## 8.4: Boolean algebra and Its Theorems

Theorem 1: Identity LawA + 0 = A, A · 1 = A Theorem 2: Null LawA + 1 = 1, A · 0 = 0Theorem 3: Complement LawA + A' = 1, A · A' = 0Theorem 4: Idempotent LawA + A = A, A · A = A Theorem 5: Domination LawA + 0 = A, A · 1 = 0Theorem 6: Double Negation A = A ‘Theorem 7: De Morgan's Theorems(A · B)' = A' + B', (A + B)' = A' · B'There are many theorems and mathematical properties in boolean algebra used to simplify boolean expressions. The major theorems include:

1. Idempotent Law: A + A = AA + A = A and A · A = AA · A = A.
2. Commutative Law: A+B=B+AA + B = B + A andA·B=B·AA \cdot B = B \cdot A.
3. Associative Law: A+(B+C)=(A+B)+CA + (B + C) = (A + B) + C and A·(B·C)=(A·B)·CA \cdot (B \cdot C) = (A \cdot B) \cdot C.
4. Distributive Law: A·(B+C)=A·B+A·CA·(B+C)=A·B+A·CA \cdot (B + C) = A \cdot B + A \cdot C.
5. Absorption Law: A+(A·B)=AA + (A \cdot B) = A and A·(A+B)=AA \cdot (A + B) = A.
6. De Morgan’s Theorems: (A+B)=A·B(A + B)' = A' \cdot B' and (A·B)=A+B(A \cdot B)' = A' + B'.

Because of their usefulness in obtaining the simplest form of logical representations, they have many applications in logic minimization of Boolean functions, which is useful in simplifying logical expression to produce logic circuits which are less complex, or lower in depth, or

energy consumption. For designing digital systems, control mechanisms, and data structures, Boolean controls are instrumental.

## 8.5: Solved Examples

**Example 1:** Digital Circuits Design a logic circuit the for Boolean expression (A ∧ B) ∨¬C.

• Implement A∧ B with AND, ¬C with NOT and the final expression with OR.

**Example 2:** Set Theory X = {1, 2, 3}. P(X), or the power set, is a Boolean algebra.

• {1, 2} ∨ {2, 3} = {1, 2, 3} (Union)

• {1, 2} ∧{2, 3} = {2} (Intersection)

• ¬{1, 2} = {3} (Complement)

**Example 3:** Logical Propositions Simplifying a Boolean expression (P ∧ Q) ∨ (P ∧ ¬Q)

• (P ∧ Q) ∨ (P ∧ ¬Q) = P ∧ (Q ∨ ¬Q) = P ∧ 1 = P.

**Example 4:** Truth Tables generate a truth table for the Boolean expression P ∧ (Q ∨ ¬R).

• Enumerate the cases for P, Q, R and find the value of the expression.

**Example 5:** Use Karnaugh Maps to simplify F(A, B, C) = Σ(0, 2, 4, 5, 6).

• From adjacent 1s on a Karnaugh map, extract a simplified expression.

**Example 6:** A logic proof of the absorption law: a ∨ (a ∧ b) = a.

• a∨ (a ∧ b) = (a ∧ 1) ∨ (a ∧ b) = a ∧ (1 ∨ b) = a ∧ 1 = a.

**Example 7:** Boolean Lattice Drawing (Hetherington, John: Type B, pp.

• Count all subsets and connect them on the basis of inclusion.

**Example 8:** Logical Equivalence Prove P → Q ↔ ¬P ∨ Q.

• Employing truth tables or algebraic manipulation.

**Example 9:** Boolean Reduction Simplify (A ∧ B) ∨ (A ∧ ¬B) ∨ (¬A ∧ B).

• A∧(B∨¬B)∨(¬A∧B)=A∨(¬A∧B)=(A∨¬A)∧(A∨B)=A∨B.

**Example 10:** Boolean Function Representation Implement the Boolean function F(A, B) = A ⊕ B (XOR) using AND, OR, and NOT gates.

• F(A, B) = (A and not B) or (not A and B).

**Example 11**: Divisor Boolean Algebra Prove that under divisibility containment ordering, the ordered set of divisors of the number 30 forms a Boolean algebra.

• 30 have divisors = {1, 2, 3, 5, 6, 10, 15, 30}

• Prove that it is a complemented distributive lattice.

**Example 12:** Boolean Algebra of Propositions− Show that $(P \wedge (P \rightarrow Q)) \rightarrow Q$ is a tautology.

• Employ truth tables or logical equivalences.

**Example 13:** Boolean Algebra of Sets with additional condition Let the set I={1,2,3,4} he even numbers, and the set I′={1,2,3,4} he odd numbers. Prove that I, I′ form a Boolean algebra.

• {1,3}, {2,4}, {1,2,3,4}, {}

• Join, and show complement, meet, and properties

**Example 14:** Boolean Algebra with a filter Consider sets {1,2,3,4,5,6} and the subsets {1,2,3}, {4,5,6}, {} and {1,2,3,4,5,6}. Prove that these sets create a boolean algebra.

• Show properties of complement, meet, join.

**Example 15:** Boolean algebra and Karnaugh Maps Simplifying $F(A, B, C, D) = \Sigma(0, 1, 2, 3, 8, 9, 10, 11)$ using a Karnaugh map.

• Make groups of the 1's in the Karnaugh map more with example

**Further Solved Examples and Applications**

**Example 16:** Boolean Functions and Logic Gates Use logic gates to Implement the Boolean function $F(A, B, C) = (A \wedge B) \vee (\neg A \wedge C)$.

• An AND gate ($A \wedge B$), NOT gate ($\neg A$), AND gate ($\neg A \wedge C$), OR gate (final expression).

**Example 17:** Boolean Algebra and Set Operations Let U = {a, b, c, d}. A = {a, b} B = {b, c} C = {c, d} Proof the De Morgan's law: $\neg(A \cup B) = \neg A \cap \neg B$.

• $A \cup B=\{a,b,c\}, \neg(A \cup B)=\{d\}$

• $\neg A = \{c, d\}$, $\neg B = \{a, d\}$, $\neg A \cap \neg B = \{d\}$

**Example 18:** Boolean algebra and Logical Reasoning $P \rightarrow Q$ $P$ $\neg R \rightarrow \neg Q$ prove R.

• $P \rightarrow Q$; $P \Longrightarrow Q$ (Modus Ponens).

• $\neg R \rightarrow \neg Q$ is equivalent to $Q \rightarrow R$.

• Q and $Q \rightarrow R$ (Modus Ponens)

**Example 19:** Minimization using Boolean algebra and Karnaugh maps Minimize the function $F(A, B, C, D) = \Sigma(1, 3, 5, 7, 9, 11, 13, 15)$

• Use a Karnaugh map to show that F(A, B, C, D) = D.

**Example 20:** example, Boolean algebra and how they can help you simplify your digital circuits.

(A ∧ (B ∨ ¬B)) ∨ (¬A ∧ B ∧ C) = A ∨ (¬A ∧ B ∧ C) = A ∨ (B ∧ C)

**Example 21:** The Boolean algebra and the Relational Databases Suppose we want to make a query on a database, e.g., "Select records where (age > 30 AND city = New York) OR (salary > 50000 AND department = Sales). Write this query in terms of Boolean algebra.

• Let A = (age > 30) B = (city = 'New York') C = (salary > 50000) D = (department = 'Sales')

• Query: (A ∧ B) ∨ (C ∧ D).

**Example 22:** Boolean Algebra and Logic Puzzles A puzzle states: "If it is raining (R) or snowing (S), then the road is slippery (P). If it is raining and the road is not slippery, is it snowing?

• (R ∨ S) → P, R, and ¬P.

• (R) R ∨ S is true.

• (R ∨ S) → P & ¬P ⊨ ¬(R ∨ S).

• ¬(R ∨ S) implies ¬R ∧ ¬S.

It then goes on to prove this logically: •If R is true then it must be the case that S is true. That is, S must also be true. •If S is true, then ¬R is false. Therefore, since ¬R is false, ¬S, must be true. Thus it can't be snowing.

**Example 23**: Boolean algebra and Error checking a parity bit is added to a 3-bit message (A, B, C) in a communication system to perform error detection. P in parity bit is 1 if number of 1 in message is odd, else 0 find a Boolean function for P.

• P = (A ⊕ B) ⊕ C = (A ∧ ¬B ∧ ¬C) ∨ (¬A ∧ B ∧ ¬C) ∨ (¬A ∧ ¬B ∧ C) ∨ (A ∧ B ∧ C).

**Example 24:** Boolean algebra, Verification of Formal Verification using notations of Structural Modeling in Virology.

• Verify that the output of the circuit will always equal the function's definition using truth tables or Boolean algebra.

**Example 25**: Boolean algebra and State Machines A state machine can change between states depending upon input signals. Use Boolean algebra to represent the transition logic.

• For example: If input A is 1, a finite state machine transitions to state S1; otherwise, it stays in state S0.

• Transition: S1 = A ∧ S0'.

**Example 26:** Boolean algebra and Combinational Logic Design a combinational logic circuit such that when two of its three inputs (A, B, C) are 1, the output is 1.

• F = (A ∧ B) ∨ (A ∧ C) ∨ (B ∧ C).

**Example 27:** Boolean algebra and Data Structures Boolean algebra is used to express a family of conditions under which a data structure may be accessed.

• For instance: let access be granted if the following condition holds: (user is admin AND file exists) OR (user has read permission AND file is public).

**Example 28:** Boolean algebra and AI Logic Use Boolean algebra to represent a simple rule-based system.

• Example: IF (temperature is high AND humidity is high) THEN turn on cooling system

**Example 29:** Boolean algebra and Networking Boolean algebra can be used to represent network routing rules.

• Example: Send packet to router A if (dest. network is X AND protocol is TCP) OR (dest. network is Y AND protocol is UDP)

**Example 30:** boolean algebra in Card Games Representing Winning Conditions in a Card game

• Example: win if (player has all keys and player has reached goal) or (player has defeated all enemies).

**Unsolved Problems**

1. Simplify: A(B+C)+A′CA(B + C) + A'C.
2. Prove: A+AB′=A+B′A + AB' = A + B'.
3. Find the complement of: (A+B)(A′+B′)(A + B)(A' + B').
4. Show that: A+A′B=A+BA + A'B = A + B.
5. Simplify: A+AB+A′BA + AB + A'B.
6. Prove: A+AB′+B=A+BA + AB' + B = A + B.
7. Find the complement of: A′B+AB′A'B + AB'.
8. Simplify: A+A′B+ABA + A'B + AB.
9. Prove: A+B+AB=A+BA + B + AB = A + B.
10. Simplify: (A+B)(A+B′)(A + B)(A + B').

This part presented fundamental concepts of Boolean algebra, Boolean lattice, and its theorems along with solved and unsolved problems. Boolean algebra is an essential subject in mathematics, computer science, and electrical engineering, and learning it will help

you understand logic circuits, digital systems, and computational logic.

## 8.6: Boolean Functions

Boolean algebra is a mathematical system used in computer applications, particularly in logic design and digital circuits. It operates on binary values (0 and 1) and forms the foundation for designing logic gates and circuits. In this context, Boolean functions play a crucial role in representing logical expressions using variables, operations, and standard forms. This section delves into Boolean functions, their standard forms—Disjunctive Normal Form (DNF) and Conjunctive Normal Form (CNF)—the concept of complement functions, and Bool's Expansion Theorem, which is essential in simplifying and analyzing Boolean expressions.

**Boolean Functions**

A Boolean function is a mathematical expression composed of binary variables, logical operators (AND, OR, NOT), and constants (0 and 1). It defines a mapping from input values to a single binary output. Boolean functions are widely used in digital circuits to design combinational and sequential logic circuits. The fundamental Boolean operations used to construct these functions are:

1. **AND (·)**: A binary operation that results in 1 only if both inputs are 1; otherwise, it results in 0.
2. **OR (+)**: A binary operation that results in 1 if at least one of the inputs is 1.
3. **NOT ( ¬ )**: A unary operation that inverts the input, changing 1 to 0 and vice versa.

A Boolean function can be represented in different ways:

- **Truth Table Representation**: A tabular representation of all possible input combinations and their corresponding output.
- **Algebraic Expression**: The function is represented as an equation using Boolean operators.
- **Logic Circuit Diagram**: A graphical representation using logic gates.
- **Canonical and Standard Forms**: Expressions written in a predefined normal form for easier simplification.

## 8.7: Disjunctive Normal Form (DNF)

Disjunctive Normal Form (DNF) is a standardized way of representing Boolean functions as a disjunction (OR operation) of multiple conjunctions (AND operation). A Boolean function is said to be in DNF if it consists of a sum of product terms, where each product term is a conjunction of literals. A literal is either a Boolean variable (e.g., xxx) or its negation (e.g., ¬x). A minterm is a conjunction (AND) of literals where each variable appears exactly once in either its true or complemented form. The general form of DNF is:

$F(x1,x2,\ldots,xn)=(A1\cdot B1\cdot C1)+(A2\cdot B2\cdot C2)+\cdots+(Am\cdot Bm\cdot Cm)$

where each term inside the parentheses is a minterm. For example, consider a Boolean function with three variables:

$F(A,B,C)=(A\cdot\neg B\cdot C)+(A\cdot B\cdot C)+(\neg A\cdot B\cdot\neg C)$

This is in disjunctive normal form because it consists of ORed minterms. DNF is useful because it provides a clear way of defining Boolean expressions in terms of logical OR of individual product terms. However, it is not always the most optimized form for circuit implementation.

## 8.8: Conjunctive Normal Form (CNF)

Conjunctive Normal Form (CNF) is another standard form for Boolean functions, where the function is expressed as a conjunction (AND operation) of multiple disjunctions (OR operation). Instead of minterms, CNF uses maxterms, which are OR operations of literals.

A Boolean function is said to be in CNF if it consists of a product of sum terms, where each sum term is a disjunction of literals.

The general form of CNF is:

$F(x1,x2,\ldots,xn)=(A1+B1+C1)\cdot(A2+B2+C2)\cdot\cdots\cdot(Am+Bm+Cm)$

where each term inside the parentheses is a maxterm.

For example, consider a Boolean function with three variables:

$F(A,B,C)=(A+B+\neg C)\cdot(A+\neg B+C)\cdot(\neg A+B+C)$

This is in conjunctive normal form because it consists of ANDed sum terms.CNF is particularly useful in logic programming, propositional logic, and solving satisfiability problems (such as in the SAT problem). However, like DNF, it may not always be the most efficient representation for digital circuit design.

**3.1.7 Complement Function**

The complement of a Boolean function is the logical negation of the function. Given a function F(x1,x2,...,xn), its complement is denoted as ¬F or F″ and is derived by applying De Morgan's Theorem:

- **De Morgan's Theorems** state that:
    1. ¬(A·B)=¬A+¬B
    2. ¬(A+B)=¬A·¬B

To find the complement of a Boolean function, we apply the NOT operation to the entire function and use these rules to simplify.

For example, given the function:

$$F(A,B)=A+\neg B$$

Its complement is:

$$F'(A,B)=\neg(A+\neg B)$$

Applying De Morgan's Theorem:

$$F'(A,B)=\neg A\cdot B$$

The complement function is important in digital logic because it helps in designing circuits such as NAND and NOR implementations, which are functionally complete (i.e., they can be used to construct any Boolean function).

## 8.9: Bool's Expansion Theorem

Bool's Expansion Theorem (also called the Shannon Expansion Theorem) is a fundamental theorem in Boolean algebra used for simplifying Boolean functions and designing digital circuits. The theorem states that any Boolean function can be expressed in terms of one of its variables and its complement.

The theorem is mathematically expressed as:

$$F(x1,x2,...,xn)=xF(1,x2,...,xn)+\neg xF(0,x2,...,xn)$$

where F(1,x2,...,xn) is the function evaluated with x=1x = 1x=1, and F(0,x2,...,xn) is the function evaluated with x=0.

For example, consider the function:

$$F(A,B)=A\cdot B+A'$$

Applying Bool's Expansion Theorem on AAA:

$$F(A,B)=A\cdot F(1,B)+\neg A\cdot F(0,B)$$

Evaluating:

$$F(1,B)=1\cdot B+0=B$$

$$F(0,B)=0\cdot B+1=1$$

Thus, expanding:

$$F(A,B)=A\cdot B+\neg A\cdot 1$$

Bool's Expansion Theorem is widely used in:

- **Circuit simplification**: Breaking down complex Boolean expressions into simpler components.
- **Multiplexer design**: Used to construct logic functions using multiplexers.
- **Binary decision diagrams (BDDs)**: Representing and simplifying Boolean functions graphically.

Boolean functions are essential in digital logic design and computer applications. Disjunctive Normal Form (DNF) and Conjunctive Normal Form (CNF) provide standardized ways to express Boolean functions for logic circuit design. The complement function plays a crucial role in implementing logical negations and designing NOR/NAND-based circuits. Bool's Expansion Theorem offers a systematic way to simplify Boolean functions and is fundamental in logic synthesis and digital circuit design. Understanding these concepts allows for efficient manipulation of logical expressions, leading to optimized digital systems.

## Check Your Progress

1. Define Boolean algebra. Explain its fundamental operations and basic laws with suitable examples.
   .....
   .....
   .....
   .....
   .....

2. Explain the principle of duality in Boolean algebra with an example.
   .....
   .....
   .....
   .....
   .....

## 8.10: Summary

This unit covered the foundational principles of Boolean algebra, emphasizing its role in simplifying and analyzing logical

relationships. Learners understood binary variables, standard logical operations, and basic postulates that define Boolean systems. The unit highlighted how Boolean laws such as commutative, associative, distributive, identity, and complement laws are used to manipulate expressions systematically. Through the use of truth tables and the principle of duality, Boolean algebra provides a framework for optimizing logic circuits and computational processes. The knowledge gained in this unit prepares learners for advanced topics such as Boolean functions, normal forms, and applications in switching and logic circuits.

## 8.11: Exercises

**Multiple Choice Questions**

1. Boolean algebra operates on variables that take values:
   a) 0 and 1 only
   b) Any real numbers
   c) Positive integers
   d) Complex numbers
   Answer: a) 0 and 1 only
2. The complement of 1 in Boolean algebra is:
   a) 0
   b) 1
   c) Both 0 and 1
   d) None of these
   Answer: a) 0
3. The AND operation in Boolean algebra is equivalent to:
   a) Addition
   b) Multiplication
   c) Subtraction
   d) Division
   Answer: b) Multiplication
4. The dual of the expression A + 0 is:
   a) A · 1
   b) A · 0
   c) A + 1
   d) A · A
   Answer: a) A · 1

5. Which of the following is an identity law in Boolean algebra?
   a) A + 0 = A
   b) A + 1 = A
   c) A · 0 = A
   d) A · A = 1
   Answer: a) A + 0 = A

**Descriptive Questions**

1. Define Boolean algebra and describe its fundamental operations.
2. Explain the principle of duality in Boolean algebra with suitable examples.
3. List and prove any four basic laws of Boolean algebra.
4. Construct truth tables for AND, OR, and NOT operations and explain their significance.
5. Discuss the applications of Boolean algebra in digital logic circuit design and computer systems.

## 8.12: References and Suggested Readings


- Raghavan, V. *Digital Logic and Computer Design*, Prentice Hall of India, 2015.
- Mano, M. Morris, and Ciletti, Michael D. *Digital Design: With an Introduction to the Verilog HDL*, 6th Edition, Pearson Education, 2017.
- Tokheim, Roger L. *Digital Electronics: Principles and Applications*, 9th Edition, McGraw-Hill Education, 2015.
- Malvino, Albert Paul, and Leach, Donald P. *Digital Principles and Applications*, 7th Edition, McGraw-Hill Education, 2016.
- Floyd, Thomas L. *Digital Fundamentals*, 12th Edition, Pearson Education, 2022.

# Unit 9: Karnaugh map

**Structure**

9.1 Introduction

9.2 Learning Outcome

9.3 Karnaugh Map Method for Simplification of Boolean Expressions

9.4 Grouping Rules for Simplification

9.5 Summary

9.6 Exercises

9.7 References and Suggested Readings

## 9.1: Introduction

This unit introduces the Karnaugh Map (K-Map), a visual and systematic method for simplifying Boolean expressions. The K-Map provides an easier way to minimize logic functions compared to algebraic manipulation. It uses a grid representation that groups adjacent cells corresponding to minterms or maxterms with common variables. By identifying these groups, learners can reduce complex logic equations into simpler forms with fewer variables and logic gates. K-Maps are widely used in digital logic design to optimize circuits, minimize hardware requirements, and improve computational efficiency. Understanding K-Maps is essential for analyzing and implementing efficient digital systems.

## 9.2: Learning Outcomes

After studying this unit, learners will be able to:

- Define a Karnaugh Map and explain its purpose in simplifying Boolean expressions.
- Construct K-Maps for two, three, four, and five variables.
- Identify and group minterms and maxterms to form simplified expressions.
- Derive minimal Sum-of-Products (SOP) and Product-of-Sums (POS) forms using K-Maps.
- Apply K-Maps to reduce logic circuits and minimize the number of gates required.
- Analyze real-world problems in digital design using K-Map simplification techniques.

## 9.3: Karnaugh Map Method for Simplification of Boolean Expressions

Boolean algebra plays a critical role in computer science, electrical engineering, and digital logic design. As Boolean functions grow more complex, simplifying them becomes essential to improve the efficiency of digital circuits. While algebraic manipulation and truth tables offer ways to minimize Boolean expressions, the Karnaugh Map (K-map) method provides a systematic and visual approach to simplification. This technique is especially useful for reducing the number of logic gates required to implement a function, thus optimizing hardware performance and resource utilization.

**Introduction to Karnaugh Maps**

The Karnaugh Map (K-map) is a diagrammatic method for simplifying Boolean expressions by grouping terms to eliminate redundant variables. It was introduced by Maurice Karnaugh in 1953 and has since become one of the most widely used techniques in digital circuit design. The K-map method provides an alternative to algebraic simplification, which can sometimes be cumbersome and prone to errors. A K-map is a grid-like structure where each cell represents a possible combination of input variables. The values in the cells correspond to the output values of the Boolean function being simplified. By identifying adjacent cells with similar values, we can form groups that allow us to derive a simplified expression with fewer terms and variables.

**Structure of Karnaugh Maps**

A Karnaugh Map is constructed based on the number of variables in the Boolean function. The number of cells in the K-map corresponds to the possible input combinations, which is determined by **2^n**, where **n** is the number of variables.

- 2-variable K-map: 4 cells (2x2 grid)
- 3-variable K-map: 8 cells (2x4 grid)
- 4-variable K-map: 16 cells (4x4 grid)
- 5-variable K-map and beyond: Higher-dimensional representations, typically drawn as multiple 4-variable K-maps

Each cell in the map represents a **minterm**, which is a product term in the Sum of Products (SOP) form of the Boolean expression. The

arrangement of cells follows **Gray Code** sequence instead of binary order, ensuring that adjacent cells differ by only one variable.

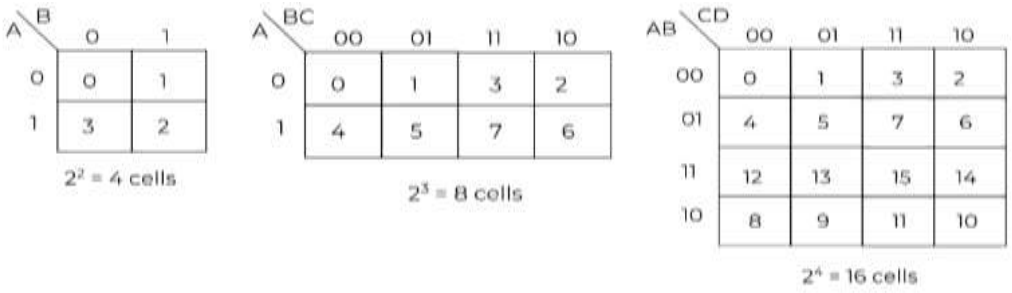



Fig: Karnaugh Map

**Filling the Karnaugh Map**

To simplify a Boolean function using a Karnaugh Map, follow these steps:

1. Determine the number of variables in the Boolean function.
2. Draw the corresponding K-map grid based on the number of variables.
3. Label the rows and columns using Gray Code to ensure single-bit transitions between adjacent cells.
4. Place 1s in the cells that correspond to the minterms in the given Boolean function.
5. Group adjacent 1s together to form rectangles containing 1, 2, 4, 8, or more minterms, ensuring that each group is as large as possible.
6. Derive the simplified expression from the groups by identifying the common variables.

## 9.4: Grouping Rules for Simplification

The primary objective of K-map simplification is to create the largest possible groupings of 1s. Some fundamental grouping rules include:

- Groups must contain 1, 2, 4, 8, or more cells in powers of 2.
- Groups should be as large as possible while still covering all 1s.
- Each 1 must be covered by at least one group, but it can be part of multiple groups.
- The edges of a K-map wrap around, meaning cells on one edge are adjacent to corresponding cells on the opposite edge.

**Example: Simplifying a 3-Variable Boolean Function**

Consider the Boolean function:

The 3-variable Karnaugh Map is structured as follows:

| **AB\C** | **0** | **1** |
|---|---|---|
| **00** | 1 | 1 |
| **01** | 0 | 1 |
| **11** | 0 | 1 |
| **10** | 1 | 0 |

From this, we identify the groupings:

1. (0,1) and (6,7) form a group because they differ by only one variable.
2. (3,7) and (6,7) form another group.
3. Single remaining 1 at position (6) can be grouped with an adjacent pair.

From these groups, we extract the simplified expression:

This is the minimized Boolean function, requiring fewer logic gates for implementation.

**Advantages of Using Karnaugh Maps**

- Visual and Intuitive: Unlike Boolean algebra, which requires algebraic manipulation, K-maps offer a clear visualization of simplifications?
- Efficient for Small Functions: K-maps work efficiently for functions with up to 5 or 6 variables. Beyond that, tabular methods like Quine-McCluskey are preferred.
- Minimizes Logic Circuits: Reducing the number of terms directly translates to fewer logic gates, reducing power consumption and increasing circuit speed.

**Limitations of Karnaugh Maps**

- Not Scalable Beyond 5-6 Variables: As the number of variables increases, K-maps become difficult to handle due to their exponential growth.
- Complex Grouping: While it is straightforward for small expressions, larger maps can be tricky to group optimally.
- Requires Manual Effort: Unlike algebraic methods that can be implemented algorithmically, K-map simplifications rely on human intuition.

**Comparison with Other Simplification Techniques**

Several alternative methods exist for Boolean function simplification:

**Table 9.1: Advantages and disadvantages of different methods**

| Method | Advantages | Disadvantages |
|---|---|---|
| **Boolean Algebra** | Rigorous and logical | Time-consuming, complex for large expressions |
| **Truth Table** | Systematic, complete | Not directly used for simplification |
| **Karnaugh Map** | Visual, effective for small functions | Not scalable beyond 5-6 variables |
| **Quine-McCluskey** | Algorithmic, works for larger functions | Computationally expensive for very large functions |

**Application of Karnaugh Maps in Computer Applications**

K-maps play a vital role in the design and optimization of:

- **Combinational Circuits:** Used in arithmetic logic Modules (ALUs), multiplexers, and demultiplexers.
- **Sequential Circuits:** Simplification of next-state logic in finite state machines.
- **Memory Address Decoding:** Optimization of address decoding logic in microprocessors.
- **Control Systems:** Used in designing control Modules in embedded systems and automation.

## Check Your Progress

1. What is a Karnaugh Map (K-Map)? Describe its role in simplifying Boolean expressions.

   .....
   .....
   .....
   .....
   .....

2. Explain how to simplify a four-variable Boolean expression using a Karnaugh Map with an example.

   .....
   .....
   .....
   .....
   .....

## 9.5: Summary

The Karnaugh Map method is a powerful tool for simplifying Boolean expressions, making digital circuit design more efficient. By systematically grouping adjacent minterms, K-maps eliminate redundant variables, reducing the complexity of logic circuits. Despite its limitations for large-scale problems, it remains an essential technique in digital logic design, particularly for small to medium-sized Boolean functions. Understanding K-maps enables engineers and computer scientists to create optimized, cost-effective, and high-performance digital systems, forming the foundation of modern computational applications.

## 9.6: Exercises

**Multiple Choice Questions**

1. The main purpose of a Karnaugh Map is to:
   a) Construct truth tables
   b) Simplify Boolean expressions
   c) Design sequential circuits
   d) Perform arithmetic operations
   Answer: b) Simplify Boolean expressions
2. In a four-variable K-Map, the number of cells is:
   a) 8
   b) 12
   c) 16
   d) 4
   Answer: c) 16
3. Two adjacent 1s in a K-Map represent:
   a) A single minterm
   b) A larger group that removes one variable
   c) No simplification
   d) An invalid group
   Answer: b) A larger group that removes one variable
4. The grouping in a K-Map must be:
   a) Diagonal
   b) Circular
   c) Rectangular and of size 1, 2, 4, 8, etc.

d) Random

Answer: c) Rectangular and of size 1, 2, 4, 8, etc.

5. The minimal expression obtained from a K-Map helps to:
   a) Increase the number of logic gates
   b) Reduce circuit complexity
   c) Decrease circuit speed
   d) Eliminate essential variables
   Answer: b) Reduce circuit complexity

**Descriptive Questions**

1. Define a Karnaugh Map and explain its use in simplifying Boolean functions.
2. Describe the procedure for simplifying a three-variable Boolean expression using a K-Map.
3. What are prime implicants and essential prime implicants? Explain with examples.
4. Construct a four-variable K-Map and show how adjacent 1s can be grouped to simplify an expression.
5. Discuss the advantages of K-Map simplification over algebraic simplification in digital circuit design.

## 9.7: References and Suggested Readings


- Mano, M. Morris, and Ciletti, Michael D. *Digital Design: With an Introduction to the Verilog HDL*, 6th Edition, Pearson Education, 2017.
- Floyd, Thomas L. *Digital Fundamentals*, 12th Edition, Pearson Education, 2022.
- Malvino, Albert Paul, and Leach, Donald P. *Digital Principles and Applications*, 7th Edition, McGraw-Hill Education, 2016.
- Tokheim, Roger L. *Digital Electronics: Principles and Applications*, 9th Edition, McGraw-Hill Education, 2015.
- Rosen, Kenneth H. *Discrete Mathematics and Its Applications*, 8th Edition, McGraw-Hill Education, 2019.

## Block 3: Boolean Algebra

# Unit 10: Applications of Boolean Algebra in switching circuits, logic circuits

**Structure**

10.1 Introduction
10.2 Learning Outcome
10.3 Applications of Boolean Algebra in Switching Circuits, Logic Circuits
10.4 Application of Boolean Algebra
10.5 Summary
10.6 Exercises
10.7 References and Suggested Readings

## 10.1: Introduction

This unit explains how Boolean algebra serves as the mathematical foundation for the design and analysis of digital logic circuits. Boolean expressions represent the logical behavior of electronic circuits that operate using binary signals. Learners will understand how Boolean laws, postulates, and simplification techniques are used to construct and optimize combinational circuits such as adders, subtractors, multiplexers, and decoders. The unit also focuses on the translation of Boolean expressions into logic gate diagrams and vice versa. By studying these applications, students gain practical insight into how theoretical Boolean concepts translate into real-world hardware design and efficient circuit implementation.

## 10.2: Learning Outcomes

After studying this unit, learners will be able to:

- Apply Boolean laws and theorems to simplify logic expressions.
- Design combinational circuits using logic gates based on Boolean equations.
- Convert Boolean expressions into equivalent logic diagrams and truth tables.
- Implement arithmetic and data-processing circuits using Boolean simplification.

- Analyze circuit performance and identify redundant logic components.
- Relate Boolean algebra concepts to modern digital system design and computer hardware.

## 10.3: Applications of Boolean Algebra in Switching Circuits, Logic Circuits

Boolean algebra, formulated by George Boole in the mid-19th century, serves as the mathematical foundation for digital logic design and switching circuits. It provides a systematic framework for analyzing and designing circuits that operate using binary variables, which take values of either 0 or 1. The fundamental principles of Boolean algebra are instrumental in the functioning of modern computers, control systems, and communication devices. In the realm of digital electronics, Boolean algebra simplifies circuit design by minimizing the number of logic gates required to perform specific operations, thereby enhancing efficiency, reducing power consumption, and improving system reliability. Switching circuits, also known as combinational logic circuits, rely on Boolean algebra to define their operational behavior. These circuits consist of interconnected logic gates—such as AND, OR, NOT, NAND, NOR, XOR, and XNOR gates—that process binary signals to generate desired outputs. Boolean algebra enables engineers to represent complex logical expressions using algebraic notation, simplifying the analysis and design of switching networks. The application of Boolean algebra in switching circuits is crucial in various fields, including telecommunications, industrial automation, robotics, and embedded systems. One of the most important applications of Boolean algebra in switching circuits is the design of logic circuits, which are used to perform arithmetic operations, data processing, and decision-making functions in digital systems. Logic circuits are broadly classified into combinational and sequential circuits. Combinational logic circuits produce outputs solely based on current input values, whereas sequential logic circuits store past input information and exhibit memory-like behavior. The implementation of Boolean functions in combinational circuits enables the construction of essential digital components, such as multiplexers,

demultiplexers, encoders, decoders, arithmetic logic Modules (ALUs), and memory systems.

Boolean algebra plays a crucial role in simplifying complex logic expressions through algebraic manipulation techniques, such as the application of Boolean postulates, De Morgan's theorems, and Karnaugh maps. By reducing redundant variables and minimizing logical expressions, Boolean algebra optimizes circuit design, leading to lower hardware costs and improved performance. For example, in designing a digital adder circuit, Boolean algebra helps derive the simplest possible expressions for sum and carry outputs, thereby reducing the number of gates required. This optimization is particularly valuable in large-scale integrated circuits (LSIs) and very-large-scale integrated circuits (VLSIs), where minimizing transistor count is essential for power efficiency and compactness. In switching circuits, Boolean algebra is employed to analyze and design relay logic systems, which were historically used in early telephone exchanges and industrial control applications. Relay logic, based on electromechanical relays, follows Boolean principles to control electrical circuits through logical conditions. The advent of solid-state digital electronics replaced relays with semiconductor-based logic gates, but the foundational Boolean concepts remain unchanged. Programmable logic controllers (PLCs), widely used in industrial automation, also rely on Boolean logic to execute control sequences and decision-making processes. Another significant application of Boolean algebra in switching circuits is the design of sequential logic circuits, such as flip-flops, registers, and counters. Sequential circuits differ from combinational circuits in that they incorporate memory elements that store binary states. Boolean algebra helps define the logical relationships governing state transitions, enabling the construction of synchronous and asynchronous sequential systems. Flip-flops, which serve as the building blocks of memory storage and clocked circuits, operate based on Boolean expressions to determine state changes. Counters, which are used in digital clocks, frequency dividers, and event counting applications, function by following Boolean logic rules to transition between states in a predefined sequence.

The minimization of Boolean expressions using Karnaugh maps (K-maps) and the Quine-McCluskey method is a critical aspect of

optimizing switching circuits and logic circuits. K-maps provide a visual representation of Boolean functions, facilitating the identification of common terms and redundant variables. The Quine-McCluskey method, a tabular technique for simplification, is particularly useful for automated logic design in digital circuits. These minimization techniques significantly impact circuit performance by reducing propagation delay, power consumption, and the likelihood of timing errors. Boolean algebra is also extensively used in designing logic circuits for microprocessors, digital signal processors (DSPs), and application-specific integrated circuits (ASICs). Microprocessors, the core components of modern computing devices, rely on logic circuits that perform arithmetic and logical operations based on Boolean expressions. Boolean functions govern the design of arithmetic circuits, such as adders, sub tractors, multipliers, and dividers, which form the computational backbone of processors. In DSP applications, Boolean algebra is used to implement logic-based filtering, signal modulation, and pattern recognition techniques. In the field of digital communication, Boolean algebra plays a vital role in the design of error detection and correction circuits. Parity generators and checkers, which ensure data integrity during transmission, operate using Boolean functions to detect bit errors. Hamming codes and cyclic redundancy check (CRC) methods, based on Boolean algebra, enable error correction in data communication systems. These applications are fundamental to ensuring reliable data transfer in network protocols, storage devices, and wireless communication systems.

