College of Engineering & Applied Sciences

CSPB 2824

Discrete Structures

Class Notes

University Of Colorado

2024

Discrete Structures - Class Notes

Inductive And Deductive Reasoning

Inductive And Deductive Reasoning

1.0.1 Assigned Reading

The reading assignments for this week is from, :

- Chapter 1.1 - Propositional Logic
- Chapter 1.2 - Applications Of Propositional Logic

1.0.2 Piazza

Must post / respond to at least **two** Piazza posts this week.

1.0.3 Lectures

The lectures for this week and their links can be found below:

- Inductive And Deductive Reasoning ≈ 22 min.
- Conjunctions And Disjunctions ≈ 43 min.
- Conditionals And Biconditionals ≈ 39 min.
- Fundamentals Of Algebra ≈ 9 min.
- Applications Of Algegra ≈ 6 min.
- Applications Of Algebra Continued ≈ 4 min.
- Negative Times A Negative? ≈ 4 min.

Below is a list of lecture notes for this week:

- Conditionals And Biconditionals Lecture Notes
- Conjunctions And Disjunctions Lecture Notes
- Field Axiom Proofs Lecture Notes

1.0.4 Assignments

The assignment for this week is:

- Assignment 1 - Logic And Reasoning

1.0.5 Quiz

The quiz's for this week can be found below:

- Quiz 1 - Logic And Reasoning

1.0.6 Chapter Summary

The first section that we are covering this week is **Section 1.1 - Propositional Logic**

Section 1.1 - Propositional Logic

Overview

Propositional logic, within the context of discrete structures, is a fundamental branch of mathematics that deals with analyzing and manipulating statements or propositions. These propositions are treated as discrete entities that can either be true or false, and they serve as building blocks for logical reasoning and problem solving in various fields, especially computer science.

In propositional logic, we focus on the relationships between propositions and how they combine to form more complex statements. Logical operators such as "and," "or," "not," "implies," and "if and only if" are used to connect propositions and create compound statements. This allows us to express relationships, conditions, and constraints in a formal and precise manner.

Conjunctions

Conjunctions are an essential part of both language and logic. In language, conjunctions are words that connect words, phrases, or clauses to show how they are related. In logic, conjunctions serve a similar purpose by allowing us to combine two or more propositions to create a compound statement. The most common conjunction in logic is "and."

In logic, a conjunction represents a statement that is true only if both of the individual statements being combined are true. This mirrors the way we use "and" in everyday language. If both parts of a conjunction are true, then the entire conjunction is true; otherwise, it's false.

Conjunction Examples

Let's consider two propositions:

- P: It is sunny today.
- Q: I will go for a picnic.

Now, we can create a compound statement using the conjunction 'and'.

$$P \wedge Q : \text{It is sunny today and I will go for a picnic.} \tag{1}$$

In this case, the compound statement "It is sunny today and I will go for a picnic" is true only if both P and Q are true. If it's indeed sunny today (P is true) and you do plan to go for a picnic (Q is true), then the entire statement "It is sunny today and I will go for a picnic" is true.

Conjunctions are particularly important in logic because they allow us to express complex conditions and relationships between different propositions. They're a fundamental tool for constructing logical statements that accurately reflect real-world situations or logical arguments.

Disjunctions

Disjunctions are another important concept in both language and logic. In language, disjunctions are words that express alternatives or choices. In logic, disjunctions allow us to combine propositions to create a compound statement that is true if at least one of the individual statements is true. The most common disjunction in logic is "or."

In logic, a disjunction is true if at least one of the individual statements being combined is true. This corresponds to the way we use "or" in everyday language to present options or alternatives.

Disjunction Example

Let's consider two propositions:

- P: It is raining today.
- Q: I will stay inside.

Now, we can create a compound statement using the conjunction 'and'.

$$P \vee Q : \text{It is raining today or I will stay indoors.} \tag{2}$$

In this case, the compound statement "It is raining today or I will stay indoors" is true if either P is true (it's raining) or Q is true (you plan to stay indoors), or if both are true.

For example, if it's indeed raining today (P is true), then the entire statement "It is raining today or I will stay indoors" is true, even if you end up going outdoors. Similarly, if you plan to stay indoors (Q is true), the statement is also true, even if it's not raining.

Disjunctions are crucial in logic because they help us express scenarios where multiple possibilities exist, and at least one of them needs to be true for the entire statement to be true. They enable us to model various conditions and decision-making processes in a structured and logical manner.

Conditional Statements

Conditional statements, often referred to as implications, are a key concept in logic and reasoning. They express a relationship between two propositions where one proposition, known as the antecedent or premise, leads to or implies the truth of the other proposition, known as the consequent or conclusion. The primary conditional operator is "if...then," and conditional statements are crucial for modeling cause-and-effect relationships and logical implications.

Conditional Statement Example

Let's consider two propositions:

- P: I study for the exam.
- Q: I will get a good grade.

We can create a conditional statement using "if...then":

$$P \rightarrow Q : \text{If I study for the exam, then I will get a good grade.} \tag{3}$$

In this case, the statement "If I study for the exam, then I will get a good grade" implies that studying (P) leads to getting a good grade (Q). If you study (P is true), then the conditional statement suggests that you will indeed get a good grade (Q is true).

However, the conditional statement does not specify what happens if you don't study (P is false). It only asserts that if you do study, there's an implication for the outcome of your grade.

Conditional statements play a pivotal role in logical reasoning, mathematics, and computer science. They're used to define rules, express relationships between variables, and create logical structures. Additionally, they're central to constructing logical proofs and arguments. Understanding conditional statements is fundamental for accurately representing cause-and-effect connections and making reasoned conclusions.

Biconditionals

Biconditionals, also known as "if and only if" statements, are another important concept in logic. They express a relationship between two propositions where both propositions are connected in such a way that if one is true, the other must also be true, and if one is false, the other must also be false. Biconditionals capture a sense of mutual exclusivity and equivalence between the two propositions.

Biconditional Statement Example

Let's consider two propositions:

- P: The cake is chocolate.
- Q: I will enjoy the dessert.

We can create a biconditional statement using "if and only if":

$$P \Leftrightarrow Q : \text{If I study for the exam, then I will get a good grade.} \tag{4}$$

In this case, the biconditional statement "I will enjoy the dessert if and only if the cake is chocolate" expresses that the enjoyment of the dessert (Q) is directly tied to the cake being chocolate (P). If the cake is indeed chocolate (P is true), then you will enjoy the dessert (Q is true). Similarly, if the cake is not chocolate (P is false), you won't enjoy the dessert (Q is false).

Conversely, if you enjoy the dessert (Q is true), the statement asserts that the cake must be chocolate (P is true), and if you don't enjoy the dessert (Q is false), the cake must not be chocolate (P is false).

Biconditionals are used to define equivalence between propositions, indicating that they are both true or both false simultaneously. They're useful for expressing situations where two conditions are inextricably linked and must hold together. In mathematics, they're used for defining properties that hold true in both directions. Understanding biconditionals is important for accurately representing scenarios where two propositions are interdependent and inseparable.

Converse Statement

The converse of a conditional statement swaps the positions of the antecedent (premise) and the consequent (conclusion) of the original statement. In other words, if you have a statement "If P, then Q," the converse would be "If Q, then P."

Converse Statement Example

The following is an example of a converse statement.

- Original Statement: If it's sunny (P), then I will go for a walk (Q).
- Converse Statement: If I go for a walk (Q), then it's sunny (P).

It's important to note that the converse might not always be logically equivalent to the original statement. In some cases, it could be true, while in others, it might not hold true.

Contrapositive Statement

The contrapositive of a conditional statement involves both negating (changing from true to false, or vice versa) and swapping the positions of the antecedent and consequent. In essence, it's the combination of the converse and the negation of each proposition.

Contrapositive Statement Example

The following is an example of a contrapositive statement.

- Original Statement: If it's raining (P), then I will stay indoors (Q).
- Contrapositive Statement: If I don't stay indoors ($\neg$Q), then it's not raining ($\neg$P).

The contrapositive is always logically equivalent to the original statement. If the original statement is true, its contrapositive is also true, and if the original statement is false, its contrapositive is false.

Inverse Statement

The inverse of a conditional statement involves negating both the antecedent and the consequent of the original statement.

Inverse Statement Example

The following is an example of an inverse statement.

- Original Statement: If I exercise (P), then I will be tired (Q).
- Inverse Statement: If I don't exercise ($\neg$P), then I won't be tired ($\neg$Q).

Like the converse, the inverse might not always be logically equivalent to the original statement. In some cases, it could be true, while in others, it might not hold true.

Understanding these related statements is valuable for analyzing the different ways that logical relationships can be manipulated and transformed. They are often used in mathematical proofs, reasoning, and decision-making processes to explore the implications of conditional statements from different angles.

The second section that we are covering this week is **Section 1.2 - Applications of Propositional Logic**

Section 1.2 - Applications of Propositional Logic

Propositional logic, a foundational concept in formal logic, holds significant applications across a diverse range of fields due to its ability to represent and analyze relationships between statements. In computer science and programming, it serves as the bedrock for constructing decision structures, enabling the creation of conditional statements and loops that define the behavior of software. This logical framework also finds a pivotal role in

digital circuit design, where it facilitates the creation of logic gates and memory units, forming the basis of modern processors and electronic systems.

The realm of artificial intelligence heavily relies on propositional logic for knowledge representation and reasoning. AI systems employ logical inference and deduction mechanisms to make informed decisions and solve complex problems. In the domain of database management, propositional logic assists in designing intricate queries that efficiently retrieve data based on specific conditions, enhancing the effectiveness of data storage and retrieval processes.

Formal verification, a critical aspect of software and hardware engineering, leans on propositional logic to ensure the correctness and compliance of systems with predetermined requirements. It enables engineers to rigorously prove that a given system behaves as intended. Moreover, in the realm of cryptography and security, propositional logic underpins the development of encryption algorithms, secure communication protocols, and authentication mechanisms, safeguarding sensitive information and digital transactions.

Beyond the technological sphere, propositional logic has broader implications. It plays a role in mathematical proofs, enabling mathematicians to construct logical chains of reasoning and establish the validity of theorems. The field of philosophy benefits from its analytical capabilities, using propositional logic to dissect complex arguments and analyze language structure. Similarly, it aids linguists in formalizing intricate linguistic constructs.

In decision analysis, propositional logic supports the modeling of intricate decision scenarios, enabling the evaluation of various choices and potential outcomes. Networks and communication systems benefit from its application in modeling network protocols, routing strategies, and data transmission. Even within interdisciplinary domains like game theory, propositional logic is instrumental in modeling strategic interactions and rational decision-making processes. Its reach extends into robotics, shaping behavior rules, path planning algorithms, and obstacle avoidance strategies for autonomous systems. From legal reasoning to philosophical analysis, propositional logic provides a structured framework to formalize arguments and explore complex concepts.

In conclusion, propositional logic's power to express relationships, conditions, and implications in a precise, systematic manner makes it an indispensable tool across diverse academic, scientific, and technological landscapes. Its applications permeate fields as varied as computer science, AI, cryptography, philosophy, and beyond, highlighting its role in advancing logical reasoning, problem-solving, and strategic decision-making.

Logic, Equivalence, And Quantifiers

Logic, Equivalence, And Quantifiers

2.0.1 Assigned Reading

The reading assignments for this week is from, :

- Chapter 1.3 - Propositional Equivalences
- Chapter 1.4 - Predicates And Quantifiers
- Chapter 1.5 - Nested Quantifiers
- Chapter 1.6 - Rules Of Inference

2.0.2 Piazza

Must post / respond to at least **two** Piazza posts this week.

2.0.3 Lectures

The lectures for this week and their links can be found below:

- Propositional Equivalence ≈ 54 min.
- Predicates And Quantifiers ≈ 1 hour, 10 min.
- Nested Quantifiers ≈ 34 min.
- Rules of Inference ≈ 1 hour, 11 min.

Below is a list of lecture notes for this week:

- Propositional Equivalence Lecture Notes
- Predicates And Quantifiers Lecture Notes
- Nested Quantifiers Lecture Notes

2.0.4 Assignments

The assignment for this week is:

- Assignment 2 - Propositional Logic

2.0.5 Quiz

The quiz's for this week can be found at:

- Quiz 2 - Predicates and Quantifiers

2.0.6 Chapter Summary

The first section that we are covering this week is **Section 1.3 - Propositional Equivalences**.

Section 1.3 - Propositional Equivalences

Overview

Propositional equivalences, also known as logical equivalences or tautologies, are fundamental concepts in propositional logic. They express the idea that two logical statements have the same truth value under all possible truth assignments to their constituent propositions. In other words, if two propositions are logically equivalent, they are either both true or both false in every possible situation.

Propositional equivalences are crucial in logic and mathematics for simplifying expressions, proving the validity of arguments, and establishing relationships between different logical statements. These equivalences are derived from the properties of logical connectives (such as negation, conjunction, disjunction, implication, and bi-conditional) and often take the form of if-and-only-if statements.

Some common examples of propositional equivalences include:

Double Negation

The negation of a negation is equivalent to the original proposition.

$$\neg(\neg p) \equiv p$$

De Morgan's Laws

These laws describe how to negate conjunctions and disjunctions.

$$\neg(p \wedge q) \equiv \neg p \vee \neg q$$
$$\neg(p \wedge q) \equiv \neg p \wedge \neg q$$

Commutative Laws

The order of propositions does not affect the outcome of conjunctions and disjunctions.

$$p \wedge q \equiv q \wedge p$$
$$p \vee q \equiv q \vee p$$

1

Associative Laws

The grouping of propositions within conjunctions and disjunctions does not change their truth value.

$$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$$
$$(p \vee q) \vee r \equiv p \vee (q \vee r)$$

Distributive Laws

These laws describe how conjunctions and disjunctions distribute over each other.

$$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$$
$$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$$

Propositional equivalences provide a powerful toolkit for transforming logical expressions and simplifying complex propositions. They play a crucial role in constructing formal proofs, enhancing problem-solving abilities, and ensuring the validity of logical arguments in various disciplines, including computer science, mathematics, philosophy, and linguistics.

Logical Equivalences

Logical equivalences, also known as propositional equivalences, are fundamental principles in propositional logic that express the idea that two logical statements have the same truth value under all possible truth assignments to their constituent propositions. These equivalences are derived from the properties of logical connectives and provide a set of rules for simplifying logical expressions, proving the validity of arguments, and establishing relationships between different logical statements.

Logical equivalences allow us to manipulate and transform logical statements while preserving their truth values. They are crucial tools for reasoning, problem-solving, and formal verification in various fields such as mathematics, computer science, philosophy, and linguistics. These equivalences enable us to navigate the intricacies of complex logical systems by breaking down propositions into simpler forms that retain their original truth values.

Logical Equivalences Example

For example, consider the logical equivalence known as the **Double Negation Law**:

$$\neg(\neg p) \equiv p$$

This equivalence states that the negation of a negation is equivalent to the original proposition. In simpler terms, if you negate a proposition and then negate it again, you end up with the same proposition. This can be demonstrated with a specific proposition:

Let p be the proposition "It is sunny today."

The negation of p is "It is not sunny today."

The negation of ¬p is "It is sunny today," which is the same as the original proposition p.

This logical equivalence can be useful in various contexts. For instance, it helps clarify that a proposition and its double negation express the same information. This understanding is essential for formalizing arguments, clarifying statements, and simplifying logical expressions to reveal their underlying structure.

In summary, logical equivalences are foundational principles that allow us to manipulate and simplify logical statements while maintaining their truth values. They play a crucial role in various disciplines, enabling us to reason rigorously, prove the validity of arguments, and navigate the complexities of logical systems.

The presented table encapsulates essential logical equivalences that unveil the intricate relationships governing propositional logic. Each entry in the table highlights a distinct law that defines the behavior of logical operations across different propositions. From the **Identity** and **Domination** laws that explore the impact of "True" (T) and "False" (F) in conjunctions and disjunctions, to the **Commutative** and **Associative** laws unveiling the symmetry and grouping properties of logical operations, these equivalences offer fundamental insights into the logic of propositions.

Furthermore, the table introduces crucial principles such as the **Double Negation** law, illustrating the profound implications of double negations, and **De Morgan's** laws, outlining how negations interact with complex logical structures. The **Absorption** laws demonstrate how propositions can interplay within conjunctions and disjunctions, while the **Negation** laws lay bare the inherent contradictions when propositions and their negations are combined. This compilation of logical equivalences provides an indispensable toolkit for simplifying and transforming logical expressions, enabling rigorous reasoning and deductive analysis across various domains of knowledge.

Equivalence Name	Equivalence
$p \wedge T \equiv p$	Identity laws
$p \vee F \equiv p$	
$p \vee T \equiv T$	Domination laws
$p \wedge F \equiv F$	
$p \vee p \equiv p$	Idempotent laws
$p \wedge p \equiv p$	
$\neg(\neg p) \equiv p$	Double negation law
$p \vee q \equiv q \vee p$	Commutative laws
$p \wedge q \equiv q \wedge p$	
$(p \vee q) \vee r \equiv p \vee (q \vee r)$	Associative laws
$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$	
$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$	Distributive laws
$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$	
$\neg(p \wedge q) \equiv \neg p \vee \neg q$	De Morgan's laws
$\neg(p \vee q) \equiv \neg p \wedge \neg q$	
$p \vee (p \wedge q) \equiv p$	Absorption laws
$p \wedge (p \vee q) \equiv p$	
$p \vee \neg p \equiv T$	Negation laws
$p \wedge \neg p \equiv F$	

The table presented introduces a set of vital logical equivalences centered around conditional statements, shedding light on the intricate relationships that govern propositional logic in the context of implications. Each entry in the table showcases a distinct equivalence, highlighting how conditional statements interact with other logical operations and propositions. From the fundamental equivalence of $p \to q$ being logically equivalent to $\neg p \vee q$, to the insightful contrapositive equivalence of $p \to q$ being the same as $\neg q \to \neg p$, these equivalences unveil the nuanced connections between antecedents and consequents in conditional statements.