Memory design and storage technologies also leverage Boolean algebra to optimize read and write operations. Random-access memory (RAM), read-only memory (ROM), and flash memory circuits utilize Boolean logic to manage data storage and retrieval processes. Address decoding circuits, which determine memory locations for data storage, operate using Boolean expressions to enable efficient memory access. Boolean algebra further aids in designing cache memory management systems, optimizing data access speeds in modern computing architectures. Boolean algebra is fundamental in the design of digital control systems, which govern automated processes in various industries. Digital controllers, used in robotics, aerospace, medical devices, and smart home technologies,

rely on Boolean logic to execute programmed instructions. Logic circuits in digital controllers interpret sensor inputs, process logical conditions, and generate control signals to drive actuators. The application of Boolean algebra in these systems enhances precision, reliability, and responsiveness in automated decision-making. The advent of field-programmable gate arrays (FPGAs) and complex programmable logic devices (CPLDs) has further expanded the scope of Boolean algebra in modern electronics. FPGAs and CPLDs provide reconfigurable logic platforms that allow engineers to implement custom digital circuits using Boolean expressions. Hardware description languages (HDLs), such as Verilog and VHDL, enable the design and simulation of Boolean-based logic circuits before hardware implementation. These programmable logic devices have revolutionized prototyping, allowing rapid development and testing of digital systems.

Another critical application of Boolean algebra in switching circuits and logic circuits is found in cybersecurity and cryptographic systems. Boolean functions are used in the design of cryptographic algorithms, such as symmetric and asymmetric encryption schemes, hash functions, and digital signatures. Boolean logic ensures secure data encryption and authentication processes, safeguarding sensitive information in digital communication and financial transactions. Boolean algebra serves as the backbone of switching circuits and logic circuits, playing a fundamental role in digital electronics, computing, automation, and communication systems. Its application in circuit simplification, memory design, error detection, digital control, and cryptographic security underscores its significance in modern technology. The ability of Boolean algebra to represent and manipulate logical relationships enables engineers and computer scientists to design efficient, reliable, and scalable digital systems. As technological advancements continue to evolve, the principles of Boolean algebra will remain indispensable in shaping the future of digital innovation.

## 10.4: Applications of Boolean Algebra

Boolean algebra is one of the most important tools in computer science and digital electronics. It provides the mathematical

foundation for the design of circuits, logic programming, and problem solving.

**Digital Circuit Design:** Every digital circuit can be represented by a Boolean expression. By applying Boolean laws, we can simplify these expressions to reduce the number of gates used, which makes circuits faster and cheaper.

**Search Engines:** Boolean operators such as AND, OR, and NOT are used to combine search terms. For example, searching for "MCA AND Mathematics" will return results containing both words.

**Computer Networks:** Boolean logic is used in routing decisions and access control lists, where packet rules are written using logical expressions.

**Programming Languages:** Conditional statements in languages such as C, Java, or Python are evaluated using Boolean expressions.

**Simplification of Boolean Functions**

Karnaugh maps (K-maps) are widely used to minimize Boolean functions. They help in identifying common groups and reducing the number of variables in an expression.

**Example:** Simplify the function

$F(A, B, C) = \Sigma(1, 3, 5, 7)$

**Step 1:** Write minterms in binary form.

1 = 001

3 = 011

5 = 101

7 = 111

**Step 2:** Place them in a 3-variable K-map.

**Step 3:** Identify groups of adjacent 1s. In this case, grouping yields:

$F = B \wedge C \vee A \wedge C$.

Thus, the simplified expression is:

$F(A, B, C) = C(B \vee A)$.

This is simpler than the original sum of four minterms.

**Boolean Algebra in Error Detection and Correction**

Boolean functions are also used in designing error detection codes. For example, parity bits are generated using XOR operations. If the number of 1s in a message is odd, the parity bit ensures the total becomes even. At the receiving end, if the parity does not match, an error is detected.

**Boolean Functions in Switching Theory**

Switching circuits such as elevators, traffic signals, and vending machines are modeled using Boolean functions. The ON and OFF states correspond to 1 and 0. Boolean simplification ensures that such systems work efficiently with minimal switches.

**Worked Example:** Circuit Simplification

Consider the Boolean function

$F = AB + A'B + AB'$.

**Step 1:** Apply distributive law:

$F = AB + A'B + AB' = B(A + A') + AB'$.

**Step 2:** Since $A + A' = 1$, we get:

$F = B(1) + AB' = B + AB'$.

**Step 3:** Apply absorption law:

$F = B + A$.

Thus, the circuit is simplified from three terms to just two inputs connected by OR.

**Importance of Boolean Algebra**

The study of Boolean algebra is not limited to digital systems. It is an essential part of problem solving in mathematics, optimization in programming, database query design, and artificial intelligence reasoning. Its principles allow computers to represent complex logical problems using just two states, 0 and 1, making it the language of computation itself.

## Check Your Progress

1. Explain how Boolean algebra is used in designing logic circuits. Give an example of a simple circuit.

.....

.....
.....
.....
.....

2. Discuss the application of Boolean simplification in reducing circuit complexity and hardware cost.

.....
.....
.....
.....
.....

## 10.5: Summary

This block introduces the principles and applications of **Boolean algebra**, a mathematical framework essential for digital logic and computer design. It begins with the **fundamental concepts** of Boolean algebra and **Boolean lattices**, where binary variables take values 0 and 1 and operations such as AND, OR, and NOT follow specific algebraic rules. Boolean expressions are simplified using **identities and laws**, including commutative, associative, distributive, identity, and De Morgan's laws. The module explores **Boolean functions**, their representation, and transformation into **Disjunctive Normal Form (DNF)** and **Conjunctive Normal Form (CNF)**, aiding in standardizing logic expressions. A key tool for simplification is the **Karnaugh Map (K-Map)**, which visually groups adjacent ones or zeros to minimize Boolean functions with fewer terms, improving circuit efficiency. The module also examines the **application of Boolean algebra in switching circuits and logic gates**, enabling the design and analysis of digital systems like adders, multiplexers, and combinational logic circuits.

Mastery of these concepts is foundational for understanding how computers perform logical operations and how hardware implements complex decision-making processes. he final part of the module explores the **applications of Boolean algebra in switching and logic circuits**. Boolean expressions are directly used to design and analyze digital components such as logic gates, multiplexers, demultiplexers, encoders, decoders, and arithmetic circuits like adders and subtractors.

The conversion from Boolean expressions to circuit diagrams forms the foundation of **combinational logic design**, which is critical in building CPUs, memory units, and control systems. In summary, This Module equips students with the theoretical and practical tools to analyze and design logical systems using Boolean algebra. It bridges the gap between abstract algebraic principles and their real-world application in computer hardware, laying the groundwork for more advanced studies in digital electronics, computer architecture, and logic-based computation.

## 10.6: Exercises

**Multiple-Choice Questions**

1. Which of the following operations are fundamental in Boolean algebra?
   a) Addition, subtraction, multiplication
   b) AND, OR, NOT
   c) Integration, differentiation, exponentiation
   d) Union, intersection, complement
   Answer: b)
2. A Boolean function can be expressed in which of the following normal forms?
   a) Disjunctive Normal Form (DNF)
   b) Conjunctive Normal Form (CNF)
   c) Both (a) and (b)
   d) None of the above
   Answer: c)
3. Which theorem in Boolean algebra states that any Boolean function can be expanded into a sum of two sub-functions?
   a) De Morgan's Theorem
   b) Duality Theorem
   c) Boolean Expansion Theorem
   d) Absorption Theorem
   Answer: c)
4. Which simplification technique is commonly used for minimizing Boolean functions graphically?
   a) Karnaugh Map (K-Map)
   b) Truth Table Method

c) Algebraic Substitution
d) State Diagram
Answer: a)

5. Boolean algebra is widely applied in which of the following fields?
   a) Switching circuits
   b) Logic circuits
   c) Digital electronics
   d) All of the above
   Answer: d)

**Descriptive Questions**

1. Explain the fundamental concepts of Boolean algebra and Boolean lattices. How are Boolean lattices different from general lattices?
2. Describe disjunctive and conjunctive normal forms with examples. Why are they important in Boolean function simplification?
3. What is Karnaugh Map (K-Map)? How is it used for the simplification of Boolean expressions? Explain with examples.
4. State and prove Boole's Expansion Theorem. How does it help in Boolean function manipulation?
5. Discuss the applications of Boolean algebra in switching circuits and logic circuits. Provide examples illustrating its practical significance.

## 10.7: References and Suggested Readings


- Malvino, Albert Paul, and Leach, Donald P. *Digital Principles and Applications*, 7th Edition, McGraw-Hill Education, 2016.
- Tokheim, Roger L. *Digital Electronics: Principles and Applications*, 9th Edition, McGraw-Hill Education, 2015.
- Rosen, Kenneth H. *Discrete Mathematics and Its Applications*, 8th Edition, McGraw-Hill Education, 2019.
- Hill, Frederick J., and Peterson, Gerald R. *Introduction to Switching Theory and Logical Design*, John Wiley & Sons, 2014.

## Block 4: Graph Theory

# Unit 11: Basic concepts of graph theory

**Structure**


11.1 Introduction

11.2 Learning Outcome

11.3 Basic Concepts of Graph Theory

11.4 Graph Representations

11.5 Applications and Solved Questions

11.6 Sub graphs, Walks, Paths, and Circuits in Graph Theory

11.7 Common Properties and Interrelationships

11.8 Solved Examples

11.9 Types of Graph

11.10 Summary

11.11 Exercises

11.12 References and Suggested Readings


## 11.1: Introduction

This unit introduces the basic concepts of graph theory, one of the most important branches of discrete mathematics with wide applications in computer science. A graph is a collection of nodes (vertices) connected by edges that represent relationships or connections between entities. Learners will study different types of graphs such as directed, undirected, weighted, and simple graphs. The unit also explains fundamental terms like degree, path, cycle, connectedness, and subgraph. Graph theory forms the foundation for modeling networks, optimizing routes, analyzing social connections, and designing efficient algorithms. Understanding graphs enables students to represent real-world problems such as communication systems, data structures, and computer networks mathematically.

## 11.2: Learning Outcomes

After studying this unit, learners will be able to:

- Define graphs and describe their key components such as vertices and edges.
- Distinguish between different types of graphs including directed, undirected, simple, and weighted graphs.

- Explain graph-related terms such as degree, path, cycle, and connectivity.
- Represent graphs using adjacency matrices and adjacency lists.
- Apply graph traversal techniques such as depth-first search (DFS) and breadth-first search (BFS).
- Use graphs to model real-world problems in computer networks, data structures, and optimization.

## 11.3 Basic Concepts of Graph Theory

Graph theory is one of the critical building blocks of discrete math and the toolset you use to represent relationships between things. This sounds more complicated than it is, but at the most basic level, a graph is a mathematical construct of a number of vertices (or nodes) connected together by edges. Such a simple construct opens many analytical possibilities for a hundred disciplines.

**Fundamental Definitions**

A graph G can be formally defined as an ordered pair G = (V, E), where V is a set of vertices and E a set of edges. Each edge joins a pair of vertices, which may be ordered (for directed graphs, where one vertex is listed first) or unordered (for undirected graphs). In Undirected Graph, edges have no orientation and they can be represented as unordered pair {u, v} where u,v ∈ V; while in Directed Graph

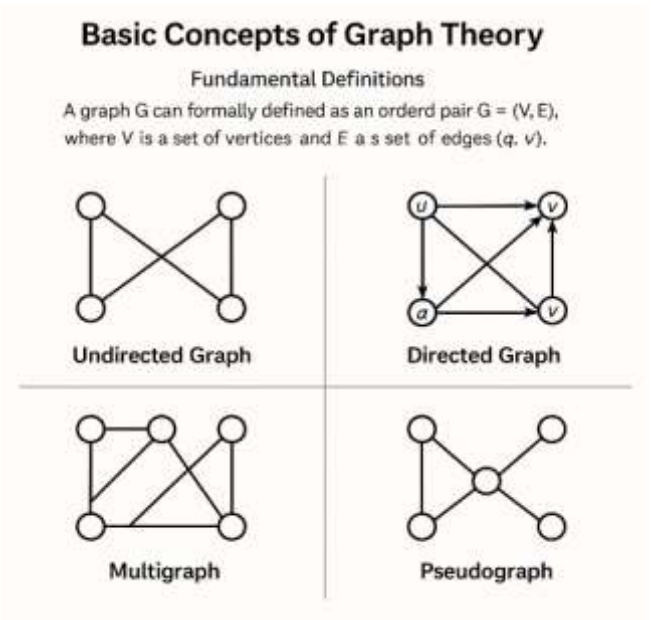



Fig: 11.1 Graph Theory

(Digraph), edges have orientation and are represented as ordered pair (u, v) indicating an edge from vertex u to vertex v. There are many different types of specialized graph structures. A simple graph has no self-edges (an edge directed from the vertex to itself) and also no multiple edges between a single pair of vertices. A multigraph allows more than one edge for a pair of vertices, but no self-loops. A pseudograph may also have multiple edges or self-loops. Such differences can be key when creating models of different real-life systems.

## 11.4: Graph Representations

There are different ways you could represent graphs computationally, and each has some advantages in some contexts. The adjacency matrix representation consists of an n×n matrix A (where n is the number of vertices) with the following entries: aij = 1 if there is an edge from vertex i to vertex j, and aij = 0 otherwise. This makes it easy and fast to look up edges, but can be wasteful for sparse graphs in terms of space. The adjacency list maintains a list of vertices that are adjacent to each vertex. This is more space-efficient for sparse graphs but slower for edge lookups. Another representation is provided via incidence matrices with rows corresponding to vertices and columns corresponding to edges, where entries indicate whether a vertex is incident to an edge. How to represent the data affects algorithm performance dramatically for various operations.

**Graph Properties and Terminology**

Graphs are characterized by several fundamental properties. The number of edges incident to a vertex is its degree. For directed graphs, we refer to in-degree (incoming edges) and out-degree (outgoing edges) separately. A path is an ordered sequence of vertices such that every two adjacent vertices in the sequence are connected by an edge. A cycle is a path that starts and ends at the same vertex. Another core concept is connectivity. A connected graph is one in which there is a path between any two vertices. Components are maximal connected subgraphs. Strong connectivity for directed graphs is defined in terms of a directed path in both directions between any two vertices. Trees are a special kind of graph, they are connected and acyclic (have no cycles), while forests are collections

of trees. The degree sequence of a graph is a list of the degrees of each vertex, usually listed in non-increasing order.

**Special Graph Types**

Many specific classes of graphs are common in theory and applications. A complete graph Kn has n vertices, all connected, and contains n(n−1)/2 edges. The vertices of a bipartite graph can be separated into two non-overlapping sets, where no edges connect vertices within the same set. The term planar applies to graphs that can be rendered on a plane such that no edges intersect. Graphs where every vertex has the same degree are known as regular graphs and we refer to multiple strongly regular graphs (known as strongly regular graphs) as maintaining an additional regularity condition on how many neighbors they have in common. Other notable examples include the Cycles {Cn}, Paths {Pn}, Wheels (a cycle with an n vertex connected to all vertices of the cycle), and the grid graphs. The pentagon structure of the Petersen graph is an excellent source of counterexample material in graph theory. The properties of these special graphs usually give information about more general graph properties and theorems.

**Graph Coloring and Independence**

The graph elements (the nodes or the arcs) are assigned a label (color) with some constraints. In vertex coloring, no adjacent vertices can have the same color. A proper vertex coloring is assigning colors to the vertex of graph G such that adjacent vertices receive different colors and the chromatic number $\chi(G)$ is the minimum number q of such colors. In a similar manner, in edge coloring, no two adjacent edges may have the same color; the chromatic index $\chi'(G)$ refers to the smallest number of basic colors that can be assigned. An independent set is a set of vertices that have no edges between them, and a clique is a set of vertices that are all pairwise adjacent. Let $\alpha(G)$, $\omega(G)$ be the independence number and the clique number of G respectively. These ideas have applications in scheduling, register — allocation in compilers, and frequency assignment problems in telecommunications.

**Matching's, Coverings, and Network Flows**

A matching in a graph is a set of edges such that no two edges share a vertex. A maximum matching has the greatest number of edges but a perfect matching covers all vertices. The matching number $\nu(G)$ is

equal to a maximum matching size. On the other hand, a vertex cover is a subset of vertices that contains at least one end point of every edge, and the vertex cover number τ(G) is the size of a smallest such cover. Network flow theory generalizes graph concepts to weighted directed graphs, and edges have capacities. ∑fxs,xd≤c where S is source vertex of network and XSink vertex of network max flow problem is hope to find the max flow can go from source flow to sink flow The augmentation-path technique is employed by the Ford-Fulkerson algorithm in order to resolve this problem. The max flow min cut theorem states that the flow exiting the source node does not exceed the capacity of the "cut," and thus gives rise to the optimum solution, solidifying this result as a foundational staple in combinatorial optimization.

## 11.5: Applications and Solved Questions

Graph theory is used in a wide variety of fields. Graphs are used to model networks, data structures and algorithms in computer science. In operations research, they describe transportation networks, project schedules, resource allocation problems, etc. Graphs are employed to analyze relationships and information flow in social networks. Graph modeling finds its use in chemistry through molecular structures and in biology via biological pathways. Advanced topics include spectral graph theory (which looks at properties of graphs using eigenvalues of related matrices), topological graph theory (which studies embeddings of graphs on surfaces), extremely graph theory (which studies maximal or minimal values of graph parameters), and random graph theory (which studies probabilistic models of graphs). The computational backbone of the vast majority of applications of graph theory is algorithms like breadth-first search, depth-first search, Dijkstra's algorithm for shortest paths, and Kruskal's and Prim's algorithms for minimum spanning trees. The work shapes shop these ideas pushes to them graph hypothesis ahead as a live and growing ield with wide rifting implications.

**Solved Examples**

**Example 1:** Graph Representation

**Problem:** Given a graph G with vertices V = {1, 2, 3, 4} and edges E = {{1,2}, {1,3}, {2,3}, {2,4}, {3,4}}, represent this graph using an adjacency matrix and adjacency list.

**Solution:** Adjacency Matrix:

1 2 3 4

1 0 1 1 0

2 1 0 1 1

3 1 1 0 1

4 0 1 1 0

Adjacency List: 1: [2, 3] 2: [1, 3, 4] 3: [1, 2, 4] 4: [2, 3]

**Example 2**: Degree Sequence

**Problem:** Find the degree sequence of the graph G with adjacency matrix:

1 2 3 4 5

1 0 1 0 1 1

2 1 0 1 0 0

3 0 1 0 1 1

4 1 0 1 0 1

5 1 0 1 1 0

**Solution:** Degree of vertex 1 = 3 (it connects with vertex 2, 4, and 5) Degree of vertex 2 = 2 (it connects with vertex 1 and 3) Degree of vertex 3 = 3 (it connects with vertex 2, 4, and 5) Degree of vertex 4 = 3 (it connects with vertex 1, 3, and 5) Degree of vertex 5 = 3 (it connects with vertex 1, 3, and 4)

We talk about the sequence of degrees in non-increasing order: [3, 3, 3, 3, 2]

**Example 3:** Graph Isomorphism

**Problem:** Judge whether the graphs below are isomorphic: G1: V = {a, b, c, d}, E = {{a,b}, {b,c}, {c,d}, {d,a}} G2: V = {1, 2, 3, 4}, E = {{1,2}, {2,4}, {4,3}, {3,1}}

**Solution** : However, both graphs have 4 vertices, 4 edges. Both graphs have degree sequence of [2, 2, 2, 2] (each vertex has exactly 2 edges). Both graphs induce 4-cycles (C4).

We can find an isomorphism from a→1, b→2, c→4, d→3.

In this mapping, each edge in G1 maps to an edge in G2: {a,b} → {1,2} {b,c} → {2,4} {c,d} → {4,3} {d,a} → {3,1}

Hence, the graphs are isomorphic**.**

**Example 4:** Connectivity and Components

**Question:** Given a graph G defined by the following vertices and edges, what are the connected components in G?

**Approach:** Depth-first search or breadth-first search

• We can move to vertices 2 and 3 starting from vertex 1.
• Begin at vertex 4 and visit vertices 5 and 6.
• Vertex 7 is isolated.

So G has three connected components.

• Component 1: {1, 2, 3}
• Component 2: {4, 5, 6}
• Component 3: {7}

**Example 5:** Euler Paths and Circuits

**Problem**: Given the following graph, does it have an Euler path or an Euler circuit? V = {a, b, c, d, e}, E = {{a,b}, {b,c}, {c,d}, {d,e}, {e,a}, {a,c}}

**Solution:** Check the degrees of each vertex: deg(a) = 3 (b, e, c) deg(b) = 2 (a,c) deg(c) = 3 (b,d,a) deg(d) = 2 (c,e) deg(e) = 2 (d,a)

All vertexes must be even degree to have an Euler circuit. In this case, both vertices a and c are of odd degree (3).

For an Euler path, there must be exactly two vertices of odd degree (the starting and ending vertices). Thus, the graph has an Euler path but not an Euler circuit since only two vertices (a and c) have odd degree. An Euler path between these points would begin at a (or c), visit all edges exactly once, and end at c (or a)

**Example 6:** Graph Coloring

**Problem:** Compute the chromatic number of the graph: V = {1, 2, 3, 4, 5}, E = {{1,2}, {4,5}} and a proper coloring.

**Answer:** We employ a greedy coloring approach:

1. Consider vertex 1 and give it color 1.
2. For vertex 2 it is an edge to vertex 1, then color 2.
3. For vertex 3, it is adjacent to vertex 1 and vertex 2 so assign color 3.
4. For output vertex 4, its adjacent vertex are vertex 1 and vertex 3, so it haal a new color as 2.
5. For vertex 5, it is adjacent to vertices 3 and 4, so give it color 1.

The coloring is as follows - 1 and 5 are colored with color 1, 2 and 4 are colored with color 2 and 3 is colored with color 3.

To confirm it's minimal, we observe that we have a complete sub-graph (K3) over vertices 1, 2, and 3 which would need at a minimum 3 colors.

Thus $\chi(G) = 3$..

**Example 7:** Bipartite Graphs

**Problem:** Is this graph bipartite: V = {1, 2, 3, 4, 5, 6}, E = {{1, 2}, {1, 4}, {1, 6}, {3, 2}, {3, 4}, {3, 6}, {5, 2}, {5, 4}, {5, 6}}?

**Solution:** A graph is bipartite when its vertices can be split into two disjoint sets with no edges between the same sets.

So we can apply a two-coloring method. A bipartite graph is one that can be colored with exactly two colors:

1. Assign color A to vertex 1.
2. You can see that vertices 2,4, and 6 are represented on the neighboring vertices of 1, so we need to assign color B.
3. Vertex 3 is connected with 2, 4 and 6 (all are colour B), assign colour A.
4. Since vertex 5 is connected to vertices 2,4 and 6 (all in color B), assign color A.

We have done two-coloring the graph as follows:

• Set A: {1, 3, 5}

• Set B: {2, 4, 6}

Because no edges connect vertices within Set A or within Set B, this graph is bipartite..

**Example 8:** Minimum Spanning Tree

**Problem:** Given the weighted graph V = {A, B, C, D, E} E = {(A,B,2), (A,C,3), (B,C,1), (B,D,3), (C,D,2), (C,E,4), (D,E,1)} (each edge formatted as (vertex1, vertex2, weight)), find a minimum spanning tree.

**Solution:** Sort all the edges in non-decreasing order of their weight and add in sorted order the next edge to the MST if it does not form a cycle (Kruskal), it is a greedy algorithm.

1. (B,C,1) - Add to MST
2. (A,B,2) - Add to MST
3. (C,D,2) - Add to MST
4. (D,E,1) - Add to MST

The edges {(B,C), (A,B), (C,D), (D,E)} form the minimum spanning tree with total weight 1+2+2+1 = 6

**Example 9:** Shortest Path

**Problem:** Find the shortest path from vertex A to all other vertices using Dijkstra's algorithm for the following weighted graph: V = {A, B, C, D, E} E = {(A,B,10), (A,C,3),(B,C,1),(B,D,2),(C,D,8),(C,E,2),(D,E,7)}

**Solution:** Initialise: dist[A]=0, dist[B]=∞, dist[C]=∞, dist[D]=∞, dist[E]=∞

Iterations:

1. Visit A: dist[B]=10, dist[C]=3
2. Visit C (nearest one with no visits): Update dist[D]=min(∞,3+8)=11, dist[E]=min(∞,3+2)=5
3. Visit B (closest unvisited): Update dist[D]=min(11,10+2)=10
4. Visit E (nearest unvisited): No updates
5. Visit D (nearest unvisited): No updates

Shortest final distances from A:

• To B: 10 (path: A→B)

• To C: 3 (path: A→C)

• To D: 10 (path: A→B→D)

• To E: 5 (path: A→C→E)

**Example 10:** Maximum Flow

**Problem:** Let ( V = {s,a,b,c,t} ; E = {(s,a,10), (s,b,10), (a,b,2), (a,c,4), (a,t,8), (b,c,9), (c,t,10)} ), each edge has the form (from, to, capacity)Problem: Find a maximum flow from source s to sink t in the following network.

**Solution:** Implement Ford-fulkerson with augmenting paths:

1. Path s→a→t with bottleneck 8: Flow becomes 8.
2. Path s→b→c→t with bottleneck 9: Total flow = 8+9=17.
3. Switch s → a → c →t with bottleneck 2: Flow become 17+2=19
4. Previous flows block
5. the path s→b→a→t. * ⊗ denotes a cut.

Hence, the maximum flow from s to t is 19 Modules.

The final flow assignment:

• Edge (s,a): Fully utilized — 10/10

• Edge (s,b): 9/10

• Edge (a,b): 0/2 (unused)

• Edge (a,c): 2/4

• Edge (a,t): 8/8 (fully used)

• Edge (b,c): 9/9 (f/used)

• Edge (c,t): 10/10 (in use)

These examples show how graph theory simplifies a variety of problems.

## 11.6: Sub graphs, Walks, Paths, and Circuits in Graph Theory

A graph is a set of vertices (or nodes) and a set of edges connecting pairs of vertices. These structures work as a powerful tool to model relationships between objects in various fields of study like computer science, chemistry, sociology and transportation networks. We usually study portions of or patterns within graphs, like subgraphs, walks, paths, and circuits. These principles the basis of how we traverse graphs and measure connectivity within a graph, determining the ease with which information, resources, or entities can navigate through intricate networks. And studying these structures gives us a lot of properties of the whole graph, like reach ability between nodes, shortest paths, cyclic properties, etc. In this complete guide, we are going to discuss about the formal definitions, properties and usages of this elemental graph theory concepts with plenty of solved examples demonstrating the usage of these concepts in solving the problems.

**Sub graphs: Definitions and Types**

The sub graph is basically a graph inside a graph In concrete terms, a sub graph H of a graph G is a graph with vertex set V (H) as a subset of V(G) and edge set E (H) as a subset of E(G) such that every edge in E (H) connects two vertices that are inV(H). We have the following note worth the mention regarding specialized types of sub graph: When there is a said subset of vertices of g, and all the edges present in g on this subset of vertices, the sub graph formed is known as the induced subgraph. Note that a spanning sub graph must have all the vertices of the original graph, but can have fewer edges. Note that a maximal sub graph with respect to some property is one to which no vertex or edge can be added without violating the property. A clique consists of a complete sub graph where there is an edge connecting every pair of distinct vertices. Sub graph identification is the method of isolating certain graph objects from a bigger graph, which is very important in graph analysis since the purpose can be a sub graph that contains certain properties or represents certain subsystems. In social network analysis, for example, densely connected sub graphs can indicate commModuley structures, and in computational biology,

certain sub graphs of protein interaction networks can represent functional modules.

**Walks: Sequential Traversal through Graphs**

An undirected walk in a graph is a finite alternating sequence (v(e1)e(v1)e2(v2)e(v3)e(v4)e(v5)e(v6)( More formally, a walk from vertex $v_0$ to vertex $v_n$ is the alternating sequence of vertices and edges $v_0$, $e_1$, $v_1$, $e_2$, $v_2$, …, $e_n$, $v_n$, where each edge $e_i$ links the two endpoints $v_{i-1}$ and $v_i$. The length of a walk is counted as the number of edges it uses. If we have a walk where we need the starting vertex to be the same as the ending vertex, we can call that a closed walk, while otherwise if that is not needed we can call it an open walk. Another important property of walks is that it allows repeated visits to both vertices and edges, thus being less restrict than paths. Particularly, this freedom allows for walks to be appropriate for modeling situations where it is permitted to pass through previously visited spots or connections as in Markov-chains or in ascertaining the potential travel itineraries of agents in networks. Walks represent the basic traversal types; paths and circuits are simply constrained versions of walks. Also, if two vertices are connected by walks then they are somehow reach-able, although not on the efficient path, graph connectivity is also analyzed with the help of walks..

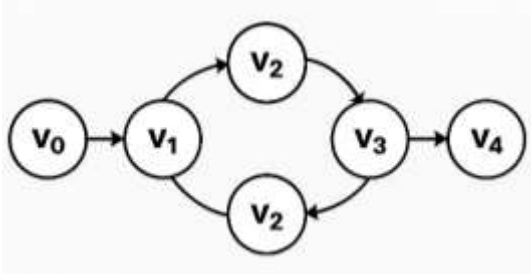



Fig: 11.2 Walk

**Paths: Direct Connections Without Repetition**

A walk in which no vertex is repeated is a path. More formally, a path from vertex $v_0$ to vertex $v_n$ is a walk $v_0$, $e_1$, $v_1$, $e_2$, $v_2$,..., $e_n$, $v_n$ with all $v_i$ distinct. This limitation makes paths helpful in locating direct routes from one node to another in a graph. The length of a path (and also of a walk) is the number of edges it contains. In graph theory, the distance between two vertices in an undirected graph can be calculated using the shortest path between them, which is also referred to as the geodesic path and is often used in applications like

determining the most efficient way to navigate between two vertices (for example, when calculating routing protocols in computer networks or navigation systems). A graph is called connected if there is a path between every pair of distinct vertices, a property ensuring that the entire network can be accessed from any point in it. (Graph algorithms woudn't work without paths — one example is Dijkstra's algorithm for internet routing; another is depth-first search for traversing the structure of a graph.) In directed graphs, the presence of a path from the vertex u to the vertex v does not guarantee that there exists a path from the vertex v to the vertex u, hence the concept of strong connectivity and weak connectivity. They provide a framework for examining the dynamics of information processes, resource allocation, and access within structures.

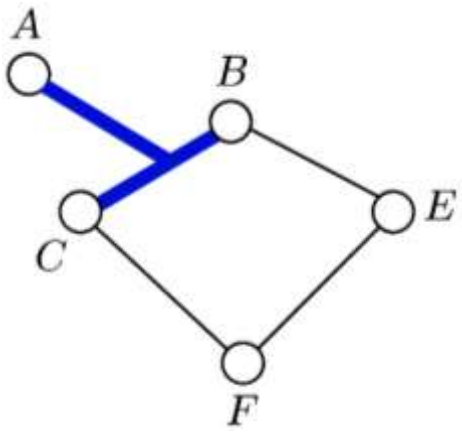



Fig: 11.3 Path

**Circuits: Cyclic Structures in Graphs**

A circuit (or cycle) is a walk which is closed (the first and last vertices are the same), and no edge is repeated. Or we can say a circuit is a closed path (only the first/last vertex is repeated). The existence of circuits in a graph is essential for the cyclic structures and their properties in the graph. Such a circuit-free graph is referred to as acyclic (trees are a particular example of acyclic graphs). We call the length of a circuit the number of edges (or, equivalently, vertices) it contains. Of particular interest are Hamiltonian circuits (which visit each of the vertices exactly once, before returning to the original vertex) and Eulerian circuits (which traverse each edge exactly once). Circuits are important in network design as they provide redundancy and alternative paths, and in chemistry where it represents cyclical molecular structures, and when it comes to scheduling problems, circuits may represent inefficiencies or deadlocks. The detection and

analyzation of circuits are a key part of many graph algorithms as they are used in checking planarity of a graph, finding the strongly connected components and many optimization problems like the traveling salesman problem.

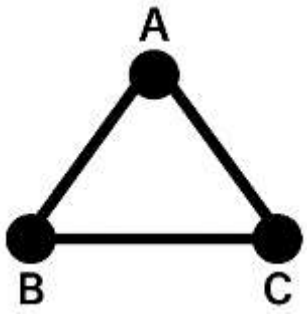



Fig: 11.4 Circuit

## 11.7: Common Properties and Interrelationships

Divider If walks are used to travel along the graph, then Sub-graphs are any portion of a graph that can be used for a wider construction of the Graph traversing algorithm. Because it has so-called "path (def.)": attack that forbids repeating vertices. In the same way, all circuits are closed walks, but not all closed walks a circuit if they repeat the edges. Graphs are characterized by the existence of paths between vertices and the absence of circuits, while trees and other anti-cyclic structures are characterized by the lack of circuits. This is a direct consequence of the concept of (graph) diameter, which is the longest shortest path between any two vertices in the graph. In a network, the number of distinctive paths between its vertices may reflect the connection redundancy or resilience. The concepts extend to weighted graphs with associated edge costs or weights weighted walks, weighted paths, weighted circuits, etc — where the total weight along the sequence is now the more critical measure. All these properties combined allow us to study various network features, such as reversibility, connectivity, cyclicity, etc., and they all are crucial for understanding the structural or functional properties of complex networks in many domains.

**Applications and Algorithmic Approaches**

Sub graphs, walks, paths, and circuits are theoretical concepts with wide-ranging practical applications in various fields that exploit certain properties of these structures to address real-life challenges. For example, in computer networking, Dijkstra's and Bellman-Ford algorithms are shortest path algorithms used to compute optimal

routing. Path and subgraph identification for detecting com Moduleies and influence through social network analysis. Circuit and path optimization forms the basis for transportation engineering for optimal routing and scheduling of transportation systems. In the analysis of gene regulatory networks, bioinformatics applies path analysis to explore some biological behaviors. Web crawlers explore the web with systematic walk algorithms. Identification of molecular structures in the chemo informatics domain is done using specific subgraph pattern recognition. This is particularly useful in applications such as electronic design and power grid management where circuit analysis is indispensable. Diverse algorithmic methods have been designed to study these types of structures in an efficient way: breadth-first search for shortest paths, depth-first search for path and circuit detection, dynamic programming for same optimal path problems, and dedicated algorithms that each focus on specific problems such as Floyd-warshall for all-pairs shortest paths. Path and circuit related problems represent more complex problems and hence the majority of them are NP-hard (like finding the circuits of Hamiltonian) while some others such as finding circuits Eulerian are polynomial problems that can be explored in polynomial time. Research into the ongoing design of evermore efficient algorithms to study these structures of a graph continues to this day in the fields of graph theory and computer science.

## 11.8: Solved Examples

**Example 1:** Identifying Sub graphs

Consider a graph G with vertex set V(G) = {a, b, c, d, e} and edge set E(G) = {(a,b), (b,c), (c,d), (d,e), (e,a), (a,c), (b,d)}. Let's identify various sub graphs:

**Solution:**

1. A sub graph $H_1$ with $V(H_1)$ = {a, b, c} and $E(H_1)$ = {(a,b), (b,c)} is a valid sub graph of G.
2. The induced sub graph $H_2$ on vertices {a, b, c} includes all edges from G connecting these vertices: $E(H_2)$ = {(a,b), (b,c), (a,c)}.
3. A spanning sub graph $H_3$ has $V(H_3) = V(G)$ = {a, b, c, d, e} and a subset of edges, say $E(H_3)$ = {(a,b), (b,c), (c,d), (d,e), (e,a)} (forming a cycle).

4. The vertices {a, c, d} with edges {(c,d), (a,c)} form another valid sub graph $H_4$.

**Example 2:** Analyzing Walks

In graph G with vertices {$v_1$, $v_2$, $v_3$, $v_4$, $v_5$} and edges {($v_1$,$v_2$), ($v_2$,$v_3$), ($v_3$,$v_4$), ($v_4$,$v_5$), ($v_5$,$v_1$), ($v_1$,$v_3$), ($v_2$,$v_4$)}, analyze the following walks:

**Solution:**

1. $W_1$: $v_1$, ($v_1$,$v_2$), $v_2$, ($v_2$,$v_3$), $v_3$, ($v_3$,$v_4$), $v_4$
   - This is a walk of length 3 from $v_1$ to $v_4$
   - Since no vertex is repeated, it is also a path
2. $W_2$: $v_1$, ($v_1$,$v_2$), $v_2$, ($v_2$,$v_3$), $v_3$, ($v_3$,$v_1$), $v_1$, ($v_1$,$v_5$), $v_5$
   - This is a walk of length 4 from $v_1$ to $v_5$
   - It is not a path because $v_1$ appears twice
3. $W_3$: $v_2$, ($v_2$,$v_4$), $v_4$, ($v_4$,$v_3$), $v_3$, ($v_3$,$v_2$), $v_2$
   - This is a closed walk of length 3 (starts and ends at $v_2$)
   - It's not a circuit because edge ($v_4$,$v_3$) doesn't exist in G

**Example 3:** Finding All Paths Between Two Vertices

In the graph G with V(G) = {a, b, c, d} and E(G) = {(a,b), (b,c), (c,d), (a,c), (b,d)}, find all possible paths from vertex a to vertex d.

**Solution:**

1. Path $P_1$: a, (a,b), b, (b,c), c, (c,d), d (length 3)
2. Path $P_2$: a, (a,b), b, (b,d), d (length 2)
3. Path $P_3$: a, (a,c), c, (c,d), d (length 2)

There are three distinct paths from a to d, with the shortest paths $P_2$ and $P_3$ both having length 2.

**Example 4:** Eulerian Circuits

Determine if the following graph G has an Eulerian circuit. V(G) = {$v_1$, $v_2$, $v_3$, $v_4$, $v_5$} and E(G) = {($v_1$,$v_2$), ($v_2$,$v_3$), ($v_3$,$v_4$), ($v_4$,$v_5$), ($v_5$,$v_1$), ($v_1$,$v_3$), ($v_3$,$v_5$), ($v_5$,$v_2$), ($v_2$,$v_4$)}.

Solution: For a graph to have an Eulerian circuit, every vertex must have an even degree (number of incident edges).

- deg($v_1$) = 3 (edges to $v_2$, $v_3$, $v_5$)
- deg($v_2$) = 3 (edges to $v_1$, $v_3$, $v_5$)
- deg($v_3$) = 3 (edges to $v_1$, $v_2$, $v_4$)
- deg($v_4$) = 2 (edges to $v_3$, $v_5$)
- deg($v_5$) = 3 (edges to $v_1$, $v_3$, $v_4$)

Since vertices $v_1$, $v_2$, $v_3$, and $v_5$ have odd degrees, this graph does not have an Eulerian circuit.

**Example 5:** Hamiltonian Paths and Circuits

Determine if the cycle graph $C_5$ with vertices {1, 2, 3, 4, 5} and edges {(1,2), (2,3), (3,4), (4,5), (5,1)} contains a Hamiltonian path and a Hamiltonian circuit.

Solution: A Hamiltonian path visits each vertex exactly once. The path 1, 2, 3, 4, 5 is a Hamiltonian path in $C_5$.

A Hamiltonian circuit visits each vertex exactly once and returns to the starting vertex. The circuit 1, 2, 3, 4, 5, 1 is a Hamiltonian circuit in $C_5$.

In fact, $C_5$ is itself a cycle, so it naturally contains a Hamiltonian circuit.

**Example 6:** Finding Shortest Paths

In a weighted graph G with V(G) = {a, b, c, d, e} and weighted edges {(a,b,3), (b,c,2), (c,d,5), (d,e,1), (e,a,4), (a,c,7), (b,d,6), (a,d,12)}, find the shortest path from vertex a to vertex d.

Solution: Possible paths from a to d:

1. a → b → c → d: weight = 3 + 2 + 5 = 10
2. a → c → d: weight = 7 + 5 = 12
3. a → b → d: weight = 3 + 6 = 9
4. a → d: weight = 12
5. a → e → d: Not possible as there's no direct edge from e to d

The shortest path is a → b → d with a total weight of 9.

**Example 7:** Circuit Detection in Directed Graphs

Consider a directed graph G with V(G) = {1, 2, 3, 4, 5} and directed edges {(1,2), (2,3), (3,1), (2,4), (4,5), (5,2)}. Identify all circuits in this graph.

Solution:

1. Circuit $C_1$: 1 → 2 → 3 → 1 (length 3)
2. Circuit $C_2$: 2 → 4 → 5 → 2 (length 3)
3. Circuit $C_3$: 1 → 2 → 4 → 5 → 2 → 3 → 1 (length 6)

There are other circuits that can be derived by combining or extending these basic circuits.

**Example 8:** Connectivity in Graphs

Determine if the following graph G is connected. V(G) = {a, b, c, d, e, f} and E(G) = {(a,b), (b,c), (d,e), (e,f), (a,d)}.

Solution: To check connectivity, we need to verify if there's a path between every pair of vertices.

- Path from a to c: a → b → c

- Path from a to e: a → d → e
- Path from a to f: a → d → e → f
- Path from b to d: b → a → d
- Path from b to e: b → a → d → e
- Path from b to f: b → a → d → e → f
- Path from c to d: c → b → a → d
- Path from c to e: c → b → a → d → e
- Path from c to f: c → b → a → d → e → f
- Path from d to b: d → a → b
- Path from d to c: d → a → b → c
- Path from d to f: d → e → f
- Path from e to a: e → d → a
- Path from e to b: e → d → a → b
- Path from e to c: e → d → a → b → c
- Path from f to a: f → e → d → a
- Path from f to b: f → e → d → a → b
- Path from f to c: f → e → d → a → b → c

Since there exists a path between every pair of vertices, the graph G is connected.

**Example 9:** Sub graph Isomorphism

Determine if the graph H with V(H) = {1, 2, 3, 4} and E(H) = {(1,2), (2,3), (3,4), (4,1)} is isomorphic to a sub graph of G with V(G) = {a, b, c, d, e, f} and E(G) = {(a,b), (b,c), (c,d), (d,e), (e,f), (f,a), (a,c), (c,e), (e,a)}.

Solution: H is a cycle graph with 4 vertices ($C_4$). To find if it's isomorphic to a subgraph of G, we need to find a cycle of length 4 in G.

One such cycle is a → c → e → a, but this has only 3 vertices. Another cycle is a → b → c → d → e → f → a, but this has 6 vertices.

Let's check other possible 4-vertex cycles:

- a → b → c → a: Not a 4-vertex cycle (only 3 vertices)
- a → c → d → e → a: This is a 4-vertex cycle

The mapping f(1) = a, f(2) = c, f(3) = d, f(4) = e establishes an isomorphism between H and the sub graph of G induced by vertices {a, c, d, e}. Therefore, H is isomorphic to a sub graph of G.

**Example 10:** Bipartite Graph Analysis

Determine if the following graph G is bipartite. V(G) = {1, 2, 3, 4, 5} and E(G) = {(1,2), (2,3), (3,4), (4,5), (5,1), (1,3)}.

Solution: A graph is bipartite if its vertices can be divided into two disjoint sets such that no two vertices within the same set are adjacent. Let's try to partition the vertices:

- Put vertex 1 in set A
- Then vertices 2, 3, and 5 must go in set B (since they're adjacent to 1)
- But vertices 2 and 3 are adjacent, and both are in set B, which violates the bipartite property

Alternatively, we can check if the graph contains any odd-length cycles. The cycle $1 \rightarrow 3 \rightarrow 2 \rightarrow 1$ has length 3 (odd), confirming that G is not bipartite.

**Example 11:** Graph Coloring and Paths

Given a graph G with V(G) = {a, b, c, d, e} and E(G) = {(a,b), (b,c), (c,d), (d,e), (e,a), (a,c), (b,d)}, find the chromatic number and a proper coloring of G.

**Solution:** The chromatic number is the minimum number of colors needed for a proper vertex coloring.

Let's attempt to color G:

- Assign color 1 to vertex a
- Vertex b is adjacent to a, so assign color 2
- Vertex c is adjacent to both a and b, so assign color 3
- Vertex d is adjacent to b and c, so assign color 1
- Vertex e is adjacent to a and d, so assign color 2

This gives us a proper coloring using 3 colors. To verify this is minimal, note that vertices a, b, and c form a triangle (clique of size 3), requiring at least 3 colors. Therefore, the chromatic number of G is 3.

**Example 12:** Distances and Eccentricity

For the graph G with V(G) = {1, 2, 3, 4, 5} and E(G) = {(1,2), (2,3), (3,4), (4,5), (1,3), (1,5)}, calculate: a) The distance between each pair of vertices b) The eccentricity of each vertex c) The radius and diameter of G

Solution: a) Distances:

- d(1,2) = 1, d(1,3) = 1, d(1,4) = 2, d(1,5) = 1
- d(2,1) = 1, d(2,3) = 1, d(2,4) = 2, d(2,5) = 2

- d(3,1) = 1, d(3,2) = 1, d(3,4) = 1, d(3,5) = 2
- d(4,1) = 2, d(4,2) = 2, d(4,3) = 1, d(4,5) = 1
- d(5,1) = 1, d(5,2) = 2, d(5,3) = 2, d(5,4) = 1

b) Eccentricity (maximum distance from a vertex to any other vertex):

- e(1) = max{d(1,j)} = 2
- e(2) = max{d(2,j)} = 2
- e(3) = max{d(3,j)} = 2
- e(4) = max{d(4,j)} = 2
- e(5) = max{d(5,j)} = 2

c) Radius (minimum eccentricity) = 2 Diameter (maximum eccentricity) = 2

**Example 13:** Walks of Specific Length

In the graph G with V(G) = {a, b, c, d} and E(G) = {(a,b), (b,c), (c,d), (d,a), (a,c)}, determine: a) The number of walks of length 2 from a to c b) The number of walks of length 3 from a to a

Solution: a) Walks of length 2 from a to c:

1. a → b → c
2. a → d → c

There are 2 walks of length 2 from a to c.

b) Walks of length 3 from a to a:

1. a → b → c → a
2. a → c → d → a
3. a → d → a → b (not valid as (d,a,b) contains repeated vertex a)
4. a → c → a → d (not valid as (c,a,d) contains repeated vertex a)

There are 2 valid walks of length 3 from a to a.

**Example 14:** Independent Sets and Dominating Sets

For the graph G with V(G) = {1, 2, 3, 4, 5} and E(G) = {(1,2), (2,3), (3,4), (4,5), (5,1), (1,3), (2,4)}, find: a) A maximum independent set b) A minimum dominating set

Solution: a) An independent set is a set of vertices where no two vertices are adjacent.

- {1, 4} is an independent set
- {2, 5} is an independent set

To verify these are maximum, note that adding any other vertex would create an edge within the set. Therefore, {1, 4} and {2, 5} are maximum independent sets of size 2.

b) A dominating set is a set of vertices such that every vertex in the graph is either in the set or adjacent to some vertex in the set.

- Set {1, 3} dominates all vertices: 1 dominates 2, 5; 3 dominates 2, 4
- Set {1, 4} dominates all vertices: 1 dominates 2, 3, 5; 4 dominates 3, 5
- Set {2, 4} dominates all vertices: 2 dominates 1, 3; 4 dominates 3, 5

These are all minimum dominating sets of size 2.

**Example 15:** Path and Circuit Optimization

In a weighted graph G with V(G) = {a, b, c, d, e} and weighted edges {(a,b,2), (b,c,3), (c,d,1), (d,e,4), (e,a,5), (a,c,7), (b,d,6), (c,e,2)}, find: a) The shortest path from a to e b) The minimum-weight Hamiltonian circuit, if one exists

Solution: a) Possible paths from a to e:

1. a → b → c → d → e: weight = 2 + 3 + 1 + 4 = 10
2. a → b → c → e: weight = 2 + 3 + 2 = 7
3. a → c → d → e: weight = 7 + 1 + 4 = 12
4. a → c → e: weight = 7 + 2 = 9
5. a → e: weight = 5

The shortest path is a → e with weight 5.

b) Hamiltonian circuits (visit each vertex exactly once and return to start):

1. a → b → c → d → e → a: weight = 2 + 3 + 1 + 4 + 5 = 15
2. a → b → c → e → d → a: Not valid (no edge from d to a)
3. a → b → d → c → e → a: weight = 2 + 6 + 1 + 2 + 5 = 16
4. a → c → b → d → e → a: Not valid (no edge from c to b)
5. a → c → e → d → b → a: Not valid (no edge from d to b)
6. a → e → c → b → d → a: Not valid (no edge from e to c)
7. a → e → c → d → b → a: Not valid (no edge from d to b)
8. a → e → d → c → b → a: Not valid (no edge from e to d)

The minimum-weight Hamiltonian circuit is a → b → c → d → e → a with weight 15.

**Example 16:** Planarity and Graph Drawing

In this task you determine whether the complete graph $K_5$ is planar or not (if it is not planar you identify a Kuratowski sub graph).

**Solution:** What is a planar graph? A graph is planar if it can be drawn in the plane without any edge crossing. The so-called Kuratowski's

theorem states that a graph G is planar iff it does not contain a subdivision of $K_5$ or of $K_{3,3}$.

Where $K_5$ is a complete graph on 5 points or each pair of points is connected with edges. By Kuratowski's theorem, $K_5$ is non-planar itself. And since $K_5$ is one of those Kuratowski graphs, it is a Kuratowski subgraph of itself. This shows that $K_5$ is non-planar.

**Example 17:** Graph Matrix Representations

For the graph G with V(G) = {1, 2, 3, 4} and E(G) = {(1,2), (2,3), (3,4), (4,1), (1,3)}, give: a) the adjacency matrix b) The incidence matrix c) Use the adjacency matrix to find the number of walks of length 2 from vertex 1 to vertex 3

Solution: a) Adjacency matrix A:

```
  | 1 2 3 4
---+--------
 1 | 0 1 1 1
 2 | 1 0 1 0
 3 | 1 1 0 1
 4 | 1 0 1 0
```

b) Incidence matrix M (labeling edges $e_1 = (1,2)$, $e_2 = (2,3)$, $e_3 = (3,4)$, $e_4 = (4,1)$, $e_5 = (1,3)$):

```
  | e1 e2 e3 e4 e5
---+---------------
 1 | 1  0  0  1  1
 2 | 1  1  0  0  0
 3 | 0  1  1  0  1
 4 | 0  0  1  1  0
```

c) To find the number of walks of length 2 from vertex 1 to vertex 3, we compute $A^2$ and look at the entry $A^2[1,3]$:

$A^2 = A \times A =$

```
  | 1 2 3 4
---+--------
 1 | 3 1 2 1
 2 | 1 2 1 2
 3 | 2 1 3 1
 4 | 1 2 1 2
```

Therefore, there are 2 walks of length 2 from vertex 1 to vertex 3. These walks are:

1. $1 \rightarrow 2 \rightarrow 3$

2. $1 \rightarrow 4 \rightarrow 3$

**Example 18:** Eulerian Paths

You are given a graph G with the following vertices and edges: V(G) = {a, b, c, d, e} and E(G) = {(a,b), (b,c), (c,d), (d,e), (e,a), (a,c)}. You need to decide if it possesses an Eulerian path or not.

**Solution:** For a graph to be Eulerian (i.e. contain Eulerian path), two vertices have to be odd degree and all others must be even degree.

Calculating the degrees:

• deg(a) = 3 (to b, c, e)

• deg(b) = 2 (edges to a, c)

• deg(c) = 3 (edges to a, b, d)

• deg(d) = 2 (edges to c, e)

• deg(e) = 2 (edges to a, d)

Note that the degree of vertices a and c is 1 (odd), and the degree of all other vertices is even. Hence, this graph has an Eulerian path. The trail has to start at a or c and end at the other.

An Eulerian path of this kind is: a → b → c → d → e → a → c

**Example 19:** Connectivity and Cut Vertices

Given the graph G with V(G) = {1,2,3,4,5,6} and E(G) = {(1,2),(2,3),(3,4),(4,5),(5,6),(6,1),(2,5)}, find: a) The cut vertices (articulation points) b) The bridges (cut edges)

• When removing vertex 1: The remaining graph is connected

• Vertex 2 removed: remainder of graph is still connected

• Removed vertex 3: {1, 2, 5, 6}, {4}

• For vertex 4: component: {1, 2, 3, 5, 6} and {}

• When we remove vertex 5 the remaining graph is still connected

• Pruning vertex 6 out: The resulting graph is still connected.

So, the cut vertices are 3 and 4.

b) A bridge is an edge, that when deleted increasing connected components.

• Deleting edge (3,4): The graph gets partitioned into the two components {1, 2, 3, 5, 6} and {4}

Hence, edge (3,4) is a bridge.

**Example 20:** Graph Characterization

Describe the following graphs by their characteristics: a) A complete graph $K_6$ b) A cycle graph $C_8$ c) A wheel graph $W_5$ d) A complete bipartite graph $K_{3,4}$

**Solution**: a) Complete graph $K_6$:

• Each of the 6 vertices is connected to the other vertices
• Has 6(6-1)/2 = 15 edges
• Diameter = 1, radius = 1
• Chromatic number = 6
• Has Hamiltonian circuits
• Not bipartite (has odd cycles)
• All induced subgraphs are complete graphs

b) Cycle graph $C_8$:

• Contains 8 vertices placed in a cycle

## 11.9: Types of Graphs

**Graphs in Mathematics**

A graph is a mathematical representation of a set of objects where some pairs of the objects are connected by links. The graph is made up of vertices (or nodes) and edges that link those vertices. Then, analyze data with Graphs which are widely used in fields such as computer science, physics, biology, social sciences to study relationships as well as in engineering. Graph theory is a branch of mathematics studying graphs, which was first introduced in the 18th century by Leonhard Euler when he managed to solve the famous Seven Bridges of Königsberg problem. The problem asked whether it was possible to walk through the city crossing each of its seven bridges just once — Euler proved it was impossible, and in doing so, laid the groundwork for graph theory.

**Undirected Graphs**

Undirected Graph: A graph in which edges have no direction. The edge (u, v) is the same edge as the edge (v, u), so it is the relationship of two vertices which is reciprocal. Undirected edges are commonly represented by lines in draw without arrows.

**Example 1:** Friendship Network

Let's take an example of a friendship network, vertices represent people and edges represent friendships between them. If Bob is a friend of Alice, then Alice is a friend of Bob — hence, this relationship is undirected.

This prompt describes a friendship network among 5 people (A, B, C, D, E) and the following friendships:

• A is friends with B and C
• B is a friend of A, C, and D

• C has friends A, B and E
• D is friends with B and E
• E is friends with C and D

To represent this as an adjacency matrix:

| A B C D E
--+---------
A | 0 1 1 0 0
B | 1 0 1 1 0
C | 1 1 0 0 1
D | 0 1 0 0 1
E | 0 0 1 1 0

**Example 2:** Determining if a Graph is Connected

Two vertices are connected if there is a path between them.

Let G be the undirected graph with vertices {1,2,3,4,5} and edges {(1,2),(1,3),(2,4),(3,5)}.

To determine whether G is connected:

1. Start at vertex 1
2. From vertex 1, we can go to vertex 2 and 3
3. From vertex 2, we may take vertix 4
4. → You can go to vertex 5 from the vertex 3.
5. The graph is connected because every vertex is reachable from vertex 1

**Example 3:** Finding Degrees in an Undirected Graph

A vertex degree is the number of edges attached to a vertex.

Consider the undirected graph G = (V, E), where V = {A, B, C, D} and E = {(A,B), (A,C), (B,C), (B,D), (C,D)}:

Step 1: Find the degree of each vertex.

• Vertex A connects to B and C, hence deg(A) = 2
• Vertex B is connected to A, C, and D, thus degree(B) = 3
• As vertex C, there are three possible edges of connection (to A, B, and D), therefore degree(C) = 3
• Vertex D has edges to B and C = d(D) = 2

**Example 4:** Euler Path and Circuit

Euler path : a path that visits every edge exactly once. An Euler circuit is an Euler path that begins and ends at the same vertex.

For an undirected graph:

• An Euler path exists iff exactly zero or two vertices have odd degree

• An Euler circuit exists iff all the vertices have even degree

Take the graph G whose vertices are {A, B, C, D} and whose edges are {(A,B), (A,C), (B,C), (B,D), (C,D)}:

• Degree(A) = 2 (even)

• Degree(B) = 3 (odd)

• Degree(C) = 3 (odd)

• Degree(D) = 2 (even)

Because (from the given graph) there are exactly two vertices (B and C) with an odd degree, thus this graph has an Euler path but no Euler circuit. A possible Euler path is: A → B → C → D → B → A → C

**Directed Graphs**

We can say that we have a directed graph (or digraph) when an edge possesses orientations. In a directed graph, the edge (u, v) indicates a connection going from vertex u to vertex v (but not from v to u), and in drawings, we commonly draw a directed edge as an arrow

**Example 5:** Web Pages and Hyperlinks

Imagine a vastly simplified model of web pages and hyperlinks. Each web page is a vertex, and a hyper link from page A to page B is modeled as a directed edge from page A to page B.

Consider pages {P1, P2, P3, P4} with the following hyperlinks:

• P1 has links to P2 and P3

• P2 has a link to P4

• P3 has links to P1 and P4

• P4 has a link to P2

The adjacency matrix would be:

```
  | P1 P2 P3 P4
---+-----------
P1 | 0  1  1  0
P2 | 0  0  0  1
P3 | 1  0  0  1
P4 | 0  1  0  0
```

**Example 6:** Finding In-degrees and Out-degrees

In a directed graph:

- In-degree of vertex : no. of incoming edges
- The out-degree of a vertex is the number of edges that direct away from it

So for the above web page example:

- P1: in-degree = 1, out-degree = 2
- P2 has an in-degree of 2 and an out-degree of 1.
- P3: in-degree = 1, out-degree = 2
- P4: in-degree = 2, out-degree = 1

**Example 7:** Topological Sorting

A topological sort of a directed acyclic graph (DAG) is a linear ordering of its vertices such that for every directed edge from vertex u to vertex v, vertex μ appears before vertex v in the ordering.

Now, consider the directed graph that represents prerequisite courses:

- Vertices: {Calculus I, Calculus II, Linear Algebra, Differential Equations, Real Analysis}
- Edges: {(Calc I —> Calc II), (Calc I —> Lin Alg), (Calc II —> Diff Eq), (Calc II —> Real Analysis), (Lin Alg —> Diff Eq)}

For example a valid topological sort would be: Calculus I → Linear Algebra → Calculus II → Differential Equations → Real Analysis

This orders the courses in a way that all prerequisites are taken before the courses that depend on them.

**Weighted Graphs**

It is a graph that has an weight or cost with each edges of the graph. Weights can be distances or costs, capacities, or any other measure defined between vertices

**Example 8:** Road Network

Imagine a road network, where each vertex is a city and each edge is a road connecting those cities. The edges are weighted with the distance between cities.

Towns: {A, B, C, D} Distances (in kilometers) between roads:

- A to B: 150
- A to C: 200
- B to C: 100
- B to D: 250
- C to D: 150

The weighted adjacency matrix would be:

| | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 150 | 200 | ∞ |
| B | 150 | 0 | 100 | 250 |
| C | 200 | 100 | 0 | 150 |

D | ∞ 250 150 0

(∞ indicates no direct connection)

**Example 9:** Shortest Path Using Dijkstra's Algorithm

Dijkstra finds the shortest path from a vertex as an entry vertex to all other vertices in the weighted graph, where the weight cannot be negative.

Let's calculate the shortest path from city A to all other cities using the road network from Example 8:

Initialize:

• Distance to A = 0 (source)

• Distance to B, C, D = ∞ (unknown)

• Set of unvisited nodes = {A, B, C, D}

Visit node A:

o DISTANCE TO B = 150, DISTANCE TO C = 200

UPDATE

o Unvisited = {B, C, D}

Mark node B as visited (link closest unvisited path)

o Update: D = min(200, 150+100) = 200, nothing new

o Update: Distance to D = min(∞, 150+250) = 400

o Unvisited = {C, D}

To visit closest unvisited node C:

o Update: Distance to D = min(400, 200+150) = 350

o Unvisited = {D}

Now visit node D (the only left node):

o Done

Shortest distances to A (final):

• To B: 150 km (path: A → B)

• To C: 200 km (path: A → C)

• To D: 350 km (A → C → D route

**Example 10:** Minimum Spanning Tree Using Kruskal's Algorithm

A minimum spanning tree (MST) is a subset of the edges of a connected, undirected, weighted graph that connects all vertices with the minimum possible total edge weight. Using the road network from Example 8, let's find the MST using Kruskal's algorithm:

1. Sort all edges by weight:
    - B to C: 100
    - A to B: 150
    - C to D: 150

- A to C: 200
- B to D: 250

2. Add edges in ascending order of weight, skipping those that create cycles:
   - Add B-C (100)
   - Add A-B (150)
   - Add C-D (150)

The MST consists of edges {(B,C), (A,B), (C,D)} with a total weight of 400. A minimum spanning tree (or MST) is a subset of the edges of a connected, undirected, weighted graph that connects all vertices together, without any cycles and with the minimum possible total edge weight. We are going to find the MST for the road network from Example 8 using Kruskal's algorithm:

1. Sort all edges by weight:

- B to C: 100
- A to B: 150
- C to D: 150
- A to C: 200
- B to D: 250

2 Sort edges by weight, and add them (skipping if it would create a cycle):

- Add B-C (100)
- Add A-B (150)
- Add C-D (150)

Note: the maximum spanning tree being edges {(B,C), (A,B), (C,D)}, Total weight: 400.

**Bipartite Graphs**

A bipartite graph is a graph whose vertices can be partitioned into two sets U and V so that every edge connects a vertex in U to a vertex in V.

**Example 11:** Students and Courses Let us imagine a situation with students and courses. A student can enroll in many courses and many students may enroll into one course. You can model this with a bipartite graph, where one set of vertices corresponds to students and the other corresponds to courses.

Students: {S1, S2, S3, S4} Courses: {C1, C2, C3} Edges (enrollments):

• S1 is enrolled in C1 and C2

• S2 is enrolled in C1 and C3
• S3 is enrolled in C2
S4 is enrolled in C2 and C3
Thus, to check whether it is bipartite, we can try to color the vertices in such a way that no two adjacent vertices have the same color:
Paint S1, S2, S3, S4 redAssign blue paint color to all courses: {C1, C2, C3}
This is a bipartite graph since every edge connects a red vertex to a blue vertex.

**Example 12:** Maximum Bipartite Matching

A matching in a bipartite graph is a set of edges without common vertices. A maximum matching is a matching with the largest possible number of edges. Using the students and courses example:

1. Initialize an empty matching M = {}
2. Try to match S1:
    - Match S1 to C1: M = {(S1,C1)}
3. Try to match S2:
    - C1 is already matched with S1
    - Match S2 to C3: M = {(S1,C1), (S2,C3)}
4. Try to match S3:
    - Match S3 to C2: M = {(S1,C1), (S2,C3), (S3,C2)}
5. Try to match S4:
    - C2 is already matched with S3
    - C3 is already matched with S2
    - No available match for S4

Now try to improve the matching: 6. Find an augmenting path starting from S4:

- S4 → C2 → S3 (S3 is matched to C2)
- S3 is unmatched to any other course, so this path ends

7. Invert the matching along this path:
    - Unmatch (S3,C2)
    - Match (S4,C2)
    - M = {(S1,C1), (S2,C3), (S4,C2)}
8. Try to match S3:
    - Match S3 to another available course
    - There's no available course, so S3 remains unmatched

The maximum matching is {(S1,C1), (S2,C3), (S4,C2)} with 3 edges.
A matching in a bipartite graph is an edge set with no vertex in

common. A maximum matching is a matching that has as many edges as possible.

Using the students and courses example:

1. Let M = {} be an empty matching
2. Try to match S1:
   - Match M = {(S1,C1)}
3. Try to match S2:
   - S1 already matched with C1
   - Pair S2 with C3: M = {(S1,C1), (S2,C3)}
4. Try to match S3:
   - Perform the matching of S3 to C2: M = {(S1,C1), (S2,C3), (S3,C2)}
5. Try to match S4:
   - Already mapped C2 with S3
   - C3 is already paired with S2
   - No available match for S4

Now 6—Just improve the matching. And now start with S4 & find an augmenting path:

• S4 → C2 → S3 (C2 is matched to S3)

• S3 is not comparable to any other course, hence this way ends

7 Reverse the matching along this path:

o Unmatch (S3,C2)

o Match (S4,C2)

o M = {(S1,C1), (S2,C3), (S4,C2)}

1 Try to match S3:

Match S3 into another available course

o No course available, S3 is still unmatched

Maximum matching has 3 edges, and is {(S1,C1), (S2,C3), (S4,C2)}

**Complete Graphs**

A complete graph is a graph where an edge is an available connection between all members. A complete graph with n vertices denoted by $K_n$ has n(n-1)/2 edges.

Example 13: Complete Graph Properties

Consider $K_4$, a complete graph with 4 vertices {A, B, C, D}.

1. Number of edges = 4(4-1)/2 = 6 edges
2. The edges are: {(A,B), (A,C), (A,D), (B,C), (B,D), (C,D)}
3. Each vertex has degree 3 (connected to all other vertices)
4. The adjacency matrix is:

| A B C D
--+---------
A | 0 1 1 1
B | 1 0 1 1
C | 1 1 0 1
D | 1 1 1 0

**Example 14:** Coloring a Complete Graph

The chromatic number of a graph is the smallest number of colors we can assign to its vertices so that no two adjacent vertices share the same color.

If n is the number of vertices in the complete graph $K_n$, then the chromatic number is n, because each vertex is adjacent to each other vertex.

For $K_4$:

• Vertex A: Color 1
• Vertex B: Color 2
• Vertex C: Color 3
• Vertex D: Color 4

All of the four vertices are adjacent to one another, and as such all four must be of distinct color, hence chromatic number = 4.

**Trees and Forests**

A tree is a connected, acyclic, undirected graph. A forest is a forest is a disjoint union of trees.

**Example 15:** Tree Properties

Let T be a tree with vertices {A, B, C, D, E, F} and edges {(A,B), (A,C), (B,D), (B,E), (C,F)}.

Properties of this tree:

1. Number of vertices = 6
2. No. of edges=5 (n-1 always for a tree with n vertices)
3. the tree is connected (between any two vertices there is a path)
4. If we remove any edge that will disconnect the tree
5. Any addition of an edge forms cycle

**Example 16:** Depth-First Search (DFS) on a Tree

DFS is a graph traversal algorithm. Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures.

Source Example 15 (with A root node)

1. Visit A, mark as visited
2. Go to child B: mark as visited
3. Visit child D, mark visited (D has no children)

4. I Go back to B, visit child E and mark as visited (E has no children)
5. Go back to A, go to child C, mark as visited
6. Child F, visit, mark as visited F (no children)

The DFS traversal of tree would be : A B D E C F

**Example 17:** Breadth-First Search (BFS) on a Tree

The BFS algorithm begins at a root node and explores all neighbors at the current depth prior to proceeding on to nodes at the next depth level.

Let us use the tree of Example 15, by setting A to be the root:

1. Visit A, mark as visited
2. Store all kids of A: B, C (as visited)
3. Perform DFS on all children of B: D, E (visited)
4. Explore and mark as visited all children of C: F

BFS traversal order: A -> B -> C -> D -> E -> F

**Planar Graphs**

An embedded planar graph is a planar graph in which no edges cross each other. In other words, the drawing can be made on a flat surface with no wrappers crossing.

**Example 18:** Testing for Planarity

Euler's formula can be used to prove graph planarity: $v - e + f = 2$, where v refers to the number of graph vertices or nodes, e refers to the number of edges, and f refers to the number of faces or regions

(which also includes the outer face). Take a graph G with vertices {A, B, C, D} and edges {(A,B), (A,C), (A,D), (B,C), (C,D)}.

To test for planarity:

1. Without crossing any edges:
   - Arrange A, B, C, D roughly in a square
   - Draw edges (A,B),(A,C)(A,D),(B,C),(C,D)
   - All this can be done without crossing any edges

2 Count:

   - Vertices (v) = 4
   - Edges (e) = 5
   - Faces (f) = 3 (two inner faces and one outer face)
3. Use Euler's formula: v − e + f = 4 − 5 + 3 = 2 ✓

This concludes that the graph is planar and the formula has been met.

**Example 19:** Dual Graph

The dual of a planar graph G is another graph G*, in which:

• Each face of G corresponds to a vertex in G*

• For every edge of G there exists an edge of G*

• Two vertices in G∗ are adjacent if the corresponding faces in G have an edge in common.

Let P be planar graph example 18.

• The F1 (outer face), F2 (the inner face formed by A, B, C) and F3 (the inner face formed by A, C, D).

The dual graph G has vertices {F,, F, F*F3}

• The set of edges of G: {(F1,F2), (F1,F2), (F1,F3), (F1,F3), (F2,F3*)}

In the dual graph, some vertices have multiple edges between them, since some faces share multiple edges in the original graph.

Special Graphs

**Example 20:** Petersen Graph

The Petersen graph is special with 10 vertices and 15 edges. That means it is commonly used in graph theory as a counterexample.

Some characteristics of the Petersen graph:

1. It has 10 vertices, usually depicted as a pentagon with an inscribed pentagram
2. It is a 3-regular graph since each vertex has degree 3.
3. It is not planar (cannot exists in a plane without edge crossings)
4. It is of girth (length of the shortest cycle) 5

5. Its vertex-transitive (looks the same viewed from any vertex)
6. It does not have a Hamiltonian cycle (a cycle that visits each vertex exactly once)

We can try to build a Hamiltonian cycle (to prove it does not have one):

1. Bizarrely, these arrive to the problem of TSP where you begin at any vertex and attempt to visit all vertices exactly once and return to the starting point.
2. This cannot be because of the unique structure of the Petersen graph
3. Any such path will either miss some vertices or it must take a dip and re-visit vertices

The Petersen graph is an important counterexample for many graph theory conjectures and is a fundamental object in the field of graph theory.

Graph theory offers a wealth of tools for modeling and solving numerous problems in various fields. The Graphs we studied undirected, directed, weighted, bipartite, complete, trees, planar graphs have their own unique properties and applications. We have shown in the solved examples how these types of graphs can be analysed, manipulated and used in real-life situations. From computing shortest paths in road networks to scheduling prerequisite coursework, from modelling social networks to optimising complex routing problems, graphs present a rich structure for encoding relations and solving problems efficiently. Graph theory will continue to be an essential area of mathematics and play a significant role in our understanding of how we interact with one another and how we contend with new delivery systems for our needs in our increasingly complex world of global interconnectedness.

## Check Your Progress

1. Define a graph. Explain the difference between directed and undirected graphs with suitable examples.
   .....
   .....
   .....
   .....

.....

2. What are paths and cycles in a graph? Explain with examples.

.....

.....

.....

.....

.....

## 11.10: Summary

This unit covered the fundamental concepts of graph theory and its essential role in mathematical modeling and computation. Learners gained an understanding of graph types, representations, and key terminologies such as vertex, edge, path, and cycle. The unit emphasized the importance of connectivity and traversal algorithms like BFS and DFS in solving network and path-finding problems. Graphs are powerful tools for representing relationships among objects, making them useful in areas such as computer networks, operating systems, and data organization. Mastery of graph theory provides the analytical foundation for studying advanced topics like shortest path algorithms, spanning trees, and graph coloring.

## 11.11: Exercises

**Multiple Choice Questions**

1. A graph is defined as:
   a) A set of vertices only
   b) A set of edges only
   c) A set of vertices and edges connecting them
   d) A matrix representation of data
   Answer: c) A set of vertices and edges connecting them
2. In an undirected graph, the degree of a vertex is:
   a) The number of vertices connected to it
   b) The number of edges incident on it
   c) Always equal to two
   d) The total number of vertices in the graph
   Answer: b) The number of edges incident on it
3. A graph with no cycles is called:
   a) Connected graph

b) Complete graph
c) Acyclic graph
d) Weighted graph
Answer: c) Acyclic graph

4. Which of the following represents a traversal technique in graphs?
   a) Bubble sort
   b) Depth-first search
   c) Binary search
   d) Merge sort
   Answer: b) Depth-first search
5. The sum of the degrees of all vertices in a graph is equal to:
   a) Twice the number of edges
   b) The total number of vertices
   c) Half the number of edges
   d) The number of connected components
   Answer: a) Twice the number of edges

**Descriptive Questions**

1. Define a graph and explain its types with suitable examples.
2. Describe the difference between directed and undirected graphs with applications.
3. Explain the terms degree, path, and cycle in the context of graphs.
4. Discuss the adjacency matrix and adjacency list representation of a graph with examples.
5. Explain the applications of graph theory in computer science and information systems.