Moreover, the table delves into more advanced equivalences, such as transposition equivalence and the negation of conditionals. These entries illustrate how conditional statements interplay with conjunctions, disjunctions, and negations to produce logical conclusions. The exportation and importation equivalences provide a deeper understanding of how conditional statements can be combined and separated within larger logical structures. By meticulously exploring these logical equivalences involving conditional statements, the table equips individuals with powerful tools to manipulate and simplify complex propositions, paving the way for sophisticated reasoning and rigorous deduction across a range of disciplines.

Equivalence	Description
$p \to q \equiv \neg p \lor q$	Conditional Equivalence
$p \to q \equiv \neg q \to \neg p$	Contrapositive Equivalence
$p \lor q \equiv \neg p \to q$	Transposition Equivalence
$p \land q \equiv \neg(p \to \neg q)$	
$\neg(p \to q) \equiv p \land \neg q$	Negation of Conditional
$(p \to q) \land (p \to r) \equiv p \to (q \land r)$	Exportation Equivalence
$(p \to r) \land (q \to r) \equiv (p \lor q) \to r$	Importation Equivalence
$(p \to q) \lor (p \to r) \equiv p \to (q \lor r)$	
$(p \to r) \lor (q \to r) \equiv (p \land q) \to r$	

The provided table introduces a comprehensive collection of logical equivalences centered around biconditional statements, shedding light on the intricate relationships governing propositional logic in the realm of mutual implications. Each entry in the table presents a distinct equivalence that showcases the nuanced interplay between propositions in biconditional statements. From the foundational equivalence of $p \leftrightarrow q$ being equivalent to both $(p \to q) \land (q \to p)$, to the illuminating negation equivalence of $\neg p \leftrightarrow \neg q$, these equivalences highlight the symmetrical nature of biconditional statements and their connection to implications.

Furthermore, the table delves into more advanced equivalences involving exclusive disjunctions and negations. The entry illustrating how $p \leftrightarrow q$ is equivalent to $(p \land q) \lor (\neg p \land \neg q)$ offers an intriguing perspective on biconditionals, emphasizing their link to conjunctions and negations. The negation of biconditional equivalence adds depth by exploring how negations can transform biconditional statements. By unveiling these complex relationships involving biconditional statements, the table equips individuals with a powerful toolkit to manipulate and simplify propositions characterized by mutual implication. These equivalences serve as valuable assets in the realm of logical reasoning, enabling the exploration of connections between propositions and reinforcing one's ability to navigate intricate logical structures across diverse academic disciplines.

Equivalence	Description
$p \leftrightarrow q \equiv (p \to q) \land (q \to p)$	Biconditional Equivalence
$p \leftrightarrow q \equiv \neg p \leftrightarrow \neg q$	Negation Equivalence
$p \leftrightarrow q \equiv (p \land q) \lor (\neg p \land \neg q)$	Exclusive Disjunction Equivalence
$\neg(p \leftrightarrow q) \equiv p \leftrightarrow \neg q$	Negation of Biconditional

Propositional Satisfiability

Propositional satisfiability is a fundamental concept in propositional logic that revolves around determining whether a given logical formula, also known as a propositional formula, can be made true by assigning appropriate truth values to its constituent propositions. In other words, it involves finding a combination of truth values for the variables in the formula that results in the entire formula evaluating to true. If such a combination exists, the formula is considered satisfiable; otherwise, if no such combination exists, the formula is unsatisfiable.

The study of propositional satisfiability has profound implications in various fields, especially in computer science and artificial intelligence. It is at the core of solving decision problems and optimization challenges, with applications in automated reasoning, theorem proving, circuit design, and even software verification. The process of determining the satisfiability of a formula often involves utilizing algorithms and techniques like truth tables, Boolean algebra, and sophisticated SAT solvers that efficiently explore the vast solution space. Understanding and analyzing propositional satisfiability not only provide insights into logical reasoning but also offer practical solutions to real-world problems by enabling the automation of complex decision-making processes.

Propositional Satisfiability Example

Consider the propositional formula: $p \land (q \lor \neg r)$

We want to determine whether there exists an assignment of truth values to the propositions p, q, and r that makes the entire formula true.

---

1. $p = \text{True}$, $q = \text{True}$, $r = \text{True}$ $p \land (q \lor \neg r) = \text{True} \land (\text{True} \lor \text{False}) = \text{True} \land \text{True} = \text{True}$
2. $p = \text{False}$, $q = \text{True}$, $r = \text{False}$ $p \land (q \lor \neg r) = \text{False} \land (\text{True} \lor \text{True}) = \text{False} \land \text{True} = \text{False}$
3. $p = \text{True}$, $q = \text{False}$, $r = \text{True}$ $p \land (q \lor \neg r) = \text{True} \land (\text{False} \lor \text{False}) = \text{True} \land \text{False} = \text{False}$

From the examples above, we can see that there are assignments of truth values that make the formula true and assignments that make it false. Since there exists at least one assignment that satisfies the formula, it is satisfiable.

This example illustrates how the concept of propositional satisfiability involves finding truth value assignments that make a propositional formula true and how such assignments impact the overall truth value of the formula.

The next section that we are covering this week is **Section 1.4 - Predicates & Quantifiers**.

Section 1.4 - Predicates & Quantifiers

Overview

Predicates and quantifiers are fundamental concepts in logic and mathematics that play a crucial role in expressing and reasoning about statements involving variables. They provide a structured framework for making statements about objects, individuals, or elements within a given domain.

Predicates are essentially statements with variables that can be either true or false depending on the values assigned to the variables. They are often used to describe properties, characteristics, or relationships among objects. For example, consider the predicate "P(x)" where "P" represents the property of being prime and "x" is a variable representing an integer. This predicate becomes true or false depending on the value assigned to "x."

Quantifiers are used to express the scope of a predicate over a certain domain. They allow us to make statements about "all" or "some" elements in a domain. There are two main types of quantifiers:

1. **Universal Quantifier ($\forall$)**: This quantifier is denoted by "$\forall$" and is used to express that a certain predicate is true for "all" elements in a given domain. For instance, the statement "$\forall x, P(x)$" asserts that the predicate "P(x)" is true for every element "x" in the domain.
2. **Existential Quantifier ($\exists$)**: This quantifier is denoted by "$\exists$" and is used to express that a certain predicate is true for "at least one" element in a given domain. For example, the statement "$\exists x, P(x)$" asserts that there exists an element "x" in the domain for which the predicate "P(x)" is true.

Predicates and quantifiers are essential tools for expressing complex logical statements, defining sets, and describing mathematical properties in a precise and systematic manner. They serve as building blocks for formalizing mathematical proofs, specifying conditions, and making assertions about various objects and entities within a given context.

Predicates

Predicates are foundational elements in logic and mathematics that enable us to express statements about variables and their relationships. A predicate is a statement that contains one or more variables and can be either true or false depending on the values assigned to these variables. Predicates allow us to describe properties, characteristics, or conditions that entities or elements may satisfy. In formal terms, a predicate is often denoted as $P(x)$, where P represents the predicate itself, and x is a variable that ranges over a specific domain.

For example, consider the predicate $P(x)$: "x is an even number." Here, x is a variable that can take on different integer values, and the predicate evaluates to true when x is an even number and false otherwise. Predicates can also involve multiple variables, allowing us to express more complex relationships. For instance, the predicate $Q(x, y)$: "x is less than y" involves two variables, x and y, and evaluates to true when x is indeed less than y.

Predicates play a pivotal role in constructing logical statements and formulating mathematical propositions. They provide a means to express conditions, make assertions, and define sets based on specific properties. Through quantifiers like universal ($\forall$) and existential ($\exists$), predicates allow us to quantify the scope of truth for statements involving variables, enabling us to reason about collections of elements in a precise and systematic manner.

Predicate Examples

Below are some examples of predicates.

---

1. $P(x)$ = "x is a prime number." - This predicate is true when the variable x represents a prime number,

and it is false for any other integer value of x.

2. $Q(x, y)$ = "x is divisible by y." - This two-variable predicate is true when x is divisible by y, and it is false otherwise. For example, $Q(10, 2)$ is true because 10 is divisible by 2.
3. $R(x)$ = "x is a positive real number." - This predicate is true for positive real numbers, and false for negative numbers, zero, and complex numbers.
4. $S(x)$ = "x is a square." - This predicate is true when x is the area of a square, and false for any other value of x.
5. $T(x, y)$ = "x is taller than y." - This two-variable predicate is true when x is taller than y in a certain context, and false otherwise.
6. $U(x)$ = "x is a vowel." - This predicate can be used for characters in an alphabet to determine if a character is a vowel or a consonant.
7. $V(x, y)$ = "x divides y evenly." - This two-variable predicate is true when x divides y without a remainder, and false otherwise.
8. $W(x)$ = "x is an even integer." - This predicate is true when x is an even integer, and false for odd integers.
9. $X(x)$ = "x is a positive solution to the equation $x^2 - 5x + 6 = 0$." - This predicate is true for the values of x that satisfy the equation, which are 2 and 3 in this case.
10. $Y(x, y)$ = "x and y are co-prime." - This two-variable predicate is true when x and y have no common factors other than 1, and false otherwise.

Quantifiers

Quantifiers are fundamental concepts in logic that enable us to express the scope of a predicate over a set of elements or objects. They allow us to make statements about "all" or "some" members of a domain and play a crucial role in formalizing logical propositions and mathematical statements.

Universal Quantifier ($\forall$): The universal quantifier, denoted by $\forall$, is used to express that a certain predicate holds true for "every" element in a given domain. For example, the statement $\forall x, P(x)$ asserts that the predicate $P(x)$ is true for every possible value of x within the specified domain. This quantifier establishes a broad claim about the entire domain and is satisfied only when the predicate is true for every individual element.

Existential Quantifier ($\exists$): The existential quantifier, denoted by $\exists$, is used to express that a certain predicate holds true for "at least one" element in a given domain. For instance, the statement $\exists x, P(x)$ indicates that there exists at least one element x in the domain for which the predicate $P(x)$ is true. This quantifier establishes the existence of elements that satisfy the predicate without making claims about all elements in the domain.

Quantifiers are essential tools for expressing and reasoning about properties, relationships, and conditions within logical and mathematical contexts. They allow us to quantify over collections of elements, formulate statements about subsets, and formalize assertions in a precise and structured manner. By combining quantifiers with predicates, we can create powerful expressions that capture the nuances of complex logical and mathematical statements, paving the way for rigorous analysis, proof, and exploration of various concepts.

Quantifier Examples

Below are some examples of quantifiers.

Universal Quantifier ($\forall$):

1. $\forall x \in \mathbb{Z}, x^2 \geq 0$ - This statement asserts that for every integer x, its square is greater than or equal to zero.
2. $\forall n \in \mathbb{N}, 2n$ is an even number. - This statement claims that every natural number multiplied by 2 results in an even number.
3. $\forall a, b \in \mathbb{R}, a + b = b + a$ - This equation expresses the commutative property of addition for all real numbers a and b.

Existential Quantifier ($\exists$):

1. $\exists x \in \mathbb{Z}, x^2 = 9$ - This statement asserts that there exists an integer x such that its square is equal to 9, which is true for $x = 3$ and $x = -3$.

2. $\exists n \in \mathbb{N}, n > 10$ - This statement states that there exists a natural number greater than 10.
3. $\exists x, y \in \mathbb{R}, x^2 + y^2 = 25$ - This equation expresses that there exist real numbers x and y such that their squares sum up to 25. For example, $x = 3$ and $y = 4$ satisfy this equation.

These examples showcase how quantifiers are used to make statements about either all elements or at least one element in a given domain. The universal quantifier $\forall$ asserts a property for every element in a set, while the existential quantifier $\exists$ asserts the existence of an element with a certain property in a set.

Here is a table that encapsulates the truth values of quantifiers.

Statement	When True?	When False?
$\forall x P(x)$	$P(x)$ is true for every x.	There is an x for which $P(x)$ is false.
$\exists x P(x)$	There is an x for which $P(x)$ is true.	$P(x)$ is false for every x.

The table presents an overview of two essential quantifiers in logic: the universal quantifier ($\forall$) and the existential quantifier ($\exists$). These quantifiers are used to express statements about properties of elements within a specified domain.

- $\forall x P(x)$: This statement is true when the predicate $P(x)$ holds true for every element x in the domain. It signifies a universal claim that the property described by $P(x)$ is applicable to all elements. On the other hand, this statement is false when there exists at least one element x for which $P(x)$ is false. In other words, if there's a single counterexample, the entire statement is false.
- $\exists x P(x)$: This statement is true when there exists at least one element x for which the predicate $P(x)$ holds true. It asserts the existence of an element that satisfies the property described by $P(x)$. Conversely, this statement is false when $P(x)$ is false for every element x in the domain. If no element satisfies the property, the statement becomes false.

These quantifiers provide a formal way to make assertions about the relationship between predicates and elements in a logical system. The universal quantifier ensures that a property applies to all elements, while the existential quantifier asserts the presence of at least one element with a specific property.

Quantifiers with Restricted Domains

Quantifiers in logic, namely the universal quantifier ($\forall$) and the existential quantifier ($\exists$), can be used with restricted domains to express statements about properties within specific subsets of a larger domain. This allows for more precise and targeted assertions.

Universal Quantifier with Restricted Domain: The statement $\forall x \in S, P(x)$ asserts that for every element x in the subset S of the domain, the predicate $P(x)$ holds true. This means that the property described by $P(x)$ applies universally within the specified subset. If there exists an element x in S for which $P(x)$ is false, the statement is false.

Existential Quantifier with Restricted Domain: The statement $\exists x \in S, P(x)$ asserts that there exists at least one element x in the subset S for which the predicate $P(x)$ holds true. This indicates that the property described by $P(x)$ is satisfied by at least one element within the specified subset. If no element in S satisfies $P(x)$, the statement is false.

Using quantifiers with restricted domains adds precision to statements, allowing us to focus on specific subsets of interest within a larger context. These quantifiers are valuable tools in expressing targeted claims about properties within constrained scopes while maintaining the logical rigor of quantification.

Precedence of Quantifiers

When working with logical statements involving multiple quantifiers, it's important to understand the precedence of quantifiers to accurately interpret and evaluate these statements. The order in which quantifiers are applied can significantly impact the meaning of a proposition.

Precedence of Universal and Existential Quantifiers: In general, universal quantifiers ($\forall$) take precedence over existential quantifiers ($\exists$) when nested within a logical statement. This means that when quantifiers are combined, the universal quantifier is applied first, followed by the existential quantifier.

For example, consider the statement $\forall x \exists y, P(x, y)$. This can be read as "For every x, there exists a y such that $P(x, y)$." Here, the universal quantifier applies to x first, and then the existential quantifier applies to y, allowing for different y values for each x.

Use of Parentheses: To avoid ambiguity and clearly convey the intended meaning, it's often wise to use parentheses to explicitly indicate the grouping of quantifiers. For instance, $\forall x (\exists y P(x, y))$ specifies that the existential quantifier is nested within the scope of the universal quantifier.

Understanding the precedence of quantifiers is crucial for correctly interpreting complex logical statements involving multiple quantifiers. Using parentheses appropriately ensures that the intended meaning is accurately conveyed and understood, facilitating effective communication of logical propositions.

Binding Variables in Quantifiers

When working with quantifiers in logic, the concept of binding variables plays a crucial role in understanding how variables are associated with quantifiers and predicates. Binding variables define the scope and range of quantified statements and influence the meaning of logical propositions.

Binding Variables in Universal Quantifiers: In a statement of the form $\forall x P(x)$, the variable x is bound by the universal quantifier $\forall$. This means that x is assigned as a placeholder within the scope of the quantifier, and the predicate $P(x)$ holds true for every possible value of x in the specified domain. The variable x cannot be used outside the scope of the quantifier.

Binding Variables in Existential Quantifiers: In a statement of the form $\exists x P(x)$, the variable x is also bound, but by the existential quantifier $\exists$. It signifies that there exists at least one value of x for which the predicate $P(x)$ holds true. Similar to the universal quantifier, the scope of the variable x is limited to the region where the quantifier operates.

Avoiding Confusion with Free Variables: It's essential to distinguish between bound variables, which are confined to the scope of quantifiers, and free variables, which are not quantified over and have independent meanings. To prevent confusion, it's common to introduce unique variables for each quantifier's scope.

Understanding the binding of variables helps in interpreting the semantics of quantified statements and clarifying the relationships between variables and predicates within logical propositions.

Logical Equivalences Involving Quantifiers

Logical equivalences involving quantifiers allow us to manipulate and transform statements while preserving their truth value. These equivalences are valuable tools for simplifying complex propositions and making deductions within logical reasoning.

Quantifier Negation Laws:

1. $\neg \forall x P(x) \equiv \exists x \neg P(x)$ - The negation of a universal quantifier is equivalent to the existential quantifier of the negation of the predicate. It expresses that there exists at least one element for which the negation of $P(x)$ is true.

2. $\neg \exists x P(x) \equiv \forall x \neg P(x)$ - The negation of an existential quantifier is equivalent to the universal quantifier of the negation of the predicate. It asserts that for every element, the negation of $P(x)$ holds true.

Quantifier Distribution Laws:

1. $\forall x(P(x) \wedge Q(x)) \equiv \forall x P(x) \wedge \forall x Q(x)$ - The conjunction of predicates under a universal quantifier is equivalent to the conjunction of the universal quantifiers of each predicate.

2. $\exists x(P(x) \vee Q(x)) \equiv \exists x P(x) \vee \exists x Q(x)$ - The disjunction of predicates under an existential quantifier is equivalent to the disjunction of the existential quantifiers of each predicate.

Logical equivalences involving quantifiers help in transforming statements into different forms while preserving their underlying meaning. These equivalences are instrumental in logical proofs, reasoning, and formalizing arguments across various domains.