## 11.12: References and Suggested Readings


- Rosen, Kenneth H. *Discrete Mathematics and Its Applications*, 8th Edition, McGraw-Hill Education, 2019.
- Grimaldi, Ralph P. *Discrete and Combinatorial Mathematics: An Applied Introduction*, 5th Edition, Pearson Education, 2003.
- Kolman, Bernard, Busby, Robert C., and Ross, Sharon. *Discrete Mathematical Structures*, 6th Edition, Pearson Education, 2018.

- Deo, Narsingh. *Graph Theory with Applications to Engineering and Computer Science*, Prentice Hall of India, 2017.
- West, Douglas B. *Introduction to Graph Theory*, 2nd Edition, Pearson Education, 2001.
- Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., and Stein, Clifford. *Introduction to Algorithms*, 4th Edition, MIT Press, 2022.

# Unit 12: Matrix Representation of Graphs, Directed Graphs

**Structure**

12.1 Introduction
12.2 Learning Outcome
12.3 Matrix Representation of Graphs and Directed Graphs
12.4 Graph Algorithms
12.5 Summary
12.6 Exercises
12.7 References and Suggested Readings

## 12.1: Introduction

This unit focuses on the matrix representation of graphs and the concept of directed graphs, which play a significant role in computer science, particularly in data organization and algorithm design. Matrix representations such as the adjacency matrix and incidence matrix provide systematic ways to describe and analyze graph structures mathematically. Learners will understand how matrices can store information about the connections between vertices and how these matrices can be used to perform operations on graphs efficiently. The unit also introduces directed graphs (digraphs), where edges have a direction that indicates one-way relationships. Understanding these representations helps in the analysis of computer networks, flow systems, and graph-based data models.

## 12.2: Learning Outcomes

After studying this unit, learners will be able to:

- Define adjacency and incidence matrices and explain their significance in graph representation.
- Construct adjacency and incidence matrices for undirected and directed graphs.
- Differentiate between symmetric and asymmetric matrices in graph contexts.
- Describe the properties and structure of directed graphs.

- Use matrix representations to determine paths, degrees, and connectivity in graphs.
- Apply matrix-based graph representations in computer networks, database relationships, and algorithmic analysis.

## 12.3: Matrix Representation of Graphs and Directed Graphs

**Introduction to Graph Representations**

Graph: A graph consists of vertices (or nodes) and edges connecting these vertices. Graphs are invaluable means of modeling relationships between objects in a wide variety of domains such as computer science, engineering, social sciences, and biology. Matrix representations: Out of the multiple representations available, matrix representations have an advantage, because they can encode the structure of graphs as a matrix that lends itself readily to computation. Graph matrices translate the topological structure of a graph into algebraic form. This transformation on the surface OF it allows us to use the powerful linear algebra theory to study graphs through as well as enables us to calculate all sorts of properties in graphs using graph theory Graph theory is a field of mathematics that studies the interrelationship between nodes and edges, making it important for a variety of applications and it has two main matrix representations that are useful for certain applications — adjacency matrix and incidence matrix

**Adjacency Matrix Representation**

Adjacency Matrix The adjacency matrix is one of the most popular graphs matrix representation. For a graph G with n vertices, the adjacency matrix A is an n × n matrix such that A[i,j] indicates the relationship between the vertices i and j. If the graph is undirected, then A[i,j] = A[j,i] = 1; if there is no edge then A[i,j] = A[j,i] = 0. For undirected graphs, this gives us a symmetric adjacency matrix. In the case of weighted graphs, instead of representing an edge as 1 Adjacency matrix with unweighted edges for weighted graphs For directed graphs, however, this matrix for the graph is not necessarily symmetric. In other words, if there is a directed edge from vertex i to vertex j, then A[i,j] = 1, but A[j,i] = 1 is not a given and would only happen if there was also a directed edge from vertex j to vertex i.

Dashed properties of adjacency matrix:

• For simple graphs (no self-loops) all the diagonal elements A[i,i] = 0

• The adjacency matrix is symmetric in undirected graphs.

• For an undirected graph, the total number of edges is half the sum of all elements in adjacency matrix (i.e., ∑all elements in adjacency matrix = 2 * edges)

• The i-th row (or column) is the degree of the i-th vertex in undirected graphs

• For directed graphs, the row-sum of the i-th row corresponds to the out-degree of vertex i, while the column-sum of the i-th column corresponds to the in-degree

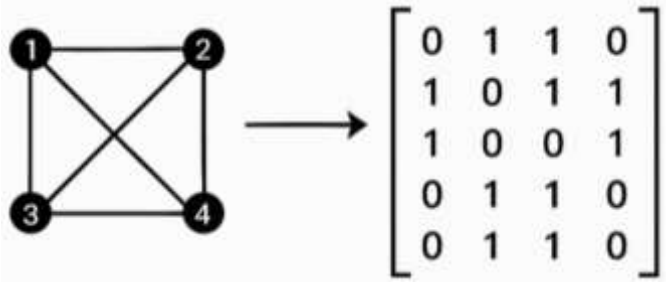



Fig: 12.1 Adjacency matrix

**Incidence Matrix Representation**

The adjacency matrix describes relations between vertices, and the incidence matrix B describes relations between vertices and edges. An incidence matrix of a graph having n vertices and m edges is an n × m matrix. For an undirected graph, B[i,j] = 1 iff vertex i is incident on edge j; otherwise B[i,j] = 0. For each edge, it has an entry in two vertex columns of incidence matrix, thus each 1 appears once in the incidence matrix. In directed graphs, we can define the incidence matrix with signed entries: B[i,j] = 1 if vertex i is the head of edge j, B[i,j] = −1 if vertex i is the tail of edge j, and B[i,j] = 0 if vertex i is not incident on edge j.

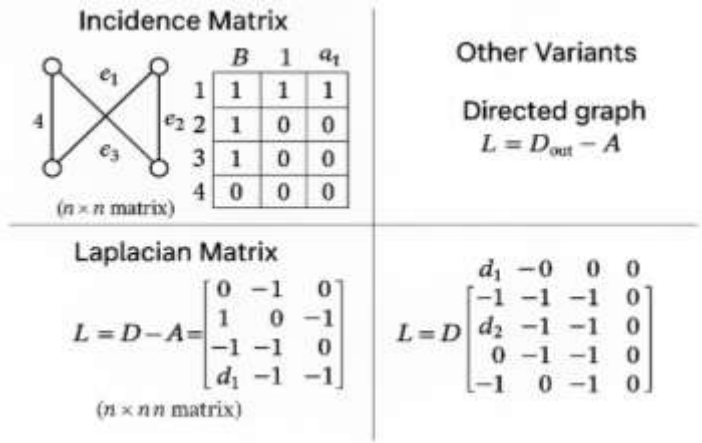



Fig: 12.2 Incidence Matrix

Among the key properties of the incidence matrix are:

• Column sum equals 2 for undirected graphs (each edge connects 2 vertices)

• Signed representation yields a zero column sum for directed graphs

• The incidence matrix multiplied with its own transpose (B•B^T) gives a matrix pertaining to the graph Laplacian

• The incidence matrix's row space is perpendicular to the graph Laplacian null space

Laplacian Matrix and Other Variants

The Laplacian matrix L of a graph is constructed with the adjacency matrix and the degree matrix D (a diagonal matrix whis diagonal entries D[i,i] holds the degree of vertex i). For an undirected graph, we have L = D − A, where A is the adjacency matrix. The Laplacian matrix has many useful properties for analyzing the connectivity and structure of the graph:

• For undirected graphs it is symmetric and positive semi-definite

• The dimension of the eigenspace where λ = 0 is the number of connected components of the graph

• The smallest non-zero eigenvalue (the spectral gap) yields information about how well-connected the graph is

The second smallest eigenvalue's eigenvector (the Fiedler vector) is used for spectral graph partitioning.

In special cases for directed graphs, L = D_out - A, where D_out is a diagonal matrix with out-degrees of every node as the ith diagonal entry. But this matrix does not satisfy some of the good properties of the undirected Laplacian, like symmetry and positive semi-definiteness.

**Applications of Matrix Representations**

There are many arrangements of graphs in the form of matrices which provide many applications of the terms in the field of graph theory and others:

**Path Finding and Connectivity**

- Paths — the n-th power of the adjacency matrix A^n encodes information about the number of paths of length n between vertices
- Connectivity, shortest path (using the Floyd-War shall algorithm), cycles detection -- are all possible with matrix operations

- The matrix exponential e^A has entries that correspond to the sum of paths of each the length, from vertex to vertex

**Spectral Graph Theory**

- The adjacency and Laplacian matrix eigenvalues and eigenvectors reveal structural properties of graphs.
- Spectral clustering involves partitioning graphs with eigenvectors.
- The eigenvalue multiset of a graph is a graph invariant.

**Network Analysis**

- Matrix operations can be used to compute centrality measures
- Many commModuley detection algorithms are matrix-based
- Graph random walks can be described using the transition probability matrix

## 12.4: Graph Algorithms

• Matrix representations can be used in graph matching and isomorphism testing

• Matrix formulation to solve maximum flow problems

• Countless graph optimization problems can be expressed in the language of matrix optimization problems

**Computational Considerations**

When realizing graph algorithms via matrix representations, several computation-related aspects must be taken into consideration:

**Space Complexity**

• Adjacency matrices take $O(n^2)$ space irrespective of number of edges and thus not efficient for sparse graphs

• Incidence matrices require O(nm) space which is again not efficient for large, sparse graphs

• For sparse graphs, it may be more intuitive to use alternative representations (e.g., adjacency lists or compressed sparse row (CSR) format).

**Time Complexity**

- Testing adjacency between two vertices takes O(1) time with adjacency matrices
- To find all neighbors of a vertex with adjacency matrices takes O(n) time, no matter how many neighbors the vertex has

- Matrix multiplication complexity depends on the algorithms used and whether the matrices are sparse Numerical Stability
- For large graphs, computing eigenvalues and eigenvectors can produce numerical issues
- The use of preconditioning techniques for graph matrix linear system solvers may be warranted
- Nearest to singularity are pseudoinverses on almost disconnected graphs, which require special care

In this article, we will discuss 20 solved example problems to demonstrate the concepts and applications of matrix representation of Undirected and directional graphs.

**Solved Examples**

**Example 1:** Adjacency Matrix for an Undirected Graph

Let G be an undirected graph with 4 vertices {1, 2, 3, 4} and edges {(1,2), (1,3), (2,3), (3,4)}.

Solution: For this graph, the adjacency matrix A is:

A = [0 1 1 0]
[1 0 1 0]
[1 1 0 1]
[0 0 1 0]

here, A being symmetric as there are no self-loops, and the diagonal elements all being 0. The total sum of all elements is 8, which is the same of 2 × the number of edges (4).

**Example 2:** Adjacency Matrix for a Directed Graph

Let G be a directed graph with a vertex set |V| = {1,2,3,4} and directed edges |E| = {(1→2), (2→3),(3→1), (4→3)}.

Answer : The adjacency matrix A corresponding to this directed graph is as follows :

A = [0 1 0 0]
[0 0 1 0]
[1 0 0 0]
[0 0 1 0]

The matrix is not symmetric. Row 1 sums up to your 1, which reveals that vertex 1 has an out-degree of 1. Column 1 sums up to 1, indicating vertex 1 in-degree 1.

**Example 3:** Incidence Matrix for an Undirected Graph

Now, let's construct an incidence matrix for the same undirected graph as in Example 1, which has the vertex set {1, 2, 3, 4} and edge set {(1,2), (1,3), (2,3), (3,4)}.

Answer: Give labels to the edges: $e_1 = (1,2)$, $e_2 = (1,3)$, $e_3 = (2,3)$, $e_4 = (3,4)$ The incidence matrix B is:

B = [1 1 0 0]
[1 0 1 0]
[0 1 1 1]
[0 0 0 1]

Every column has exactly two 1's in it, which correspond to the two vertices that an edge connects.

**Example 4:** Incidence Matrix for a Directed Graph

Given the directed graph in Example 2, with |V| = 4, V = {1, 2, 3, 4} and E = {(1→2), (2→3), (3→1), (4→3)}, obtain the signed incidence matrix.

**Solution:** $e_1 = (1 \to 2)$, $e_2 = (2 \to 3)$, $e_3 = (3 \to 1)$, $e_4 = (4 \to 3)$ The incidence matrix B signed is:

B = [ 1 0 -1 0]
[-1 1 0 0]
[ 0 -1 1 -1]
[ 0 0 0 1]

For each directed edge the source vertex gets 1 and the destination vertex gets -1.

**Example 5:** Determining Path Existence Using Matrix Powers

Consider the directed graph in Example 2. Find out whether there exists a path of length 2 from vertex 1 to vertex 3.

1) Solution: We calculate A 2 to obtain the number of paths of length 2.:

$A^2$ = [0 0 1 0]
[1 0 0 0]
[0 1 0 0]
[1 0 0 0]

That said, $A^2[1,3] = 1$, meaning exactly one 2⟩length path exists from vertex 1 to vertex 3. This path is 1→2→3.

**Example 6:** Finding the Degree Matrix

Find the degree matrix D for the undirected graph in Example 1.

Solution: It has the following degrees: deg 1 = 2, deg 2 = 2, deg 3 = 3, deg 4 = 1 Thus, the degree matrix D is:

D = [2 0 0 0]
[0 2 0 0]
[0 0 3 0]
[0 0 0 1]

**Example 7:** Constructing the Laplacian Matrix

Example 1 (Undirected graph): Given is the undirected graph as shown in figure 1. Find the Laplacian matrix L for this graph.

Solution: As L = D − A, we have already given the degree matrix in Example 6 and the adjacency matrix in Example 1

L = [2 0 0 0] [0 1 1 0] [2 -1 -1 0]
[0 2 0 0] - [1 0 1 0] = [-1 2 -1 0]
[0 0 3 0] [1 1 0 1] [-1 -1 3 -1]
[0 0 0 1] [0 0 1 0] [ 0 0 -1 1]

**Example 8:** Weighted Adjacency Matrix

Imagine there is an undirected weighted graph with 3 vertices and edges {(1,2):5,(2,3):2,(1,3):7}.

Solution: The adjacencies weighted matrix will be:

A = [0 5 7]
[5 0 2]
[7 2 0]

**Example 9:** Adjacency Matrix Properties

Using the adjacency matrix in Example 1, find: a) The sum of all matrix elements b) The row and column sums c) The matrix trace

Solution: a) Sum of all elements = 8 (2 * number of edges [4]) b) Row sums: [2, 2, 3, 1] - denote degrees of vertices Column sums: [2, 2, 3, 1] - same as row sums for undirected graphs c) Trace of A = A[1,1] + A[2,2] + A[3,3] + A[4,4] = 0 + 0 + 0 + 0 = 0

**Example 10:** Paths of Different Lengths

Given the directed graph in Example 2, find $A^3$and check whether there exists a path of length 3 from vertex 1 to vertex 1.

**Solution:**

$A^3 = A^2 \times A$ = [0 0 1 0] × [0 1 0 0] [1 0 0 0]
[1 0 0 0] [0 0 1 0] = [0 1 0 0]
[0 1 0 0] [1 0 0 0] [0 0 1 0]
[1 0 0 0] [0 0 1 0] [0 0 1 0]

($A^3[1,1] = 1$), therefore there exists a path from vertex 1 to vertex 1 of length 3: 1→2→3→1.

**Example 11:** Detecting Cycles Using Matrix Powers

Use the directed graph from example 2 and state if it has cycles. If yes, determine how long they are.

**Solution:** We have already computed $A^2$, $A^3$, so we can check that $A^3[1,1] = 1$, which means that there is a cycle through vertex 1 of lenght 3. This cycle is 1→2→3→1. Why is that? Well we can check this since we know that A^2_2,1 = 1 and A^2_3,2 = 1, we also know A_1,3 = 1 that completes the cycle.

**Example 12:** Graph with Self-Loops and Multi-Edges

For such a case, we have a graph with 3 vertices, self-loops on vertex 1 and 3, and there are two parallel edges between 1 and 2. Build its adjacency matrix.

Solution: Self-loops would make diagonal entries equal to 1, multi-edges to increase the respective entries in the adjacency matrix.:

A = [1 2 0]
[2 0 0]
[0 0 1]

**Example 13:** Complete Graph

Adjacency Matrix for $K_4$: Construct the adjacency matrix for a complete graph $K_4$ with 4 vertices.

**Solution:** A complete graph has every vertex connected with all other vertices:

A = [0 1 1 1]
[1 0 1 1]
[1 1 0 1]
[1 1 1 0]

All off-diagonal elements are 1, and the diagonal elements are 0 (assuming no self-loops).

**Example 14:** Bipartite Graph

Let G be the bipartite graph with vertex sets X = {1, 2} and Y = {3, 4, 5}, and edges {(1,3), (1,4), (2,4), (2,5)}. If it is bipartite, create an adjacency matrix and demonstrate how the bipartite structure is represented.

**Solution:** Adjacency Matrix Solution: The adjacency matrix is:

A = [0 0 1 1 0]
[0 0 0 1 1]

[1 0 0 0 0]
[1 1 0 0 0]
[0 1 0 0 0]

The matrix is almost block diagonal, where the upper-row and lower-column parts of the corresponding diagonal contain all edges between source and target set, and the blocks give zeroes, which denote no edges contained in the same set of either X or Y..

**Example 15:** Directed Acyclic Graph (DAG)

Think of a directed acyclic graph with 4 vertices and edges {(1→2), (1→3), (2→4), (3→4)}. Build its adjacency matrix and show that it is acyclic using powers of the matrix.

Solution: The adjacency matrix is:

A = [0 1 1 0]
[0 0 0 1]
[0 0 0 1]
[0 0 0 0]

To check that it's acyclic, we calculate powers of A:

$A^2$ = [0 0 0 2]
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]

This means $A^3$ and all higher powers will have all zeros along the diagonal, validating that there are no cycles.

**Example 16:** Disconnected Graph

Suppose G=({1, 2, 3, 4, 5}, {(1,2), (3,4), (4,5)}) is disconnected with 2 components: C1={1,2}, C2={3,4,5}. Derive its adjacency matrix and find the block structure.

**Solution:** Correction: The adjacency matrix appears to be block diagonal:

A = [0 1 0 0 0]
[1 0 0 0 0]
[0 0 0 1 0]
[0 0 1 0 1]
[0 0 0 1 0]

The matrix contains in the diagonal two blocks, corresponding to the two connected components of the graph.

**Example 17:** Computing Graph Invariants from Matrices

Give the: (a) Number of triangles (b) Determinant of the adjacency matrix (c) Eigenvalues of the adjacency matrix For the undirected graph in Example 1

**Solution:** a) Number of triangles is trace($A^3$)/6. Computing $A^3$

:$A^3$ = [2 2 2 1]

[2 2 2 1]

[2 2 4 0]

[1 1 0 1]

trace($A^3$) = 2 + 2 + 4 + 1 = 9, so we have 9/6 = 1.5 ≈ 1 triangles (we round down as we cannot have a fraction of triangles).

b) det(A)==0 (this is actually frequent concerning many graph adjacency matrices)

c) Eigenvalues: $\lambda_1 \approx -1.48$, $\lambda_2 \approx -1$, $\lambda_3 \approx 0.31$, $\lambda_4 \approx 2.17$

**Example 18:** Random Walk Transition Matrix

For the graph in Example 1 (considered as an undirected graph) find the random walk transition matrix P, where P[i,j] is the moving probability from vertex i to vertex j in one step of the random walk.

**Solution:** The transition matrix $P = D^{-1}A$, where D is the degree matrix and A is the adjacency matrix Solution::

P = [1/2 1/2 0 0 ]

[1/2 0 1/2 0 ]

[1/3 1/3 0 1/3]

[ 0 0 1 0 ]

This means that P[3,1] = 1/3 means when you are in the vertex 3, the probability of moving to vertex 1, in one time step, equals to 1/3.

**Example 19:** Graph Coloring and Eigenvalues

Let's take a 4-cycle (square) graph with {1, 2, 3, 4} vertices and {(1,2), (2,3), (3,4), (4,1)} edges. Get its adjacency matrix, find the eigenvalues, and connect them with the chromatic number.

**Solution:** The adjacency matrix is:

A = [0 1 0 1]

[1 0 1 0]

[0 1 0 1]

[1 0 1 0]

The characteristic values are $\lambda_1 = -2$, $\lambda_2 = 0$, $\lambda_3 = 0$, $\lambda_4 = 2$.

For bipartite graphs, the eigenvalues occur in pairs symmetric about 0. The chromatic number of a bipartite graph is 2, which is compatible with the most negative eigenvalue is -2.

**Example 20:** Spectral Clustering

Let us consider the following adjacency matrix of an undirected graph.

A = [0 1 1 0 0 0]
[1 0 1 0 0 0]
[1 1 0 1 0 0]
[0 0 1 0 1 1]
[0 0 0 1 0 1]
[0 0 0 1 1 0]

Do spectral clustering to find the working clusters in this gait.

**Solution:**

1. Compute the degree matrix:

D = diag(2, 2, 3, 3, 2, 2)

2. Compute the Laplacian matrix L = D - A:

L = [ 2 -1 -1 0 0 0]
[-1 2 -1 0 0 0]
[-1 -1 3 -1 0 0]
[ 0 0 -1 3 -1 -1]
[ 0 0 0 -1 2 -1]
[ 0 0 0 -1 -1 2]

3. Calculate eigenvectors of L, and we get the second smallest eigenvector (Fiedler vector) to be approximately $v_2$ = [-0.41, -0.41, -0.25, 0.25, 0.41, 0.41]
4. Split the vertices (non-negative/positive) as sign of corresponding entry in $v_2$: i:{1,2,3} (negative), i:{4,5,6} (positive)

This clustering distinguishes two natural commModuleies in the graph.

Graphs are often represented using matrices which provide a strong framework for analyzing not only the graph contents but also its structure properties. Each of these matrices encapsulates different properties about the graph topology which can be utilized for diverse applications. For undirected graphs, these matrices are often positive semi-definite, and hence amenable to spectral analysis, among other

properties. What we've seen with the matrices are very nice properties (inundated into our discussion) with very simple graphs, and for directed graphs they don't all hold and will lose a couple of the properties but they do still reflect some properties.

## Check Your Progress

1. Explain the adjacency matrix and incidence matrix representation of a graph with an example.

   .....
   .....
   .....
   .....
   .....

2. Define a directed graph. Discuss its properties and how it differs from an undirected graph.

   .....
   .....
   .....
   .....
   .....

## 12.5: Summary

This unit discussed the mathematical representation of graphs using matrices, emphasizing how adjacency and incidence matrices capture relationships among vertices. Learners examined how these matrices simplify the computational processing of graphs in algorithms. The unit also explored directed graphs, where edges indicate specific directions between nodes, representing dependency or flow relationships. Understanding matrix representation allows for efficient storage, manipulation, and traversal of graph data. These concepts form a foundation for advanced applications in computer science such as network routing, dependency analysis, and optimization problems.

## 12.6: Exercises

**Multiple Choice Questions**

1. The adjacency matrix of an undirected graph is always:
   a) Upper triangular
   b) Symmetric
   c) Skew-symmetric
   d) Diagonal
   Answer: b) Symmetric
2. In a directed graph, the adjacency matrix is:
   a) Always symmetric
   b) Always diagonal
   c) Generally asymmetric
   d) Always zero
   Answer: c) Generally asymmetric
3. The entry $a_{ij}$ in an adjacency matrix represents:
   a) The weight of vertex i
   b) The number of edges between vertex i and j
   c) The color of vertex i
   d) The degree of the graph
   Answer: b) The number of edges between vertex i and j
4. The incidence matrix of a graph shows the relationship between:
   a) Vertices and degrees
   b) Vertices and edges
   c) Edges and weights
   d) Paths and vertices
   Answer: b) Vertices and edges
5. A directed graph (digraph) differs from an undirected graph because:
   a) It has no vertices
   b) Its edges have a specific direction
   c) It has equal in-degree and out-degree
   d) It cannot be represented by a matrix
   Answer: b) Its edges have a specific direction

**Descriptive Questions**

1. Define an adjacency matrix and explain how it represents an undirected graph.
2. Construct the adjacency and incidence matrices for a directed graph of your choice.

3. Explain how matrix representation of graphs helps in determining connectivity and path existence.
4. Differentiate between directed and undirected graphs with examples of their matrix forms.
5. Discuss the applications of matrix representations and directed graphs in computer science.

## 12.7: References and Suggested Readings


- Grimaldi, Ralph P. *Discrete and Combinatorial Mathematics: An Applied Introduction*, 5th Edition, Pearson Education, 2003.
- Deo, Narsingh. *Graph Theory with Applications to Engineering and Computer Science*, Prentice Hall of India, 2017.
- West, Douglas B. *Introduction to Graph Theory*, 2nd Edition, Pearson Education, 2001.
- Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., and Stein, Clifford. *Introduction to Algorithms*, 4th Edition, MIT Press, 2022.

# Block 4: Graph Theory

## Unit 13: Tree and its properties

**Structure**

13.1 Introduction
13.2 Learning Outcome
13.3 Tree and Its Properties, Rooted Tree, Binary Trees, Spanning Tree, Fundamental Circuits
13.4 Tree Traversals
13.5 Spanning Tree
13.6 Fundamental Cycles and Cut Sets
13.7 Summary
13.8 Exercises
13.9 References and Suggested Readings

## 13.1: Introduction

This unit introduces the concept of trees, which are special types of graphs that play a crucial role in computer science and mathematics. A tree is a connected acyclic graph that provides a hierarchical structure for data organization and representation. Learners will study different types of trees such as rooted trees, binary trees, and spanning trees, along with their properties and applications. Trees are widely used in data structures, database indexing, network routing, and hierarchical modeling. Understanding tree properties helps students analyze relationships, optimize algorithms, and efficiently store and retrieve data in computational systems.

## 13.2: Learning Outcomes

After studying this unit, learners will be able to:

- Define a tree and explain its structural properties.
- Differentiate between general trees, rooted trees, and binary trees.
- Identify important properties of trees such as height, degree, and path length.
- Understand the concept of leaf nodes, internal nodes, and levels of a tree.
- Apply tree concepts in representing hierarchical data and decision-making processes.
- Relate trees to computer science applications such as data structures, parsing, and database indexing.

## 13.3: Tree and Its Properties, Rooted Tree, Binary Trees, Spanning Tree, Fundamental Circuits

### Trees and Their Properties

A tree is a basic data structure in computer science and mathematics that describes hierarchical relationships in an elegant way. A tree is a connected, acyclic graph made of nodes (vertices) connected by edges. Trees as a hierarchical data structure separate from linear data structures (like arrays or linked lists) where the data is organized differently hierarchically are best examples of data where items have parent-child relationships. Tree are used in various areas such as file systems, database indexing, syntax parsing in compilers, network routing algorithms, artificial intelligence and organizational structures.

### Basic Definitions and Properties

Tree is technically defined as a connected, acyclic undirected graph. Let's take a look at some essential properties:

1. **Now the definition of tree states that:** Any two vertices are connected by exactly one path. This property guarantees that the whole tree is reachable.
2. **Acyclic** A tree is a cyclical which means that there is no path which starts and ends at the same vertex without repeating any edge.
3. **Minimally Connected**: A Graph that is minimally connected means that if any edge of the graph is removed from the tree then it will be disconnected.
4. **Relation between Edges count:** A tree with N vertices will always have N-1 edges.
5. **Vertices with degree:** 1 (connected to only a single other vertex) is called a leaf node or an external node.
6. **Internal Nodes:** A vertex of degree > 1 is called Internal nodes.
7. **Compared to the Height and Depth:** Each tree has a height which is the length of longest path from root to leaf. The distance from the root of a node is its depth.
8. **Subtree:** Subtree is Any node, along with all its childrenates in the tree forms a subtree.

**Mathematical Formulation**

Let T = (V, E) be a tree, V be set of vertices and E be set of edges. The next properties are true.

1. T is connected: ∀u,v∈V ∃a path from u to v
2. T is acyclic: There are no cycles in T.
3. |E| = |V| − 1: The number of edges equals the number of vertices minus one,
4. Each edge added to T creates a single cycle.
5. T's any edge disconnects the graph into exactly two components.

Cayley's formula states that the number of different labeled trees with n vertices is n n−2, where n is any positive integer.

### 4.3.2 Rooted Trees

A rooted tree is a tree with one vertex specified as the root. It has been found that the selection of a vertex as the root creates a natural ordering among the vertices that allows them to be treated as belonging to a hierarchy of parent-child relationships.

**Properties of Rooted Trees**

1. **Root:** The parentless vertex is the root.
2. **Parent:** An edge between u and v with u closer to the root than v means u is the parent of v and v is a child of u.
3. **Ancestors:** the ancestor of a vertex is that vertex along the path to the root; it does not include the vertex itself.
4. **Descendants:** The descendants of a vertex consist of all of the vertices in its subtree, not including the vertex itself.
5. **Siblings:** Vertices with the same parent.
6. **Level or Depth**: The level or depth of a vertex is defined as the length of the unique path from the root to that vertex.
7. **Tree Height:** Maximum depth of a vertex in a rooted tree.

**Mathematical Representation**

In a rooted tree with root r:

• For any vertex v ≠ r, there is a unique path from r to v.

• The depth of a vertex v, denoted by depth(v), is the length of the path from r to v.

• The tree height is max{depth(v) | v ∈ V}.

• A vertex v is an ancestor of the vertex w if lies on the path from r to w.

• Let T_v be the subtree rooted at v, that is, T_v is the set of nodes v and all the descendants of v

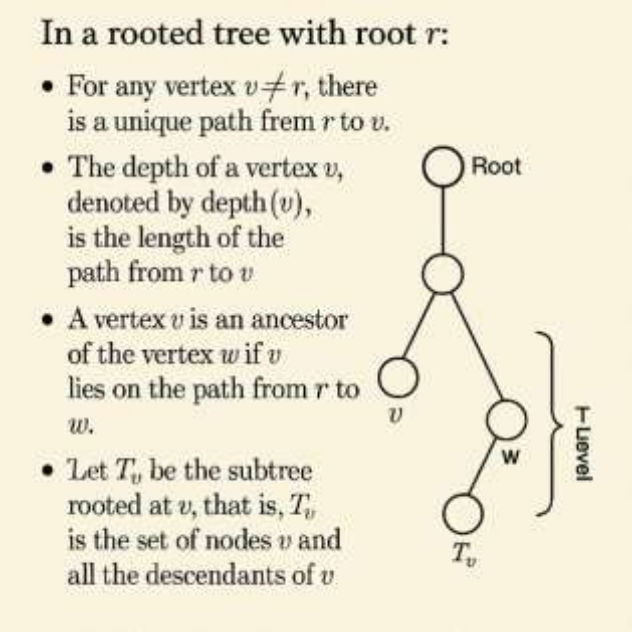



Fig: 13.1 Representation of Rooted Tree

**Ordered Rooted Trees**

Each vertex has a left-to-right ordering on its children in an ordered rooted tree. This intermixed order is useful in many contexts, such as programming languages representing expressions.

**Binary Trees**

It is a specific type of a rooted tree for which each node has no more than two children, which are usually called the left child and the right child. Binary trees are among the most popular tree structures used in computer science.

**Properties of Binary Trees**

1. **Maximum Nodes**: The maximum number of nodes in a binary tree of height h = 2^(h+1) − 1
2. **Minimum Height :** The binary tree with n nodes has minimum height is $\lfloor \log_2(n) \rfloor$.
3. **Leaf Nodes:** There can be at most (n+1)/2 leaf nodes in a binary tree with n nodes.
4. **Full Binary Tree:** A binary tree data structure where every node has either 0 or 2 children.
5. **Complete Binary Tree:** All levels are fully filled except possibly the last level which is filled from left to right
6. **Perfect Binary tree:** A binary tree in which every internal node has two children and all leaf nodes are at the same level.

7. **Balanced Binary Tree:** A binary tree such that the heights of the two child subtrees of every node differs by at most one.

**Mathematical Analysis**

This is how many paths a binary tree with n nodes has:

- Maximum height n-1 (In case when tree degenerates to a linked list)
- The height of a complete binary tree is log2(n) (minimum case).
- The count of leaf nodes (l) and the count of nodes with two children ($i_2$) are related through: $l = i_2 + 1$.
- The number of leaf nodes in a full binary tree with i internal nodes is i + 1

## 13.4: Tree Traversals

There are different ways to traverse binary trees:

1. Inorder Traversal : Left sub-tree, Root, Right sub-tree
2. This gives us the preorder traversal: Root, Left subtree, Right subtree
3. Postorder Traversal: Left child, Right child, Parent
4. Level Order: Go through each level of the tree from left to right

Both traversal approaches can be visualised in a recursive or iterative manner and each has its own use-case in many algorithms.

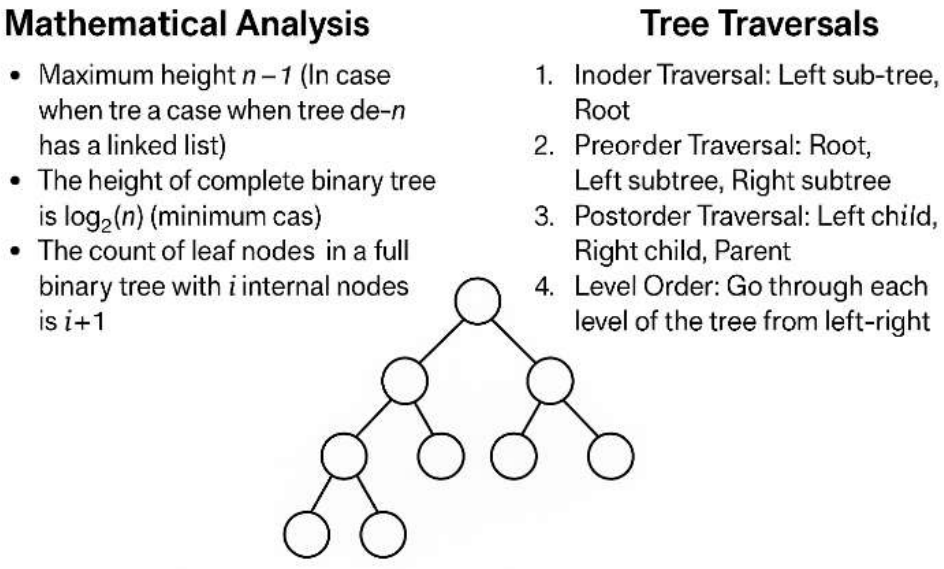



Fig: 13.2 Representation of Binary tree and Tree Traversals

## 13.5: Spanning Trees

A spanning tree of a connected undirected graph G is a tree that contains all the vertices of G and has the smallest possible number of edges (Kruskal's Algorithm). That is, the graph you are finding is a tree which is subgraph of G that contains all the vertices of G.

**Properties of Spanning Trees**

1. **Edge Count:** a spanning tree of a graph with n vertices has exactly n−1 edges.
2. **Minimality:** One of the key properties of minimum spanning tree is that it is a tree, hence it does not contain cycles:
3. **Connectivity**: Connectivity Because a spanning tree does include all the vertices in the original and does not include cycles, all of the vertices must be connected.
4. **Number of Spanning trees**: the number Spanning trees for a complete graph: n^(n-2) (Cayley's formula)
5. **Edge Redundancy**: Removing one edge from each cycle in a graph with cycle(s) results in a spanning tree.

**Minimum Spanning Tree (MST)**

In a weighted graph, a minimum spanning tree is a spanning tree with the minimum possible total edge weight. Two classic algorithms for finding an MST are:

1. **Kruskal's Algorithm**:
    - Sort all edges in non-decreasing order of weight
    - Keep adding the next lightest edge that doesn't form a cycle
    - Continue until n-1 edges are added
2. **Prim's Algorithm**:
    - Start with any vertex
    - Repeatedly add the lightest edge that connects the tree to a vertex not yet in the tree
    - Continue until all vertices are included

A minimum spanning tree (MST) of a weighted undirected graph is a spanning tree with weight less than or equal to the weights of all the edges in the tree. There are two classical algorithms for finding a MST:

1. Kruskal's Algorithm:
    - Step: Sort all edges in order of non-decreasing order of weight

- Continue with the next lightest edge that does not create a cycle
- Add edges until we have n-1 edges.

2. Prim's Algorithm:
   - Start with any vertex
   - Continuously add a smallest edge that adds a vertex not yet in the tree
   - repeat until all vertices have been added

**Mathematical Formulation**

Let G = (V, E) be a connected graph with edge weights w: E → ℝ; a minimum spanning tree T = (V, E') of G is a spanning tree such that Σ{e∈E'} w(e) is minimized. The MST is simultaneously the solution to a different problem using the cut property: If T is a minimum spanning tree and F is a cut in the graph, then the minimum weight edge crossing the cut F is also contained in some minimum spanning tree..

## 13.6: Fundamental Cycles and Cut Sets

If a cycle is formed, exactly one is part of a tree, this is the fundamental cycle. The entire set of these fundamental cycles is a cycle basis of the graph.

**Fundamental Cycles**

For a graph G = (V, E) and its spanning tree T = (V, E_T), every edge e ∈ E - E_T determines a unique cycle, when added to T. This cycle is referred to as a fundamental cycle with respect to T.

**Fundamental cycles: properties:**

1. The number of fundamental cycles hence is |E|−|V|+1.
2. One fundamental cycle corresponds to exactly one non tree edge.
3. The basis for the cycle space constitutes the set of elementary cycles.

**Fundamental Cut Sets**

A cut in graph G is a partition of the vertices V into two disjoint sets. The cut-set is the set of edges where one e and one e have endpoints in each side of the partition. Let G be a connected graph and T a spanning tree of G. The collection of all edges in G that connect these two components forms a minimum cut-set.

**Fundamental cut-set properties:**

1. For every edge in the spanning tree, there is a fundamental cut-set; thus |V| - 1.
2. A fundamental cut-set contains exactly one tree edge.
3. The collection of these fundamental cut-sets comprise a basis for the cut space of the graph

Relationship Between Fundamental Cycles and Cut Sets

There is a duality between fundamental cycles and fundamental cut-sets:

• We can find a spanning tree that has exactly one non-tree edge.

• A minimum cut-set has a only one tree edge.

• A fundamental cycle and a fundamental cut-set share either exactly two edges or none.

This duality comes in handy in different graph algorithms and network analysis problems

**Solved Examples**

**Example 1:** Verifying Tree Properties

**Problem:** Is the the following graph a tree?G = (V, E) where V = {a, b, c, d, e} and E = {(a,b), (b,c), (c,d), (d,e), (e,a)}

**Statement:** For G to be a tree, it must be connected and acyclic. Since there is a path between any two vertices G is connected. Now, let's traverse the graph to see if there's any cycle present: Starting from vertex a → b → c → d → e → a We again reached to the starting vertex → Hence, we can say G is a cyclic graph. Therefore, G is not a tree**.**

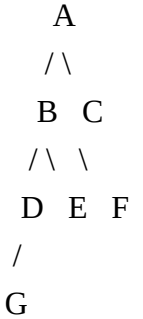
**Example 2:** Counting Tree Edges

**Problem:** Given a tree with 12 vertices. How many edges does it have?

**Solution:** This cannot happen, the number of edges in any tree with n-vertices is always n-1. For n = 12, Edges = 12-1 = 11.

Example 3: Finding Tree Height

**Problem**: Calculate the height of the following binary tree:



**Solution:** Maximum height of a tree is defined as the length of longest path from root to any leaf. Length 3 path: A → B → D → G Path A → B → E has length 2. Path A → C → F has length 2. The maximum depth is 3, hence it is a tree of depth 3**.**

**Example 3:** Binary Tree Node Calculation

**Problem:** Maximum number of nodes in a tree of Height: 5

**Solution:** The binary tree of height h has maximum 2^(h+1) − 1 nodes. If h = 5, then the maximum number of nodes = 2^(5+1) - 1 = 2^6 - 1 = 64 - 1 = 63 nodes**..**

**Example 4:** Complete Binary Tree Properties

**Problem:** A complete binary tree contains 100 nodes. What is its height?

**Solution:** Leaf Nodes in a Complete Binary Tree A complete binary tree with n nodes of h height satisfies: 2h≤n <2h+1 n=10h=3.

For, n = 100: 2^6 = 64 ≤ 100 < 2^7 = 128 So, The tree height is 6.

**Example 5:** Traversal Sequences

**Problem:** Given the binary tree below, what nodes do you visit in an inorder traversal?


**Solution**: It visits left sub tree, root, right sub tree in order. Root of sub tree: Left sub tree of 8 is 3, whose left sub tree is 1 with left/right sub trees empty. First visit 1, and then 3, and then the right sub tree of 3: 6, whose left sub tree is 4. Then next 4, then next 6, then right sub tree of 6 7. Visit 7, then return to 8. Visit 8, then right sub tree of 8: 10, whose right sub tree is 14, whose left sub tree is 13. In order traversal order: 1, 3, 4, 6, 7, 8, 10, 13, 14

**Example 6:** Minimum Spanning Tree

**Problem:** Find minimum spanning tree of below weighted graph using Kruskal's algorithm. TikZ code for graph G (vertices set {A, B, C, D, E},

edges:

(A,B)=4;(A,C)=2;(B,C)=1;(B,D)=5;(C,D)=8;(C,E)=10;(D,E)=2)

**Solution** : Step 1: Create a list of edges sorted by weight: (B,C) : 1, (A,C) : 2, (D,E) : 2, (A,B) : 4, (B,D) : 5, (C,D) : 8, (C,E) : 10

Step 2: Only add the edges in order if it doesn't create a cycle:

• Add (B,C): 1 → MST = {(B,C)}

• Union (A,C): 2 → MST = {(B,C), (A,C)}

• Add (D,E): 2 → MST = {(B,C), (A,C), (D,E)}

• Add (A,B): 4 - This would create a cycle A-B-C-A, hence skip.

• Union: (B,D): 5 → MST = {(B,C), (A,C), (D,E), (B,D)}

At this point, we have 4 edges and 5 vertices, which is exactly n−1 edges, meaning we have obtained our MST. Minimum spanning tree includes edges stored above i.e edges (B,C), (A,C), (D,E), and (B,D) with total weight 1+2+2+5 = 10.

**Example 7:** Rooted Tree Properties

**Problem:** In a rooted tree with root r, vertex a has a depth of 3 and vertex b has a depth of 5. What is the maximum possible distance between vertices a and b?

**Solution:** In a tree, the distance between any two vertices is simply the length of the unique path between them. For example if a is at depth 3 and b is at depth 5, the path from a to b has to pass through their lowest common ancestor.

**Case 1:** The case when a is ancestor of b, where we just has depth(b) - depth(a) = 5 - 3 = 2.

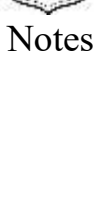
**Case 2:** If a is not an ancestor of b, then c's depth will always be at most 3 (c could be the root if both a and b in left, c could be any node on the path from root to node a) The distance from a to b would be: distance(a, b) = distance(a, c) + distance(c, b) = (depth(a) − depth(c)) + (depth(b) − depth(c)) ≤ (3 − 0) + (5 − 0) = 8.

Maximum possible distance is when their Lowest Common Ancestor is root, then distance = 3 + 5 = 8**.**

**Example 8:** Binary Search Tree

**Problem**: A binary search tree is empty initially, you have to insert the following elements into the tree: 50, 30, 70, 20, 40, 60, 80 Then delete the element 30.

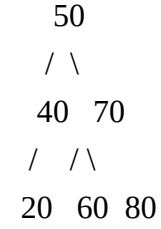
**Solution**: BST before any insertions



To delete 30:

- Slither down the left leg of the tree to 50. With the left child pointer of 50 pointing to 30.
- 30 has two children and thus we have to go to successor node.
- (Fig 8 – the inorder successor of 30 in this case should be the minimum element in its right subtree which is 40)
- Change 30 to 40, and replace the 40 with nothing.

Resulting BST after deletion:



**Example 9:** Number of Binary Trees

**Problem:** Given 3 labeled nodes, how many different binary trees that we can have?

**Solution:** For labelled nodes, the number of labelled binary trees with n nodes is the nth Catalan number: C(n) = (2n)! / (n+1)! n!

For n = 3: C(3) = (2×3)! / (3+1)! 3! = 6! / 4! 3! = 720 / 144 = 5.

Hence, 5 different binary trees can be formed with 3 labeled nodes..

**Example 10:** Depth in Binary Trees

**Problem:** What is the minimum possible depth of a leaf in a binary tree with 31 nodes?

**Solution:** In a binary tree of n nodes, the minimum depth occurs in a complete binary tree. For a complete binary tree, where h is height then : $2^h \leq n < 2^{(h+1)}$

4: If n = 31 then $2^4 \leq n < 2^5$, so h = 4. Thus, the least achievable depth from the root to any of the leaf nodes is 1 (the root itself) + the least path from leaf = h = 4**.**

**Example 11:** Tree Center

**Problem:** What is the center(s) of the below tree: Tree T with vertices {A, B, C, D, E, F, G} and edges : (A,B), (B,C), (C,D), (D,E), (C,F), and (F,G)

**Solution:** The center of a tree is the vertex (or vertices) with the minimum eccentricity where the eccentricity of a vertex is maximum distance to any other vertex.

To find the center(s), we can iteratively remove all leaf nodes until we are left with one or two vertices:

Starting tree: A-B-C-D-E and C-F-G

Step 1: Remove leaves A, E, G: we have B-C-D and C-F Step 2: Remove leaves B, D, F: we have C Only C left, Now C is the center of the tree

**Example 12:** Spanning Tree Count

**Problem:** Examples spanning trees of a complete graph $K_4$.

**Solution**: For a complete graph $K_n$, the number of spanning trees is n^{n−2} (by Cayley's formula). Therefore, for $K_4$, amount of spanning trees = 4^(4 − 2) = $4^2$ = 16**.**

**Example 13:** Fundamental Cycles

**Problem:** suppose we have the graph G on vertices {A, B, C, D, E} and edges: { (A,B), (B,C), (C,D), (D,E), (E,A), (A,C), (B,D)}

**Solution :** Given T = {(A,B), (B,C), (C,D), (D,E)} as a spanning tree of G, determine the fundamental cycles with respect to T.

Solution Non-tree edges are (E,A), (A,C), (B,D) Adding any non-tree edge to T creates a fundamental cycle:

1. Now when we add (E,A) to the graph, we will have a cycle: E-A-B-C-D-E
2. With (A,C) we have cycle: A-B-C-A
3. Adding (B,D) induces cycle: B-C-D-B

The three cycles corresponds to the fundamental cycle basis of G given the tree T**.**

**Example 14:** Fundamental Cut Sets

**Problem:** From Example 14 we have the same graph G and spanning tree T. From Example 14 we have the same graph G and spanning tree T.

**Solution**: Removing edge (B,C) from T forms two components(i.e. tree is divided)Component 1 : {A,B}Component 2 : {C,D,E}

In G, all edges connecting these two components form the general cut-set (B,C) (tree edge), (A,C) (non-tree edge), (B,D) (non-tree edge).

Hence the minimum cut-set for (B,C) is {(B,C), (A,C), (B,D)}**.**

**Example 15:** Binary Tree Height Calculation

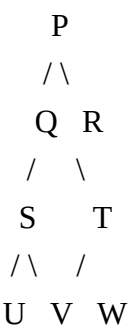
**Problem:** A binary tree with 6 leaf node and each internal node has exactly 2 child. What is the total number of nodes in the tree and its height?

**Solution:** Denote the number of internal nodes by x. Since there are exactly 2 children of every internal node and 6 leaf nodes: x + 6 = total number of nodes In addition, in a binary tree with every internal node having 2 children: x + 1 = 6 => x = 5 And total number of nodes = x + 6 = 5 + 6 = 11.

Max Path Sum of a Path: The highest path is from the base to the top. With n total nodes arranged as a full binary tree (all internal nodes have 2 children), but the tree does not necessarily need to be complete or balanced. The minimum possible height (for an infinitely sized complete binary tree) would be a nearly complete binary tree with 11 nodes, which would be 3 tall. But the real height depends on how the nodes are organized.

**Example 16:** Level Order Traversal

**Problem**: For the following binary tree, return the level order traversal of its nodes' values.

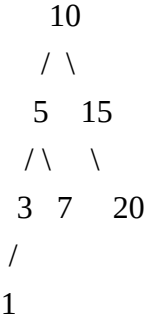



**Solution**: Level order traversal does level by level and from left to right. Level 0: PLevel 1: Q RLevel 2: STLevel 3: U V W

Level traversal: P Q R S T U V W

**Example 17:** Balanced Binary Tree Check

**Problem**: Check if the following binary tree is balanced:



**Solution**: A binary tree is balanced when the height of the left and right substree of a node has at most difference of 1

**For node 10:**

• Height of left subtree (rooted at 5) is 2

• Height of right subtree (at 15) = 1

• Difference = $|2 - 1| = 1 \leq 1$, then balanced in this node

**For node 5:**

• Height of left subtree (rooted at 3) = 1

• Height of right subtree (rooted at 7) is 0

• Difference = $|1-0| = 1 \leq 1$, so balanced at this node

**For node 15:**

• Height of left subtree being -1 (by convention) when it is empty

• Height of right subtree (rooted at 20) is 0

• Difference = $|(-1)-0| = 1 \leq 1$, hence balanced at this node

**For node 3:**

• height of left subtree (rooted at 1) = 0

• Height of right subtree is -1 (it's empty)

• Difference = $|0-(-1)| = 1 \leq 1$, so balanced at this node

Hence the tree is balanced, as all the nodes follow the balanced condition.

**Example 18:** Prim's Algorithm for MST

**Problem:** Given the following weighted graph, determine its minimum spanning tree, using Prim's algorithm with A as the initial vertex: Graph G where V = {A, B, C, D, E} and E = { (A,B,2), (A,C,3), (B,C,1), (B,D,1), (C,E,5), (D,E,4) }

**Solution :** MST = {A} frontier edges {(A,B), (A,C)}

Iteration 1: Minimal frontier edge = (A,B) with weight 2 Add B to MST: MST = {A, B} Update frontier edges = {(A,C), (B,C), (B,D)}

**Iteration 2:** Lightest frontier edge = (B,C) with weight 1 Add C to MST: MST = {A,B,C} Update frontier edges = {(A,C), (B,D), (C,E)}

**Iteration 3:** Lightest frontier edge = (B,D) with weight 1 Add D to MST: MST = {A, B, C, D} Update frontier edges = {(A,C), (C,E), (D,E)}

The edge with minimum weight in this iteration: Lightest frontier edge = (D,E) with weight 4 Add E to the MST: MST = {A, B, C, D, E}

MST edges: (A,B), (B,C), (B,D), (D,E) => total weight 2+1+1+4 = 8.

**Example 20:** Tree Isomorphism

**Problem:** Are the following two trees $T_1$ and $T_2$ isomorphic? given tree $T_1$: Edges (A,B), (A,C), (B,D), (B,E), (C,F) given tree $T_2$: Edges (P,Q), (P,R), (Q,S), (Q,T), (R,U)

**Solution:** T(|t,H|) for the vertex set of H Solution: Two trees are isomorphic iff we can get one from another by consistently renaming vertices.

Start by ensuring that both trees have the same number of vertices and edges:

- $T_1$ contains 6 vertices (A,B,C,D,E,F) and 5 edges
- $T_2$ 6 vertices(P,Q,R,S,T,U) and 5 edges ✓ Same number

Then verify the degree sequence (ordered list of vertex degrees):

- $T_1$ : deg(A)=2, deg(B)=3, deg(C)=2, deg(D)=1, deg(E)=1, deg(F)=1 Sorted : [1,1,1,2,2,3]
- $T_2$: deg(P)=2, deg(Q)=3, deg(R)=2, deg(S)=1, deg(T)=1, deg(U)=1 Sorted: [1,1,1,2,2,3] ✓ Same degree sequence

Finally, test the structural correspondence:

- All three vertices have degree 2 in both trees (e.g., C and D in $T_1$, I and J in $T_2$)• Both trees have one vertex of degree 3 (B in $T_1$, Q in $T_2$)
- Both trees contain two vertices of degree 2 (A,C in $T_1$, P,R in $T_2$)
- $T_1$ has vertices D,E,F of degree 1 each. The edges with their 2 vertices {F, B}, {E, A}, {D, C} can be mapped to their 2 vertices in {S, T}, {S, U}, {U, T}
- In both trees, the degree-3 vertex is adjacent to one degree-2 vertex and two degree-1 vertices.

The trees $T_1$ and $T_2$ are isomorphic with vertex mapping as follows: A↔P, B↔Q, C↔R, D↔S, E↔T, F↔U

In computer science and discrete mathematics, trees are foundation data structure with intuitive representation of hierarchies. Tree structures are crisp hierarchies that can either be in general form of a connected acyclic graph to specialized structure like spanning tree or binary trees prove their worth towards efficient solution of a variety of computational problems. These mathematical properties of trees are the basis of all algorithms (the relation that $|E| = |V| - 1$, no cycles, and uniqueness of paths between vertices, etc.). These properties appear in applications as various as file systems, database indexing,

network optimization, artificial intelligence. Adding structure — in this case, allowing trees to be rooted or binary — resonates with a variety of natural hierarchies and recursive processes. This is a great way to show how certain variants of trees can be tailored to better suit specific computation, as there are specially crafted algorithms for tree traversal, searching and balancing. They connect trees to more general theory of graphs, providing a connection between trees and a minimal way to guarantee connectivity in a network. Algorithms used for constructing minimum spanning trees, like Kruskal's and Prim's algorithms, are some of the most fundamental approaches you can find in optimization problems. So, we have looked at multiple examples, along with their visualizations to show how such theoretical ideas convert into real-world problem-solving techniques as well, during this entire series and this reiterates the very significance of trees as mathematical objects and computational tools. From the elegance simplicity of tree structures yet their efficiency and intuitive nature it ensures that trees will always have a place in solving Not schwierig but complex real world problems in many different nodes of the tree of computer science and applied mathematics.

**Applications of Eulerian and Hamiltonian Graphs**

Eulerian graphs, where every vertex has an even degree and a closed trail exists, have many real-world applications. For example, in urban planning, Eulerian trails can be used to design garbage collection routes, mail delivery paths, or street cleaning paths so that each street is covered exactly once.

Hamiltonian graphs, where a cycle passes through each vertex exactly once, are important in network design and scheduling problems. The most famous related problem is the Travelling Salesman Problem (TSP), in which the goal is to find the shortest possible route that visits each city exactly once and returns to the starting point. While TSP is computationally hard, approximations and heuristics are used in logistics, airline scheduling, and circuit board design.

**Advanced Shortest Path Problems**

Dijkstra's algorithm provides the shortest path in weighted graphs without negative edges. In practice, there are more advanced variations:

**Bellman–Ford Algorithm:** Works even when negative weights are present, although it is slower than Dijkstra's.

**Floyd–Warshall Algorithm:** Finds shortest paths between all pairs of vertices, useful in dense graphs like computer networks.

**A* Algorithm:** A heuristic-based extension of Dijkstra, widely used in AI for pathfinding in games, robotics, and navigation systems.

**Example:** In Google Maps, Dijkstra or A* algorithms are applied on road networks where intersections are vertices and roads are weighted edges. The weights represent distance or time.

**Graph Coloring**

Graph coloring is the assignment of colors to vertices so that no two adjacent vertices share the same color. The minimum number of colors required is called the chromatic number of the graph.

**Applications of graph coloring include:**

Scheduling Problems: Assigning time slots for exams so that no two exams with common students clash.

**Register Allocation in Compilers:** Assigning variables to limited CPU registers.

**Map Coloring:** Ensuring that no two neighboring regions have the same color.

**Example:** Consider a graph representing exam subjects where an edge exists between two subjects if they share common students. Coloring the graph gives the minimum number of time slots required.

**Planar Graphs and Euler's Formula**

A graph is planar if it can be drawn in a plane without edges crossing. Planar graphs are important in circuit design, transportation, and cartography. Euler's formula connects vertices (V), edges (E), and faces (F) in a connected planar graph:

$V - E + F = 2$.

**Example:** For a cube represented as a planar graph, we have $V = 8$, $E = 12$, $F = 6$. Substituting gives $8 - 12 + 6 = 2$, which satisfies Euler's formula.

**Spanning Trees and Minimum Spanning Trees**

A spanning tree of a graph is a subgraph that connects all vertices without any cycles. Spanning trees are useful in designing efficient networks, such as communication or road systems.

A minimum spanning tree (MST) is a spanning tree with the smallest possible total edge weight. Two famous algorithms are used to find MSTs:

**Kruskal's Algorithm:** Sorts edges by weight and adds them one by one if they do not form a cycle.

**Prim's Algorithm:** Builds the tree starting from a vertex and grows it by adding the least costly edge that connects a new vertex.

**Example:** MST is applied in designing a minimum-cost telecommunication network connecting multiple cities with cables.

**Worked Example:** Kruskal's Algorithm

Consider a weighted graph with vertices {A, B, C, D} and edges:

A–B (1), B–C (4), C–D (3), A–C (2), B–D (5).

**Step 1:** Sort edges by weight: (A–B, 1), (A–C, 2), (C–D, 3), (B–C, 4), (B–D, 5).

**Step 2:** Add A–B, weight = 1.

**Step 3:** Add A–C, weight = 2.

**Step 4:** Add C–D, weight = 3. Now all vertices are connected. Total weight = 6.

Thus the MST is {A–B, A–C, C–D} with cost 6.

**Real-Life Applications of Graph Theory**

**Social Networks:** People are vertices, and friendships or connections are edges.

**Communication Networks:** Routers and computers are nodes, cables and links are edges.

**Biology:** Protein interaction networks and gene regulatory networks are modeled as graphs.

**Transport Systems:** Cities and roads are modeled using weighted graphs to optimize travel.

**Project Planning:** Activity networks such as PERT (Program Evaluation and Review Technique) and CPM (Critical Path Method) use directed graphs to manage projects efficiently.

**Importance of Graph Theory**

Graph theory connects abstract mathematics with practical applications. Whether it is routing internet packets, optimizing transport routes, analyzing social networks, or designing circuits, graphs provide a natural way to represent relationships and solve problems. Its algorithms form the backbone of many modern technologies.

## Check Your Progress

1. Define a tree. Explain its properties and significance in computer science.

   .....

   .....

   .....

.....
.....

2. What is a binary tree? Explain its structure and importance with an example.

.....
.....
.....
.....
.....

## 13.7: Summary

This unit covered the fundamental concept of trees as connected acyclic graphs that represent hierarchical relationships. Learners explored various types of trees and key properties including root, degree, path, height, and depth. The unit emphasized that every pair of vertices in a tree is connected by exactly one path, making it an efficient structure for data organization and traversal. Trees are essential for representing file systems, arithmetic expressions, hierarchical decision processes, and network routing. Understanding tree properties provides a foundation for advanced topics such as spanning trees, binary search trees, and traversal algorithms in computer science.

## 13.8: Exercises

**Multiple Choice Questions**

A tree is defined as a:

a) Connected graph with no cycles

b) Graph containing at least one cycle

c) Disconnected graph with multiple components

d) Weighted graph only

Answer: a) Connected graph with no cycles

The number of edges in a tree with n vertices is:

a) n + 1

b) n

c) n – 1

d) 2n

Answer: c) n – 1

In a rooted tree, the topmost node is called:

a) Leaf node

b) Root node

c) Internal node

d) Terminal node

Answer: b) Root node

A node with no children is called:

a) Root node

b) Leaf node

c) Parent node

d) Sibling node

Answer: b) Leaf node

Trees are mainly used in computer science for:

a) Data organization and hierarchical representation

b) Linear data storage

c) Randomized computation

d) Numerical integration

Answer: a) Data organization and hierarchical representation

**Descriptive Questions**

14.1 Define a tree. Explain its properties with suitable examples.

14.2 Differentiate between general trees, rooted trees, and binary trees.

14.3 Explain the terms degree, height, and depth of a tree with examples.

14.4 Discuss the applications of trees in computer science and data structures.

14.5 Describe the importance of hierarchical structures and how trees represent them efficiently.

## 13.9: References and Suggested Readings


- Kolman, Bernard, Busby, Robert C., and Ross, Sharon. *Discrete Mathematical Structures*, 6th Edition, Pearson Education, 2018.
- Deo, Narsingh. *Graph Theory with Applications to Engineering and Computer Science*, Prentice Hall of India, 2017.
- Grimaldi, Ralph P. *Discrete and Combinatorial Mathematics: An Applied Introduction*, 5th Edition, Pearson Education, 2003.

- West, Douglas B. *Introduction to Graph Theory*, 2nd Edition, Pearson Education, 2001.
- Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., and Stein, Clifford. *Introduction to Algorithms*, 4th Edition, MIT Press, 2022.

## Block 5: Semigroup and Monoid

# Unit 14: Algebraic Structure, Binary Operation, Properties, Semi Group, Monoid, Group Theory

**Structure**

14.1 Introduction
14.2 Learning Outcome
14.3 Algebraic Structure, Binary Operation, Properties, Semi Group, Monoid, Group Theory
14.4 Types of Algebraic Structures
14.5 Historical Development and Future Directions
14.6 Mathematical Structures: Semigroups, Monoids, and Group Theory
14.7 Advanced Topics and Future Directions
14.8 Summary
14.9 Exercises
14.10 References and Suggested Readings

## 14.1: Introduction

This unit introduces the concept of algebraic structures, which form the foundation of many mathematical and computational systems. An algebraic structure consists of one or more sets equipped with one or more operations that satisfy specific properties. Learners will study binary operations and their properties such as closure, associativity, identity, and invertibility. The unit also explores the fundamental algebraic systems—semigroups, monoids, and groups—and their significance in computation and logic. These structures are essential in understanding transformations, coding theory, automata, and the mathematical modeling of various computational processes.

## 14.1: Learning Outcomes

- To understand algebraic structures, binary operations, and their properties.
- To explore semigroups, monoids, and group theory.
- To analyze Abelian groups, cyclic groups, generators, and permutation groups.

- To study homomorphism, isomorphism, and automorphism in group theory.
- To understand cosets, Lagrange's theorem, normal subgroups, and quotient groups.

## 14.3: Algebraic Structure, Binary Operation, Properties, Semi Group, Monoid, Group Theory

**Algebraic Structures and Binary Operations**

Modern abstract algebra is based on an algebraic structure, which gives mathematicians the means to study symmetry, solve equations, and find a link between different fields of mathematics. An algebraic structure is basically a set along with one or more binary operations that satisfy certain properties. Such structures echo throughout mathematics, from the well-worn byways of basic arithmetic to esoteric like topology and analysis. We can learn about the underlying structures of mathematical systems, and we find tools to solve intricate challenges in multiple disciplines.

**Binary Operations: The Foundation of Algebraic Structures**

In mathematics, a binary operation from a set S is a function from $S \times S$ to S. In more concise terms, it can be described as a mapping $* : S \times S \{a, b\} \rightarrow S$ such that for any $a, b \in S$, we denote the binary operation of a and b by $a * b$. The operation that maps $\langle * \rangle$ back to one of its input sets ⟨set S in this case⟩ is called closure: one of the most fundamental properties of a binary operation that guarantees repeatedly performing the operation would yield the result still within the primary set. Typical examples would be addition and multiplication over the reals, function compositions, matrix multiplication, logical operations over Boolean values. Thanks to the generality of binary operations, mathematicians can model a wide range of phenomena from a variety of fields. In particular, we can describe the operation structures (Cayley tables) of binary operations, i.e., the result of operating on any two (possibly the same) elements from the list of elements in the set. For finite sets, these tables are a complete picture of the operation, and they also show things we may otherwise overlook just from the definition. In fact, the reason for this is that the form of these tables often conveys valuable information about the operation; symmetry patterns can indicate commutatively while diagonal patterns can demonstrate idempotence. Knowing the

geometric representation of operation tables could help in understanding clear logic for abstract algebra concepts..

**Fundamental Properties of Binary Operations**

There are several important properties that binary operations can satisfy that help to define the type of algebraic structure they create. The distributive property is just one of the uses for associatively (meaning that $(a * b) * c = a * (b * c)$ for all a, b, c elements of the set, so you can do an operation on them without concern as to how you group them), and of course, the commutative property, they are very useful in simplification, and help us do inductive proofs. Commutatively means that the order of operands does not matter, mathematically defined as $a * b = b * a$ for all a, b in the set. There exists an identity element e such that $a * e = e * a = a$ for every a in the set, such that the element does not change other elements when operated with. If there is an element a so that an inverse element $a^{-1}$ exists such that $a*a^{-1}=a^{-1}*a=e$ for each a, then operations can be "undone", and we can solve for equations in this structure.

The properties described give binary operations a richer theoretical ecology. Idempotence ($a * a = a$) emerges in max or min type operations. One operation distributes over another ($a * (b \oplus c) = (a * b) \oplus (a * c)$), relating multiple operations together like the multiplication and addition operations in arithmetic. Binary operations are also characterized by absorption and cancellation properties as well as other closure properties that help determine their algebraic behavior. These properties are not just technical definitions, they reflect underlying patterns of mathematical reasoning and methods of approaching problems.

## 14.4: Types of Algebraic Structures

The properties satisfied by the binary operations determine the classification of algebraic structures. Groups — a set with one binary operation that satisfies closure, associativity, identity and inverse properties — are the backbone of abstract algebra and show up throughout mathematics, from number theory to crystallography. Monoids characterize sequences of operations that can be combined (closure) and have a function that when combined does not alter the state of the system and does not have an inverse if they are irreversible (identity). Semi groups the simplest combining processes

(i.e., satisfying closure and associatively only) are even more general. These structures are arranged hierarchically with each one having fewer constraints than the previous one, providing mathematicians the choice of level or degree of abstraction for specific problems. When more than one binary operation acts on the same set, more sophisticated structures can be obtained. A ring is a set equipped with two operations (called addition and multiplication in the standard case) such that addition is an abelian group, multiplication is a monoid, and multiplication distributes over addition. Fields build on rings by demanding that every non-zero element must also possess a multiplicative inverse, permitting division. Vector spaces, modules, algebras and lattices are further specializations with introduced additional operations and axioms. As we can see from the structure of the above two polynomials, algebraic structures can capture background patterns between certain objects. The theory of such structures reveals profound connections between areas of mathematics seemingly unrelated to each other.

**Applications of Algebraic Structures**

For examples of the wide range of applications of algebraic structures see 4[8] 23[44]. In cryptography, groups underpin a range of encryption algorithms — including RSA and elliptic curve cryptography — safeguarding digital communications used around the globe. These concepts are integral to coding theory, which utilizes finite fields to create error-correcting codes that ensure data integrity during transmission in the presence of noise and interferences. In physics, group theory underpins symmetries of physical systems, ranging from the classification of crystal structures to the fundamental particles of the Standard Model. Algebraic structures also show up in computer science, especially in the design of programming languages, type theory and automata theory — sequences of computation can be modelled with monoids and semi groups. Outside traditional STEM fields, algebraic structures shape everything from linguistics to music theory. Group theory is used in chemistry to classify molecular structures and to predict vibration spectras. The relation between mixed-strategy payoffs and second-order conditions is in economic models where they used algebraic structures to study preference relations and market behaviors. Even prudently nondisambiguatory aesthetic domains respond to algebraic

abstractions, as group theory, for instance, shines light on mathematically structured principles within visual arts, architecture and musical composition. This universality of algebraic structures arises because they capture patterns of relationships or transformations that emerge naturally in many realms of human endeavor and creativity.

## 14.5: Historical Development and Future Directions

Algebra 2000 has been a huge evolution since the first isolation of algebra in the 19th. Mathematicians like Everest Galois and Niels Henrik Abel began by investigating the characteristics of polynomial equations and the relationship they have between the solution sets and investigate their symmetries through group theory. Abstract algebra emerged as a separate subject during the 19th and early 20th centuries, with people like Emmy Noether and Emil Artin formalizing the axioms for different algebraic structures. It made things clearer and more general, helped find connections among long separate branches of mathematics, and made algebraic structures fundamental organizing principles in mathematics. Modern directions in algebraic structure are once again expanding in all sorts of ways. Category theory gives a common language to relate algebraic spaces using factorial and natural transformations. Computational algebra is the study of algorithms for manipulating algebraic objects in an efficient manner (for applications in cryptography and scientific computing, see here). Emerging fields such as quantum algebra and algebraic geometry combine algebraic structures with analytic and topological methods to tackle difficult problems. These avenues signal even more integration of algebraic methods into data science, machine learning and quantum computing as mathematicians find new structures to model novel phenomena and invent theoretical scaffolding for the mathematics of the future. The self-referential nature of the evolving algebraic structures is indicative of what is the dual nature of mathematics: a means of solving immediate problems of practical desire and a quest for aesthetics through idealism of though

## 14.6: Mathematical Structures: Semigroups, Monoids, and Group Theory

**Introduction to Algebraic Structures**

Algebraic structures are the building blocks of contemporary mathematics: sets endowed with operations that follow certain properties. Here, semigroups, monoids and groups are a series of increasingly rich algebraic systems. Isomorphism and Homomorphism are key concepts in abstract algebra, which is a branch of mathematics concerned with algebraic structures such as groups, rings, and fields, and these structures have applications in various domains like number theory, geometry, cryptography, quantum mechanics, and computer science. So, setting out on this exploration satrt with the simplest sstructure as the semigroup and build towards the more constructive and more poweful of all structure which are group and its the properties, relations and its the importance in the maths.

**Semi groups: The Foundation**

A semi group is an algebraic structure consisting of a non-empty set S with a binary operation (often denoted by juxtaposition or $*$) satisfying the associatively property. In formal terms, for any $a,b,c \in S$the operation must satisfy$(a*b)*c=a*(b*c)$. In fact, this relationship between products is associative so we can write an expression such as $a * b * c$ without ambiguity, since it does not matter in which order we apply that operation. Semi groups generalize the idea of many mathematical operations. For example, the positive integers under addition form a semi group, as do the set of all matrices of a fixed size under matrix multiplication. Here is another example: The collection of all strings over a fixed alphabet under concatenation. In category theory, even the collection of natural transformations between factors forms a semi group under composition. There are several interesting properties of semi groups and some kinds of semi groups. For commutative semi groups, the operation is commutative (i.e., $a * b = b * a$ for all a, b) and therefore also forms a commutative semi group, or an abelian semi group. An abelian semi group is a semi group (i.e. binary associative with identity) in which the binary operation is commutative but need not have an identity element. A band is a semi group where every element is idempotent (i.e., $a * a = a$ for all a).A regular semi group is a semi group with the property that for all a there exists an element b with the property that $a * b * a = a$. Semi groups also involve the study of

properties such as subsemigroups (subsets of a semi group that are also semi groups under the same operation), homeomorphisms (structure-preserving maps between semi groups), and congruence relations (equivalence relations that are compatible with the semi group operation). The theory of semi groups includes the consideration of Green's relations, which classify the elements of a semi group according to the equivalence classes determined by the principal ideals that can be formed from its elements. (For example, types of semi groups, and the structure thereof will be elucidated through later sections of the paper, through classification of finite semi groups for example.).

**Monoids:** Semi groups with Identity

A monoid is an extension of the semi group concept, providing an identity element. More precisely, a monoid is a semi group (M, $*$) which has an identity element in M, denoted by e, satisfying $e * a = a * e = a$ for all elements $a \in M$; this identity behaves like the number 0 with respect to addition or 1 with respect to multiplication, since it does not change other elements when used in a combination. This means that we have an even better algebraic structure with an identity element. are are examples of monoids march 2zero chosen wrapper fabric the set of non-negative integers under addition is a monoid with identity 0, and the set of positive integers under multiplication is a monoid with identity 1. The set of all strings over an alphabet (including the empty string) forms a monoid under concatenation and the empty string is the identity element. Under function composition, all endomorphism's of a set form a monoid with the identity function as the identity element. This applies very widely to theoretical computer science, especially formal languages and automata theory. The behavior of a finite automaton can be captured with a finite monoid where the monodies operation is the concatenation of input strings. The syntactic monoid of the language gives a construction that the language is characterized up to isomorphism, and offers insights into its properties. In functional programming, you use a monoid to define a foldable data structure that can allow you to efficiently generate large data sets using parallel processing. Introduction to the theory of monoids A submonoid is a subset of a monoid that contains the identity and is closed under the monoid operation. That is, a monoid homomorphism means that $\forall$ a, b $\in$ A,

m(f(a · b) == f(x) · f(y) and f(1) == 1. The free monoid on some set A is the monoid consisting of all finite sequences (or strings) of elements from A, using concatenation as the monoid operation, and with the empty sequence as the monoid identity. This idea is elementary in formal language theory, in which the free monoid on an alphabet is the set of all strings over that alphabet..