Negating Quantified Expressions

Negating quantified expressions involves understanding how to negate statements that contain universal ($\forall$) and existential ($\exists$) quantifiers. Properly negating these statements is essential for accurately representing the opposite of the original propositions.

Negating Universal Quantifiers:

1. $\neg(\forall x P(x)) \equiv \exists x \neg P(x)$ - The negation of a universally quantified statement is equivalent to the statement that there exists at least one element for which the predicate is false.

Negating Existential Quantifiers:

1. $\neg(\exists x P(x)) \equiv \forall x \neg P(x)$ - The negation of an existentially quantified statement is equivalent to the statement that the predicate is false for every possible element.

Avoiding Ambiguity: Negating quantified expressions requires careful handling of negations and the rearrangement of quantifiers. It's important to maintain clarity and prevent confusion by correctly negating both the quantifiers and the predicates.

Negating quantified expressions is a crucial skill in logic and is employed when working with logical proofs, solving problems, and establishing the negation of statements involving quantifiers.

De Morgan's Laws for quantifiers are powerful tools that aid in comprehending the negations of quantified statements. These laws offer equivalences for negating statements involving universal ($\forall$) and existential ($\exists$) quantifiers.

Negation	Equivalent Statement	When Is Negation True?
$\neg\exists x P(x)$	$\forall x \neg P(x)$	For every x, $P(x)$ is false.
$\neg\forall x P(x)$	$\exists x \neg P(x)$	There is an x for which $P(x)$ is false.

These laws empower us to simplify and manipulate quantified statements when dealing with negations. They provide a clear comprehension of how negations impact the quantifiers and predicates within logical propositions.

The next section that we are covering this week is **Section 1.5 - Nested Quantifiers**.

Section 1.5 - Nested Quantifiers

Overview

Nested quantifiers are a powerful concept in mathematical logic and formal reasoning, allowing us to express intricate statements involving multiple variables and their relationships. These quantifiers occur when one quantifier is placed within the scope of another, creating a nested structure that captures complex conditions and relationships in mathematical expressions.

When working with nested quantifiers, the order in which they appear plays a significant role in determining the overall meaning of the statement. For instance, a statement with a universal quantifier ($\forall$) followed by an existential quantifier ($\exists$) signifies that for every value of the first variable, there exists a value of the second variable that satisfies the given condition. On the other hand, the reverse order, with an existential quantifier followed by a universal quantifier, implies that there exists a value of the first variable that works for all values of the second variable.

Understanding the scope and interplay of nested quantifiers is essential for precise mathematical reasoning. The correct interpretation of these nested structures ensures accurate communication of complex mathematical concepts, which is vital in various fields including mathematics, computer science, and formal logic.

The Order of Quantifiers

The order of quantifiers is a fundamental concept in mathematical logic and predicate calculus that determines how we interpret statements involving multiple quantifiers, such as universal ($\forall$) and existential ($\exists$) quantifiers. The order in which these quantifiers appear in a statement significantly influences the meaning and implications of that statement.

1. **Universal Quantifiers First ($\forall x \exists y$)**: When universal quantifiers precede existential quantifiers, it means that for every value of the first variable (x), there exists at least one value of the second variable (y) that satisfies the given condition. This order expresses that the property holds uniformly across all instances of the first variable.

2. **Existential Quantifiers First ($\exists x \forall y$)**: When existential quantifiers appear before universal quantifiers, it signifies that there exists at least one value of the first variable (x) for which the property holds true for every value of the second variable (y). This order asserts the existence of a specific instance that satisfies the condition universally.

The order of quantifiers is crucial in precise mathematical reasoning, as it can affect the validity and interpretation of mathematical statements. It allows mathematicians, logicians, and computer scientists to convey complex relationships and conditions accurately. Understanding how the order of quantifiers influences the meaning of statements is essential when working with predicate logic and formal reasoning.

Order of Quantifiers Example

Consider the following example that illustrates the importance of the order of quantifiers in mathematical statements:

---

1. **Universal Quantifier First:** $\forall x \exists y(x + y = 5)$
   - This statement asserts that for every value of x, there exists at least one value of y such that the sum of x and y equals 5. In other words, for any chosen x, you can always find a suitable y to satisfy the equation $x + y = 5$.
2. **Existential Quantifier First:** $\exists x \forall y(x + y = 5)$
   - In contrast, this statement asserts that there exists at least one value of x such that, for all values of y, the sum of x and y equals 5. It means that there is a single value of x that, when combined with any y, always results in $x + y = 5$.

The order of quantifiers changes the interpretation of the statement. In the first case, it means that for every choice of x, you can find a suitable y, while in the second case, it means there exists a specific x that works for all y. This example demonstrates how the order of quantifiers influences the meaning of mathematical statements and highlights the importance of correctly interpreting them in various contexts.

In mathematical logic, the use of quantifiers allows us to make statements about elements in a given domain. Quantifiers come in different forms, including universal ($\forall$) and existential ($\exists$) quantifiers. This table illustrates the various quantifications involving two variables and provides an understanding of when these statements hold true and when they are false.

Statement	When True?	When False?
$\forall x \forall y P(x, y)$	$P(x, y)$ is true for every pair x, y.	There is a pair x, y for which $P(x, y)$ is false.
$\forall y \forall x P(x, y)$	$P(x, y)$ is true for every pair x, y.	There is a pair x, y for which $P(x, y)$ is false.
$\forall x \exists y P(x, y)$	For every x, there is a y for which $P(x, y)$ is true.	There is an x such that $P(x, y)$ is false for every y.
$\exists x \forall y P(x, y)$	There is an x for which $P(x, y)$ is true for every y.	For every x, there is a y for which $P(x, y)$ is false.
$\exists x \exists y P(x, y)$	There is a pair x, y for which $P(x, y)$ is true.	$P(x, y)$ is false for every pair x, y.
$\exists y \exists x P(x, y)$	There is a pair x, y for which $P(x, y)$ is true.	$P(x, y)$ is false for every pair x, y.

This table illustrates various quantifications of two variables and explains when they are true or false in mathematical logic. The quantifiers $\forall$ and $\exists$ are used to make statements about all elements or at least one element in a given domain, and the table clarifies the conditions under which these statements hold or fail to hold.

Negating Nested Quantifiers

In mathematical logic, nested quantifiers are used to make statements about elements in a given domain, often involving both universal ($\forall$) and existential ($\exists$) quantifiers. When we negate such statements, we change the meaning of the statement while preserving the structure of nested quantifiers. The process of negating nested quantifiers involves reversing the original statement's meaning and requires applying De Morgan's Laws and carefully negating the predicates within the quantified expressions.

Negating nested quantifiers can be challenging, as the order and arrangement of quantifiers significantly impact the resulting negated statement's meaning. A precise understanding of the logical structure of the original statement and the correct application of negation rules are essential to ensure the accuracy of the negated statement.

Overall, negating nested quantifiers is a fundamental skill in mathematical logic and formal reasoning, enabling mathematicians and logicians to reason about complex statements involving multiple levels of quantification and draw precise conclusions about mathematical structures and systems.

Negation of Nested Quantifiers Example

Consider the following example illustrating the negation of nested quantifiers:

---

1. **Original Statement:** $\forall x \exists y(x + y = 5)$
   - This statement asserts that for every value of x, there exists at least one value of y such that the sum of x and y equals 5. In other words, for any chosen x, you can always find a suitable y to satisfy the equation $x + y = 5$.

2. **Negated Statement:** $\exists x \forall y(x + y \neq 5)$

   - To negate the original statement, we first apply De Morgan's Laws to reverse the meaning of the quantifiers, leading to $\exists x \neg(\exists y(x + y = 5))$. Then, we further negate the statement inside to obtain this negated form.
   - This negated statement asserts that there exists a value of x for which, for every possible value of y, the sum of x and y is not equal to 5. In other words, there is at least one value of x that makes the sum $x + y$ different from 5 for all possible values of y. This illustrates how the negation of nested quantifiers changes the original statement's meaning.

The last section that we are covering this week is **Section 1.6 - Rules of Inference**.

Section 1.6 - Rules of Inference

Overview

Rules of Inference are fundamental principles used in deductive reasoning to draw valid conclusions from given premises or statements. These rules serve as the foundation of formal logic and are indispensable tools in various fields, including mathematics, philosophy, computer science, and more.

Key principles and types of Rules of Inference include:

1. **Modus Ponens**: If we have a conditional statement $P \to Q$ and we know that P is true, we can infer that Q is true. It represents a straightforward form of reasoning where if a condition is met, the consequent must follow.

2. **Modus Tollens**: This rule works in reverse to Modus Ponens. If we have a conditional statement $P \to Q$ and we know that Q is false, we can conclude that P must be false as well. It's a way of reasoning by ruling out scenarios.

3. **Hypothetical Syllogism**: If we have two conditional statements, $P \to Q$ and $Q \to R$, we can derive a third conditional statement, $P \to R$. This rule allows us to chain together conditional statements to reach more complex conclusions.

4. **Disjunctive Syllogism**: If we have a disjunction $P \vee Q$ and we know that one of the disjuncts (either P or Q) is false, we can conclude that the other disjunct must be true. It's a way of reasoning through exclusion.

5. **Conjunction**: If we know that both P and Q are true, we can conclude that their conjunction $P \wedge Q$ is also true. This rule allows us to combine true statements into a compound statement.

6. **Addition**: Given a statement P, we can infer the disjunction $P \vee Q$ for any statement Q. This rule is a way of introducing new possibilities.

7. **Simplification**: If we have a conjunction $P \wedge Q$, we can separately infer both P and Q. It's a way of breaking down complex statements into simpler components.

8. **Resolution**: In propositional logic, this rule allows us to simplify complex disjunctive statements by resolving contradictory disjuncts. For example, from $P \vee Q$ and $\neg P \vee R$, we can infer $Q \vee R$ when P and $\neg P$ contradict each other.

These Rules of Inference provide a structured and systematic approach to logical reasoning, enabling us to make valid deductions and reach well-supported conclusions based on given information. They are essential tools for problem-solving, formal proof construction, and ensuring the soundness of arguments in various domains.

Rules of Inference for Propositional Logic

In deductive reasoning, **Rules of Inference** are fundamental principles used to draw valid conclusions from given premises or statements. These rules are crucial tools in various fields, including mathematics, logic, computer science, and philosophy. They provide a systematic way to reason and make logical deductions.

Here are some essential Rules of Inference:

Rule of Inference	Tautology	Name
p $p \to q$ $\therefore q$	$(p \land (p \to q)) \to q$	Modus Ponens
$\neg q$ $p \to q$ $\therefore \neg p$	$(\neg q \land (p \to q)) \to \neg p$	Modus Tollens
$p \to q$ $q \to r$ $\therefore p \to r$	$((p \to q) \land (q \to r)) \to (p \to r)$	Hypothetical Syllogism
$p \lor q$ $\neg p$ $\therefore q$	$((p \lor q) \land \neg p) \to q$	Disjunctive Syllogism
p $\therefore p \lor q$	$p \to (p \lor q)$	Addition
$p \land q$ $\therefore p$	$(p \land q) \to p$	Simplification
p q $\therefore p \land q$	$((p) \land (q)) \to (p \land q)$	Conjunction
$p \lor q$ $\neg p \lor r$ $\therefore q \lor r$	$((p \lor q) \land (\neg p \lor r)) \to (q \lor r)$	Resolution

The table provides a comprehensive overview of fundamental rules of inference within propositional logic, each accompanied by its associated tautology and name. These rules serve as the foundational elements of deductive reasoning, facilitating the derivation of valid conclusions based on provided premises. Modus Ponens, the initial rule, stipulates that if a statement p holds true and p implies q, then q is also true. In contrast, Modus Tollens focuses on negations, asserting that if the negation of a statement q is true and p implies q, then the negation of p is true.

Hypothetical Syllogism enables the concatenation of implications, allowing us to infer that if p implies q and q implies r, then p implies r. Disjunctive Syllogism addresses disjunctions, specifying that if either p or q is true (but not both) and p is false, then q must be true. Furthermore, Addition asserts that if p holds true, then p or any other statement q is also true. Simplification confirms that the simultaneous truth of both p and q implies that p is true, while Conjunction establishes that when both p and q hold true, the statement $p \land q$ is also true. Lastly, Resolution tackles intricate disjunctions by deducing that if p or q is true and simultaneously the negation of p or r holds, we can conclude that q or r is true. These rules are indispensable for structuring reasoning and validating arguments and mathematical proofs within propositional logic.

Rules of Inference for Quantified Statements

The Rules of Inference for Quantified Statements play a crucial role in predicate logic, facilitating the derivation of valid conclusions from statements that involve universal ($\forall$) and existential ($\exists$) quantifiers. These rules offer a systematic approach to formalizing arguments and mathematical proofs. Universal Instantiation (UI) allows us to replace universally quantified variables ($\forall x$) with specific values, permitting conclusions based on individual instances. Conversely, Universal Generalization (UG) lets us generalize specific statements to universally quantified ones, affirming their validity across the entire domain. Existential Instantiation (EI) enables us to substitute existentially quantified variables ($\exists x$) with specific values, demonstrating the existence of at least one element satisfying the predicate. Existential Generalization (EG) asserts the existence of an element that satisfies a predicate, a vital step in many proofs.

Additionally, the Modus Ponens and Modus Tollens rules, adapted for quantified statements, allow us to infer conclusions from universally quantified conditional statements and their negations, respectively. Hypothetical Syllogism for Quantifiers enables the chaining of universally quantified conditional statements, facilitating the derivation of conclusions through interconnected implications. Universal Transposition (UT) and Existential Transposition (ET) provide further flexibility by allowing the swapping of quantified variables' positions, aiding in the transformation of quantified expressions. In summary, these rules form the cornerstone of predicate logic, offering a structured framework for reasoning about quantified statements and laying the foundation for rigorous mathematical proofs.

Rule of Inference	Name
$\forall x P(x)$ $\therefore P(c)$ for an arbitrary c	Universal instantiation
$P(c)$ for an arbitrary c $\therefore \forall x P(x)$	Universal generalization
$\exists x P(x)$ $\therefore P(c)$ for some element c	Existential instantiation
$P(c)$ for some element c $\therefore \exists x P(x)$	Existential generalization

The table above provides a concise overview of essential rules of inference for quantified statements in first-order logic. These rules are foundational for deductive reasoning when dealing with statements that involve universal quantifiers ($\forall$) and existential quantifiers ($\exists$).

The rules encompass Universal Instantiation, Universal Generalization, Existential Instantiation, and Existential Generalization. Universal Instantiation allows us to derive specific instances from universally quantified statements, while Universal Generalization permits the generalization of a specific statement to a universally quantified form. On the other hand, Existential Instantiation enables the introduction of specific instances for existentially quantified statements, and Existential Generalization allows the generalization of a specific statement to an existentially quantified form. These rules are fundamental tools for logical reasoning and play a crucial role in proving theorems and making inferences in various domains of mathematics and computer science.

Proof Techniques

Proof Techniques

3.0.1 Reading Assignment

The reading assignments for this week is from, :

- Chapter 1.7 - Introduction To Proofs
- Chapter 1.8 - Proof Methods And Strategy

3.0.2 Piazza

Must post / respond to at least **two** Piazza posts this week.

3.0.3 Lectures

The lectures for this week and their links can be found below:

- Intro To Proofs ≈ 13 min.
- More Proofs ≈ 30 min.
- Proof Techniques & MW Help ≈ 26 min.
- Additional Proofs By Contradiction ≈ 28 min.

Below is a list of lecture notes for this week:

- Algebra And Log Rule Review Lecture Notes
- Proof Tutorial Lecture Notes
- Rosen Proofs Lecture Notes

3.0.4 Assignments

The assignment for this week is:

- Assignment 2 - Propositional Logic

3.0.5 Quiz

The quiz's for this week can be found at:

- Quiz 3 - Proof techniques

3.0.6 Chapter Summary

The first section that we are covering this week is **Section 1.7 - Introduction To Proofs**.

Section 1.7 - Introduction To Proofs

Overview

Proof techniques are essential tools in mathematics and logic for verifying the validity of statements. They offer various approaches to establishing the truth of mathematical claims. Common techniques include direct proof, where conclusions are logically derived from premises, proof by contradiction, which assumes the opposite of the statement and finds a contradiction, mathematical induction for statements involving natural numbers, proof by contrapositive for proving the opposite of a statement, proof by exhaustion for finite cases, proof by counterexample to disprove statements, and proof by construction to demonstrate the existence of objects or solutions. These techniques enable mathematicians and logicians to provide rigorous evidence for mathematical arguments, with the choice of method depending on the nature of the statement being proven.

Direct Proofs

Direct proofs are a fundamental technique in mathematics and logic used to establish the truth of a mathematical statement or theorem. In a direct proof, one starts with the given premises, often called axioms or assumptions, and logically derives a conclusion or a new statement based on these premises. The goal is to show that the conclusion is true under the provided conditions.

The key steps in a direct proof typically involve applying established mathematical principles, theorems, and rules of logic to demonstrate that the conclusion logically follows from the given premises. Direct proofs are characterized by their clarity, simplicity, and the absence of any assumptions that contradict the premises.

In essence, direct proofs provide a straightforward and convincing way to demonstrate the validity of mathematical statements, making them an essential tool in mathematical reasoning and problem-solving. They form the foundation of mathematical proofs and are widely used across various branches of mathematics, from algebra and calculus to geometry and number theory.

Direct Proofs Example

Theorem: If n is an even integer, then n^2 is also an even integer.

Proof: Let's assume that n is an even integer. By definition, this means that there exists an integer k such that $n = 2k$.

Now, we want to prove that n^2 is also an even integer. We can express n^2 as:

$$n^2 = (2k)^2 = 4k^2$$

Since k is an integer, $4k^2$ is also an integer. Therefore, n^2 can be expressed as $n^2 = 2(2k^2)$, where $2k^2$ is an integer. This means that n^2 is divisible by 2, making it an even integer. Thus, we have shown that if n is an even integer, then n^2 is also an even integer, which concludes our proof.