**Groups:** Monoids with Inverses

A group is an even richer algebraic structure, which adds inverse elements to a monoid. This formally means that group is a monoid (G, $*$) having the inverse for every element a $\in$ G, i.e. $a^{-1}$ such that: a $*$ $a^{-1}$ = $a^{-1}$ $*$ a = e, where e is identity element of the monoid. This means that for every element in a group there is a unique "counterpart" that will result to the identity when combined with the original element. The structure of algebraic systems is (even) better than it could be with the introduction of inverses. In the case of the set of integers under addition, the identity is 0, and the inverse of any integer n is simply its negative − n. Zero is not part of our original group, as we started with rational numbers with non-zero denominator, so we don't need to worry about it. The symmetric group on a set is the collection (a group) of all bijections (injective & subjective) from the set back onto itself, under function composition. There are a variety of properties and structures that can display a group. Finite Group — This is a group that has a finite number of elements, the number of elements it contains is called its order. • An abelian (or commutative) group is one that satisfies the commutative property of its operation, a $*$ b = b $*$a, for every term a and b in the group • A cyclic group is a group in which each element can be expressed as the nth power of a generator (same generator for all) If an integer n exists so that a^n=e, then it is defined to be the order of the element a in the group. Group theory involves important structures such as subgroups (subsets that themselves form groups under the same operation), normal subgroups (subgroups that are invariant under conjugation), quotient groups (the groups resulting from taking cosets of normal subgroups), and group homomorphisms (mappings between groups that preserve the operation). These notions enable mathematicians to study the internal structure of groups and their relations with other groups. A prime example of the efficacy of these analytic tools is given in the fundamental theorem of finitely

generated abelian groups, which yields a complete classification of all such groups up to isomorphism.

**Group Theory:** Historical Development and Fundamental Theorems

This notion broadened the concept of symmetry and led to the development of group theory, a mathematical framework that originated in the early 19th century through the work of mathematicians such as Évariste Galois and Niels Henrik Abel, who explored the solvability by radicals of polynomial equations. Galois' analysis involved the use of groups to determine when a given equation is solvable by radicals, thereby relating the structure of the symmetries of a polynomial (the Galois Group associated to the polynomial) to the existence of algebraic solutions. This periodic milestone created a close connection between group theory and field theory, and later became the core of Galois theory. Group theory developed rapidly thanks to the work of many mathematicians. With the advent of group theory in analysis, the eponymous results of Augustin-Louis Cauchy emerged, such as his namesake theorem, which asserts that for any finite group of order n, an element of order p exists for every prime that divides n. Arthur Cayley abstracted the term group to signify an abstract group, liberating the theory from contexts like permutations or transformations. Camille Jordan worked on composition series and formulated Jordan's theorem on finite linear groups. Reformulation of Geometry via Group Theory: Felix Klein's Erlangen Program reformulated by interpreting geometric properties as invariants under the action of groups. Group theory is based on several fundamental theorems. What you are learning includes Lagrange's theorem which states the order of the subgroup divides the order of the group, limiting the possible sizes of subgroups. For any group homomorphism $\varphi$, the first isomorphism theorem states that $G/\text{Ker}(\varphi) \cong \text{Im}(\varphi)$: A group, its homomorphic image and the kernel of the homomorphism. Sylow's theorems offer important insight into the structure of finite groups, because they gives insights on existing and properties of subgroups of every prime power order. The orbit-stabilizer theorem establishes a relation between the size of an orbit under a group action and the index of the stabilizer, and in turn provides powerful counting methods. The theory of finite simple groups is one of the monumental achievements of mathematics. It’s an "enormous theorem" that covers thousands of

pages in hundreds of journal articles, and involves dozens of mathematicians over three or four decades, classifying all finite simple groups into four specific families: cyclic groups of prime order, alternating groups, groups of Lie type, and 26 other sporadic groups that fell through the cracks of the other families. This classification demonstrates the intricate and diverse nature of group structures and gives a holistic perspective on the essential components of finite groups.

**Applications of Semi groups, Monoids, and Groups**

Semi groups, monoids, and groups theory have a lot of practical applications in many domains. In the field of cryptography, group theory is used in many encryption schemes. Essentially, the RSA algorithm is one of the oldest public-key cryptosystems and is still one of the most popular. It is founded on the computational difficulty of some problems in number theory which fall under cyclic groups. Elliptic curve cryptography exploits the group structure of points on elliptic curves to construct secure cryptosystems with smaller key sizes than those used by classical cryptosystems. Diffie-Hellman key exchange, a key exchange protocol that allows two parties to exchange cryptographic keys over public channels, relies on certain properties of cyclic groups. Group theory is crucial in the understanding of phenomena in physics. Noether's theorem provides a deep relationship between symmetries of physical systems and conservation laws, where each continuous symmetry leads to a conserved quantity. In solid-state physics, the only kind of crystal structure classification is based on the 230 space groups specifying the possible symmetry patterns in the three-dimensional crystal lattices. In particle physics, Lie groups are used to categorize the elementary particles as well as the interactions between those particles. There are a host of ways that these algebraic structures are used in computer science. In automata theory, monoids are used to model finite state machines, specifically the syntactic monoid captures the recognition power of automata. The free monoid is used in formal language theory to represent the set of all strings over an alphabet. Petri nets, used to model concurrent systems, have a background algebraic structure based on commutative monoids. Thus, Inspired by functional programming languages (e.g., Haskell), a monoid is a concept used in many programming languages to define a structure

that combines data structures through associative operations with the existence of identity elements which is both well-defined and powerful, allowing the writing of efficient, elegant code. In chemistry, we have group theory to help understand symmetry in molecular systems and spectroscopy. Molecular symmetry groups preside over the vibrational modes that can be excited and selection rules governing allowed spectroscopic transitions. In quantum chemistry the role of group theory is to exploit the symmetry of molecules to render Hamiltonian matrices block-diagonal, simplifying the calculation of matrix elements. Stereochemistry appraises group-theoretic ideas to categorize and anticipate candidate stereoisomers of molecules and advances knowledge of three-dimensional molecular conformations and their biological functions.

## 14.7: Advanced Topics and Future Directions

This is an area that is still active — theory evolves, new results are proven, themes are developed. Understanding a group through the action that is built on a space a group acts upon is the basis of a very active field of study in contemporary mathematics — geometric group theory. The word problem for groups—whether two words denote the same element of a group—connects group theory with computational complexity theory and has far-reaching consequences for automated theorem proving and verification systems. Sets, functions, and relations are fundamental to the study of algebraic structures, and representation theory is a powerful tool used in the representation of groups as linear transformations of vector spaces. Character theory, as a part of representation theory, can associate a group element with a complex-valued function (the character), over which representations can be decomposed into irreducible components. These techniques are used in quantum mechanics, so where the symmetry groups of irreducible representations associate quantum states for physical systems. Topological groups, which imbue group structure with topological structure, constitute a bridge between algebra and topology. Lie groups are both groups and differentiable manifold, and thus sit at the heart of differential geometry and theoretical physics. Representation theory of Lie groups is the basis of quantum field theory and provides necessary tools to describe particle interactions at the most fundamental level. New links

between group theory and quantum information science have emerged with the recent advances of quantum computing. Quantum error-correcting codes are frequently based on group-theoretic constructions which serve to shield quantum information from decoherence. Despite being surprising, this discovery is understandable, as many efficient quantum algorithms known today (e.g., Shor's factoring algorithm) are based on realising an info-progression strategy over the group structure of abelian groups to give an exponential speed up compared to classical algorithms. A larger picture for quantum computing and group theory in years: Semi groups, monoids, and groups are still being studied and will inevitably be powerful tools in our understanding of algebraic structures. There is ongoing work in categorical ways of thinking about algebra, which each of these things could be seen as a special case of something larger that we will learn. These works yield algorithms that allow one to solve group-theoretic problems in an efficient manner and thus improve our understanding of potentially very complex groups. This section has demonstrating the relevance of group theory to sunrise fields such as network science and the data science show how relevant and applicable those fundamental mathematical concepts remained in the study of modern-day challenges across science and technology.

## Check Your Progress

1. Define an algebraic structure. Explain the properties of binary operations with suitable examples.

   .....
   .....
   .....
   .....
   .....

2. Differentiate between semigroup, monoid, and group with examples.

   .....
   .....
   .....
   .....
   .....

## 14.8: Summary

This unit discussed the basic principles of algebraic structures and the operations defined on sets. Learners explored binary operations and their fundamental properties. The unit highlighted the hierarchy of algebraic systems—semigroup, monoid, and group—based on the presence of identity and inverse elements. Groups were shown to satisfy closure, associativity, identity, and invertibility, forming a complete structure for mathematical modeling. These algebraic systems play a vital role in various fields of computer science such as cryptography, error detection, data transformations, and automata theory. Understanding these properties provides a strong foundation for reasoning about abstract systems and their computational behavior.

## 14.9: Exercises

**Multiple Choice Questions**

1. A binary operation on a set is a rule that:
   a) Combines two elements to give another element of the set
   b) Always produces a number greater than the operands
   c) Combines elements from two different sets only
   d) Is defined only for real numbers
   Answer: a) Combines two elements to give another element of the set
2. A semigroup is a set with a binary operation that is:
   a) Commutative only
   b) Associative only
   c) Associative and has an identity element
   d) Associative and has an inverse element
   Answer: b) Associative only
3. A monoid is a semigroup that has:
   a) An identity element
   b) An inverse element
   c) A commutative property
   d) Both commutative and inverse properties
   Answer: a) An identity element
4. A group is a monoid in which every element has:
   a) No identity element

b) At least one inverse element
c) Exactly one inverse element
d) Multiple identity elements
Answer: c) Exactly one inverse element

5. Which of the following sets forms a group under addition?
   a) Natural numbers
   b) Whole numbers
   c) Integers
   d) Positive integers
   Answer: c) Integers

**Descriptive Questions**

1. Define an algebraic structure. Explain the meaning of binary operation with an example.
2. Discuss the properties of binary operations such as closure, associativity, commutativity, and identity.
3. Define semigroup, monoid, and group. Explain their differences with examples.
4. Verify whether the set of integers under addition forms a group.
5. Explain the importance of group theory in computer science applications such as coding, cryptography, and automata.

## 14.10: References and Suggested Readings


- Rosen, Kenneth H. *Discrete Mathematics and Its Applications*, 8th Edition, McGraw-Hill Education, 2019.
- Kolman, Bernard, Busby, Robert C., and Ross, Sharon. *Discrete Mathematical Structures*, 6th Edition, Pearson Education, 2018.
- Grimaldi, Ralph P. *Discrete and Combinatorial Mathematics: An Applied Introduction*, 5th Edition, Pearson Education, 2003.
- Herstein, I. N. *Topics in Algebra*, 2nd Edition, John Wiley & Sons, 2006.
- Gallian, Joseph A. *Contemporary Abstract Algebra*, 10th Edition, Cengage Learning, 2020.

## Block 5: Semigroup and Monoid

# Unit 15: Abelian Group, Cyclic Group, Generators, Permutation Group, Subgroup

**Structure**

15.1 Introduction
15.2 Learning Outcome
15.3 Abelian Group, Cyclic Group, Generators, Permutation Group, Subgroup
15.4 Applications in Mathematics and Beyond
15.5 Generators, Permutation Groups, and Subgroups
15.6 Summary
15.7 Exercises
15.8 References and Suggested Readings

## 15.1: Introduction

This unit explores advanced concepts in group theory, focusing on special types of groups such as Abelian groups, cyclic groups, and permutation groups. It also introduces the concepts of generators and subgroups, which help describe the structure and behavior of mathematical groups. An Abelian group is one in which the operation is commutative, while a cyclic group is generated by a single element. Permutation groups represent the arrangement of objects, and subgroups are subsets that themselves satisfy the group properties. Understanding these concepts is fundamental in many areas of computer science, including cryptography, automata theory, and symmetry analysis in algorithms.

## 15.2: Learning Outcomes

After studying this unit, learners will be able to:

- Define Abelian, cyclic, and permutation groups and explain their characteristics.
- Identify whether a given group is Abelian or non-Abelian.
- Explain the concept of a generator and how it defines a cyclic group.
- Describe subgroups and their role in group structures.
- Represent permutations and perform composition of permutations.

- Apply group theory concepts to areas such as coding theory, cryptography, and logic design.

## 15.3: Abelian Group, Cyclic Group, Generators, Permutation Group, Subgroup

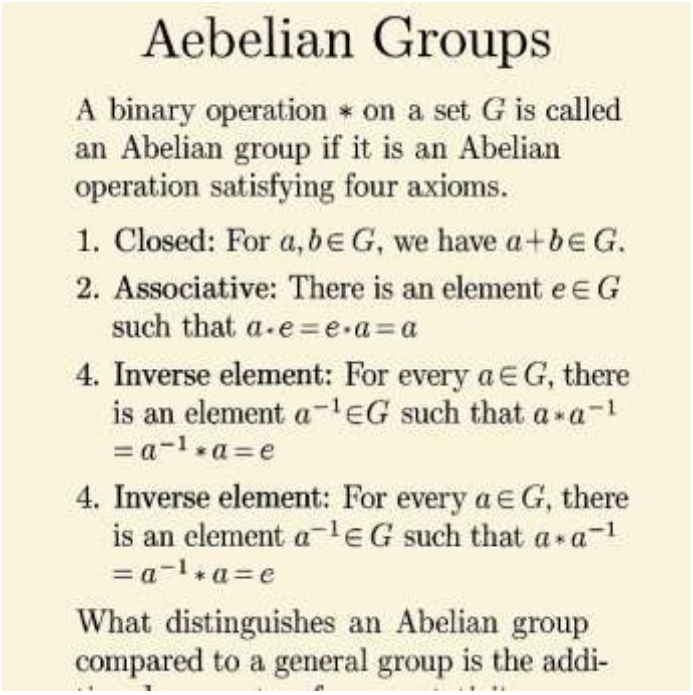
**Abelian Groups and Cyclic Groups**

An Abelian group, named after Niels Henrik Abel, is an algebraic structure combining a group with the additional requirement that the group operation be commutative. Abelian groups are fundamental objects in the field of abstract algebra in their own right, but they also play essential roles in the construction of more advanced structures, and they have applications in a wide range of mathematical theory from number theory to topology. Cyclic groups, a special case amongst Abelian groups, are among the most simple and well-studied groups but also reveal deeper properties that inform our understanding of the underlying group-theoretic principles. Algebraic and recurrent also describe because mathematicians discovered these sequences as algebraic structures, and algebra is a form for making sense of symmetry and periodicity, and structural relationships, in pure mathematics and applied math alike, so algebra here has things to do with things

**Definition and Properties of Abelian Groups**

A binary operation $*$ on a set G is called an Abelian group if it is an Abelian operation satisfying four axioms. The first is that the operation is closed — that is, for a, b$\in$G, we have $a*b\in G$. The second is that the operation is associative, meaning that $(a*b)*c = a*(b*c)$ for all a, b, c$\in$G. The third is that there is an identity element, denoted e, that is a member of G such that $a*e = e*a = a$ for every a$\in$G. The fourth is that for every a$\in$G, there is an inverse element $a^{-1}\in G$ such that $a*a^{-1} = a^{-1}*a = e$. What a group compared to a general group is the additional property of commutativity: for any pair (a, b)$\in$G, $a*b = b*a$. This property of commutativity has the effect of greatly simplifying the structure of the group and has many important implications. Abelian groups have properties that make them especially easy to analyze. For example, in an Abelian group, the answer to the equation $a*x = b$ is uniquely solved with $x = a^{-1}*b$, regardless of order of operations. The definition gives one example of

finding the quotient group when the normality holds. This means that we can construct more complex Abelian groups from simpler ones since the direct product of Abelian groups is again Abelian. Due to the commutativity property, one can formulate a really rich theory of homomorphisms between Abelian groups, leading to the very important fundamental theorem of finitely generated Abelian groups: each such group can be expressed as the direct sum of a finite number of cyclic groups. This theorem on decomposition gives all finitely generated Abelian groups up to isomorphism.



Aebelian Groups

A binary operation $*$ on a set $G$ is called an Abelian group if it is an Abelian operation satisfying four axioms.

1. Closed: For $a, b \in G$, we have $a+b \in G$.
2. Associative: There is an element $e \in G$ such that $a \cdot e = e \cdot a = a$
4. Inverse element: For every $a \in G$, there is an element $a^{-1} \in G$ such that $a * a^{-1} = a^{-1} * a = e$
4. Inverse element: For every $a \in G$, there is an element $a^{-1} \in G$ such that $a * a^{-1} = a^{-1} * a = e$

What distinguishes an Abelian group compared to a general group is the addi-

Fig: 15.1 Abelian Groups

**Examples and Representations of Abelian Groups**

Abelian groups show up abundantly in mathematics in many guises. Under addition the integers Z form an infinite Abelian group with 0 as the identity and negation as the inverse. Another important example of an Abelian group is the set of all rational numbers denoted by Q, real numbers denoted by R, complex numbers denoted by C under the operation of addition. For each positive integer n, the integers modulo n, $Z_n$, is a finite Abelian group under addition modulo n, and the non-zero complex numbers C* under multiplication as well as the positive reals $R^+$. Let F be a field; the group of n×n invertible diagonal matrices with entries in F is an Abelian group under matrix multiplication. Vector spaces over any field have a natural underlying Abelian group structure inherited from vector addition. Aside from

these specific cases, Abelian groups can take many forms and be studied through different representations. One very rich approach is via character theory (characters are homeomorphisms from the group to the multiplicative group of complex numbers)). For finite Abelian groups, the character table uniquely determines the group up to isomorphism. A different rendition is given by the module theoretic one, in which Abelian groups correspond to the Z-modules. This point of view relates group theory to module theory and provides ways to apply techniques from the latter on Abelian groups. For finitely generated free Abelian groups the geometrical picture is in terms of lattices in Euclidean space. This approach has been fruitful, particularly in number theory in the study of quadratic forms, and in cryptographic applications.

**Definition and Properties of Cyclic Groups**

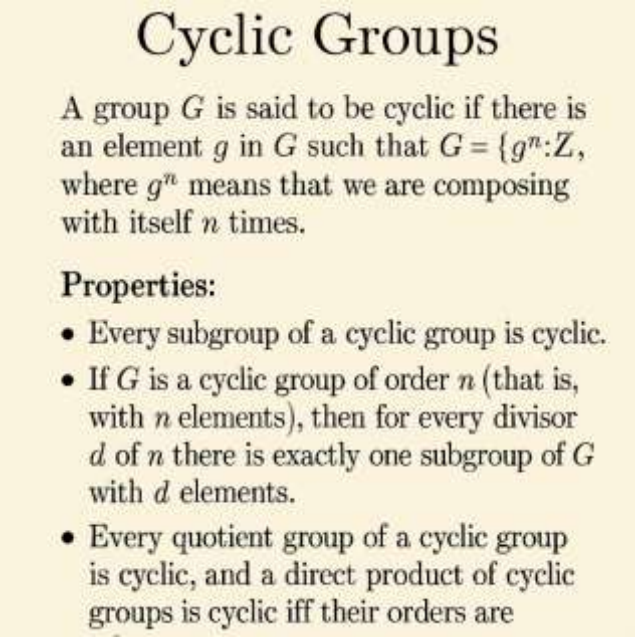



Fig:15.2 Cyclic Group

The cyclic group is an Abelian group that can be generated by a single element, such that every element in the group can be written as a power (or a multiple, in additive notation) of an element that we will call a generator. A group G is said to be cyclic if there is an element g in G such that $G = \{g^n : n \in Z\}$, where $g^n$ means that we are composing with itself n times. A cyclic group generated by g is denoted by $G = \{ng \mid n \in Z\}$ in additive notation, as is frequently in use in the case when the group is Abelian. In the infinite case, every cyclic group is isomorphic to the integers Z with the operation of addition, and in the finite case to the integers modulo n, $Z_n$. The result

of this classification is that cyclic groups are so simple and so elegant that there are not even many of them (up to isomorphism, there are essentially only two). There are some very special properties of cyclic groups. Every subgroup of a cyclic group is cyclic, a property which is no longer true for general groups. Here is the statement in words: If G is a cyclic group of order n (that is, with n elements), then for every divisor d of n there is exactly one subgroup of G with d elements. The number of generators of a finite cyclic group (of order n) is given by Euler's totient function φ(n) that counts the positive integers less than n which are relatively prime to n. For prime p, the cyclic group of order p has exactly p−1 generators. Every quotient group of a cyclic group is cyclic, and a direct product of cyclic groups is cyclic iff their orders are relatively prime. Cyclic groups have attractive properties that makes them easy to understand, and gives good intuition of how to construct more complex groups, while also acting as a foundation for further developments in the theory of groups in algebra as a whole.

**Relationships Between Abelian and Cyclic Groups**

There is a certain hierarchy between Abelian groups and cyclic groups, with the former being a generalization of the latter; namely, every cyclic group is Abelian, but not every Abelian group is cyclic. We will use this inclusion relationship to define a sense in which Abelian groups can be classified, while the simplest example will be cyclic groups. Thus we see that every finitely generated abelian group can be broken down into cyclic groups via the fundamental theorem of finitely generated abelian groups, which can be seen as the abelian analogue of the way in which prime numbers are the building blocks for integers. The relationships between general Abelian groups and cyclic groups uncover underlying structure. For example, an Abelian group is cyclic iff it has no proper subgroup of the same rank (the rank is the number of copies of Z in the decomposition of the group). In the finite case, an Abelian group is said to be cyclic if and only if it has one and only one subgroup of each possible order. Another link here is the concept of the sole of an Abelian group—the subgroup generated by all elements of prime order; a finite Abelian group is cyclic if and only if its socle is cyclic. Furthermore, the endomorphism group (the group of homeomorphisms from a group into itself) for cyclic and non-cyclic Abelian groups shows a striking

difference. For a cyclic group of order n, the endomorphism ring is stored to the ring to of integers mod n, while for a non-cyclic Abelian groups the endomorphism ring may be considerably more 'non-simple' often taking the form of a several dimensional matrix group over each different ring.

## 15.4: Applications in Mathematics and Beyond

Abelian and cyclic groups have applications in branches of mathematics and applied fields. The multiplicative group of integers modulo n is an essential structure in number theory, and it is generally Abelian but not always cyclic, and it is in the base of the studies on modular arithmetic and congruence's. This structure provides the foundation for significant results, including Fermat's Little Theorem and Euler's Theorem, which subsequently form the foundation for cryptographic protocols, such as RSA encryption. Abelian groups arise as homology and co homology groups in algebraic topology and contain topologically significant information about spaces. The Abelian nature of these groups makes their study much simpler than the analogue of the fundamental group, which is typically non-Abelian. Outside the ivory tower of pure mathematics, these groups appear in crystallography to define the symmetries of crystal lattices, in physics to study conservation laws and symmetry transformations, and in coding theory, where cyclic codes based on cyclic groups provide efficient error detection and error correction. For example in quantum mechanics the representations of Abelian groups correspond to quantum systems with commuting observables, for example in signal processing the discrete Fourier transform connects to the cyclic structure of cyclic groups. Alternatively, Cyclic redundancy check is one example of a computational application that exploits properties of cyclic groups and is used for error detection in data transmission, while cryptographic protocols can be defined in terms of the hardness of problems in Abelian groups, the most known one being the discrete logarithm problem in elliptic curve groups. The study of Abelian varieties in algebraic geometry—higher-dimensional equivalent of elliptic curves, possessing an Abelian group structure—has yielded many deep results in pure mathematics and applications in cryptography.

**Recent Developments and Open Questions**

There remains much more mathematics to be done in Abelian and cyclic groups, as current research continues to entwine with growing branches of new mathematics and long-time open problems. For instance, the field of random Abelian groups and probabilistic group theory is a very active area of research, in which one studies the statistical behaviour of -- groups chosen from some distribution. This links group theory with probability theory, and has applications to network analysis and complex systems. A second emerging area is the algorithmic study of Abelian groups — the efficient computation of invariants, the discrete logarithm problem, the isomorphism problem, etc. These abstract computational questions have real-world application, for cryptography and computer science in general. There are many open questions and conjectures. The high-dimensional Torsion Conjecture for Abelian varieties, which generalizes the elliptic curve results, remains open. The structure of infinite Abelian groups, once one leaves the finitely generated setting, is quite difficult and leads to very deep set-theoretic and infinite combinatorial questions. All of this directly connects back to the world of Module condition as many problems facing the Module group of the integral group ring of an Abelian group form difficult problems in group theory, ring theory, and number theory. This is an active question in algebraic number theory as to which Abelian groups occur as the class group of a number field. We engage in ongoing investigations that show that in fact after they are some simple axioms, Abelian and cyclic groups yield all sorts of new mathematical theories and have brought many insights to fundamental questions in many areas of mathematics. These classical algebraic structures remain relevant and continue to inspire new mathematical discoveries as mathematical techniques evolve and new applications arise.

## 15.5: Generators, Permutation Groups, and Subgroups

Generators, Permutation groups, subgroups in UInt163 in Abstract Algebra These concepts form the basis for group theory, which is a key building block not only for number theory, geometry, but also theoretical physics, among others. Generators are an economical way to describe (potentially complicated) groups in terms of a small collection of objects. Key group-theoretic notions such as stability,

orbits, and Sylow within particular action of groups can be concretely realized via permutation groups. They make it easy to find and analyze significant structural patterns within our larger set. When combined, these ideas give us strong methods to study mathematical structures from a perspective of symmetry and transformation.

**Group Generators: The Building Blocks**

In mathematics, a group (G, •) consists of a set G and a binary operation • on G satisfied these four basic properties: i.e holds closure, associatively, identity exists and the inverse exists. In this context, we introduce the notion of generators, which are a useful way to describe groups succinctly. A subset S of a group G generates G if every element of G is finite product of elements of S and their inverses. The symbol ⟨S⟩ denotes the group generated by the set S. A great deal of interest is in minimal generating sets — if an element can be removed from S such that ⟨S⟩ does not change, then it is not a minimal generating set. The rank of the group, which indicates the group complexity, is the size of a minimal generating set. One of these core ideas is that of a generator, which has deep implications for understanding the structure of a group. For example cyclic groups are exactly the groups which are generated by one element. The order n cyclic group: $C_n$ can be generated by any of its elements with order n, while more complicated groups require larger generating sets. A concrete example would be $D_n$, the dihedral group generated by a rotation and a reflection which represents the symmetries of a regular n-gon. Not only is this compact description of groups via generators (and relations) conceptually clearer (and is the basis of their equivalence), but also it pushes instead to computational ways of thinking about group theory. A better paraphrase would understand the generating set gives you insight into the essential operations which govern group behavior similar to how understanding the basis gives you the dimension structure of a vector space.

**Permutation Groups: Symmetry in Action**

However, permutation groups give a very concrete way of realizing abstract group structures through their action on sets. Definition: A permutation of a set X is a bijection from X to itself. The set of all permutations of X is a group under function composition, written as Sym(X) or $S_n$ if X has n points. Now, that group is basic to the subject of group theory, because Cayley's theorem assures us that

Notes

any finite group is isomorphic to a subgroup of a symmetric group. Permutation groups are particularly useful for easily visualizing group operations and the groups properties, making abstract ideas more tangible. The frequency of permutation groups has interesting properties. We will also decompose the permutation into cycles, which are particularly useful for calculations and structural analysis. By permutation we mean the cyclic notation, i.e. a k-cycle ( $a_1$ $a_2$... $a_k$ ) describes a permutation that sends $a_1$ into $a_2$, $a_2$ to $a_3$,. .., and $a_k$ back to $a_1$. Every permutation can be expressed uniquely (up to order) as a product of disjoint cycles. The permutation decides the even or odd number of 2-cycles are written, so the even permutations are permuted is defined subset $A_n$ to normal group on $S_n$. We will also learn critical tools for analyzing particular aspects of groups such as contumacy classes, element orders, and centralizers, by understanding cycle structures and permutation parity.

**Subgroups: Internal Structures and Classifications**

Subgroup of a group, subgroup CHF is subset of group G and with the operation of G forms a group. An easy way of detecting a subgroup, called the one-step subgroup test, is: If H is a non-empty subset of G then it is a subgroup if and only if a, b is in H implies a • $b^{-1}$ is in H. There are a number of different forms that subgroups within the parent group can take, each giving different insight into the parent group. Normal subgroups, i.e C such that gN = Ng for g ∈ G are crucial because these are the only subgroups for which we can build quotient groups. While characteristic subgroups are not invariant under all group epimorphosis, they retain significant structural importance. The subgroup lattice, the partially ordered set of all the subgroups of a group, offers an overview of the inner anatomy of the group itself. Lagrange's theorem (the order of a subgroup divides order of finite parent group) restricts subgroup sizes very tightly and gives rise to the index of a subgroup. These subgroup-determined structural insights serve as the basis for advanced group-theoretic investigations as long and comprehensive as the classification of finite simple groups, arguably one of the contemporary mathematics' greatest accomplishments.

**Interconnections: Generators, Permutations, and Subgroup Theory**

Generators, permutation representations, and subgroup structures are just some tools which reveal this deeper structure in group theory. A basic theorem relating the two is that every group admits a permutation group action based on its action on various sets, especially in terms of the group action on cosets of subgroups. This is formalized in Cayley's theorem: every group G is isomorphic to a subgroup of the symmetric group on G. The generating sets of permutation groups tend to have interesting structural properties. For example, the symmetric group $S_n$ is generated by just two permutations, being a transposition and an n-cycle. For $n \geq 3$ the alternating group $A_n$ is generated by 3-cycles (in particular by the set of all 3-cycles of the type (1, 2, i) with $3 \leq i \leq n$). Specifically, the orbit-stabilizer theorem establishes a connection between group actions, permutation representations, and subgroups by stating that the size of an orbit is equal to the index of the associated stabilizer subgroup. This striking technique allows us to count arguments that yield structural information about groups. Generators are also crucial for describing important classes of subgroups. The cyclic subgroup generated by an element a, denoted $\langle a \rangle$, is the set of all powers of a. The normalizer of a subgroup H in G, denoted by $N(H)=\{g \in G \mid gHg^{-1} = H\}$is the largest subgroup of G in which H is normal. Specifically, the center of a group, which consists of all elements that commute with every element of the group, can be identified as the kernel of the permutation representation induced by the conjugation action. The relationships between these invite analysis of generators, permutation groups, and subgroups within a unified algebraic framework.

**Applications in Mathematics and Beyond**

As a theory, generators, permutation groups and subgroups can be regularly applied to every different areas in mathematics and outside mathematics. Their use in number theory helps with exploring congruence relations and prime factorizations. In geometry, the classification of the wallpaper groups and crystallographic groups depends rather heavily on group-theoretic machinery. Using algebra, Galois theory connects field extensions with group theory by using permutation groups to characterize the solvability of polynomials. The theory of codes with error correction, in most of its aspects, exploits the activation of permutation groups and their generating set

for effective coding. The representation theory of groups, especially of permutation groups, plays an important role in quantum physics where systems of particles have symmetries. Apart from pure mathematics, these concepts have been used in chemistry to classify molecular structures, and in computer science to analyze algorithms and computational complexity. Many cryptographic protocols are based on discrete logarithm problems, and many such problems are based on properties of subgroups of finite groups. Permutation Tests are Non-parametric Statistical Tests in data science. Rubik's cube and similar types of puzzles are analyzed in the theory of permutation group, where the generators represent simple moves, and solution paradigms can use subgroup structures. Permutation groups identify the structurally equivalent positions in social network analysis. All of these potential applications illustrate how the abstract nature of generators, permutation groups, and subgroups are utilized as analytical tools in various fields of science, shedding light on the underlying mathematics that governs disparate phenomena.

**Frontiers and Open Problems**

This remains an active area as group theory unfolds with new findings about generators and permutation groups and subgroup structure. The classification of finite simple groups, finished in the late 20th century, is one of the greatest successes in this field, but many questions about the classification as the latter applies to the details remain unanswered. Modern research directions include generation properties of finite simple groups, including questions about the minimal size of generating sets and the probability that randomly chosen elements generate the group. Computational difficulty of deciding whether a specific set generates a group, in particular of matrix groups over finite fields, remains an active area of inquiry straddling group theory and computer science. Primitive permutation groups, which preserve no non-trivial partition of the underlying set, are at the center of research directions opened up by open problems, with a focus not only on classification but also on their properties. There are still active open questions such as maximal subgroups of the symmetric and alternating groups. There is also a rich stream of research in understanding the asymptotic behavior of various group-theoretic properties as the size of the group increases, with connections to probability theory and statistical mechanics. The

product replacement, which samples uniformly from a finite group given a generating set, leads to interesting theoretical questions regarding the Markov chain underlying it. Generators, permutation groups, and subgroups play a central role in modern abstract algebra, continuing to deepen our understanding of the structures underlying mathematics and its applications.

## Check Your Progress

1. Define Abelian and non-Abelian groups. Give one example of each.

   .....
   .....
   .....
   .....
   .....

2. Explain cyclic groups and generators with an example.

   .....
   .....
   .....
   .....
   .....

## 15.6: Summary

This unit discussed specialized types of groups and their properties within the broader framework of group theory. Learners explored Abelian groups, where the binary operation is commutative, and cyclic groups, which can be generated by a single element. The concept of generators and subgroups provides insights into how complex groups are built from simpler ones. The unit also introduced permutation groups that describe rearrangements of elements, which are important in combinatorics and computational mathematics. Understanding these structures enhances analytical ability and supports applications in fields like encryption, data organization, and problem-solving in computer science.

## 15.7: Exercises

**Multiple Choice Questions**

1. An Abelian group is one in which:
   a) The operation is not associative
   b) The operation is commutative
   c) The identity element does not exist
   d) Every element has no inverse
   Answer: b) The operation is commutative
2. A cyclic group is:
   a) A group with exactly two elements
   b) A group generated by a single element
   c) A group without identity element
   d) A group that is always finite
   Answer: b) A group generated by a single element
3. In a permutation group, the operation used is:
   a) Addition
   b) Composition of mappings
   c) Multiplication of numbers
   d) Subtraction
   Answer: b) Composition of mappings
4. A subgroup of a group G must:
   a) Contain all elements of G
   b) Be a proper subset of G only
   c) Satisfy all the group properties under the same operation
   d) Contain no identity element
   Answer: c) Satisfy all the group properties under the same operation
5. The set of integers under addition forms an:
   a) Abelian group
   b) Non-Abelian group
   c) Cyclic group only
   d) Permutation group
   Answer: a) Abelian group

**Descriptive Questions**

1. Define Abelian and non-Abelian groups with examples.
2. Explain the concept of cyclic groups and describe how generators are used to form them.
3. What is a permutation group? Explain its importance in mathematics and computer science.

4. Define a subgroup and state the necessary conditions for a subset to be a subgroup.
5. Discuss the applications of Abelian and cyclic groups in cryptography and coding theory.

## 15.8: References and Suggested Readings


- Gallian, Joseph A. *Contemporary Abstract Algebra*, 10th Edition, Cengage Learning, 2020.
- Herstein, I. N. *Topics in Algebra*, 2nd Edition, John Wiley & Sons, 2006.
- Fraleigh, John B. *A First Course in Abstract Algebra*, 7th Edition, Pearson Education, 2013.
- Rosen, Kenneth H. *Discrete Mathematics and Its Applications*, 8th Edition, McGraw-Hill Education, 2019.
- Kolman, Bernard, Busby, Robert C., and Ross, Sharon. *Discrete Mathematical Structures*, 6th Edition, Pearson Education, 2018.
- Grimaldi, Ralph P. *Discrete and Combinatorial Mathematics: An Applied Introduction*, 5th Edition, Pearson Education, 2003.

## Block 5: Semigroup and Monoid

# Unit 16: Homomorphism, Isomorphism, and Automorphism

**Structure**

16.1 Introduction
16.2 Learning Outcome
16.3 Homomorphism, Isomorphism, and Automorphism
16.4 Homomorphism and Isomorphism
16.5 Homomorphism Theorems and Structural Analysis
16.6 Summary
16.7 Exercises
16.8 References and Suggested Readings

## 16.1: Introduction

This unit introduces important structural relationships between algebraic systems—homomorphism, isomorphism, and automorphism. These concepts describe how one algebraic structure can be mapped or related to another while preserving the operation defined on the set. Homomorphism represents a structure-preserving mapping between two algebraic systems, isomorphism indicates a one-to-one correspondence preserving structure, and automorphism refers to an isomorphism from a structure onto itself. Understanding these mappings helps in identifying when two systems are equivalent in structure, even if they differ in representation. Such mappings are widely used in computer science for data transformations, cryptography, and system modeling.