Proof By Contraposition

Proof by contraposition is a powerful technique in mathematical and logical reasoning used to establish the validity of a conditional statement (if-then statement). It is based on the idea that to prove an implication of the form "if P, then Q," you can instead prove its contrapositive, which is "if not Q, then not P." The contrapositive is logically equivalent to the original statement, and proving it can sometimes be simpler or more intuitive.

1. **Start with a Conditional Statement:** Begin with a conditional statement of the form "if P, then Q," where P and Q are propositions or statements.
2. **Formulate the Contrapositive:** Rewrite the statement in its contrapositive form, which is "if not Q, then not P." In other words, negate both the consequent (Q) and the antecedent (P).
3. **Prove the Contrapositive:** To establish the truth of the original statement, prove the contrapositive instead. This can be done using various proof techniques, such as direct proof, contradiction, or other appropriate methods.
4. **Conclude the Original Statement:** Once you have successfully proven the contrapositive, you can conclude the truth of the original statement. If "if not Q, then not P" holds, then "if P, then Q" also holds, as they are logically equivalent.

Proof by contraposition is particularly useful when direct proof seems challenging or when it simplifies the argument. It is a valuable tool in mathematical proofs, logic, and various areas of science and engineering. By proving the contrapositive, you establish the validity of conditional statements efficiently and logically.

Proof by Contraposition Example

Theorem: If n is an even integer, then n^2 is also an even integer.

Proof: We will prove the contrapositive of the statement, which is: If n^2 is an odd integer, then n is also an odd integer.

Let's assume that n is an even integer, which implies that $n = 2k$ for some integer k. Now, consider the square of n:

$$n^2 = (2k)^2 = 4k^2$$

Since k is an integer, $4k^2$ is also an integer, and it's clear that $4k^2$ is even because it can be expressed as $2(2k^2)$.

Now, let's prove the contrapositive. If n^2 is an odd integer, we can write it as $n^2 = 2m + 1$ for some integer m. Substituting n^2 with $4k^2$, we get:

$$4k^2 = 2m + 1$$

Rearranging this equation gives:

$$2(2k^2) = 2m + 1$$

Subtracting 1 from both sides:

$$2(2k^2) - 1 = 2m$$

Now, we have shown that $2m$ is an even integer, which means m is also even since an even integer multiplied by 2 results in an even integer.

So, $m = 2p$ for some integer p. Substituting this back into the equation:

$$2(2k^2) - 1 = 2(2p)$$

Simplifying:

$$4k^2 - 1 = 4p$$

Now, we have expressed $4k^2 - 1$ as $4p$, which is even, and this implies that $4k^2 - 1$ is also even.

Since $4k^2 - 1$ is even, n^2 (which is equal to $4k^2 - 1$) must be even, which contradicts our initial assumption that n^2 is odd. Therefore, our assumption that n is even is false. Hence, we have proved the contrapositive: If n^2 is an odd integer, then n is also an odd integer.

As a result, by contraposition, we have established the original statement: If n is an even integer, then n^2 is also an even integer.

Proof By Contradiction

Proof by Contradiction, also known as reductio ad absurdum, is a powerful mathematical technique used to establish the truth of a statement by assuming the opposite and demonstrating that this assumption leads to a logical contradiction. This method rests on the principle that if assuming the negation of a statement results in an inconsistency or absurdity, then the original statement must be true.

In a proof by contradiction, you start with the negation of the statement you want to prove and then proceed to derive a contradiction from this assumption. This contradiction could be anything that clearly violates the laws of logic or mathematics, such as showing that 1 equals 0 or that a number is both even and odd. Since such contradictions cannot exist, the only logical conclusion is that the initial assumption (the negation of the statement you aimed to prove) must be false. Consequently, the original statement is proven to be true.

Proofs by contradiction are valuable in mathematics and logic because they provide a systematic way to establish the truth of a proposition when direct or other proof methods are challenging or not readily available. They are particularly useful in proving the existence of mathematical objects or properties and are a fundamental tool in various areas of mathematics and computer science.

Proof by Contradiction Example

Theorem: There is no largest prime number.

Proof: Let's assume, for the sake of contradiction, that there exists a largest prime number, denoted as p. We will now derive a contradiction from this assumption.

Consider the number $N = p + 1$. Now, N is either prime or not prime. There are two possibilities to explore:

Case 1: N is prime. In this case, we have found a prime number (N) larger than our assumed largest prime (p), which contradicts our initial assumption.

Case 2: N is not prime. This means that it must be divisible by some integer greater than 1, but less than or equal to p, because p is assumed to be the largest prime. However, if N is divisible by an integer greater than 1 and less than or equal to p, it cannot be prime. This also contradicts our initial assumption that N is prime.

In both cases, we arrive at a contradiction. Therefore, our initial assumption that there exists a largest prime number (p) must be false. Hence, there is no largest prime number.

The last section that we are covering this week is **Section 1.8 - Proof Methods And Strategy**.

Section 1.8 - Proof Methods And Strategy

Overview

Proof methods and strategies are essential tools in mathematics and logic, providing systematic approaches to establish the validity of mathematical statements and arguments. Three fundamental proof techniques include direct proof, proof by contraposition, and proof by contradiction.

Direct proof is a foundational approach in mathematics, where one starts with given premises and employs logical reasoning to demonstrate that a specific conclusion logically follows from those premises. It relies on axioms, definitions, and previously proven theorems to construct a coherent argument that validates the desired result. Direct proof serves as the cornerstone of mathematical proof and is widely employed across various mathematical disciplines.

Proof by contraposition is a potent strategy where mathematicians aim to prove a statement by showing that its contrapositive, which is the negation of the statement's conclusion implying the negation of its hypothesis, is true. By establishing the contrapositive, they indirectly validate the original statement's correctness. This approach becomes particularly valuable when a direct proof may be challenging or less apparent.

Proof by contradiction involves assuming the opposite of the statement to be proven and demonstrating that this assumption leads to a logical contradiction. If the assumption indeed results in a contradiction, it implies that the original statement must be true. This method is particularly effective in proving the non-existence of solutions or disproving the presence of specific properties. These proof techniques serve as the foundation of mathematical reasoning, enabling mathematicians to explore mathematical concepts, validate conjectures, and contribute to the advancement of mathematical knowledge.

Exhaustive Proof

Exhaustive proof is a rigorous mathematical technique involving the meticulous examination of every possible case or scenario within a given problem domain to establish the validity of a statement. This method, often used in discrete mathematics, ensures that no cases are left unverified, providing a high level of certainty in the proof's correctness. While exhaustive proofs are considered airtight, they can be time-consuming and labor-intensive, making them less practical for large or complex problem spaces. Mathematicians often choose more efficient proof techniques when possible, such as direct proofs or proof by contradiction.

Exhaustive Proof Example

Theorem: For any positive integer n, if n is divisible by 2 and 3, then n is divisible by 6.

Proof: To prove this statement, we will examine every possible case for n. Since we want to show that n is divisible by both 2 and 3, we need to consider all positive integers that satisfy this condition.

1. **Case 1:** $n = 6$
   In this case, n is both divisible by 2 and 3, and therefore, it is divisible by 6.
2. **Case 2:** $n = 12$
   Again, n is divisible by both 2 and 3, making it divisible by 6.
3. **Case 3:** $n = 18$

Once more, n satisfies the conditions of being divisible by 2 and 3, leading to its divisibility by 6.

4. **Case 4:** $n = 24$
   In this case, n is divisible by both 2 and 3, hence it is divisible by 6.

5. **Case 5:** $n = 30$
   Finally, n is divisible by 2 and 3, making it divisible by 6.

We have systematically examined all positive integers that are divisible by 2 and 3, and in each case, we found that they are also divisible by 6. Therefore, by exhausting all possibilities, we have proven that for any positive integer n divisible by 2 and 3, n is indeed divisible by 6.

Proof By Cases

Proof by cases is a valuable mathematical proof technique that involves breaking down a complex problem into distinct, manageable scenarios or cases. Each case is examined individually to determine whether the statement holds true under those specific conditions. By considering all possible cases exhaustively, mathematicians can establish the validity of a theorem or proposition for all potential situations. This method provides a structured approach for tackling intricate problems, ensuring that every possible scenario is thoroughly analyzed and contributing to the overall rigor of mathematical proofs.

Proof by Cases Example

Theorem: For any integer n, if n is even, then n^2 is also even.

Proof: We'll prove this theorem by considering two cases.

Case 1: n is even. In this case, we can express n as $n = 2k$ for some integer k. Now, let's calculate n^2:

$$n^2 = (2k)^2 = 4k^2$$

Since k is an integer, $4k^2$ is also an integer, and n^2 can be expressed as $n^2 = 2(2k^2)$, where $2k^2$ is an integer. Therefore, n^2 is even when n is even.

Case 2: n is odd. In this case, we can express n as $n = 2k + 1$ for some integer k. Now, let's calculate n^2:

$$n^2 = (2k + 1)^2 = 4k^2 + 4k + 1$$

We observe that n^2 is also an odd integer because it can be expressed as $n^2 = 2(2k^2 + 2k) + 1$, where $2k^2 + 2k$ is an integer.

In both cases, we've shown that if n is even or odd, n^2 is either even or odd, respectively. This exhaustively proves that for any integer n, if n is even, then n^2 is also even.

Leveraging Proof By Cases

Proof by cases is a strategy frequently employed in mathematical and logical reasoning, allowing mathematicians to tackle complex problems by breaking them down into distinct scenarios or cases and providing proofs for each scenario individually. By methodically considering all possible cases, this approach ensures the thorough examination of a theorem or problem, enhancing its rigor and credibility. Proof by cases is particularly valuable when one overarching method cannot address all potential situations, making it an essential tool for mathematicians seeking to establish the validity of their arguments across diverse scenarios and conditions.

Proof by Cases Example

Theorem: For any integer n, n^2 is either even or odd.

Proof: Let's consider two cases.

Case 1: n is even. In this case, we can write n as $n = 2k$ for some integer k. Now, let's examine n^2:

$$n^2 = (2k)^2 = 4k^2$$

Since k is an integer, $4k^2$ is also an integer. Thus, n^2 is even.

Case 2: n is odd. In this case, we can write n as $n = 2k + 1$ for some integer k. Now, let's examine n^2:

$$n^2 = (2k + 1)^2 = 4k^2 + 4k + 1 = 2(2k^2 + 2k) + 1$$

Again, since k is an integer, $2(2k^2 + 2k)$ is an integer. Therefore, n^2 is odd.

In both cases, we have shown that n^2 is either even or odd, concluding our proof by cases.

Without Loss Off Generality

(WLOG) is a frequently used term in mathematical proofs. It enables us to simplify the reasoning process by considering a specific case or subset of cases. Essentially, it allows for the assumption of a particular scenario that does not compromise the overall generality of the proof. Typically employed when dealing with multiple cases, WLOG permits us to choose one case and demonstrate that the result applies universally to all analogous cases, as the argument's generality encompasses the entire set. This technique enhances the clarity and conciseness of mathematical proofs.

Existence Proofs

Existence proofs are a fundamental part of mathematical reasoning used to demonstrate that a particular mathematical object or solution to a problem indeed exists within a given context. These proofs aim to show that there is at least one example or instance that satisfies a particular property or condition. Existence proofs come in various forms, including constructive and non-constructive methods.

In a constructive existence proof, mathematicians provide a method or algorithm to explicitly construct an example that satisfies the given conditions. This type of proof not only establishes the existence of the object but also provides a way to find it. On the other hand, non-constructive existence proofs demonstrate that an object with the desired properties must exist, without necessarily providing a way to find it. These proofs often rely on logical reasoning and principles like contradiction.

Existence proofs are crucial in various mathematical disciplines, including number theory, set theory, and calculus, and they play a pivotal role in establishing the existence of solutions to equations, mathematical structures, or objects in abstract spaces. They help mathematicians and researchers affirm the reality of mathematical concepts and provide assurance that specific mathematical entities are not mere theoretical constructs but tangible components of mathematical reality.

Existence Proof

Theorem: There exists at least one prime number with a last digit of 7.

Proof: Let's consider all the natural numbers that have a last digit of 7, i.e., numbers of the form $n = 10k + 7$, where k is a non-negative integer. We will show that at least one of these numbers is prime.

Suppose, for the sake of contradiction, that none of these numbers is prime. That would mean that for each such number n, there exists a positive integer $m > 1$ such that n is divisible by m.

Now, let's construct a new number N by multiplying all these positive integers m together:

$$N = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot \ldots$$

Since there are infinitely many natural numbers ending in 7, there are infinitely many factors in N. But notice that none of these numbers can be 2 or 5 because they don't end in 7. So, N is greater than 1, and it's divisible by a prime number greater than 1.

This contradicts the fundamental theorem of arithmetic, which states that every natural number greater than 1 can be uniquely expressed as a product of prime numbers. Therefore, there must be at least one prime number with a last digit of 7. This completes the proof.

Uniqueness Proofs

Uniqueness proofs are a type of mathematical argument used to establish that a particular object or solution is unique within a given context or set of conditions. These proofs are typically employed when it's necessary to demonstrate that there is only one possible solution or object that satisfies a specific property or criterion.

In uniqueness proofs, the primary goal is to show that if there were two or more distinct objects or solutions that met the defined conditions, it would lead to a contradiction or inconsistency. This contradiction often arises from the assumption that multiple solutions exist when, in fact, only one can exist. By demonstrating that any two

supposed solutions would be equivalent or identical, these proofs establish the uniqueness of the solution, providing mathematical certainty that no other possibilities exist within the specified context.

Uniqueness proofs are widely utilized across various mathematical disciplines, including algebra, calculus, number theory, and more. They play a crucial role in validating the existence of unique solutions or objects, strengthening the foundations of mathematical reasoning and problem-solving.

Uniqueness Proof Example

Theorem: For any positive real number a, there exists a unique positive real number x such that $x^2 = a$.

Proof: To prove the uniqueness of the square root, we'll first establish the existence of such a number. Suppose a is a positive real number. Consider the set $S = \{x \in \mathbb{R}^+ \mid x^2 = a\}$. By definition, S contains at least one element since a itself satisfies $x^2 = a$ when $x = \sqrt{a}$. Therefore, there exists at least one positive real number x such that $x^2 = a$.

Now, let's prove uniqueness by contradiction. Assume there are two distinct positive real numbers, x and y, both satisfying $x^2 = a$ and $y^2 = a$. Without loss of generality, assume $x < y$. Then, consider the difference $y - x$. Since both x and y are positive, this difference is also positive. Now, consider the square of this difference:

$$(y - x)^2 = y^2 - 2xy + x^2 = a - 2xy + a = 2a - 2xy.$$

Since x and y both satisfy $x^2 = a$ and $y^2 = a$, we have $2a - 2xy = a - 2xy = 0$. However, this implies that $(y - x)^2 = 0$, which in turn implies $y - x = 0$, or $y = x$. This contradicts our assumption that x and y are distinct. Therefore, there cannot be two distinct positive real numbers x and y that both satisfy $x^2 = a$. Hence, the square root of a positive real number a is unique.

This uniqueness proof demonstrates that for any positive real number a, there exists a unique positive real number x such that $x^2 = a$.

Looking For Counterexamples

Looking for counterexamples is a crucial strategy in mathematical proof and problem-solving. It involves searching for specific instances or cases that contradict a given statement or conjecture. The goal is to demonstrate that a universally quantified statement (one that claims something is true for all elements in a set) is false by providing a single example where it does not hold.

This proof strategy is particularly effective when dealing with universally quantified statements. By finding a counterexample, you show that the statement is not true for all cases, thereby invalidating it. Counterexamples can be especially useful in identifying the limits or boundary conditions of a mathematical assertion, helping to refine statements and theories.

In practice, mathematicians often rely on counterexamples to challenge conjectures, test the robustness of theorems, and explore the boundaries of mathematical concepts. By disproving a statement through counterexamples, mathematicians gain valuable insights into the nature of mathematical truths and refine their understanding of various mathematical principles.

Counterexample Example

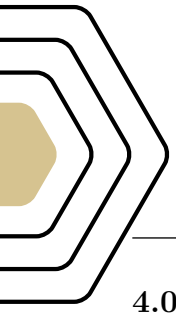
Claim: For all positive integers n, $n^2 + n$ is always prime.

Counterexample: Let's look for a counterexample. Consider the case when $n = 4$. In this case, $n^2 + n = 4^2 + 4 = 20$. However, 20 is not a prime number, as it has divisors other than 1 and itself (2 and 10).

So, we have found a counterexample (when $n = 4$) where $n^2 + n$ is not prime, which disproves the claim that it is always prime for all positive integers n.

Therefore, the claim is false, and we have successfully used a counterexample to demonstrate its invalidity.

Sets And Functions



Sets And Functions

4.0.1 Assigned Reading

The reading assignments for this week is from, :

- Chapter 2.1 - Sets
- Chapter 2.2 - Set Operations
- Chapter 2.3 - Functions
- Chapter 4.1 - Divisibility And Modular Arithmetic

4.0.2 Piazza

Must post / respond to at least **two** Piazza posts this week.

4.0.3 Lectures

The lectures for this week and their links can be found below:

- Sets And Set Operations ≈ 1 hour, 2 min.
- Functions ≈ 40 min.
- Modular Arithmetic Review ≈ 5 min.
- Modular Arithmetic With Scopes And Cups ≈ 7 min.
- Best Function You've Never Heard Of Part 1 ≈ 6 min.
- Best Function You've Never Heard Of Part 2 ≈ 10 min.
- Make A Slide Rule ≈ 8 min.
- Algorithms ≈ 36 min.
- Countable And Uncountable Sets ≈ 18 min.

Below is a list of lecture notes for this week:

- Algorithms Lecture Notes
- Countable And Uncountable Sets Lecture Notes
- Functions Lecture Notes
- Modular Scoops Lecture Notes
- Sets And Set Operations Lecture Notes

4.0.4 Assignments

The assignment for this week is:

- Assignment 4 - Sets And Functions

4.0.5 Quiz

The quiz's for this week can be found at:

- Quiz 4 - Sets And Modulo

4.0.6 Chapter Summary

The first section that we are covering this week is **Section 2.1 - Sets**.