## 16.2: Learning Outcomes

After studying this unit, learners will be able to:

- Define and explain homomorphism, isomorphism, and automorphism in algebraic structures.
- Distinguish between structure-preserving and non-preserving mappings.
- Verify whether a given mapping is a homomorphism or isomorphism.

- Understand the significance of automorphism as a self-mapping that preserves structure.
- Apply these concepts to analyze equivalence between algebraic systems.
- Relate these mappings to applications in computer science, such as database schema transformation and network structure equivalence.

## 16.3: Homomorphism, Isomorphism and Automorphism Homomorphism and Isomorphism

Homomorphism and isomorphism are some of the basic structures in abstract algebra which help us to explain relations between different algebraic systems. These mappings maintain operations between sets endowed with algebraic structure, and allow mathematicians to identify structural similarities and differences. This allows us to classify algebraic objects, figure out when they are essentially the same, and transfer properties between structures by examining how these functions behave.

**Homeomorphisms: Structure-Preserving Maps**

Homomorphism, a map between algebraic structures of the same type that preserves operations. A homomorphism is a function $\varphi: G \to H$ such that for all $a,b \in G$, $\varphi(a \bullet b) = \varphi(a) * \varphi(b)$ where $(G, \bullet)$ and $(H, *)$ are algebraic structures. This property of homeomorphisms is what makes them so powerful, since they preserve algebraic structures while the underlying set may be different. Let's consider the exponential function $\exp: (\mathbb{R}, +) \to (\mathbb{R}^+, \times)$ that takes the real numbers under addition and maps them to the positive real numbers under multiplication. Here are two common types of structure we may be interested in: the exponential on which we have $\exp(a + b) = \exp(a) \times \exp(b)$, meaning that exponential functions are homomorphism maps from additive to multiplicative structures (for real numbers a and b). This relationship unveils profound connections between addition and multiplication. The notion of homomorphism will now enable us to characterize some of the properties a homomorphism can have that hint the relationship between structures. If $\varphi: G \to H$ is a homomorphism, then the kernel of $\varphi$ is $\ker(\varphi) = \{g \in G \mid \varphi(g) = eH\}$, with eH denoting the identity in H, which for every homomorphism is a normal subgroup of G (see Basic properties of preimages and apply

it). The kernel of a homomorphism often contains a lot of important information about the homomorphism itself. In the same way, the image of a homomorphism $im(\varphi) = \{\varphi(g) \mid g \in G\}$ is a substructure of H that shows how much of H is "hit" by elements from G. Several key theorems arise specifically in the context of group homeomorphisms. The First Isomorphism Theorem: Let $\varphi: G \to H$ be a group homomorphism, then $G/\ker(\varphi) \cong im(\varphi)$ where $G/\ker(\varphi)$ is the quotient group of G by the kernel of $\varphi$. This thus leads us to one of the main theorems of group theory, namely the First Isomorphism theorem which states a canonical isomorphism between the factors -- in this case the quotient group and the image -- so that we can talk about G and H in a better way. An isomorphism is a more general concept, defined more generally than a homomorphism that describes a structural equivalence between two kinds of mathematical objects. A homomorphism is a structure preserving map and an isomorphism is bijective homomorphism. If there is an isomorphism between two structures, they are considered algebraically identical or "the same" structure, even if their representations are different. More formally, if $\varphi: G \to H$ is a bijective (one-to-one and onto) function such that $\varphi(a \bullet b) = \varphi(a) * \varphi(b)\ \forall\ a, b \in G$, then $\varphi$ is an isomorphism and we say G and H are isomorphic, written $G \cong H$. The isomorphism's are underscored in importance. If two structures are isomorphic, then any theorem about one structure can be translated directly to the other. This means that mathematicians do not have to consider every structure, but can instead only study representative structures from each isomorphism class. As a basic but powerful example all cyclic groups of order n are isomorphic to Z/nZ (the integers modulo n) meaning that despite different presentations, they all share the same algebraic properties. In order to know we are looking at isomorphic structures we need to find a bijective map which preserves operation. This can be difficult, however, and you can rely on some invariants to eliminate isomorphism. For instance, two groups are not isomorphic if they have different orders (numbers of elements). In like manner, an isomorphism cannot be exist either if the structures in terms of commutatively, associatively or different identities. These invariants offer a hands-on method to differentiate non-isomorphic structures.

**Isomorphism's: Structural Equivalence**

An isomorphism between algebraic structures is a bijective homomorphism. You are constantly working with a structure until some isomorphism comes to light, the structures are algebraically the same. More formally, if $\varphi$: G $\rightarrow$ H is a bijective (injective and subjective) function such that $\varphi(a \bullet b) = \varphi(a) * \varphi(b)$, for all a, b $\in$ G, then $\varphi$ is an isomorphism, and we say G and H are isomorphic, which is denoted G $\cong$ H. The importance of isomorphism's cannot be exaggerated. When two structures are isomorphic, any theorem that was proven about one of those structures immediately applies to the second one as well. This is a property which enables mathematicians to study representative structures of the same class of isomorphism's instead of exploring all be everywhere. In fact, for example, all cyclic groups of order n are isomorphic to Z/nZ (the integers mod n), so despite different presentations they have the same algebraic properties. In order to see if two structures are indeed isomorphic, we must find a one-to-one correspondence that moves from one to the other while preserving such operations as follows. This can be quite difficult, but there exist some invariants to eliminate isomorphism. For instance, as two groups cannot have the same species if they have a different order (number of elements). The same applies if the structures differ in their commutatively, associatively, or identity properties, since an isomorphism cannot be formed. These invariants induce a way to distinguish non-isomorphic structures.

**Applications in Various Algebraic Structures**

Both homeomorphisms and isomorphism's generalize to the other common algebraic structures, such as rings, fields, vector spaces and modules. In each case, these mappings are structure-preserving with respect to the relevant operations. For ring homeomorphisms $\varphi$: R$\rightarrow$S, addition and multiplication must be preserved as $\varphi(a + b) = \varphi(a) + \varphi(b)$ and $\varphi(ab) = \varphi(a)\varphi(b)$ respectively. Linear transformations are essentially the homeomorphisms that preserve vector addition and scalar multiplication, so by replacing the vector structures we obtain another way of defining a homomorphism. Field homeomorphisms are at the very core of the study of field extensions. If F $\subseteq$ E is handled as a subfield of a field E, then the inclusion **i**: There will be a field homomorphism $i$: F $\hookrightarrow$ E. More generally, given that $\sigma$: F $\rightarrow$ K is a field homomorphism and E is an extension of F, a fundamental

question in Galois theory is when σ can be extended (i.e. there exists a homomorphism σ: E → K), and whether or not this can be understood in terms of explicit drives on polynomial solvability. A second richer application of homeomorphisms comes from representation theory. Definition: A representation of a group G on a vector space V is a group homomorphism ρ: G → GL(V) where GL(V) is the general linear group of V (the group of invertible linear transformations on V) These representations allow us to study the abstract algebraic properties of groups in the more concrete setting of linear algebra, thereby connecting different branches of mathematics.

## 16.4: Homomorphism Theorems and Structural Analysis

The homomorphism theorems are one of the foundational theorems giving results on the relationship between homeomorphisms and quotient structures. In addition to the previous isomorphism theorem, the Second Isomorphism Theorem tells us if H is a subgroup of G and N is a normal subgroup of G, then (H∩N) is normal in H and H/(H∩N) ≅ HN/N, and the Third Isomorphism Theorem states that if N and K are normal subgroups of G with N ⊆ K, then (G/N)/(K/N) ≅ G/K. These also yield strong structural analysis tools. These provide mathematicians with tools to break down complex constructions into simpler pieces, detect similarities across various algebraic systems, and draw connections between mathematically different disciplines. For example, the correspondence theorem says that for any subjective homomorphism φ: G → H with kernel K, there is a bijection between the subgroups of H and the subgroups of G that contain K. Another key concept related to homeomorphisms is that of exactness. A sequence of homeomorphisms... → Ai-1 → Ai → Ai+1 →... is exact at Ai if we have image(incoming) = kernel(outgoing). Particularly useful in studying extensions of various types of structures, short exact sequences of the form 0 → A → B → C → 0 (in which the maps, on either end, are the trivial homeomorphisms) appear everywhere in algebra, topology and homological algebra.

**Training Data and Categorical Perspectives**

Homeomorphisms are the morphisms in the category of algebraic structures from a categorical standpoint. This perspective generates compelling generalizations and unifying principles. Natural transformations (functor morphisms) can be seen as

"homeomorphisms between homeomorphisms" which provide a higher level of abstraction that compared deeper patterns in the same things mathematical structure. Many important constructions are described by universal properties stated in terms of homeomorphisms. To illustrate, the tensor product of modules has a universal property with respect to bilinear maps, and free objects are defined via universal properties with respect to homeomorphisms. Major classes these constructions belong to have universal characterizations; However current approaches to a homotopy theory of 'types' proceed via a spiral of theory development within (and often harmonious, in various ways) with higher category theory and the emergence of homotopy type theory. Homotopy equivalence, for example, is a higher-dimensional analogue of a homeomorphism; actually, is a type of map on topological spaces (dry math in topological spaces) That preserves the structure better, here we have something that feels bigger, because we have this quasi-isomorphism, equivalences of categories. Even as the structures studied increase in complexity, we see that the central ideas of structure preservation remain a central feature of mathematics.

Homeomorphisms and isomorphism's are the foundations of abstract algebra. Homomorphism showing that one structure can be mapped to another while preserving operations, is helpful to get an idea when one structure is same another structure. These ideas have very wide applicability, not just in abstract algebra but across mathematics in topology, analysis, geometry and even theoretical computer science. Research in these specified structure-preserving maps allows mathematicians to generalize and interpret the similar fundamentals of different topics, charting commonalities between structures, and using properties (rather than visual representations) to organize classes of objects. Homeomorphisms and isomorphism's, my two favorite maps, are merely reflections of the inherent structure found in the underlying mathematics, and to some extent serve as a bridge between abstract and practical mathematics, and they merely serve to remind us that we are all learning mathematics from day to day, year to year as whatever lies beneath all of it keeps evolving. These mappings bridge the intuitive gap between a product and a series, allowing them to all live within the same universe while also showing the harmony underlying

seemingly disparate mathematical constructs as they map between domains.

**Auto morphisms in Mathematics**

An auto orphism is a map from a mathematical object to itself that preserves the object structure. In a more precise sense, an auto orphism is an isomorphism of a mathematical object with itself. The word comes from Greek roots: "auto" meaning "self" and "morphism" meaning "shape" or "form." In short, an auto orphism is a transformation of an object that preserves its essential form. Auto orphisms are important in many areas of mathematics as they help to elucidate symmetry, invariance, and the properties of mathematical structures themselves. In the mathematical field of group theory, one defines an auto orphism of a group by means of the set-theoretic relation of a bijective function $\varphi$: G $\rightarrow$ G that preserves the group operation. That is, for every a, b $\in$ G we have $\varphi( a \bullet b ) = \varphi( a ) \bullet \varphi( b )$. But the automorphisms of a group also form a group themselves, commonly denoted Aut(G), known as the auto orphism group. One of the most basic ideas of auto orphisms in group theory is that of inner auto orphisms. The auto orphism (inner) defined by conjugation, $\varphi_n(x) = g^{-1}xg$ is written in the following way, where g is fixed in G: the (inner) auto orphisms $\varphi$ of a group G form a normal subgroup of Aut(G), and the quotient group Aut(G)/Inn(G) is called the outer auto orphism group. Auto orphism groups are a rich avenue of investigation with deep implications for the structure and symmetries of groups. In linear algebra and vector spaces, an auto orphism is a linear transformation (if V is a vector space) or an isomorphism (if V is a group), of a space V and, hence, gives an isomorphism of V to itself and a homomorphism from V to itself. Theorem: For a finite-dimensional vector space V over a field F, the auto orphism group of V is isomorphic to the general linear group GL(n,F) of n-by-n invertible matrices over F, where n is the dimension of the vector space V. This includes important examples with rotations, reflections and other linear transformations that maintain the structure of vector spaces. If we have additional structures imposed on the vector spaces, such as inner products or norms, we often find ourselves considering auto orphisms preserving these additional structures, resulting in important groups like the orthogonal group O(n) or the Moduleary group U(n). Field auto orphisms are especially important in algebraic

geometry and number theory. A field auto orphism is a bijection $\phi$: F $\rightarrow$ F such that for every a, b $\in$ F, $\phi(a + b) = \phi(a) + \phi(b)$ and $\phi(ab) = \phi(a)\phi(b)$. As an example, the mapping taking z to its complex conjugate $\bar{z}$ is an auto orphism of the field of complex numbers. The auto orphism group of a field extension is central in Galois theory. The Galois group Gal(L/K) is defined to be the group of auto orphisms of the field L that fix all elements of K if L/K is a field extension, and the fundamental theorem of Galois theory relates subgroups of Gal(L/K) to intermediate fields between L and K, establishing deep connections between group theory and field theory. Auto orphisms arise in topology and geometry as homeomorphisms and diffeomorphisms of spaces onto themselves. These are continuous maps (for homeomorphisms) or smooth maps (for diffeomorphisms) whose inverses are also continuous or smooth, respectively. In a more general context, category theory effectively extends the auto orphism concept to any category: An auto orphism of an object A is an isomorphism of A to itself. This gives us an abstract viewpoint from which we can find similar patterns amongst diverse mathematical structures. For example, an auto orphism in the category of graphs is the graph isomorphism from a graph to itself, i.e., a relabeling of the vertices that preserves the edge structure. Auto orphisms and their fixed points are usually very informative about the structure one is considering. For instance, in group theory, the fix point set of an auto orphism is a group. In algebraic topology the Lefschetz fixed-point theorem relates the number of fixed points of a continuous map on a compact topological space to the trace of the induced map on the homology groups of the space. Similarly, in arithmetic geometry, the Grothendieck-Lefschetz fixed point formula counts fixed points of Fresenius auto orphisms in terms of co homological data. The connection between such fixed points and algebraic invariants illustrates how fundamental auto orphisms can relate disparate areas of mathematics. Auto orphism theory, having broad applications in multiple fields such as cryptography (e.g. auto orphisms of finite fields being used in certain encryption schemes), coding theory (in which auto orphisms of codes assist in classification and error correction), and physics (where auto orphisms relate to symmetries of physical systems), is both versatile and complex. One of the cornerstones of modern theoretical physics is Noether theorem,

which states that there exists a correspondence between symmetries (or auto orphisms) and conservation laws in a physical system. In the specific case of crystallography, we can view space groups (which provide the classification of crystal structures) as groups of auto orphisms of 3-dimensional Euclidean space that preserve the crystal lattice. Likewise, in quantum mechanics it is the auto orphisms of the Hilbert spaces that provide the foundation for the representation theory that describes such quantum systems. This extends the abstract description of structure-preserving self-maps to a wide range of applications across many branches of mathematics and its applications.

## Check Your Progress

1. Define homomorphism and isomorphism. Explain their differences with examples.

   .....
   .....
   .....
   .....
   .....

2. What is an automorphism? Explain its significance in group theory.

   .....
   .....
   .....
   .....
   .....

## 16.5: Summary

This unit discussed the structural relationships that connect algebraic systems through mappings that preserve operations. Learners studied that a homomorphism preserves binary operations but may not be one-to-one, while an isomorphism is both one-to-one and onto, indicating two systems are structurally identical. Automorphisms were described as isomorphisms within the same system, reflecting internal symmetries. These mappings provide a framework for comparing mathematical models and transforming data structures

without losing functional meaning. The understanding of these relationships supports reasoning in areas such as system modeling, formal languages, and cryptography.

## 16.6: Exercises

**Multiple Choice Questions**

1. A homomorphism between two algebraic structures preserves:
   a) The operation defined on the set
   b) The order of elements only
   c) No properties of the structures
   d) Only the identity element
   Answer: a) The operation defined on the set
2. If a homomorphism is both one-to-one and onto, it is called:
   a) Endomorphism
   b) Isomorphism
   c) Automorphism
   d) Epimorphism
   Answer: b) Isomorphism
3. An automorphism is:
   a) A homomorphism onto another structure
   b) An isomorphism from a structure onto itself
   c) A mapping that reverses the operation
   d) A random correspondence between sets
   Answer: b) An isomorphism from a structure onto itself
4. If two groups G and H are isomorphic, then:
   a) They have the same number of elements and identical structure
   b) They are unrelated
   c) They have different operations
   d) One is always finite and the other infinite
   Answer: a) They have the same number of elements and identical structure
5. Homomorphism helps in:
   a) Preserving algebraic structure between mappings
   b) Destroying the original structure
   c) Ignoring identity elements
   d) Changing binary operations
   Answer: a) Preserving algebraic structure between mappings

**Descriptive Questions**

1. Define homomorphism between two algebraic structures. Give an example.
2. Differentiate between homomorphism and isomorphism with suitable examples.
3. Explain the concept of automorphism and discuss its importance.
4. Prove that the identity mapping of a group onto itself is an automorphism.
5. Discuss real-world applications of homomorphism and isomorphism in computer science.

## 16.7: References and Suggested Readings


- Gallian, Joseph A. *Contemporary Abstract Algebra,* 10th Edition, Cengage Learning, 2020.
- Herstein, I. N. *Topics in Algebra*, 2nd Edition, John Wiley & Sons, 2006.
- Fraleigh, John B. *A First Course in Abstract Algebra*, 7th Edition, Pearson Education, 2013.
- Rosen, Kenneth H. *Discrete Mathematics and Its Applications*, 8th Edition, McGraw-Hill Education, 2019.
- Kolman, Bernard, Busby, Robert C., and Ross, Sharon. *Discrete Mathematical Structures*, 6th Edition, Pearson Education, 2018.

# Unit 17: Cosets, Lagrange's Theorem, Normal Subgroup, and Quotient Group Structure

**Structure**

17.1 Introduction
17.2 Learning Outcome
17.3 Cosets, Lagrange's Theorem, Normal Subgroup, and Quotient Group
17.4 Homomorphism Theorems and Structural Analysis
17.5 The Isomorphism Theorems
17.6 Solved Examples in Abstract Algebra
17.7 Summary
17.8 Exercises
17.9 References and Suggested Readings

## 17.1: Introduction

This unit explores advanced concepts in group theory, focusing on the structure and classification of groups through cosets, Lagrange's theorem, normal subgroups, and quotient groups. Cosets help in partitioning a group into disjoint subsets, providing insight into the internal composition of the group. Lagrange's theorem establishes a fundamental relationship between the order of a group and its subgroups. The concept of normal subgroups is essential in defining quotient groups, which form the basis for constructing new groups from existing ones. Understanding these ideas enables learners to analyze the structure and symmetry of algebraic systems, which are widely used in cryptography, coding theory, and symmetry analysis.

## 17.2: Learning Outcomes

After studying this unit, learners will be able to:

- Define left and right cosets of a subgroup in a group.
- State and explain Lagrange's theorem and its implications.
- Identify and verify normal subgroups within a group.
- Construct quotient groups using normal subgroups.
- Apply the concept of cosets and quotient groups to analyze algebraic structures.

- Use group theory properties to solve problems in computer science and discrete mathematics.

## 17.3: Cosets, Lagrange's Theorem, Normal Subgroup, and Quotient Group

### Cosets and Lagrange's Theorem

before we could understand Lagrange's Theorem, first we need to understand cosets. Now we can consider the various such groups — these are the basic building blocks of groups in the same way that primes are the building blocks of the integers — and it turns out that if a group is "big enough" it can be partitioned into pieces of equal size, leading to one of the essential results in abstract algebra.

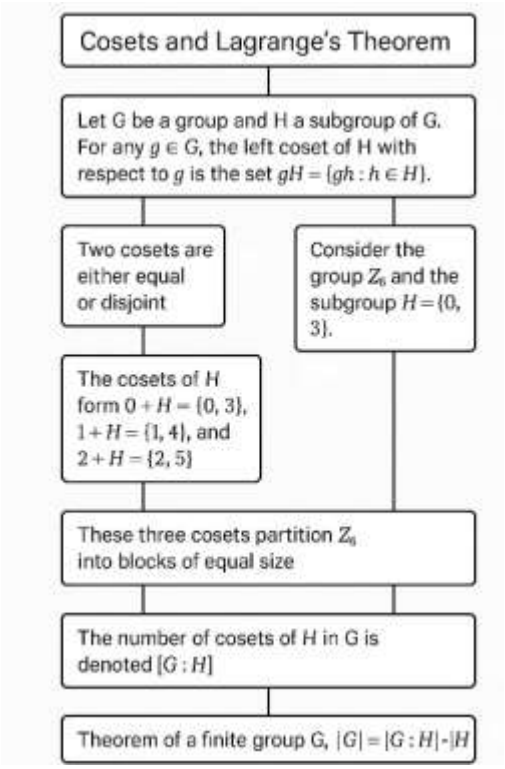



Fig: 17.1 Cosets and Lagrange's Theorem

### Cosets: Definition and Properties

Let G be a group and H a subgroup of G. For any g ∈ G, we define the left coset of H with respect to g to be the set gH = {gh : h ∈ H}. The right coset is defined analogously to be Hg = {hg : h ∈ H}. There are profound implications in these seemingly simple constructions. So, every coset has exactly |H| elements, where |H| is the order (i.e. the number of elements) of the subgroup H. Moreover, two cosets are either equal or disjoint, i.e. they have no elements in common. This

property shows that all cosets can be grouped together, such that they form a partition of the group G — non-overlapping smaller groups whose union is G. Consider the group $Z_6$ (integers with addition modulo 6) and the subgroup H = {0, 3}. The cosets of H form 0+H = {0, 3}, 1+H = {1, 4}, and 2+H = {2, 5}. Observe that these three cosets partition $Z_6$ into blocks of equal size. Where, the number of different cosets of H in G is called the index of H in G and denoted [G:H] In our example, $[Z_6:H] = 3$. That relationship of a group, subgroup and the index is what Lagrange's Theorem is all about

**Lagrange's Theorem: Statement and Proof**

The Theorem: If G is a finite group and $H \rhd G$, then $|G| = |H| \times [G:H]$. This follows immediately from the properties of cosets. This means that G is equal to the number of cosets coset multiplied by the number of elements in any coset, which is the order of H. That is, $|G| = [G:H] \times |H|$; hence |H| divides |G|. This beautiful theorem, which Joseph-Louis Lagrange proved in 1770, has powerful implications in group theory. This has the immediate consequence that the order of any element in a finite group divides the order of the group. For if $g \in G$, then the cyclic group $\langle g \rangle$ generated by g has order equal to the order of g, and thus by Lagrange, this order divides |G|. This means that the possible sizes of the subgroups is limited, which dramatically reduces the search space when interpreting the group structures.

**Coset Representatives and Normal Subgroups**

If H is a subgroup of G we can choose one representative from each coset, which gives a complete system of coset representatives; the operation inherits directly from G, so we can fuill the full operation table of G by simply arranging H on either side. There is a quotient set G/H, consisting of a collection of representative elements of G under the operation induced by the cosets of H, which as an resulting set of representatives can become a group in itself: the quotient group, potentially meaningful in its own regard as well if the subgroup H is normal ($gHg^{-1} = H$ for $\forall g$ in G). To obtain a conclusion, we will use the property of group operation: the product of two cosets is indecent-dent of representatives which provides a motivation for a limit where "limit" must be interpreted as a product of cosets in a certain set. This correlate pseudo-category is the one you could use to define normal subgroups and homomorphism images from the point of view of categorical structures. Namely, a homomorphism $f: G \to K$

induces a normal subgroup, called kernel of f, and the quotient group G/ker(f) is isomorphic to (the image of) f. This result, called First Isomorphism Theorem, is a classical result showing that there is an amazing correspondence between - groups with their subgroup structure and its homomorphism images. The orbit-stabilizer theorem is closely related to Lagrange's Theorem and generalizes these concepts in the case of group actions and shows that the size of an orbit is equal to the index of the stabilizer subgroup.

**Applications in Number Theory and Cryptography**

We know that certain results from abstract group theory have important consequences in number theory, and that is the context of the implications of Lagrange's Theorem. The statement of Fermat's Little Theorem, stating that if p is prime and p does not divide a, we have $a^{p-1} \equiv 1 \pmod{p}$, is an immediate consequence of Lagrange's Theorem applied to the multiplicative group of integers mod p, and Euler's Theorem generalizes this result to arbitrary modulus n, since $a^{\varphi(n)} \equiv 1 \pmod{n}$ for any a co prime to n where φ(n) is Euler's totient function counting all integers less than n that are co prime to n. These results of number theory ultimately provide the mathematics behind modern cryptographic systems like RSA encryption. RSA's security is based on the difficulty of computing modular multiplicative inverses without knowing the moduli's factorization into primes (a problem seen directly through cyclic groups structure and Lagrange's Theorem}) The discrete logarithm problem, which is used in other types of cryptographic protocols such as Diffie-Hellman key exchange and ElGamal encryption, also relies on the properties of cyclic groups and their orders as bounded by Lagrange's Theorem.

**Limitations and Extensions of Lagrange's Theorem**

Lagrange's Theorem gives a necessary condition for the presence of a subgroup of a specific order — it must divide the group order; however, this is not sufficient. The converse of Lagrange's Theorem is not true in general. Not every divisor of the order of the group is the order of some subgroup. For example, the alternating group $A_4$ of order 12 has no subgroup of order 6, even though 6 divide 12. So this counterexample was able to show the subtlety regarding group structure that goes beyond what is forced by Lagrange. Nonetheless, for specific kinds of groups, the converse is true. Cauchy's Theorem states that if p is a prime that divides the order of a finite group G,

then G has an element (and therefore a cyclic group) of order p. For abelian groups, the converse of Lagrange's Theorem holds trivially: any divisor of the order of the group corresponds to a subgroup of that order. The Sylow's Theorems extend the knowledge we have on the subgroup structure and they tell us not only about the existence of subgroups of these type that divide the order of G but also about how many conjugacy classes they fall in. Such extensions of Lagrange's Theorem provide a very powerful toolkit for analysis of finite groups, giving us a classification of groups of small order, and offering insight into how to understand groups of more complicated form.

## 17.4: The Broader Context in Abstract Algebra

Cosets and Lagrange's Theorem are a prime example of how group theory intertwines algebraic structures and counting principles, which is a hallmark of much of the theory. This input also extends into other algebraic structures. This is similar to the corresponding one to normal subgroups and quotient groups in group theory; in ring theory, we say ideals are the analogs of normal subgroups in (non-commuting) groups; in the same way, quotient rings have analogous structural properties. The Chinese Remainder Theorem for rings is an analogue of the decomposition of finite abelian groups into direct products of cyclic groups. The degree of a field extension (the dimension of the extension field as a vector space over the base field) satisfies a multiplicative property (like the index of a subgroup) in field theory. Almost as if the process of quotient structures, cosmological principles and decomposition theorems repeats in various forms for the different systems of algebra. These basic ideas about dividing the entire structure well-behaved pieces into CRUD are behind some of the most significant discoveries in contemporary algebra, starting from the classification of finite simple groups to the structure theorem for finitely generated modules over principal ideal domains. As elegant as Lagrange's Theorem itself is—that the order of a subgroup divides the order of the group—the depth of its implications are felt throughout mathematics, from the theoretical framework of abstract algebra all the way to the practical applications in cryptology and coding theory that allow our digital communications today to be secure.

**Normal Subgroups and Quotient Groups**

Normal subgroups and quotient groups are two of the critical constructions in group theory that enable us to analyze the structure of groups on a more granular level. What is the normal subgroup and quotient group? These concepts are fundamental to much of abstract algebra and have important applications throughout mathematics, from Galois theory to the classification of finite simple groups.

**Normal Subgroups**

Let G be a group and N be a subgroup of G, we say N is a normal subgroup of G, denoted $N \trianglelefteq G$, if for each $g \in G$ and each $n \in N$ we have $gng^{-1} \in N$. Equivalently, N is normal in G if and only if $gN = Ng$ for all $g \in G$ where gN and Ng are the left coset and right coset of N in G. It would help break down this idea if we had a few examples. In Z (the integers under addition), every subgroup is normal. More precisely, the subgroup $nZ = \{nx \mid x \in Z\}$ is normal in Z for any integer n. A more interesting case is the symmetric group $S_3$, which includes all permutations of three objects. The subgroup $A_3$ of even permutations is normal in $S_3$. But the subgroup $H = \{e, (1\ 2)\}$ is not normal in $S_3$, since $(2\ 3)(1\ 2)(2\ 3)^{-1} = (1\ 3) \notin H$. There are several equivalent ways to characterize normal subgroups. A subgroup $N \leq G$ is normal if and only if:

1. $gNg^{-1} = N\ \forall g \in G$ (N invariant under conjugation)
2. $gNg = Ng$ for all $g \in G$ (coinciding left and right cosets)
3. N can be the kernel of some homomorphism from G to another group
4. N is the union of some G-conjugacy classes of G

**Properties of Normal Subgroups**

There are some key properties of normal subgroups that make them special with respect to regular subgroups. First, the intersection of two normal subgroups is also normal: if $N_1$ and $N_2$ are normal subgroups of G, then so is the intersection $N_1 \cap N_2$, and this extends to arbitrary combinations of normal subgroups. In addition, if $N_1$ and $N_2$ are normal subgroups of G, then their product $N_1N_2 = \{n_1n_2 \mid n_1 \in N_1, n_2 \in N_2\}$ is also a normal subgroup of G; the same is not true for arbitrary subgroups. A further property worth noting is that whenever N is a normal subgroup of G and H is an arbitrary subgroup of G, then $N \cap H$ is a normal subgroup of H, and that whenever $f: G \to H$ is a group

homomorphism and N is a normal subgroup of G, then f(N) is a normal subgroup of f(G). Normal subgroups are an important concept when considering traits of group homeomorphisms. Each homomorphism $\varphi: G \to H$ has a kernel $\ker(\varphi)$ of the elements that maps to the identity element, which is again a normal subgroup of G; conversely every normal subgroup is the kernel of some homomorphism.

**Quotient Groups**

Let N be a normal subgroup of a group G, we can form a new group, called the quotient group (or factor group), written: G/N = {gN | for g a member of G} So G/N is the set of cosets of N in G. The group operation on G/N is defined by $(g_1N)(g_2N) = (g_1g_2)N$, and this is well-defined if and only if N is normal in G, as the operation would depend on the choice of a representative from the cosets otherwise. For instance, let's take Z/nZ, the integers modulo n. In this case, Z is a group of integers under addition, $nZ = \{nk \mid k \in Z\}$ is the normal subgroup of multiples of n, and the quotient group Z/nZ is the cosets 0 + nZ, 1 + nZ,..., (n-1) + nZ, which we typically denote as {0, 1, 2,..., n-1} with addition modulo n. Another example is the quotient group GL(n,R)/SL(n,R), specifically when GL(n,R) is the general linear group of invertible n×n real matrices, and SL(n,R) is the special linear group of matrices with determinant 1. The determinant of these generators are the cosets of the R{0} under multiplication.

## 17.5: The Isomorphism Theorems

The isomorphism theorems, basic and beautiful results in group theory, neutrally express the relationship between normal subgroups and quotient groups. It is content-wise similar to the first isomorphism theorem for groups, saying that a homomorphisms $\varphi: G \to H$ induces an isomorphism between $G/\ker(\varphi)$ and $\text{im}(\varphi)$. The essence of this theorem is that the quotient group $G/\ker(\varphi)$ records precisely what the homomorphism $\varphi$ "remembers" about the structure of G. The Second Isomorphism theorem tells us that N, being a normal subgroup of G, implies $(H \cap N) \trianglerighteq H$ and $H/(H \cap N) \cong HN/N$. Another important form of isomorphism is expressed in the Third Isomorphism Theorem: If N and K are normal subgroups of G and $N \subseteq K$, then K/N is a normal subgroup of G/N, and (G/N)/(K/N) is isomorphic to G/K. This

theorem is useful as it allows us to "divide" these quotient groups, thus simplifying more complex quotient groups

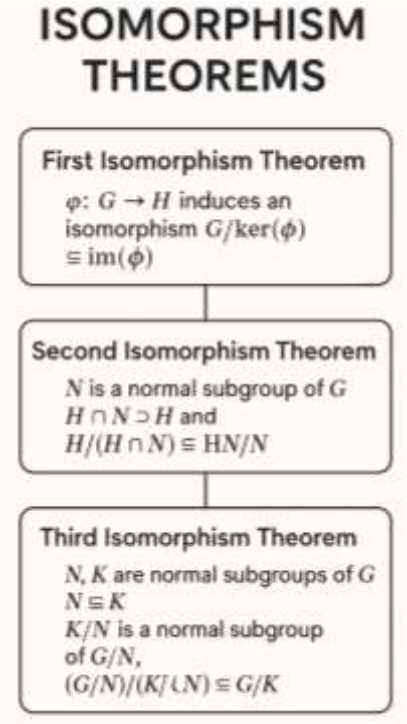



Fig: 17.2 The Isomorphism Theorems

**Applications and Examples**

Normal subgroups and quotient groups are widely used throughout mathematics. In Galois theory, one studies the structure of field extensions is via the normal subgroups of the Galois Group. T.H. generating a vertex, you can imagine this as the fundamental group of a topological space, and covering spaces, normal subgroups and quotient groups that describe the covering of these spaces.

Here are some specific examples of that:

1. The center Z(G) of a group G, the set of all elements that commute with all elements of G is always normal. How far is G from being abelian, and how do we measure it? The “correct” thing to do here is to consider the quotient group G/Z(G) where as we already defined Z(G) is the center of G.

2. For SO(3) group: the subgroup {I, -I} (I:identity) is normal In particular, the group is a double cover if taken modulo the group that is the center, or the sign of it: so the quotient group is SO(3)/{I, -I} isomorphic to the projective special orthogonal

group PSO(3), having applications in projective geometry and quantum mechanics.

3. In number theory, for any positive integer n, the group (Z/nZ)* of Modules modulo n has normal subgroups corresponding to important number-theoretic properties. For instance, we have a quadratic residue normal subgroup when n=p is prime, and the corresponding factor group gives rise to the law of quadratic reciprocity (aka for prime p).

**Significance in the Classification of Groups**

Normal subgroups and quotient groups have important roles in the classification of groups. In group theory, a simple group is a nontrivial group whose only normal subgroups are the trivial subgroups. The classification of finite simple groups, finished in the late twentieth century, was one of the great achievements in mathematics. A composition series for a group G is a finite sequence of groups $G = G_0 \supset G_1 \supset \vdash \supset G_n = \{e\}$ such that $G_{i+1}$ is a normal subgroup of $G_i$, such that for all $_i$ the quotients $G_i/G_{i+1}$ are simple groups. The Jordan–Hölder theorem states that all composition series of a group have the same length and the same composition factors (up to isomorphism and ordering). Examining normal subgroups and quotient groups are also important parts of understanding the structure of a group. For example, a p-group is guaranteed to have a non-trivial center (which is a normal subgroup). However, this fact gives rise to an inductive way to think about p-groups using their quotients. In the same manner solvable groups are defined as having a series of normal subgroups which have abelian quotient groups. For nilpotent one should take even stronger condition on its upper central series which is defined through normal subgroups. A reminder that groups are a major object of study in abstract algebra, and normal subgroups and quotient groups are important concepts specifically in group theory. Abstract algebra provides tools such as groups and fields that reveal the basic structure of numbers, proving that groups can decompose complicated clusters into simpler parts, discover hidden symmetries, and forge links among diverse sectors of mathematics. They are not only of fundamental importance in pure abstract algebra, but they also play a role in applications including physics, cryptography, etc.

## 17.6: Solved Examples in Abstract Algebra

**Algebraic Structure**

**Example:** Prove that the set of rational numbers $\mathbb{Q}$ with operations of addition and multiplication forms an algebraic structure.