Section 2.1 - Sets

Overview

In the realm of discrete mathematics, sets play a foundational role, serving as one of the fundamental mathematical concepts. A set is essentially a collection of distinct elements, denoted within curly braces {}. These elements can encompass a wide range of entities, such as numbers, letters, symbols, or even other sets. For instance, a set of natural numbers could be represented as $\{1, 2, 3, 4, \ldots\}$, while a set of uppercase letters might appear as $\{A, B, C, \ldots\}$.

One of the key principles that govern sets is the concept of equality. Two sets are considered equal if and only if they contain the exact same elements. For example, if we have a set $A = \{1, 2, 3\}$ and another set $B = \{3, 1, 2\}$, they are still deemed equal since their elements are identical. Sets also have a unique feature known as cardinality, which measures the number of elements within a set. In the case of sets A and B mentioned earlier, their cardinality is 3 since they both contain three elements.

Among the essential ideas in set theory are the notions of subsets, union, and intersection. A set A is considered a subset of another set B if every element in A is also present in B. The union of two sets, denoted by $A \cup B$, combines all distinct elements from both sets into a new set. Conversely, the intersection of two sets, denoted by $A \cap B$, contains only the elements that are common to both A and B. These concepts provide the foundation for various mathematical operations and problem-solving techniques in discrete mathematics.

Sets also have a counterpart known as the empty set $\emptyset$, which contains no elements. The existence of the empty set is crucial for mathematical reasoning and allows us to represent situations where there are no items in a particular category. For instance, the set of natural numbers less than zero would be represented as an empty set: $\emptyset$.

In summary, sets are fundamental entities in discrete mathematics, serving as a means to organize and manipulate collections of elements. Their properties, including equality, cardinality, subsets, union, intersection, and the existence of the empty set, play a crucial role in various branches of discrete mathematics, including combinatorics, set theory, probability, and graph theory. Understanding sets is essential for effective problem-solving and mathematical reasoning in these fields.

Subsets

Subsets are a foundational concept in set theory and discrete mathematics. In essence, a subset is a set that comprises only elements that are also part of another set, known as the "superset." Symbolically, if every element within set A is also found in set B, then A is considered a subset of B, denoted as $A \subseteq B$. Subsets find extensive use in various mathematical and logical contexts, helping categorize elements based on specific properties or criteria. Whether analyzing integers, real numbers, or more complex structures, subsets enable mathematicians to break down intricate problems into manageable parts. Additionally, subsets can include the "empty set" ($\emptyset$ or {}), which contains no elements and is considered a subset of every set, making it a fundamental concept in mathematical proofs and reasoning. Moreover, the "power set" of a set A, denoted as $P(A)$, encompasses all possible subsets of A, including A itself and the empty set, often applied in combinatorics and discrete mathematics to explore various combinations and permutations.

Size Of A Set

In the realm of set theory, determining the size or cardinality of a set is a fundamental concept. The size of a set refers to the number of elements it contains. It is denoted as $|A|$, where A represents the set. The cardinality of a set can be finite or infinite. For finite sets, counting the elements directly provides the cardinality. However, for infinite sets, the cardinality is assessed differently, often by establishing a correspondence with a known infinite set, such as the natural numbers.

The concept of cardinality is essential for comparing sets and establishing relationships between them. Two sets are considered equal in size if their cardinalities are the same, regardless of the elements within them. Set operations, such as union and intersection, can also be analyzed in terms of cardinality, allowing mathematicians to make precise statements about the sizes of resulting sets. Additionally, cardinality plays a pivotal role in combinatorics, probability theory, and various branches of mathematics where counting and measuring the size of sets are integral to solving problems and proving theorems.

Cartesian Products

In discrete mathematics, the Cartesian product is a fundamental concept that allows us to construct new sets from existing ones. Given two sets A and B, the Cartesian product $A \times B$ is defined as the set of all ordered pairs where the first element comes from set A and the second element comes from set B. Symbolically, $A \times B = \{(a, b) \mid a \in A, b \in B\}$.

The Cartesian product is a versatile tool used in various mathematical and computer science applications. It is commonly employed in geometry to represent points in the Cartesian coordinate system. In set theory, it plays a crucial role in defining relations between sets and functions. In computer science, Cartesian products are used to model the state spaces of systems, such as finite automata.

Cartesian products can be extended to more than two sets. For example, the Cartesian product of three sets $A \times B \times C$ consists of ordered triples, and so on for higher dimensions. Understanding Cartesian products and their properties is essential for solving problems involving ordered pairs, relations, and multi-dimensional data structures.

Set Notation With Quantifiers

Set notation and quantifiers are essential tools in discrete mathematics for describing and making statements about collections of elements. The universal quantifier ($\forall$) is employed to express statements that hold true for every element within a given set. For instance, $\forall x \in A, P(x)$ asserts that the predicate $P(x)$ is valid for all elements x in set A. Conversely, the existential quantifier ($\exists$) is used to indicate that at least one element in a set satisfies a particular predicate. For example, $\exists x \in A, P(x)$ implies that there exists an element x in set A for which $P(x)$ is true.

Set notation, in conjunction with quantifiers, provides a concise and precise way to represent complex statements about sets and their elements. The set-builder notation, $\{x \in A \mid P(x)\}$, signifies the set of all elements in A for which the predicate $P(x)$ holds. This combination of tools is indispensable in various mathematical fields, such as set theory, logic, and proof theory. It enables mathematicians to systematically and rigorously reason about collections of objects, making it a fundamental aspect of discrete mathematics.

The next section that we are covering this week is **Section 2.2 - Set Operations**.

Section 2.2 - Set Operations

Overview

In discrete mathematics, sets are fundamental objects used to represent collections of elements. Set operations allow us to manipulate and analyze sets in various ways. The primary set operations include union, intersection, complement, and difference.

The union of two sets, denoted by $A \cup B$, contains all the unique elements that belong to either set A or set B, or both. It combines elements from both sets without duplication. The intersection of two sets, denoted by $A \cap B$, consists of all the elements that are common to both set A and set B. In other words, it contains only the elements that appear in both sets.

The complement of a set, denoted by $\neg A$ or A', contains all the elements from the universal set that do not belong to set A. It represents everything outside of set A within the universal set. The difference between two sets, denoted by $A - B$, contains all the elements that belong to set A but not to set B. It essentially removes the elements of set B from set A.

Set operations are crucial tools in various mathematical disciplines, including set theory, probability, and discrete mathematics. They help us perform tasks like combining sets, finding common elements, and defining subsets. These operations follow specific rules and properties that enable precise and systematic reasoning about sets and their relationships, making them essential for solving a wide range of mathematical problems.

Set Identities

In the realm of set theory, set identities are fundamental principles and relationships that govern the behavior of sets and their operations. These identities help mathematicians manipulate sets and express complex relationships with clarity and precision. Some of the key set identities include the identity laws, domination laws, complement laws, double complement law, and De Morgan's laws.

The identity laws state that the union of a set with the universal set is the universal set itself, and the intersection of a set with the universal set is the set itself. These laws highlight the role of the universal set as an identity

element for set operations.The domination laws state that the union of a set with the empty set is the set itself, and the intersection of a set with the universal set is the empty set. These laws underscore the influence of the empty set as a dominating element in set operations.

The complement laws describe the relationship between a set and its complement. They state that the union of a set with its complement is the universal set, and the intersection of a set with its complement is the empty set. These laws highlight the complementary nature of sets. The double complement law states that the complement of the complement of a set is the set itself. It emphasizes that taking the complement of a set twice returns the original set.De Morgan's laws provide a powerful tool for expressing the complement of union and intersection. They state that the complement of the union of two sets is equal to the intersection of their complements, and the complement of the intersection of two sets is equal to the union of their complements. These laws enable the simplification of complex set expressions.

Set identities serve as a foundation for proving theorems and making deductions in set theory and related mathematical disciplines. They provide a systematic way to reason about sets and their properties, making them indispensable tools for solving problems in various areas of mathematics.

Set Identities

Here is a table of set identities that are similar to propositional logic.

Identity Name	Set Identity
Identity laws	$A \cap U = A$ $A \cup \emptyset = A$
Domination laws	$A \cup U = U$ $A \cap \emptyset = \emptyset$
Idempotent laws	$A \cup A = A$ $A \cap A = A$
Complementation law	$\overline{\overline{A}} = A$
Commutative laws	$A \cup B = B \cup A$ $A \cap B = B \cap A$
Associative laws	$A \cup (B \cup C) = (A \cup B) \cup C$ $A \cap (B \cap C) = (A \cap B) \cap C$
Distributive laws	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
De Morgan's laws	$A \cup B = A \cap B$ $A \cap B = A \cup B$
Absorption laws	$A \cup (A \cap B) = A$ $A \cap (A \cup B) = A$
Complement laws	$A \cup A = U$ $A \cap A = \emptyset$

Generalized Unions And Intersections

In the realm of set theory and discrete mathematics, Generalized Unions and Intersections offer a versatile approach to dealing with collections of sets. These operations extend the principles of regular unions and intersections to handle entire families of sets, making them invaluable tools in various mathematical contexts.

The Generalized Union, often represented as $\bigcup$, allows us to combine the elements of multiple sets within a given family of sets. This operation is particularly useful when dealing with infinite sets or an unspecified number of sets. The result of $\bigcup$ is a new set containing all unique elements found in any set within the family, without repetition.

On the other hand, the Generalized Intersection, denoted as $\bigcap$, enables us to find elements that are common to all sets in a given family. Like its union counterpart, the Generalized Intersection accommodates infinite or unspecified sets within the family. The outcome of $\bigcap$ is a new set that includes only the elements present in every set of the family.

These Generalized Unions and Intersections adhere to the same fundamental principles as regular unions and intersections, including the commutative and associative properties. They play a pivotal role in various mathematical disciplines, such as topology, set theory, and real analysis, where dealing with diverse and unbounded sets is commonplace. Whether handling countable or uncountable collections, these operations provide a robust framework for reasoning about sets and their relationships, allowing mathematicians to explore intricate concepts and solve complex problems.

The next section that we are covering this week is **Section 2.3 - Functions**.

Section 2.3 - Functions

Overview

Functions in discrete mathematics are a cornerstone concept with broad applicability in diverse mathematical and computational domains. These mathematical constructs are essentially relationships that link two sets: a domain (comprising all possible input values) and a codomain (encompassing all potential output values). A crucial characteristic of functions is their unique mapping property, where each element from the domain corresponds to precisely one element in the codomain. This mapping, typically represented as $f : A \to B$ for a function f with domain A and codomain B, allows for the systematic analysis, modeling, and problem-solving across numerous mathematical and computer science disciplines.

Functions come with several key attributes. They begin with the definition of their domain and codomain, with the codomain often serving as a broader set that includes all possible output values. The range of a function denotes the subset of the codomain containing the actual output values produced by the function. Additionally, functions can be categorized based on their behavior. A function is one-to-one (or injective) if it assigns distinct values to different domain elements, ensuring that no two domain elements map to the same codomain element. Conversely, a function is onto (or surjective) when every element in the codomain has at least one corresponding element in the domain.

Function notation, such as $f(x)$, is used to denote the relationship between elements in the domain and their corresponding elements in the codomain. This mathematical concept serves as a foundational tool across various mathematical domains, including algebra, calculus, number theory, and combinatorics. Moreover, in computer science, functions are instrumental for modeling algorithms, data transformations, and program behaviors. By studying functions and their properties, mathematicians and computer scientists gain the fundamental tools needed to analyze, model, and solve an extensive range of mathematical and computational problems.

One-to-One And Onto Functions

One-to-One and Onto functions are fundamental concepts in discrete mathematics and play a crucial role in understanding the relationships between sets and their mappings. A function is considered one-to-one, or injective, if each element in its domain maps to a unique element in its codomain. In simpler terms, no two distinct elements in the domain can map to the same element in the codomain. This property ensures that there is a one-to-one correspondence between elements in the domain and their images in the codomain, making it possible to invert the function when restricted to its range.

Conversely, a function is termed onto, or surjective, when its range covers the entire codomain. In other words, every element in the codomain has a pre-image in the domain under the function. Onto functions are exhaustive in their mappings, leaving no "gaps" or uncovered elements in the codomain. When a function is both one-to-one and onto, it is referred to as a bijection. Bijections establish a bijective correspondence between the elements of the domain and codomain, providing a complete and reversible mapping.

One-to-One and Onto functions are crucial in various mathematical fields, including algebra, calculus, and combinatorics. They allow mathematicians to analyze the behavior of functions, determine their invertibility, and explore the properties of sets and their relationships. In computer science and data analysis, these concepts are essential for understanding data transformations, encryption algorithms, and database operations. One-to-One and Onto functions provide powerful tools for modeling real-world situations and solving mathematical and computational problems effectively.

Inverse Functions And Compositions Of Functions

In discrete mathematics, inverse functions and compositions of functions are fundamental concepts. An inverse function, denoted as f^{-1}, is a function that "undoes" the actions of another function. For any element x in the domain, f maps it to y, while f^{-1} maps y back to x. The properties $f(f^{-1}(x)) = x$ for all x in the domain and $f^{-1}(f(x)) = x$ for all x in the codomain of f must hold for them to be considered inverses.

Compositions of functions involve applying one function to the output of another. If we have two functions f and g, the composition $g \circ f$ means applying f first and then g. Compositions help us understand how multiple functions combine and their combined effects. It's expressed as $(g \circ f)(x) = g(f(x))$. The order of composition matters, as $g \circ f$ may not be the same as $f \circ g$.

These concepts are vital in various branches of mathematics, such as calculus, linear algebra, and abstract algebra, enabling us to analyze transformations, solve equations, and explore relationships between functions. In computer science and engineering, they are essential for modeling systems, designing algorithms, and optimizing data processing. Inverse functions and compositions provide a powerful framework for understanding complex mappings and transformations in both mathematical and real-world contexts.

The last section that we are covering this week is **Section 4.1 - Divisibility And Modular Arithmetic**.

Section 4.1 - Divisibility And Modular Arithmetic

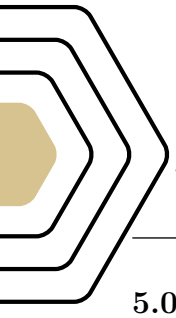
Overview

In discrete mathematics, divisibility and modular arithmetic are fundamental concepts with widespread applications. Divisibility refers to the property of one integer being divisible by another without leaving a remainder. For integers a and b, if there exists an integer k such that $a = bk$, then b is said to divide a, denoted as $b|a$. Divisibility plays a crucial role in number theory, prime factorization, and understanding the properties of integers.

Modular arithmetic, on the other hand, deals with remainders when integers are divided. It operates within a fixed modulus m and considers integers to be congruent if they have the same remainder when divided by m. This concept is denoted as $a \equiv b \mod m$. Modular arithmetic has various applications, including cryptography, computer science (in algorithms and data structures), and number theory (in solving Diophantine equations).

These concepts provide tools for solving equations, understanding patterns in number sequences, and working with cyclic data in real-world applications. Divisibility and modular arithmetic are essential in many mathematical disciplines and have practical implications in various fields, making them valuable tools for mathematicians, scientists, and engineers.

Algorithms And Number Properties



Algorithms And Number Properties

5.0.1 Assigned Reading

The reading assignments for this week is from, :

- Chapter 2.6 - Matrices
- Chapter 3.1 - Algorithms
- Chapter 3.2 - The Growth Of Functions
- Chapter 4.2 - Integer Representation And Algorithms

5.0.2 Piazza

Must post / respond to at least **two** Piazza posts this week.

5.0.3 Lectures

The lectures for this week and their links can be found below:

- Algorithms ≈ 36 min.
- Complexity ≈ 50 min.
- Number Systems (Introduction) (F/S) ≈ 13 min.
- Conversion From Decimal To Binary (F/S) ≈ 9 min.
- Conversion Algorithm From Decimal To Binary (F/S) ≈ 11 min.
- Conversion Algorithm From Decimal To Any Base (F/s) ≈ 9 min.
- Binary / Hex To Decimal (F/S) ≈ 11 min.
- More Complexity And Matrix Multiplication ≈ 44 min.

Below is a list of lecture notes for this week:

- Algorithms Lecture Notes
- Complexity Lecture Notes
- Old School Sorting Lecture Notes

5.0.4 Assignments

The assignment for this week is:

- Assignment 5 - Algorithms And Number Properties

5.0.5 Quiz

The quiz's for this week can be found at:

- Quiz 5 - Algorithms And Matrices

5.0.6 Chapter Summary

The first section that we are covering this week is **Section 2.6 - Matrices**

Section 2.6 - Matrices

Overview

Matrices are fundamental mathematical structures used to represent and manipulate data in various fields, including mathematics, physics, computer science, engineering, and more. A matrix is essentially a rectangular grid of numbers, symbols, or expressions arranged in rows and columns. It is a versatile tool for solving systems of linear equations, performing transformations, and analyzing data.

Matrices have several key components and properties:

1. **Rows and Columns:** A matrix is defined by its dimensions, which specify the number of rows and columns it contains. For example, a matrix with three rows and two columns is referred to as a "3x2 matrix."

2. **Elements:** The entries within a matrix are called elements. These elements can be real numbers, complex numbers, or even variables and symbols.

3. **Equality:** Two matrices are considered equal if they have the same dimensions and if their corresponding elements are equal. This forms the basis for matrix algebra.

4. **Addition and Subtraction:** Matrices of the same dimensions can be added or subtracted by adding or subtracting their corresponding elements. This operation is done element-wise.

5. **Scalar Multiplication:** A matrix can be multiplied by a scalar (a single number) by multiplying all of its elements by that scalar.

6. **Matrix Multiplication:** Matrix multiplication is a more complex operation. It involves multiplying the rows of one matrix by the columns of another matrix and summing the results. Notably, matrix multiplication is not commutative, meaning that the order of multiplication matters.

7. **Identity Matrix:** The identity matrix is a special square matrix with ones on its main diagonal (from the top-left to the bottom-right) and zeros elsewhere. Multiplying any matrix by the identity matrix leaves the matrix unchanged.

8. **Transpose:** The transpose of a matrix is obtained by swapping its rows and columns. This operation is denoted by adding a superscript "T" to the matrix symbol.

9. **Determinant:** The determinant of a square matrix is a scalar value that provides information about the matrix's invertibility and the scale factor of linear transformations it represents.

10. **Inverse:** Not all matrices have inverses, but square matrices that are invertible have an inverse matrix that, when multiplied by the original matrix, yields the identity matrix.

Matrices are used in a wide range of applications, including solving systems of linear equations, representing geometric transformations, analyzing networks, performing data transformations, and much more. They are a fundamental concept in linear algebra, and their versatility makes them a powerful tool for various mathematical and computational tasks.

The next section that we are covering this week is **Section 3.1 - Algorithms**.

Section 3.1 - Algorithms

Overview

Algorithms are step-by-step procedures or sets of rules for solving specific problems or performing tasks. They are at the heart of computer science and play a pivotal role in various fields, including mathematics, engineering, data science, and artificial intelligence. Algorithms are designed to take inputs, process them through a series of well-defined steps, and produce desired outputs efficiently and accurately.

Key aspects of algorithms include:

1. **Problem Solving:** Algorithms are developed to address specific problems or tasks. They can range from simple tasks like sorting a list of numbers to complex challenges like route optimization in logistics.

2. **Efficiency:** Efficiency is a critical consideration in algorithm design. An efficient algorithm accomplishes its task with minimal resource usage, such as time and memory. Efficient algorithms often have low time complexity, meaning they execute quickly, even for large inputs.

3. **Correctness:** An algorithm must produce correct results for all valid inputs. Rigorous testing and mathematical proofs are used to ensure correctness.

4. **Determinism:** Algorithms are deterministic, meaning that given the same input, they will produce the same output every time.

5. **Termination:** An algorithm must eventually halt and produce an output. Infinite loops or non-terminating algorithms are considered faulty.

6. **Analysis:** Analyzing an algorithm involves evaluating its performance, resource usage, and scalability. This analysis helps compare algorithms and select the most suitable one for a given problem.

7. **Data Structures:** Algorithms often work in conjunction with data structures, which are used to store and organize data efficiently. Common data structures include arrays, linked lists, trees, and graphs.

8. **Recursion:** Some algorithms use recursion, where a function calls itself to solve smaller instances of a problem. Recursion can simplify complex problems but requires careful handling to avoid infinite recursion.

9. **Complexity Theory:** Complexity theory categorizes algorithms based on their resource usage. It classifies algorithms as polynomial or exponential, helping understand their computational limits.

10. **Optimization:** Algorithms can be optimized to improve efficiency or reduce resource consumption. Optimization techniques may involve algorithmic improvements or parallel computing.

11. **Applications:** Algorithms find applications in various domains, including computer graphics, cryptography, machine learning, data analysis, and network routing.

12. **Algorithmic Paradigms:** Different algorithmic paradigms, such as divide and conquer, dynamic programming, and greedy algorithms, offer systematic approaches to problem-solving.

13. **Algorithm Libraries:** Libraries and frameworks, like those in programming languages, provide pre-implemented algorithms, allowing developers to leverage well-tested solutions.

Algorithms are the foundation of modern computing and enable the automation of tasks, from simple calculations to complex decision-making processes. Their study and development are central to advancing technology and solving real-world challenges.

The next section that we are covering this week is **Section 3.2 - The Growth of Functions**.

Section 3.2 - The Growth of Functions

Overview

Algorithmic complexity refers to the analysis of how an algorithm's performance scales concerning the size of its input data. It provides insights into how efficiently an algorithm operates and how its runtime or resource consumption grows as the input size increases. Big O notation is a mathematical notation used to express algorithmic complexity in a simplified and standardized way. It allows us to categorize algorithms based on their upper bounds regarding time and space requirements.

Key points about algorithmic complexity and Big O notation include:

1. **Time Complexity:** Time complexity measures the number of basic operations (such as comparisons or assignments) performed by an algorithm concerning the input size. It quantifies how the runtime of an algorithm increases as the input grows. Algorithms are classified into categories like constant time ($O(1)$), linear time ($O(n)$), quadratic time ($O(n^2)$), and more complex complexities ($O(n \log n)$, $O(2^n)$) based on their behavior.

2. **Space Complexity:** Space complexity evaluates the memory usage of an algorithm in relation to the input size. It describes how additional memory requirements grow as the input size increases. Similar to time complexity, algorithms can be categorized as using constant space ($O(1)$), linear space ($O(n)$), or more complex space complexities.

3. **Big O Notation (O):** Big O notation provides an upper bound on the worst-case time or space complexity of an algorithm. It describes how an algorithm's performance scales asymptotically (for very large inputs). For example, if an algorithm has a time complexity of $O(n^2)$, it means that its runtime will not grow faster than a quadratic function of the input size.

4. **Best, Average, and Worst Cases:** Algorithms may have different complexities depending on the input data's characteristics. The best-case complexity represents the most efficient scenario, while the worst-case complexity describes the least efficient scenario. The average-case complexity provides an average estimate of performance over all possible inputs.

5. **Comparing Algorithms:** Big O notation is invaluable for comparing and selecting algorithms for specific tasks. When faced with multiple algorithms to solve the same problem, choosing the one with the lower Big O complexity often leads to better overall performance.

6. **Trade-offs:** Optimizing one aspect of an algorithm's complexity may lead to trade-offs in other areas. For example, reducing time complexity may increase space complexity. Analyzing these trade-offs is essential for algorithm design.

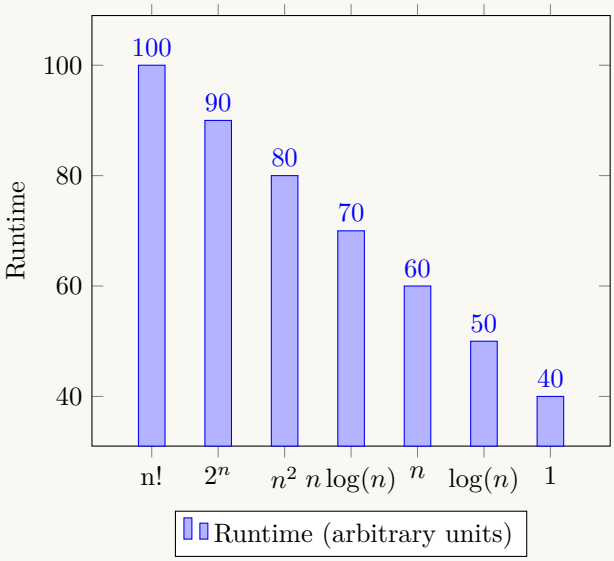
7. **Practical Considerations:** While Big O notation is a useful tool, it simplifies the analysis. Practical considerations, like constant factors and hidden constants, may impact real-world performance. Profiling and benchmarking are essential for fine-tuning algorithms.

8. **Complexity Classes:** Complexity classes categorize problems based on their inherent computational difficulty. Classes like P (problems solvable in polynomial time) and NP (non-deterministic polynomial time) are central to computer science and the theory of computation.

Algorithmic complexity and Big O notation provide a structured approach to evaluating and comparing algorithms' efficiency and scalability. They are essential tools for computer scientists, software engineers, and developers seeking to design and implement high-performance algorithms for a wide range of applications.

Runtime Examples

Below is an image that depicts examples of runtime operations in algorithms.



The graph illustrates the runtime comparison of various functions with different algorithmic complexities. Each function is represented on the x-axis, including $n!$, 2^n, n^2, $n\log(n)$, n, $\log(n)$, and 1. On the y-axis, the runtime is depicted in arbitrary units. The graph uses a ybar plot, and each bar corresponds to one of the functions, showing how their runtimes compare. As expected, $n!$ (factorial) has the highest runtime, followed by 2^n and n^2. Conversely, constant-time complexity (1) has the lowest runtime, and other functions, like $n\log(n)$ and n, fall in between. This graph provides a visual representation of how different algorithmic complexities impact runtime.

Below is a table that summarizes commonly used terminology for the complexity of algorithms.

Complexity	Terminology
$\Theta(1)$	Constant Complexity
$\Theta(\log(n))$	Logarithmic Complexity
$\Theta(n)$	Linear Complexity
$\Theta(n\log(n))$	Linearithmic Complexity
$\Theta(n^b)$	Polynomial Complexity
$\Theta(b^n)$, where $b > 1$	Exponential Complexity
$\Theta(n!)$	Factorial Complexity

The last section that we will cover this week is **Section 4.2 - Integer Representation And Algorithms**

Section 4.2 - Integer Representation And Algorithms

Overview

In mathematics and computer science, various number bases are used to represent and work with numbers, each with its unique properties and applications. Here, we'll explore four common bases: decimal, hexadecimal, octal, and binary.

- **Decimal (Base 10)** - Decimal is the base most commonly used by humans. It's a base-10 system, meaning it has ten digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. In a decimal number, each digit's position signifies a power of 10. For example, in the number 365, the digit 5 is in the one's place, 6 is in the ten's place, and 3 is in the hundred's place.

- **Hexadecimal (Base 16)** - Hexadecimal, often used in computing, is a base-16 system. It uses sixteen symbols: 0-9 for values 0 to 9 and A-F for values 10 to 15. Hexadecimal is compact and frequently used to represent binary data in a more human-readable format. For example, the decimal number 255 is represented as FF in hexadecimal.

- **Octal (Base 8)** - Octal is a base-8 system, employing eight symbols: 0-7. It's less common today but was widely used in early computing. Octal numbers are sometimes used in Unix file permissions, where each digit represents three bits of data.
- **Binary Base (Base 2)** - Binary is the fundamental base for computers, using only two symbols: 0 and 1. It's the language of digital circuits, where each bit represents an on/off state. Computers store and manipulate data in binary form, making it critical in computer science.

Each of these bases serves specific purposes in different fields, with decimal being the most familiar to people in everyday life, while hexadecimal, octal, and binary are more commonly used in computer science and digital technology. Understanding these bases is essential for working with various types of data and programming languages.

Exam 1

Exam 1

6.0.1 Piazza

Must post / respond to at least **two** Piazza posts this week.

6.0.2 Lectures

The lectures for this week and their links can be found below:

- Jupyter Lab Tutorial ≈ 8 min.
- Divisibility Card Trick ≈ 2 min.
- Secrets To The Amazing Trick ≈ 12 min.

Below is a list of lecture notes for this week:

- Exam 1 Charts Lecture Notes
- Jupyter Lab Resources
- Markdown Cheat Sheet

Number Theory And Primes

Number Theory And Primes

7.0.1 Assigned Reading

The reading assignments for this week is from, :

- Chapter 4.1 - Divisibility And Modular Arithmetic
- Chapter 4.2 - Integer Representation And Algorithms
- Chapter 4.3 - Primes And Greatest Common Divisors

7.0.2 Piazza

Must post / respond to at least **two** Piazza posts this week.

7.0.3 Lectures

The lectures for this week and their links can be found below:

- Modular Arithmetic ≈ 5 min.
- Congruences ≈ 5 min.
- Modular Operator vs Congruence ≈ 4 min.
- Arithmetic and Modular Operator ≈ 10 min.
- Congruences Vs. Equality ≈ 7 min.
- Congruences Vs. Equality (Differences) ≈ 4 min.
- Prime Number And Fundamental Theorem Of Arithmetic (Recap) ≈ 13 min.
- Facts About Primes ≈ 14 min.
- Tests For Primality ≈ 9 min.
- GCDs Basic Definition ≈ 8 min.
- GCDs, LCMs and Prime Factorization ≈ 14 min.
- Euclid's Algorithm for GCD ≈ 13 min.
- Euclid's Algorithm [Continued] ≈ 7 min.
- Euclid's Algorithm With Jugs ≈ 7 min.
- Square And Mod - A Simple Introduction To Fast ModularExponentiation ≈ 10 min.
- Fast Modular Exponentiation ≈ 15 min.

7.0.4 Assignments

The assignment for this week is:

- Assignment 6 - Number Theory And Primes

7.0.5 Quiz

The quiz's for this week can be found at:

- Quiz 6 - Number Theory And Primes

7.0.6 Chapter Summary

The first section that we are covering this week is **Section 4.1 - Divisibility And Modular Arithmetic**.

Section 4.1 - Divisibility And Modular Arithmetic

Overview

Divisibility and modular arithmetic are fundamental concepts in number theory, a branch of mathematics that explores the properties and relationships of integers. These concepts lay the foundation for understanding how numbers can be divided and organized into congruence classes, with wide-ranging applications in mathematics and various other fields. Here, we delve into the core principles of divisibility and modular arithmetic and explore their significance in number theory and practical applications.

Key Concepts

1. **Definition of Divisibility:** Divisibility is a fundamental concept in number theory that explores the relationships between integers concerning their ability to be divided by one another without leaving a remainder. In mathematical terms, we say that an integer a is divisible by another integer b, denoted as $b|a$, when there exists an integer q such that $a = bq$. This concept is fundamental because it lays the groundwork for understanding how integers are interconnected in terms of multiplication and division. If no such integer q exists, then b is not a divisor of a, and we denote this as $b \nmid a$.

2. **Properties of Divisibility:** Divisibility exhibits several important properties. For instance, if an integer a is divisible by both b and c, it is also divisible by their greatest common divisor (GCD), represented as $\gcd(b, c)$. Furthermore, if a is divisible by b and b is divisible by c, then a is also divisible by c. Additionally, the concept of divisibility plays a key role in understanding the least common multiple (LCM) of integers, denoted as $\text{lcm}(b, c)$.

3. **Prime Factorization:** Every positive integer can be expressed uniquely as a product of prime numbers, known as its prime factorization. This concept is at the heart of divisibility since it allows us to determine the divisors of a number efficiently. By decomposing a number into its prime factors, we gain valuable insights into its divisibility properties, making prime factorization a crucial tool in number theory.

4. **Modular Arithmetic:** Modular arithmetic is a specialized field of number theory that deals with the arithmetic of remainders. It introduces the concept of congruence, where two numbers share the same remainder when divided by a fixed integer m. Modular arithmetic, denoted as $a \equiv b \pmod{m}$, is an essential aspect of number theory and has widespread applications in various domains.

5. **Congruence Classes:** In modular arithmetic, numbers are organized into congruence classes, often represented by $\pmod{m}$, where m represents the modulus. Each congruence class contains integers that share the same remainder when divided by m. For example, the congruence class $\pmod{5}$ includes numbers like $0, 5, 10, -5, -10$, and so on. These classes are fundamental in modular arithmetic and help simplify computations.

6. **Modular Operations:** Modular arithmetic defines operations like addition, subtraction, multiplication, and exponentiation modulo m. These operations follow specific rules and properties, making it possible to perform calculations within congruence classes. Modular arithmetic's systematic approach to these operations is essential for various applications, especially in computer science, cryptography, and coding theory.

7. **Applications:** Modular arithmetic finds applications in a wide range of fields, including computer science, cryptography, coding theory, and number theory. It is used in algorithms and protocols for data encryption, error detection and correction, and generating pseudorandom numbers. Its practical utility extends to computer systems, where it helps manage memory allocation and addressing.

8. **Fermat's Little Theorem and Euler's Totient Function:** These two key results rely on modular arithmetic. Fermat's Little Theorem states that if p is a prime number and a is an integer not divisible by p, then $a^{p-1} \equiv 1 \pmod{p}$. This theorem has significant implications in cryptography and number theory. Euler's Totient function, denoted as $\phi(n)$, is employed to calculate the number of positive integers less than n that are coprime to n. It is fundamental in solving problems related to modular arithmetic and number theory.

Divisibility and modular arithmetic are foundational concepts in number theory, intertwined in their importance and application. Divisibility forms the basis for understanding the relationships between integers, while modular arithmetic offers a versatile framework for working with remainders and congruences. Together, they provide the foundation for tackling complex mathematical problems and finding solutions in various fields, from cryptography to computer science. These concepts are not only essential for theoretical mathematics but also have practical implications in real-world problem-solving.

The next section that we are covering this week is **Section 4.2 - Integer Representations And Algorithms**.

Section 4.2 - Integer Representations And Algorithms

Overview

Integer representations and algorithms are fundamental topics in computer science and mathematics, serving as the backbone for various computational tasks. Understanding how integers are represented in binary form and knowing how to perform arithmetic operations on them efficiently is crucial in computer systems, cryptography, and many other areas. This summary explores the key concepts related to integer representations and algorithms, shedding light on their significance and real-world applications.

Key Concepts

1. **Binary Representation:** Integers are often represented in binary form, which uses only two digits: 0 and 1. In this representation, each digit's position has a weight corresponding to a power of 2, allowing for efficient conversions and bitwise operations.

2. **Two's Complement:** Two's complement is a widely-used representation for signed integers. It allows both positive and negative numbers to be represented using a fixed number of bits. Negative numbers are obtained by taking the complement (flipping the bits) of the corresponding positive number and adding 1.

3. **Overflow:** Integer overflow occurs when the result of an arithmetic operation exceeds the maximum representable value for a given number of bits. Handling overflow is critical to ensure accurate computations in computer programs.

4. **Bitwise Operations:** Bitwise operations (AND, OR, XOR, NOT, shifts) provide a way to manipulate individual bits in integers. They are used in various algorithms, including those for data compression, encryption, and optimization.

5. **Addition and Subtraction:** Algorithms for addition and subtraction involve working with binary representations, carry bits, and borrowing. Efficient methods, such as binary addition trees and two's complement subtraction, are used in hardware and software implementations.