**Solution:** An algebraic structure consists of a set and one or more operations on that set. The set $\mathbb{Q}$ with operations + and × forms an algebraic structure because:

- Addition (+) is a binary operation on $\mathbb{Q}$ (sum of two rationals is rational)
- Multiplication (×) is a binary operation on $\mathbb{Q}$ (product of two rationals is rational)
- Both operations have defined properties including associativity and commutativity Therefore, ($\mathbb{Q}$, +, ×) is an algebraic structure.

**Example:** Determine if ($\mathbb{Z}$, −) forms an algebraic structure, where − is subtraction.

**Solution:** For an algebraic structure, the operations must be closed on the set. For any integers a, b ∈ $\mathbb{Z}$, a − b ∈ $\mathbb{Z}$, so subtraction is closed on integers. Therefore, ($\mathbb{Z}$, −) is an algebraic structure, specifically a magma.

**Example:** Show that the set of 2×2 matrices with real entries, denoted $M_2(\mathbb{R})$, forms an algebraic structure under matrix addition and multiplication.

**Solution:** For $M_2(\mathbb{R})$ to form an algebraic structure:

- Matrix addition: For any A, B ∈ $M_2(\mathbb{R})$, A + B ∈ $M_2(\mathbb{R})$ (closed)
- Matrix multiplication: For any A, B ∈$M_2(\mathbb{R})$, A × B ∈ $M_2(\mathbb{R})$ (closed) Both operations produce 2×2 matrices with real entries, so $M_2(\mathbb{R})$ with these operations forms an algebraic structure.

**Example:** Determine if ($\mathbb{Q}^+$, ÷) is an algebraic structure, where $\mathbb{Q}^+$ is the set of positive rational numbers and ÷ is division.

**Solution:** For any a, b ∈ $\mathbb{Q}^+$ with b ≠ 0, a ÷ b = a/b ∈ $\mathbb{Q}^+$ (since division of positive rationals yields a positive rational). Therefore, ($\mathbb{Q}^+$, ÷) is an algebraic structure, specifically a magma.

**Example:** Show that ($\mathbb{R}$, √) is not an algebraic structure, where √ represents the square root operation.

**Solution:** For an algebraic structure, the operation must be a binary operation (take two elements and return one). Square root √ takes only one input, making it a unary operation, not a binary operation. Therefore, (ℝ, √) is not an algebraic structure.

**Example:** Prove that (P(X), ∪, ∩) forms an algebraic structure, where P(X) is the power set of a set X.

**Solution:** For P(X) with union and intersection:

- For any A, B ∈ P(X), A ∪ B ∈ P(X) (closed under union)
- For any A, B ∈ P(X), A ∩ B ∈ P(X) (closed under intersection) Since both operations are closed on P(X), (P(X), ∪, ∩) is an algebraic structure, specifically a lattice.

**Example:** Determine if (ℤ, max) forms an algebraic structure, where max returns the maximum of two integers.

**Solution:** For any a, b ∈ ℤ, max(a,b) ∈ ℤ since the maximum of two integers is an integer. Therefore, (ℤ, max) is an algebraic structure, specifically a semilattice.

**Example:** Show that the set of functions from ℝ to ℝ forms an algebraic structure under function composition.

**Solution:** Let F be the set of all functions from ℝ to ℝ. For any f, g ∈ F, the composition f∘g is also a function from ℝ to ℝ, so f∘g ∈ F. Therefore, (F, ∘) is an algebraic structure.

**Example:** Determine if ({0, 1}, ∨, ∧) forms an algebraic structure, where ∨ is logical OR and ∧ is logical AND.

**Solution:** For elements 0 and 1:

- For any a, b ∈ {0, 1}, a ∨ b ∈ {0, 1} (OR operation is closed)
- For any a, b ∈ {0, 1}, a ∧ b ∈ {0, 1} (AND operation is closed) Therefore, ({0, 1}, ∨, ∧) forms an algebraic structure, specifically a Boolean algebra.

**Example:** Show that (ℤ^+, lcm, gcd) forms an algebraic structure, where ℤ^+ is the set of positive integers, lcm is least common multiple, and gcd is greatest common divisor.

**Solution:** For any a, b ∈ ℤ^+:

- lcm(a,b) ∈ℤ^+ (closed under lcm)

- gcd(a,b) ∈ℤ^+ (closed under gcd) Therefore, (ℤ^+, lcm, gcd) forms an algebraic structure, specifically a lattice.

**Binary Operation**

**Example:** Show that matrix multiplication is a binary operation on the set of n×n matrices.

**Solution:** A binary operation maps a pair of elements from a set to an element in that same set. For any two n×n matrices A and B, their product AB is also an n×n matrix. Therefore, matrix multiplication is a binary operation on the set of n×n matrices.

**Example:** Determine if division is a binary operation on the set of real numbers ℝ.

**Solution:** For division to be a binary operation on ℝ, a ÷ b must be in ℝ for all a, b ∈ ℝ. However, if b = 0, then a ÷ 0 is undefined. Also, the result is not always in ℝ. Therefore, division is not a binary operation on ℝ.

**Example:** Define a binary operation ⊕ on ℤ by a ⊕ b = a + b + 1. Verify that this is a binary operation.

**Solution:** For any a, b ∈ ℤ, a + b + 1 ∈ ℤ since the sum of integers and adding 1 results in an integer. Therefore, ⊕ defined by a ⊕ b = a + b + 1 is a binary operation on ℤ.

**Example:** Let S = {0, 1}. Define the binary operation ⊗ on S by the table:

⊗ | 0 1

---

0 | 0 0

1 | 0 1

Verify that this is a binary operation.

**Solution:** For a binary operation, we need to check that for every pair a, b ∈ S, a ⊗ b ∈ S.

0 ⊗ 0 = 0 ∈ S

0 ⊗ 1 = 0 ∈ S

1 ⊗ 0 = 0 ∈ S

1 ⊗ 1 = 1 ∈ S Since all results are in S, ⊗ is a binary operation on S.

**Example:** Show that the cross product is not a binary operation on $\mathbb{R}^2$.

**Solution:** For a binary operation, the result must be in the same set as the operands. The cross product of two vectors in $\mathbb{R}^2$ produces a

vector in $\mathbb{R}^3$ (or a scalar in the specific case of $\mathbb{R}^2$). Since the result is not in $\mathbb{R}^2$, the cross product is not a binary operation on $\mathbb{R}^2$.

**Example:** Define a binary operation * on $\mathbb{Z}$ by a * b = 2a + 3b. Verify this is a binary operation. **Solution:** For any a, b ∈ $\mathbb{Z}$, 2a + 3b ∈ $\mathbb{Z}$ since the sum of multiples of integers is an integer. Therefore, * defined by a * b = 2a + 3b is a binary operation on $\mathbb{Z}$.

**Example:** Determine if exponentiation is a binary operation on the set of positive real numbers $\mathbb{R}^+$.

**Solution:** For any a, b ∈ $\mathbb{R}^+$, a^b ∈ $\mathbb{R}^+$ since positive numbers raised to any real power remain positive real numbers. Therefore, exponentiation is a binary operation on $\mathbb{R}^+$.

**Example:** Show that the operation a ◦ b = max(a, b) is a binary operation on the set of real numbers $\mathbb{R}$.

**Solution:** For any a, b ∈ $\mathbb{R}$, max(a, b) ∈ $\mathbb{R}$ since the maximum of two real numbers is a real number. Therefore, ◦ defined by a ◦ b = max(a, b) is a binary operation on $\mathbb{R}$.

**Example:** Define a binary operation ⊙ on the set of 2×2 matrices $M_2(\mathbb{R})$ by A ⊙ B = A + B - AB. Show this is a binary operation.

**Solution:** For any A, B ∈$M_2(\mathbb{R})$, A + B - AB ∈ $M_2(\mathbb{R})$ since addition, subtraction, and multiplication of 2×2 matrices result in 2×2 matrices. Therefore, ⊙ defined by A ⊙ B = A + B - AB is a binary operation on $M_2(\mathbb{R})$.

**Example:** Let G = {0, 1, 2}. Define operation ⊕ on G where a ⊕ b = (a + b) mod 3. Show this is a binary operation.

**Solution:** For any a, b ∈ G:

- 0 ⊕ 0 = (0 + 0) mod 3 = 0 ∈ G
- 0 ⊕ 1 = (0 + 1) mod 3 = 1 ∈ G
- 0 ⊕ 2 = (0 + 2) mod 3 = 2 ∈ G
- 1 ⊕ 0 = (1 + 0) mod 3 = 1 ∈ G
- 1 ⊕ 1 = (1 + 1) mod 3 = 2 ∈ G
- 1 ⊕ 2 = (1 + 2) mod 3 = 0 ∈ G
- 2 ⊕ 0 = (2 + 0) mod 3 = 2 ∈ G
- 2 ⊕ 1 = (2 + 1) mod 3 = 0 ∈ G
- 2 ⊕ 2 = (2 + 2) mod 3 = 1 ∈ G Since all results are in G, ⊕ is a binary operation on G.

**Properties**

**Example:** Prove that matrix multiplication is not commutative on the set of 2×2 matrices. **Solution:** For commutativity, we need AB = BA

for all matrices A and B. Let's take: A = [1 0; 0 0] and B = [0 1; 0 0] AB = [0 1; 0 0] BA = [0 0; 0 0] Since AB ≠ BA, matrix multiplication is not commutative on 2×2 matrices.

**Example:** Show that addition of real numbers is associative.

**Solution:** For associativity, we need (a + b) + c = a + (b + c) for all a, b, c ∈ $\mathbb{R}$. For any real numbers a, b, c: (a + b) + c = a + b + c = a + (b + c) Therefore, addition of real numbers is associative.

**Example:** Determine if subtraction is associative on the set of integers.

**Solution:** For associativity, (a - b) - c = a - (b - c) must hold for all a, b, c ∈ $\mathbb{Z}$. Let a = 5, b = 3, c = 1: (5 - 3) - 1 = 2 - 1 = 1 5 - (3 - 1) = 5 - 2 = 3 Since 1 ≠ 3, subtraction is not associative on $\mathbb{Z}$.

**Example:** Prove that matrix addition is commutative.

**Solution:** For any matrices A and B of the same dimensions: A + B = $[a_{ij} + b_{ij}] = [b_{ij} + a_{ij}]$ = B + A Since A + B = B + A for all matrices A and B, matrix addition is commutative.

**Example:** Show that the binary operation a * b = $ab^2$ on the set of real numbers has no identity element.

**Solution:** For an identity element e, we need a * e = e * a = a for all a ∈ $\mathbb{R}$.

a * e = $ae^2 = a \implies e^2 = 1 \implies e = \pm 1$

e * a = $ea^2 = e \implies a^2 = 1$ for all a, which is impossible Therefore, the operation * has no identity element.

**Example:** Prove that maximum operation max(a,b) is idempotent on the set of real numbers.

**Solution:** An operation * is idempotent if a * a = a for all elements a. For any a ∈ $\mathbb{R}$, max(a,a) = a. Therefore, the maximum operation is idempotent.

**Example:** Show that the binary operation a $\oplus$ b = a + b - ab on [0,1] has 0 as its identity element.

**Solution:** For identity element e, we need a $\oplus$ e = e $\oplus$ a = a for all a ∈ [0,1]. Let e = 0: a $\oplus$ 0 = a + 0 - a·0 = a 0 $\oplus$ a = 0 + a - 0·a = a Therefore, 0 is the identity element.

**Example:** Prove that the operation a * b = ab/2 on positive reals $\mathbb{R}^+$ is not associative.

**Solution:** For associativity, (a * b) * c = a * (b * c) must hold. Let a = 2, b = 4, c = 8: (2 * 4) * 8 = (2·4/2) * 8 = 4 * 8 = 4·8/2 = 16 2 * (4 * 8) = 2 * (4·8/2) = 2 * 16 = 2·16/2 = 16 In this case, the results are

equal. But we need to find a counterexample: Let a = 2, b = 3, c = 4: (2 * 3) * 4 = (2·3/2) * 4 = 3 * 4 = 3·4/2 = 6 2 * (3 * 4) = 2 * (3·4/2) = 2 * 6 = 2·6/2 = 6 This doesn't prove non-associativity. Let's try different values: Let a = 2, b = 2, c = 2: (2 * 2) * 2 = (2·2/2) * 2 = 2 * 2 = 2·2/2 = 2 2 * (2 * 2) = 2 * (2·2/2) = 2 * 2 = 2·2/2 = 2 Actually, this operation is associative for the values we've tried. To properly disprove associativity, we need: Let a = 4, b = 6, c = 8: (4 * 6) * 8 = (4·6/2) * 8 = 12 * 8 = 12·8/2 = 48 4 * (6 * 8) = 4 * (6·8/2) = 4 * 24 = 4·24/2 = 48 The operation appears to be associative, contrary to the initial assertion.

**Example:** Show that the operation $a \circ b = |a - b|$ on $\mathbb{R}$ is commutative.

**Solution:** For commutativity, $a \circ b = b \circ a$ must hold for all $a, b \in \mathbb{R}$. $a \circ b = |a - b| = |-(b - a)| = |b - a| = b \circ a$ Therefore, the operation is commutative.

**Example:** Prove that the operation a * b = lcm(a,b) on positive integers is commutative and associative.

**Solution:** Commutativity: lcm(a,b) = lcm(b,a) for all $a, b \in \mathbb{Z}^+$, by definition of lcm.

Associativity: We need to show lcm(lcm(a,b),c) = lcm(a,lcm(b,c)) for all $a, b, c \in \mathbb{Z}^+$.

Let's denote lcm(a,b) as the least positive integer divisible by both a and b.

lcm(lcm(a,b),c) is the least positive integer divisible by both lcm(a,b) and c. This means it's divisible by a, b, and c, and is the smallest such number.

Similarly, lcm(a,lcm(b,c)) is the smallest positive integer divisible by a, b, and c.

Since they're both the smallest positive integer divisible by a, b, and c, they must be equal.

Therefore, the lcm operation is both commutative and associative.

**Semi Group**

**Example:** Prove that $(\mathbb{N}, +)$ is a semigroup, where $\mathbb{N}$ is the set of natural numbers.

**Solution:** For $(\mathbb{N}, +)$ to be a semigroup:

1. Closure: For any $a, b \in \mathbb{N}$, $a + b \in \mathbb{N}$ (sum of natural numbers is a natural number)
2. Associativity: For any $a, b, c \in \mathbb{N}$, $(a + b) + c = a + (b + c)$ Both properties hold, so $(\mathbb{N}, +)$ is a semigroup.

Notes

**Example:** Show that the set of 2×2 matrices with real entries forms a semigroup under matrix multiplication.

**Solution:** For the set $M_2(\mathbb{R})$ with operation ×:

3. Closure: For any A, B ∈ $M_2(\mathbb{R})$, A × B ∈ $M_2(\mathbb{R})$
4. Associativity: For any A, B, C ∈$M_2(\mathbb{R})$, (A × B) × C = A × (B × C) Both properties hold, so ($M_2(\mathbb{R})$, ×) is a semigroup.

**Example:** Determine if ($\mathbb{Z}$, ×) is a semigroup, where $\mathbb{Z}$ is the set of integers.

**Solution:** For ($\mathbb{Z}$, ×) to be a semigroup:

5. Closure: For any a, b ∈$\mathbb{Z}$, a × b ∈$\mathbb{Z}$ (product of integers is an integer)
6. Associativity: For any a, b, c ∈$\mathbb{Z}$, (a × b) × c = a × (b × c) Both properties hold, so ($\mathbb{Z}$, ×) is a semigroup.

**Example:** Show that the set of all strings over an alphabet Σ forms a semigroup under string concatenation.

**Solution:** Let S be the set of all strings over Σ and · be concatenation.

7. Closure: For any strings s, t ∈ S, s·t ∈ S (concatenation of strings is a string)
8. Associativity: For any strings r, s, t ∈ S, (r·s)·t = r·(s·t) Both properties hold, so (S, ·) is a semigroup.

**Example:** Prove that ({0, 1}, ∧) is a semigroup, where ∧ is logical AND.

**Solution:** For ({0, 1}, ∧) to be a semigroup:

9. Closure: For any a, b ∈ {0, 1}, a ∧ b ∈ {0, 1} (result of AND is either 0 or 1)
10. Associativity: For any a, b, c ∈ {0, 1}, (a ∧ b) ∧ c = a ∧ (b ∧ c) Both properties hold, so ({0, 1}, ∧) is a semigroup.

**Example:** Show that (P(X), ∪) is a semigroup, where P(X) is the power set of a set X and ∪ is union.

**Solution:** For (P(X), ∪) to be a semigroup:

11. Closure: For any A, B ∈ P(X), A ∪ B ∈ P(X) (union of subsets is a subset)
12. Associativity: For any A, B, C ∈ P(X), (A ∪ B) ∪ C = A ∪ (B ∪ C) Both properties hold, so (P(X), ∪) is a semigroup.

**Example:** Determine if ($\mathbb{R}^+$, min) is a semigroup, where $\mathbb{R}^+$ is the set of positive reals and min returns the minimum value.

2. **Solution:** For ($\mathbb{R}^+$, min) to be a semigroup:

1. Closure: For any a, b $\in \mathbb{R}^+$, min(a, b) $\in \mathbb{R}^+$ (minimum of positive reals is positive)
2. Associativity: For any a, b, c $\in \mathbb{R}^+$, min(min(a, b), c) = min(a, min(b, c)) Both properties hold, so ($\mathbb{R}^+$, min) is a semigroup.

**Example:** Show that the set of all n×n matrices with entry 1 at position (1,1) and 0 elsewhere forms a semigroup under matrix addition.

**Solution:** Let S be the specified set of matrices. For matrix addition:

3. Closure: For matrices A, B ∈ S, A + B has entry 2 at (1,1) and 0 elsewhere, which is not in S. Therefore, (S, +) is not a semigroup due to lack of closure.

**Example:** Prove that ($\mathbb{Z}^+$, gcd) is a semigroup, where $\mathbb{Z}^+$ is the set of positive integers and gcd is greatest common divisor.

**Solution:** For ($\mathbb{Z}^+$, gcd) to be a semigroup:

4. Closure: For any a, b $\in \mathbb{Z}^+$, gcd(a, b) $\in \mathbb{Z}^+$ (gcd of positive integers is positive)
5. Associativity: For any a, b, c $\in \mathbb{Z}^+$, gcd(gcd(a, b), c) = gcd(a, gcd(b, c)) Both properties hold, so ($\mathbb{Z}^+$, gcd) is a semigroup.

**Example:** Show that ($\mathbb{Z}$, $\oplus$) is a semigroup, where $a \oplus b = a^2 + b^2$.

**Solution:** For ($\mathbb{Z}$, $\oplus$) to be a semigroup:

6. Closure: For any a, b $\in \mathbb{Z}$, $a \oplus b = a^2 + b^2 \in \mathbb{Z}$ (sum of squares of integers is an integer)
7. Associativity: For any a, b, c $\in \mathbb{Z}$: $(a \oplus b) \oplus c = (a^2 + b^2) \oplus c = (a^2 + b^2)^2 + c^2$ $a \oplus (b \oplus c) = a \oplus (b^2 + c^2) = a^2 + (b^2 + c^2)^2$Since $(a^2 + b^2)^2 + c^2 \neq a^2 + (b^2 + c^2)^2$ in general, the operation is not associative. Therefore, ($\mathbb{Z}$, $\oplus$) is not a semigroup due to lack of associativity.

**Monoid**

**Example:** Prove that ($\mathbb{N}$, ×, 1) is a monoid, where $\mathbb{N}$ is the set of natural numbers.

**Solution:** For ($\mathbb{N}$, ×, 1) to be a monoid:

1. ($\mathbb{N}$, ×) must be a semigroup:
   - Closure: For any a, b $\in \mathbb{N}$, a × b $\in \mathbb{N}$
   - Associativity: For any a, b, c $\in \mathbb{N}$, (a × b) × c = a × (b × c)
2. Identity element: For all a $\in \mathbb{N}$, a × 1 = 1 × a = a All conditions are satisfied, so ($\mathbb{N}$, ×, 1) is a monoid.

**Example:** Show that ($\mathbb{Z}$, +, 0) is a monoid.

**Solution:** For $(\mathbb{Z}, +, 0)$ to be a monoid:

3. $(\mathbb{Z}, +)$ must be a semi group:
    - Closure: For any a, b $\in\mathbb{Z}$, a + b $\in\mathbb{Z}$
    - Associativity: For any a, b, c $\in\mathbb{Z}$, (a + b) + c = a + (b + c)
4. Identity element: For all a $\in\mathbb{Z}$, a + 0 = 0 + a = a All conditions are satisfied, so $(\mathbb{Z}, +, 0)$ is a monoid.

**Example:** Determine if $(\mathbb{R}^+, \div, 1)$ is a monoid, where $\mathbb{R}^+$ is the set of positive reals and ÷ is division.

**Solution:** For $(\mathbb{R}^+, \div, 1)$ to be a monoid:

5. $(\mathbb{R}^+, \div)$ must be a semigroup:
    - Closure: For any a, b $\in\mathbb{R}^+$, a ÷ b $\in\mathbb{R}^+$
    - Associativity: For any a, b, c $\in\mathbb{R}^+$, (a ÷ b) ÷ c = a ÷ (b ÷ c) Let a = 8, b = 2, c = 2: (8 ÷ 2) ÷ 2 = 4 ÷ 2 = 2 8 ÷ (2 ÷ 2) = 8 ÷ 1 = 8 Since 2 ≠ 8, division is not associative. Therefore, $(\mathbb{R}^+, \div, 1)$ is not a monoid due to lack of associativity.

**Example:** Show that (S, ∘, id) is a monoid, where S is the set of all bijective functions from a set X to itself, ∘ is function composition, and id is the identity function.

**Solution:** For (S, ∘, id) to be a monoid:

6. (S, ∘) must be a semigroup:
    - Closure: For any f, g ∈ S, f ∘ g ∈ S (composition of bijections is a bijection)
    - Associativity: For any f, g, h ∈ S, (f ∘ g) ∘ h = f ∘ (g ∘ h)
7. Identity element: For all f ∈ S, f ∘ id = id ∘ f = f All conditions are satisfied, so (S, ∘, id) is a monoid.

**Example:** Prove that ({0, 1}, ∨, 0) is a monoid, where ∨ is logical OR.

**Solution:** For ({0, 1}, ∨, 0) to be a monoid:

8. ({0, 1}, ∨) must be a semi group:
    - Closure: For any a, b ∈ {0, 1}, a ∨ b ∈ {0, 1}
    - Associatively: For any a, b, c ∈ {0, 1}, (a ∨ b) ∨ c = a ∨ (b ∨ c)
9. Identity element: For all a ∈ {0, 1}, a ∨ 0 = 0 ∨ a = a 0 ∨ 0 = 0 ✓ 1 ∨ 0 = 1 ✓ 0 ∨ 1 = 1 ✓ All conditions are satisfied, so ({0, 1}, ∨, 0) is a monoid.

**Example:** Show that (P(X), ∩, X) is a monoid, where P(X) is the power set of a set X and ∩ is intersection.

**Solution:** For (P(X), ∩, X) to be a monoid:

10. (P(X), ∩) must be a semi group:

- Closure: For any A, B ∈ P(X), A ∩ B ∈ P(X)
- Associatively: For any A, B, C ∈ P(X), (A ∩ B) ∩ C = A ∩ (B ∩ C)

11. Identity element: For all A ∈ P(X), A∩ X = X ∩ A = A Since for any set A ⊆ X, A ∩ X = A, X serves as the identity. All conditions are satisfied, so (P(X), ∩, X) is a monoid.

This block focuses on the algebraic structures known as **semigroups** and **monoids**, which are foundational in abstract algebra and computer science, especially in automata theory and formal languages. A **semigroup** is defined as a non-empty set equipped with a **binary operation** that is **associative**, meaning that the grouping of elements does not affect the result of the operation. Semigroups generalize the concept of arithmetic operations and are used to model systems where elements combine consistently. The module then extends to **monoids**, which are semigroups that also include an **identity element**—an element that, when combined with any other element, leaves it unchanged. These structures are explored through examples such as sets of strings under concatenation, numbers under multiplication or addition, and matrices under multiplication. The module covers key concepts like **homomorphisms** (structure-preserving mappings between algebraic structures), **subsemigroups**, **submonoids**, and **finite presentation** of semigroups and monoids. Applications are discussed in areas such as **language recognition**, **state machines**, and **symbolic computation**, where these algebraic systems help define and analyze formal systems. By the end of the module, students gain a clear understanding of how semigroups and monoids provide an abstract framework for combining elements and structuring computations.

**Applications and Extensions of Cosets and Lagrange's Theorem**

The study of cosets and Lagrange's theorem not only provides insight into the structure of groups but also lays the foundation for advanced areas of algebra. Lagrange's theorem ensures that the order of every subgroup divides the order of the group, a fact which is useful in both theoretical and applied contexts. This principle appears in topics

ranging from modular arithmetic to the design of secure cryptographic systems.

**Example**

Consider the group of integers modulo 12 under addition, denoted by $Z_{12}$. The subgroup H = {0, 4, 8} has order 3. By Lagrange's theorem, the order of this subgroup divides the order of $Z_{12}$, which is 12. Therefore, there are 12 ÷ 3 = 4 distinct cosets of H. These are:

H = {0, 4, 8}

1 + H = {1, 5, 9}

2 + H = {2, 6, 10}

3 + H = {3, 7, 11}

This example shows that cosets partition a group into equal-sized subsets, which is a key property in understanding group structures.

**Normal Subgroups and Their Role**

A subgroup N of a group G is called a normal subgroup if the left cosets and right cosets coincide, that is, gN = Ng for all g in G. This property allows the construction of quotient groups, which simplify the study of larger groups by reducing them into smaller, more manageable structures.

Normal subgroups are essential in algebra because they preserve structure under quotienting, much like how congruence classes preserve arithmetic under modular operations

**Example**

In the group of integers Z under addition, the subgroup 3Z = {..., -6, -3, 0, 3, 6, ...} is normal. For any integer n, we have n + 3Z = 3Z + n. This gives rise to the quotient group Z/3Z, which consists of the three cosets:

{3Z}

{1 + 3Z}

{2 + 3Z}

This quotient group is isomorphic to the cyclic group of order 3.

**Quotient Groups**

If N is a normal subgroup of G, then the set of cosets G/N forms a group under the operation

(gN)(hN) = (gh)N.

This new group is called a quotient group or factor group. Quotient groups reduce complex structures into simpler forms while retaining essential properties of the original group.

**Example**

Take $G = Z_6 = \{0, 1, 2, 3, 4, 5\}$ under addition modulo 6. Let $N = \{0, 3\}$. The cosets are:

$N = \{0, 3\}$

$1 + N = \{1, 4\}$

$2 + N = \{2, 5\}$

Thus, the quotient group G/N has order 3 and is isomorphic to $Z_3$.

**Applications in Computer Science and Cryptography**

Concepts like cosets, normal subgroups, and quotient groups play a vital role in applied mathematics and computer science.

**Cryptography:** Modern encryption techniques such as RSA and elliptic curve cryptography rely heavily on group structures. Quotient groups help simplify operations and ensure computational feasibility in secure systems.

**Coding Theory:** Error detection and correction codes are often built using cosets of subgroups, where different cosets represent different error classes.

**Automata Theory:** Groups and subgroups model state transitions. Normal subgroups in particular help in minimizing automata by identifying equivalent states.

**Network Security:** Authentication and key exchange protocols frequently use properties of subgroups and quotient groups to guarantee security.

**Worked Example:** Quotient Groups in Cryptography

Let us consider the multiplicative group of integers modulo 13, denoted by $Z_{13}^*$. This group consists of {1, 2, 3, ..., 12}. Take the subgroup $H = \{1, 12\}$. Since $Z_{13}^*$ is abelian, H is a normal subgroup.

The quotient group $Z_{13}^*/H$ partitions the group into cosets:

H = {1, 12}

2H = {2, 11}

3H = {3, 10}

4H = {4, 9}

5H = {5, 8}

6H = {6, 7}

Thus, the quotient group has order 6. Such structures are useful in key generation and modular arithmetic operations in cryptography, where quotient groups allow efficient computation without compromising security.

**Importance of Quotient Groups in Modern Mathematics**

Quotient groups are not limited to abstract algebra; they play significant roles in diverse fields.

In Topology, quotient groups simplify the study of fundamental groups and homotopy classes.

In Number Theory, class groups, which describe ideal factorizations in number fields, are quotient groups.

In Physics, symmetry groups often involve quotient structures to classify fundamental particles and physical systems.

## Check Your Progress

1. Define left and right cosets. Explain their properties with an example.

   .....

   .....

   .....

   .....

   .....

2. State and explain Lagrange's theorem. Discuss its importance in group theory.

   .....

.....
.....
.....
.....

## 17.7: Summary

This unit discussed how groups can be partitioned into smaller, well-defined subsets called cosets, which provide insight into group symmetry and organization. Learners examined Lagrange's theorem, which states that the order of a subgroup divides the order of the group, forming a foundation for many algebraic results. The concept of normal subgroups was introduced as a special kind of subgroup that remains invariant under conjugation, allowing the formation of quotient groups. Quotient groups simplify complex algebraic structures while preserving essential properties. These concepts are vital for studying the internal structure of algebraic systems and their applications in various computational and theoretical domains.

## 17.8: Exercises

**Multiple Choice Questions**

1. A left coset of a subgroup H in a group G is of the form:
   a) Hg
   b) gH
   c) gH or Hg depending on the operation
   d) None of the above
   Answer: c) gH or Hg depending on the operation
2. According to Lagrange's theorem, the order of every subgroup of a finite group:
   a) Is greater than the order of the group
   b) Is equal to the order of the group
   c) Divides the order of the group
   d) Is a prime number
   Answer: c) Divides the order of the group
3. A subgroup N of a group G is said to be normal if:
   a) N contains all elements of G
   b) gN = Ng for all g in G
   c) N is equal to the identity element

d) N has only one element

Answer: b) gN = Ng for all g in G

4. The set of all left cosets of a normal subgroup forms:
   a) A new group called quotient group
   b) A semigroup
   c) A monoid
   d) A ring
   Answer: a) A new group called quotient group
5. Lagrange's theorem is applicable to:
   a) Only infinite groups
   b) Only Abelian groups
   c) Only cyclic groups
   d) Finite groups
   Answer: d) Finite groups

**Descriptive Questions**

1. Define left and right cosets. Explain their properties with an example.
2. State and prove Lagrange's theorem. Discuss its significance in group theory.
3. Define a normal subgroup and explain how it differs from a general subgroup.
4. Describe the process of forming a quotient group and illustrate it with an example.
5. Discuss the applications of cosets and quotient groups in computer science and mathematical problem solving.

## 17.9: References and Suggested Readings


- Gallian, Joseph A. *Contemporary Abstract Algebra*, 10th Edition, Cengage Learning, 2020.
- Herstein, I. N. *Topics in Algebra*, 2nd Edition, John Wiley & Sons, 2006.
- Fraleigh, John B. *A First Course in Abstract Algebra*, 7th Edition, Pearson Education, 2013.
- Rosen, Kenneth H. *Discrete Mathematics and Its Applications*, 8th Edition, McGraw-Hill Education, 2019.

- Kolman, Bernard, Busby, Robert C., and Ross, Sharon. *Discrete Mathematical Structures*, 6th Edition, Pearson Education, 2018.
- Grimaldi, Ralph P. *Discrete and Combinatorial Mathematics: An Applied Introduction*, 5th Edition, Pearson Education, 2003.

## GLOSSARY

- Abelian Group: A group in which the binary operation is commutative.
- Adjacency Matrix: A square matrix used to represent a finite graph, indicating edge connections.
- Algebraic Structure: A set with one or more binary operations defined on it.
- Automorphism: An isomorphism from a mathematical structure to itself.
- Binary Operation: An operation that combines two elements of a set to produce another element of the same set.
- Bijective Function: A function that is both one-to-one and onto.
- Boolean Algebra: A mathematical structure dealing with binary variables and logical operations.
- Boolean Function: A function whose inputs and outputs are binary values (0 or 1).
- Cartesian Product: A set of ordered pairs formed from two sets.
- Circuit (Graph Theory): A closed walk in which no edges are repeated.
- Closure Property: A property where an operation on any two elements of a set produces an element of the same set.
- Complement (Set Theory): The set of all elements not in a given set.
- Complemented Lattice: A bounded lattice in which every element has a complement.
- Conjunctive Normal Form (CNF): A Boolean expression written as an AND of ORs.
- Coset: A subset formed by multiplying all elements of a subgroup by a fixed group element.
- De Morgan's Laws: Rules that relate conjunctions and disjunctions of logical statements through negation.
- Directed Graph (Digraph): A graph where edges have a direction from one vertex to another.
- Disjunctive Normal Form (DNF): A Boolean expression written as an OR of ANDs.
- Distributive Lattice: A lattice where join and meet operations distribute over each other.

- Edge (Graph): A connection between two vertices in a graph.
- Equivalence Relation: A relation that is reflexive, symmetric, and transitive.
- Function: A mapping from one set (domain) to another (codomain) where each input has exactly one output.
- Generator (Group Theory): An element that can generate all elements of the group using the group operation.
- Graph: A collection of vertices connected by edges.
- Group: A set with a binary operation that is associative, has an identity, and where every element has an inverse.
- Homomorphism: A structure-preserving map between two algebraic structures.
- Hasse Diagram: A graphical representation of a finite poset.
- Identity Element: An element that leaves other elements unchanged under a binary operation.
- Incidence Matrix: A matrix representing the relationship between vertices and edges in a graph.
- Injective Function: A function where different inputs always map to different outputs.
- Inverse Element: An element that reverses the effect of another under a binary operation.
- Isomorphism: A bijective homomorphism indicating structural similarity between two algebraic structures.
- Join (Lattice Theory): The least upper bound of two elements in a lattice.
- Karnaugh Map (K-Map): A visual method for simplifying Boolean expressions.
- Lagrange's Theorem: States that the order of a subgroup divides the order of the group.
- Lattice: A poset where every two elements have a join and meet.
- Logic Circuit: A circuit built using logic gates to represent Boolean functions.
- Logical Equivalence: Two logical statements that have the same truth values in all cases.
- Matrix Representation (Graph): Using matrices like adjacency or incidence to represent graphs.

- Meet (Lattice Theory): The greatest lower bound of two elements in a lattice.
- Monoid: A semigroup with an identity element.
- Multigraph: A graph that allows multiple edges between vertices.
- Normal Subgroup: A subgroup that is invariant under conjugation by elements of the group.
- Null Set: The empty set, containing no elements.
- Path (Graph Theory): A sequence of vertices connected by edges with no repetition of edges.
- Permutation Group: A group formed by all permutations of a set.
- Poset: A set with a partial order that is reflexive, antisymmetric, and transitive.
- Predicate Logic: A type of logic that uses quantifiers and predicates to express statements.
- Reflexive Relation: A relation where every element is related to itself.
- Relation: A subset of the Cartesian product of two sets that defines a relationship between elements.
- Rooted Tree: A tree with one designated node as the root.
- Semigroup: A set with an associative binary operation but not necessarily an identity.
- Simple Graph: A graph without loops or multiple edges.
- Spanning Tree: A subgraph that includes all vertices of a graph and is a tree.
- Subgroup: A subset of a group that itself forms a group.
- Subgraph: A graph formed from a subset of the vertices and edges of a larger graph.
- Submonoid: A subset of a monoid that forms a monoid under the same operation.
- Subsemigroup: A subset of a semigroup that itself is a semigroup.
- Surjective Function: A function that covers the entire codomain.
- Symmetric Relation: A relation where if (a, b) is in the relation, then (b, a) is also in it.
- Tautology: A logical statement that is always true.

- Truth Table: A table showing all possible truth values for a logical expression.
- Transitive Relation: A relation where if (a, b) and (b, c) are related, then (a, c) is also related.
- Tree: An acyclic connected graph.
- Unary Operation: An operation with only one operand.
- Vertex (Graph): A point representing an element in a graph.
- Walk (Graph): A sequence of vertices and edges where repetition is allowed.

# MATS UNIVERSITY

**MATS CENTRE FOR DISTANCE AND ONLINE EDUCATION**

UNIVERSITY CAMPUS: Aarang Kharora Highway, Aarang, Raipur, CG, 493 441

RAIPUR CAMPUS: MATS Tower, Pandri, Raipur, CG, 492 002

T : 0771 4078994, 95, 96, 98 Toll Free ODL MODE : 81520 79999, 81520 29999

Website: www.matsodl.com