6. **Multiplication and Division:** Multiplication and division of integers require more complex algorithms, such as long multiplication and long division. Optimized techniques like Karatsuba multiplication and Newton-Raphson division enhance efficiency.

7. **Modular Arithmetic:** Modular arithmetic is used to perform arithmetic operations within a specified modulus. It finds applications in cryptography (RSA encryption), hashing, and pseudorandom number generation.

8. **Euclidean Algorithm:** The Euclidean algorithm computes the greatest common divisor (GCD) of two integers efficiently. It plays a crucial role in simplifying fractions, solving linear Diophantine equations, and ensuring data integrity.

9. **Number Bases:** Understanding different number bases, such as hexadecimal and octal, is important for programmers and engineers. These bases are used in low-level programming, memory addressing, and data representation.

10. **Floating-Point Representation:** Floating-point representation is used to represent real numbers with a fractional part. It involves a sign bit, exponent, and mantissa and is fundamental in scientific and engineering computing.

11. **Applications:** Integer representations and algorithms find applications in various fields, including computer graphics, cryptography, database systems, network protocols, and scientific simulations. Efficient algorithms are essential for optimizing software and hardware performance.

In conclusion, integer representations and algorithms are the building blocks of computer science and mathematics. They underlie the fundamental operations performed by computers and are essential for designing efficient algorithms and data structures. Mastery of these concepts empowers professionals in fields ranging from software engineering to cybersecurity, enabling them to solve complex problems and develop robust systems. An understanding of integer representations and algorithms is invaluable in the digital age, where computational efficiency and accuracy are paramount.

The last section that we will be covering this week is **Section 4.3 - Primes and Greatest Common Divisors**.

Section 4.3 - Primes and Greatest Common Divisors

Overview

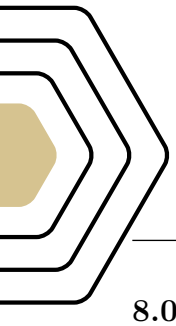
Primes and greatest common divisors (GCD) are fundamental concepts in number theory with wide-ranging applications in various mathematical and computational domains. Prime numbers, as the building blocks of integers, play a crucial role in encryption, data compression, and random number generation. GCD, on the other hand, is a foundational tool for solving Diophantine equations, simplifying fractions, and ensuring data integrity. This summary explores the key concepts related to primes and GCD, highlighting their importance and real-world implications.

Key Concepts

1. **Prime Numbers:** Prime numbers are natural numbers greater than 1 that have no divisors other than 1 and themselves. They play a fundamental role in number theory and cryptography.
2. **Composite Numbers:** Composite numbers are integers greater than 1 that are not prime, meaning they have divisors other than 1 and themselves.
3. **Prime Factorization:** Prime factorization is the process of expressing a composite number as a product of prime numbers. It is the basis for understanding the internal structure of integers.
4. **Sieve of Eratosthenes:** The Sieve of Eratosthenes is an algorithm for efficiently finding all prime numbers up to a given limit. It eliminates multiples of each prime as it iterates through the integers.
5. **Greatest Common Divisor (GCD):** The GCD of two integers is the largest positive integer that divides both of them without a remainder. It is denoted as $\text{GCD}(a, b)$ or $\gcd(a, b)$.
6. **Euclidean Algorithm:** The Euclidean algorithm is an efficient method for finding the GCD of two integers. It involves repeated division with remainder and is a fundamental tool in number theory and cryptography.
7. **Diophantine Equations:** Diophantine equations are equations where solutions are sought in integers. The GCD and the Euclidean algorithm are used to solve linear Diophantine equations.
8. **Relatively Prime Numbers:** Two integers are relatively prime if their GCD is 1. Relatively prime numbers have no common divisors other than 1.
9. **Applications:** Prime numbers and GCD have practical applications in cryptography (RSA encryption), random number generation, error detection and correction, and hashing algorithms. They are essential tools in ensuring data security and integrity.

Primes and greatest common divisors are foundational concepts in number theory with a significant impact on mathematics, computer science, and cryptography. Prime numbers are the basis for secure encryption, while the GCD is a versatile tool for solving equations and ensuring data accuracy. Understanding these concepts is essential for professionals in fields such as cybersecurity, algorithm design, and numerical analysis. The study of primes and GCD continues to be a fertile ground for mathematical research and innovation, with implications for both theoretical and practical aspects of computation and data security.

Cryptography



Cryptography

8.0.1 Assigned Reading

The reading assignments for this week is from, :

- Chapter 4.3 - Primes And Greatest Common Divisors
- Chapter 4.4 - Solving Congruences
- Chapter 4.6 - Cryptography

8.0.2 Piazza

Must post / respond to at least **two** Piazza posts this week.

8.0.3 Lectures

The lectures for this week and their links can be found below:

- Bezout Theorem ≈ 10 min.
- Proof of Bezout Theorem ≈ 16 min.
- Relatively Prime Numbers ≈ 4 min.
- Properties of Relatively Prime Numbers ≈ 8 min.
- Prime Vs. Relatively Prime ≈ 9 min.
- Extended Euclid's Algorithm ≈ 13 min.
- Modular Inverse ≈ 10 min.
- Modular Congruences ≈ 9 min.
- Simultaneous Congruences ≈ 11 min.
- Two Simultaneous Congruences And Chinese Remainder Theorem (Part I) ≈ 8 min.
- Two Simultaneous Congruences And Chinese Remainder Theorem (Part II) ≈ 9 min.
- Chinese Remainder Theorem (Multiple Congruences) ≈ 8 min.
- IPython Notebook For Chinese Remainder Theorem
- Proof of Fermat's Little Theorem ≈ 14 min.
- Fermat Little Theorem ≈ 4 min.
- Intro to Cryptography ≈ 18 min.
- Public Key Encryption and RSA ≈ 35 min.

Below is a list of lecture notes for this week:

- Public Key Encryption And RSA Lecture Notes
- Intro To Cryptography Lecture Notes

8.0.4 Assignments

The assignment for this week is:

- Assignment 7 - Cryptography

8.0.5 Quiz

The quiz's for this week can be found at:

- Quiz 7 - Cryptography

8.0.6 Chapter Summary

The first section that we are covering this week is **Section 4.3 - Primes And Greatest Common Divisors**.

Section 4.3 - Primes And Greatest Common Divisors

Prime Numbers

Prime numbers are fundamental in number theory, known for their unique properties. They are the building blocks of integers, as every positive integer greater than 1 can be expressed uniquely as a product of prime factors. This property is known as the Fundamental Theorem of Arithmetic.

Prime numbers have been a subject of fascination for mathematicians for centuries. They are extensively studied, and open problems like the distribution of prime numbers, the Twin Prime Conjecture, and the Riemann Hypothesis are closely related to primes.

In computer science, prime numbers find applications in various algorithms, such as those for hashing, searching, and cryptography. For example, hashing functions often rely on prime numbers to distribute data uniformly.

Greatest Common Divisor (GCD)

The GCD is a fundamental concept that extends beyond just two numbers. It can be used to find the GCD of multiple integers, which is particularly important in number theory and cryptography.

The Extended Euclidean Algorithm is a powerful tool for finding the GCD and Bézout coefficients (integers s and t) that satisfy the equation ax + by = GCD(a, b). This algorithm plays a crucial role in solving Diophantine equations, which have applications in cryptography and coding theory.

Modular arithmetic, a branch of number theory, heavily relies on GCD calculations. It helps solve problems involving remainders and congruences and is used in computer science for tasks like data validation and error detection.

Primes and GCD are foundational concepts with wide-reaching applications in mathematics, computer science, cryptography, and data integrity. Their study continues to inspire mathematicians and computer scientists alike, contributing to advancements in algorithms, encryption, and number theory. Understanding these concepts is essential for solving complex problems in these domains and ensuring the security and reliability of digital systems.

Euclidean Algorithm

The Euclidean Algorithm is a fundamental concept in number theory and mathematics, known for its simplicity and utility in finding the greatest common divisor (GCD) of two integers. Named after the ancient Greek mathematician Euclid, this algorithm has been used for over two millennia and remains a cornerstone of modern mathematics. The primary goal of the Euclidean Algorithm is to efficiently determine the largest positive integer that divides two given integers without leaving a remainder.

Algorithm Overview

The Euclidean Algorithm operates on the principle of repeated subtraction. Given two positive integers, a (the larger number) and b (the smaller number), it repeatedly subtracts b from a until a becomes smaller than b. At this point, the algorithm swaps the values of a and b, ensuring that a is always the larger number. The process continues until b becomes zero. The value of a at this stage represents the GCD of the original two integers.

Mathematically, the steps of the Euclidean Algorithm can be summarized as follows:

1. Start with two positive integers, a and b.

2. Divide a by b and find the remainder, denoted as r.
3. Replace a with b and b with r.
4. Repeat steps 2 and 3 until b becomes zero.
5. The value of a at this point is the GCD of the original two integers.

Key Properties and Applications

The Euclidean Algorithm possesses several key properties and applications:

1. **Efficiency:** The algorithm's efficiency is one of its most notable features. It operates in a time complexity that is proportional to the number of bits required to represent the numbers a and b. This makes it highly efficient even for large integers.
2. **GCD Computation:** The primary purpose of the Euclidean Algorithm is to calculate the GCD of two integers. It is invaluable in simplifying fractions and solving Diophantine equations (equations with integer solutions), both of which have widespread applications in various fields, including number theory, cryptography, and computer science.
3. **Extended Euclidean Algorithm:** The Extended Euclidean Algorithm is an extension of the basic algorithm that not only calculates the GCD but also finds the Bézout coefficients, integers s and t, such that $as + bt = \text{GCD}(a, b)$. This extended version plays a crucial role in solving linear Diophantine equations and has applications in cryptography and coding theory.
4. **Modular Arithmetic:** The Euclidean Algorithm is frequently employed in modular arithmetic, where it is used to find modular inverses and solve congruences. It is a foundational tool in the field of number theory and has applications in cryptography, error detection, and data validation.

The Euclidean Algorithm is a timeless mathematical tool that continues to find applications in various domains, including number theory, computer science, and cryptography. Its elegance and efficiency make it an essential component of algorithms used in digital systems, ensuring the secure and efficient functioning of modern technology. Whether simplifying fractions, solving Diophantine equations, or securing digital communications, the Euclidean Algorithm remains a cornerstone of mathematical problem-solving. Understanding its principles and applications is essential for anyone working with numbers and computation.

Bézout Coefficients: Understanding the Fundamental Relationship in Number Theory

In the realm of number theory, Bézout coefficients hold a pivotal role as they provide a profound understanding of the greatest common divisor (GCD) of two integers. Named after the French mathematician Étienne Bézout, these coefficients offer insights into the linear combination of two integers that results in their GCD. This concept has diverse applications, from solving Diophantine equations to cryptography and modular arithmetic, making it a cornerstone in mathematics.

The Basics of Bézout Coefficients

At its core, Bézout's identity states that for any two integers, a and b, there exist integers s and t such that $as + bt = \text{GCD}(a, b)$. In other words, Bézout coefficients s and t represent the linear combination of a and b that yields their GCD. This fundamental relationship is the basis for the Extended Euclidean Algorithm, a powerful tool for calculating GCDs and finding modular inverses.

Applications in Number Theory

1. **Diophantine Equations**: Diophantine equations are equations with integer solutions, and they arise in various mathematical contexts. Bézout coefficients play a crucial role in solving linear Diophantine equations of the form $ax + by = c$, where a, b, and c are integers. These coefficients provide a systematic method for finding solutions to such equations.
2. **Modular Arithmetic**: In modular arithmetic, Bézout coefficients are used to find modular inverses. For a given integer a and modulus m, the modular inverse a^{-1} exists if and only if $\text{GCD}(a, m) = 1$. Bézout coefficients s and t allow us to express this modular inverse as $a^{-1} \equiv s \mod m$.
3. **Cryptography**: Bézout coefficients have applications in cryptographic algorithms, especially in RSA encryption. They are used to compute the private key from the public key, ensuring secure communication and data protection.

Bézout coefficients are a fundamental concept in number theory with far-reaching implications in mathematics and its applications. They enable us to understand the underlying structure of GCDs, solve Diophantine equations, work with modular arithmetic, and strengthen the foundations of cryptography. The elegance and versatility of Bézout's identity continue to influence various fields, making it a topic of enduring importance in mathematics.

The next section that we are covering this week is **Section 4.4 - Solving Congruences**.

Section 4.4 - Solving Congruences

Overview

Modular arithmetic, a branch of number theory, deals with the fascinating world of congruences—equations that express the concept of numbers having the same remainder when divided by a fixed integer, known as the modulus. Solving congruences involves finding solutions that satisfy these modular equations, and it has applications in various fields, from cryptography to algebraic structures.

The Essence of Congruences

At the core of solving congruences is the notion of congruence relation, denoted as $a \equiv b \mod m$, which signifies that integers a and b leave the same remainder when divided by m. This relation divides the integers into congruence classes, paving the way for understanding patterns and solving modular equations.

Methods for Solving Congruences

1. **Direct Computation**: For simple congruences, direct computation involves exploring possible values within the congruence class to identify the solutions. This method is practical for small moduli and linear equations.

2. **Modular Arithmetic Rules**: Leveraging modular arithmetic rules, such as addition, subtraction, multiplication, and exponentiation, simplifies solving more complex congruences. These rules allow for step-by-step manipulation to arrive at solutions.

3. **Chinese Remainder Theorem (CRT)**: The CRT is a powerful tool for solving systems of simultaneous congruences. It breaks down a complex modular problem into simpler subproblems, which can be solved individually, and then combines their solutions to obtain the final solution.

4. **Extended Euclidean Algorithm**: When working with modular inverses, the Extended Euclidean Algorithm plays a vital role. It helps find the multiplicative inverse of an integer modulo m, enabling the solution of congruences involving division.

Applications in Mathematics and Beyond

Solving congruences has broad applications in mathematics and beyond:

- **Cryptography**: In modern cryptography, congruences are employed to secure communications and data encryption. Algorithms like RSA rely on the difficulty of factoring large composite numbers, which involves solving congruences.

- **Algebraic Structures**: Congruences are used to explore algebraic structures, particularly in the study of rings, fields, and group theory. They provide insights into the properties and behaviors of mathematical objects.

- **Number Theory**: Congruences are deeply intertwined with number theory, facilitating the exploration of properties related to prime numbers, divisibility, and modular arithmetic.

Solving congruences is a fundamental and versatile skill in mathematics, offering a structured approach to handling modular equations. From number theory to cryptography and algebraic structures, the ability to decipher congruences plays a crucial role in understanding mathematical concepts and solving real-world problems.

Fermat's Little Theorem

Fermat's Little Theorem is a captivating result in number theory, named after the French mathematician Pierre de Fermat, who first stated it in the 17th century. This theorem offers profound insights into the behavior of remainders when raising integers to a power, and it has significant applications in various mathematical and computational domains.

The Essence of Fermat's Little Theorem

The theorem is expressed as follows: For any prime number p and an integer a not divisible by p, a^{p-1} is congruent to 1 modulo p, represented as $a^{p-1} \equiv 1 \mod p$. In simpler terms, when you raise an integer a to the power $p-1$, and then divide the result by the prime p, the remainder is always 1.

Illustrating the Power of Primality

Fermat's Little Theorem shines a spotlight on the concept of primality. It helps identify prime numbers, as they are the only integers for which the theorem holds true. When applied to a composite number, the result can be anything other than 1, making it a handy tool for testing primality.

Applications in Number Theory and Cryptography

1. **Primality Testing**: Fermat's Little Theorem serves as a basis for primality tests, including the Fermat primality test. By verifying whether the theorem holds for a given number, one can assess its primality with high confidence.

2. **RSA Encryption**: In the field of cryptography, the theorem plays a pivotal role in the RSA encryption algorithm. It forms the foundation for generating secure public and private keys, ensuring the confidentiality of data in modern secure communications.

3. **Group Theory**: The theorem has deep connections to group theory, where it reveals the properties of groups formed by elements and powers in modular arithmetic.

Generalizations and Extensions

Fermat's Little Theorem has several generalizations and extensions, including Euler's Totient Theorem and Carmichael's Theorem, which provide further insights into modular arithmetic and prime-powered behavior.

Fermat's Little Theorem stands as a testament to the elegance and power of number theory. Its simple yet profound statement continues to impact various branches of mathematics and serves as a cornerstone in the development of secure cryptographic systems.

The next section that we are covering this week is **Section 4.6 - Cryptography**.

Section 4.6 - Cryptography

Overview

Cryptography is the science and practice of encoding and decoding information to protect its confidentiality, integrity, and authenticity. It plays a pivotal role in ensuring the security and privacy of digital communications, transactions, and data storage in the modern information age.

The Pillars of Cryptography

1. **Confidentiality**: Cryptography ensures that unauthorized individuals cannot access or understand sensitive information. Encryption is a fundamental technique used to transform plaintext into ciphertext, which can only be deciphered by those with the appropriate decryption key.

2. **Integrity**: Cryptographic mechanisms are employed to detect any unauthorized modifications or tampering of data during transmission or storage. Hash functions and digital signatures are examples of tools used to verify data integrity.

3. **Authentication**: Cryptography provides methods for verifying the identities of individuals, devices, or entities in a digital interaction. Public key infrastructure (PKI) and digital certificates are commonly used to establish trust in online transactions.

Key Concepts in Cryptography

1. **Symmetric and Asymmetric Cryptography**: Cryptographic algorithms are categorized as either symmetric (using a single key for both encryption and decryption) or asymmetric (utilizing a pair of public and private keys). Symmetric cryptography is efficient for data encryption, while asymmetric cryptography is essential for secure key exchange and digital signatures.

2. **Cryptanalysis**: This is the art of breaking cryptographic systems, often through mathematical analysis and computational techniques. Cryptanalysts aim to discover vulnerabilities and weaknesses in encryption algorithms.

3. **Cryptographic Protocols**: These are standardized procedures and rules for secure communication. Protocols like SSL/TLS (for secure web browsing) and IPsec (for secure network communication) are widely used to protect data during transmission.

4. **Quantum Cryptography**: With the advent of quantum computing, the field of cryptography faces new challenges. Quantum cryptography harnesses the principles of quantum mechanics to create theoretically unbreakable encryption schemes.

Applications of Cryptography

1. **Secure Communication**: Cryptography safeguards email, instant messaging, voice calls, and other digital communications from eavesdropping and interception.

2. **E-commerce and Online Banking**: It underpins secure online transactions, enabling consumers to shop online and perform banking operations with confidence.

3. **Government and Military Use**: Cryptography is crucial for protecting classified information, national security, and military communications.

4. **Data Protection**: Cryptographic techniques are applied to secure sensitive data in databases, cloud storage, and backups.

5. **Blockchain Technology**: Cryptography is central to blockchain, ensuring the integrity and immutability of data in decentralized ledgers like Bitcoin.

Challenges and Evolving Landscape

Cryptography is a dynamic field that continually adapts to emerging threats and technologies. Quantum computing, the rise of cyberattacks, and the need for privacy-preserving algorithms are among the ongoing challenges in the world of cryptography.

Cryptography is the guardian of digital trust, enabling secure and private interactions in an interconnected world. Its innovations and advancements are instrumental in safeguarding sensitive information and shaping the future of cybersecurity.

RSA

RSA, named after its inventors Ron Rivest, Adi Shamir, and Leonard Adleman, is a widely used asymmetric encryption algorithm that plays a crucial role in securing digital communications, data protection, and authentication.

Key Concepts in RSA

1. **Asymmetric Cryptography**: RSA uses a pair of keys—a public key for encryption and a private key for decryption. The security of RSA relies on the mathematical difficulty of factoring large composite numbers.

2. **Key Generation**: To set up RSA encryption, a user generates a public-private key pair. The public key is shared with anyone, while the private key is kept confidential.

RSA Encryption Process

1. **Key Exchange**: In a typical scenario, Alice wants to send an encrypted message to Bob. Bob shares his public key (containing the modulus n and encryption exponent e) with Alice.
2. **Message Encryption**: Alice takes Bob's public key and uses it to encrypt her plaintext message M. She raises M to the power of e modulo n, resulting in ciphertext C:
$$C \equiv M^e \pmod{n}$$
3. **Sending Ciphertext**: Alice sends the ciphertext C to Bob.

RSA Decryption Process

1. **Message Retrieval**: Bob receives the ciphertext C.
2. **Decryption**: Using his private key (containing the modulus n and decryption exponent d), Bob decrypts C to obtain the original message M:
$$M \equiv C^d \pmod{n}$$
3. **Message Verification**: Bob now has access to the plaintext message M.

Security and Applications

RSA's security is based on the difficulty of factoring the product of two large prime numbers. As such, it is considered highly secure when key lengths are sufficiently large. RSA is widely used in secure email, online banking, digital signatures, and secure communication protocols like SSL/TLS.

Key Management

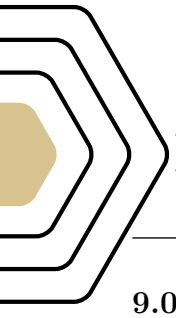
Key management is a crucial aspect of RSA. Key pairs must be generated securely, and private keys must be safeguarded to prevent unauthorized access. Periodic key rotation and secure key storage practices are essential.

Quantum Computing and RSA

The advent of quantum computing poses a potential threat to RSA. Quantum computers may efficiently factor large numbers, rendering RSA encryption insecure. Post-quantum cryptography research is actively exploring alternative encryption algorithms to withstand quantum attacks.

RSA encryption and decryption provide a robust framework for secure communication and data protection in the digital world. Understanding the principles behind RSA is essential for implementing secure information exchange and safeguarding sensitive data.

RSA Project



RSA Project

9.0.1 Piazza

Must post / respond to at least **two** Piazza posts this week.

9.0.2 Lectures

The lectures for this week and their links can be found below:

- Jupyter Lab Resources ≈ 8 min.
- Time Complexity ≈ 50 min.
- Best Practices For Main Function ≈ 11 min.

Below is a list of lecture notes for this week:

- Markdown Cheatsheet
- Simple Comment Guide
- Jupyter Lab Resources

9.0.3 Assignments

The assignment for this week is:

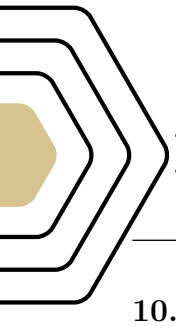
- RSA Project

9.0.4 Quiz

The quiz's for this week can be found at:

- Quiz 8 - RSA Project

Induction And Recursion



Induction And Recursion

10.0.1 Assigned Reading

The reading assignments for this week is from, :

- Chapter 2.4 - Sequences And Summations
- Chapter 5.1 - Mathematical Induction
- Chapter 5.4 - Recursive Algorithms

10.0.2 Piazza

Must post / respond to at least **two** Piazza posts this week.

10.0.3 Lectures

The lectures for this week and their links can be found below:

- Sequences, Induction, And Recursion ≈ 20 min.
- Visually Sum 1 + 2 + 3 ... = n (n + 1) / 2 ≈ 3 min.
- Sum Of The Odds Makes Squares Numbers? ≈ 2 min.
- Grand Finale - Sum Of The Cubes ≈ 5 min.
- Sequences ≈ 23 min.
- Induction Basics ≈ 25 min.
- Recursion And Memoization ≈ 41 min.
- Binary Search: Correctness ≈ 34 min.

Below is a list of lecture notes for this week:

- Sequences Lecture Notes
- Induction Lecture Notes

10.0.4 Assignments

The assignment for this week is:

- Assignment 8 - Induction

10.0.5 Quiz

The quiz's for this week can be found at:

- Quiz 9 - Induction

10.0.6 Chapter Summary

The first section that we are covering this week is **Section 2.4 - Sequences And Summations**.

Section 2.4 - Sequences And Summations

Overview

Sequences and summations are fundamental concepts in mathematics, particularly in discrete mathematics and calculus. They are used to describe and analyze patterns, series, and the accumulation of values over a range of terms or elements. Here's a summary of sequences and summations:

1. **Sequences**:
   - A sequence is an ordered list of numbers or elements written in a specific order.
   - Each element in a sequence is referred to as a term, typically denoted as a_n or x_n, where n represents the position of the term.
   - Sequences can be finite or infinite, depending on whether they have a finite or infinite number of terms.
   - Common types of sequences include arithmetic sequences, geometric sequences, and recursive sequences.
2. **Arithmetic Sequences**:
   - In an arithmetic sequence, each term is obtained by adding a constant difference, called the common difference (d), to the previous term.
   - The general form of an arithmetic sequence is $a_n = a_1 + (n-1)d$.
3. **Geometric Sequences**:
   - In a geometric sequence, each term is obtained by multiplying the previous term by a constant ratio, called the common ratio (r).
   - The general form of a geometric sequence is $a_n = a_1 \cdot r^{(n-1)}$.
4. **Summations**:
   - A summation represents the sum of a sequence of terms and is denoted by the Greek letter sigma ($\sum$).
   - The index variable, typically i or n, specifies the position of the term being summed.
   - The lower and upper limits of the summation indicate the range of terms to be added.
   - Summations are used to find the total accumulation or sum of values in a sequence.
5. **Common Summation Notations**:
   - The most common summation notation is $\sum_{i=1}^{n} a_i$, which represents the sum of terms a_i from $i = 1$ to n.
   - Summations can also involve functions, such as $\sum_{i=1}^{n} f(i)$, where $f(i)$ is a function of the index variable.
6. **Properties of Summations**:
   - Linearity: $\sum(a_i + b_i) = \sum a_i + \sum b_i$
   - Constant Multiple: $\sum c \cdot a_i = c \cdot \sum a_i$
   - Changing Limits: $\sum_{i=a}^{b} a_i = \sum_{i=1}^{b} a_i - \sum_{i=1}^{a-1} a_i$
7. **Applications**:
   - Sequences and summations are used in various fields, including physics, computer science, and finance, to model and analyze real-world phenomena.
   - They play a crucial role in understanding series, limits, and convergence, as well as in solving mathematical and computational problems.

Sequences describe ordered lists of terms, while summations represent the accumulation of those terms. Understanding these concepts is essential in mathematics and its applications, providing valuable tools for solving problems and analyzing patterns.

The second section that we will be covering this week is **Section 5.1 - Mathematical Induction**.

Section 5.1 - Mathematical Induction

Overview

Mathematical induction is a powerful proof technique used in mathematics to establish the truth of an infinite number of statements, typically involving natural numbers. It is based on two fundamental principles: the base case and the induction step. Here's a summary of mathematical induction:

1. **Base Case:**
   - The first step in a proof by mathematical induction is to establish the truth of the statement for a specific value, often the smallest value, of the parameter under consideration. This initial condition is called the base case.
   - The base case serves as the starting point, demonstrating that the statement holds true for at least one value.
2. **Induction Hypothesis:**
   - After proving the base case, the next step is to assume that the statement is true for an arbitrary but fixed value of the parameter. This assumption is known as the induction hypothesis.
   - The induction hypothesis provides the basis for making conclusions about other values of the parameter.
3. **Induction Step:**
   - The core of mathematical induction is the induction step. It involves demonstrating that if the statement is true for one value (the induction hypothesis), then it must also be true for the next value of the parameter.
   - In other words, the induction step establishes an implication: if the statement holds for any value k, it also holds for $k + 1$.
4. **Generalization:**
   - By repeating the induction step indefinitely, mathematical induction allows us to generalize the statement's truth to an infinite range of values of the parameter.
   - This powerful technique enables mathematicians to prove properties and theorems that hold true for all natural numbers, such as properties of sequences and series.
5. **Examples Of Usage:**
   - Mathematical induction is frequently used to prove statements involving natural numbers, such as properties of sums, products, and divisibility.
   - It is employed in various branches of mathematics, including number theory, combinatorics, and calculus.
   - It is also a fundamental tool for proving the correctness of algorithms and computer programs.
6. **Strong Induction:**
   - While standard mathematical induction assumes that the statement holds for a single value and concludes its truth for the next value, strong induction allows the assumption that the statement holds for all previous values up to the current one.
   - Strong induction provides an even more versatile tool for proving statements.

Mathematical induction is a proof technique used to establish the truth of statements for an infinite range of values, primarily in the context of natural numbers. It relies on the base case, the induction hypothesis, and the induction step to generalize the statement's validity. Mathematical induction is a fundamental and widely applied method in mathematics and computer science.

The last section that we will be covering this week is **Section 5.4 - Recursive Algorithms**.

Section 5.4 - Recursive Algorithms

Overview

Recursive algorithms are a fundamental concept in computer science and mathematics, enabling the solution of complex problems by breaking them down into smaller, self-similar subproblems. These algorithms are implemented using recursive functions, which call themselves, solving smaller instances of the same problem with base cases specifying when to terminate. Recursive algorithms exhibit a tree-like structure, often leading to elegant and concise solutions for problems like computing factorials or Fibonacci numbers. While recursion offers simplicity and readability, it can be memory-intensive for deep recursion. Techniques like tail recursion and dynamic programming can mitigate these issues, making recursive algorithms a versatile tool in problem-solving, especially for inherently recursive problems and data structures like trees and graphs. Understanding recursion and designing effective recursive solutions are essential skills in computer science.

1. **Introduction**:
   - Recursive algorithms are a fundamental concept in computer science and mathematics.
   - They involve solving problems by breaking them down into smaller, self-similar subproblems.
2. **Recursive Function**:
   - A recursive algorithm is implemented using a recursive function, which is a function that calls itself.
   - In each recursive call, the function solves a smaller instance of the same problem.
3. **Base Case**:
   - Every recursive algorithm must have a base case or termination condition.
   - The base case specifies when the recursion should stop, preventing infinite recursion.
4. **Recursive Case**:
   - The recursive case defines how the problem is divided into smaller subproblems and solved recursively.
   - It typically involves calling the function with modified input, moving closer to the base case.
5. **Example - Factorial**:
   - An example of a recursive algorithm is calculating the factorial of a number.
   - Base Case: $n! = 1$ when $n = 0$.
   - Recursive Case: $n! = n \cdot (n-1)!$.
6. **Tree-Like Structure**:
   - Recursive algorithms often have a tree-like structure, where each node represents a function call.
   - The root node is the initial call, and child nodes represent recursive calls.
7. **Recursion vs. Iteration**:
   - Recursive and iterative solutions may exist for many problems.
   - Recursion provides an elegant and concise solution for certain problems but may have higher memory usage due to function call overhead.
8. **Examples**:
   - Common recursive algorithms include computing Fibonacci numbers, traversing trees and graphs, and solving divide-and-conquer problems like merge sort and quicksort.
9. **Tail Recursion**:
   - Tail recursion is a specific form of recursion where the recursive call is the last operation in the function.
   - Some programming languages optimize tail-recursive functions to reduce stack space usage.
10. **Advantages and Disadvantages**:
    - Advantages of recursive algorithms include simplicity, readability, and suitability for inherently recursive problems.

- Disadvantages include potential performance issues with deep recursion and increased memory usage.

11. **Dynamic Programming**:
    - Dynamic programming is a technique that combines recursion with memoization (caching) to optimize recursive algorithms by avoiding redundant calculations.
12. **Recursive Thinking**:
    - Understanding recursion requires thinking in terms of dividing problems into smaller instances and understanding how they relate to each other.

Recursive algorithms are a powerful problem-solving technique that involves solving problems by breaking them into smaller, similar subproblems. They are widely used in computer science and mathematics and can provide elegant solutions to various problems. Understanding recursion and designing effective recursive algorithms are important skills for programmers and mathematicians.

Combinatorics And Binomial

Combinatorics And Binomial

11.0.1 Assigned Reading

The reading assignments for this week is from, :

- Chapter 6.1 - The Basics Of Counting
- Chapter 6.2 - The Pigeonhole Principle
- Chapter 6.3 - Permutations And Combinations
- Chapter 6.4 - Binomial Coefficients And Identities
- Chapter 6.5 - Generalized Permutations And Combinations

11.0.2 Piazza

Must post / respond to at least **two** Piazza posts this week.

11.0.3 Lectures

The lectures for this week and their links can be found below:

- Product Rule ≈ 10 min.
- Product Rule Examples ≈ 14 min.
- Sum Rule ≈ 7 min.
- Sum + Product Rule Examples ≈ 14 min.
- Inclusion-Exclusion Principle ≈ 6 min.
- Division Rule ≈ 16 min.
- Pigeon Hole Principle ≈ 11 min.
- Generalized PHP ≈ 7 min.
- Permutations And Combinations: An Introduction ≈ 14 min.
- Counting Permutations ≈ 10 min.
- Counting Combinations ≈ 13 min.
- Permutations Of Repeated Objects ≈ 11 min.
- Binomial Theorem ≈ 6 min.
- Pascal's Triangle ≈ 9 min.
- Sum Of Binomial Coefficients ≈ 6 min.
- Proofs Using Pigeon-Hole Principle ≈ 7 min.
- Slick Proofs Using Pigeon Hole Principle ≈ 9 min.
- Ramsey's Theorem ≈ 11 min.
- Proof Of Binomial Theorem ≈ 8 min.
- Counting Solutions To Sum Of Variables - Part 1 ≈ 6 min.

- Counting With Sum Of Variables - Part 2 ≈ 3 min.
- Counting With Repetitions ≈ 8 min.
- Counting With Repetitions : Unordered ≈ 11 min.
- Further Properties Of Binomial Coefficients ≈ 7 min.

11.0.4 Assignments

The assignment for this week is:

- Assignment 9 - Counting And Binomials

11.0.5 Quiz

The quiz's for this week can be found at:

- Quiz 10 - Counting

11.0.6 Chapter Summary

The first section that we are covering this week is **Section 6.1 - The Basics Of Counting**.

Section 6.1 - The Basics Of Counting

Overview

This section introduces the fundamental principles of counting which are essential in discrete mathematics and computer science for analyzing algorithms, computing probabilities, and solving enumeration problems. The chapter is divided into various subtopics, each explaining different counting techniques and their applications.

Basic Counting Principles

The section starts with the basic counting principles:

- The *product rule* explains that if a task can be divided into a sequence of two subtasks that are independent, the total number of ways to perform this task is the product of the number of ways to perform each subtask.
- The *sum rule* applies to situations where a task can be performed by one of two mutually exclusive methods.
- The *subtraction rule* or inclusion-exclusion principle corrects for overcounting when two or more ways to do a task overlap.
- The *division rule* states that if a task can be done in n ways, and there are d indistinguishable ways to do it, then there are n/d distinct ways to do the task.

Applications and Examples

Numerous examples illustrate the application of these rules, such as assigning offices to employees, creating license plates, counting functions from one set to another, and constructing variable names in programming languages.

More Complex Counting Problems

The chapter also delves into more complex counting problems that require a combination of the above rules, such as:

- Counting the number of different passwords that meet specific criteria.
- Determining the number of possible IPv4 addresses.
- Avoiding overcounting when sets have elements in common.

Key Takeaways

- The principles of counting are essential for solving a wide range of problems in discrete mathematics and computer science.
- The product rule, sum rule, subtraction rule, and division rule are the foundational techniques for enumerative combinatorics.
- Examples and applications provided in the section highlight the practical significance of these counting principles.

The next section that we will be covering this week is **Section 6.2 - The Pigeonhole Principle**.

Section 6.2 - The Pigeonhole Principle

Overview

The section discusses the Pigeonhole Principle and its generalized version, which are fundamental concepts in combinatorics. The basic Pigeonhole Principle is introduced with the classic example of pigeons and pigeonholes: If $k + 1$ objects are placed into k boxes, then at least one box contains two or more objects.

This principle is formalized in **Theorem 1: The Pigeonhole Principle**, which states: If k is a positive integer and $k + 1$ or more objects are placed into k boxes, then there is at least one box containing two or more of the objects.

The section provides several illustrative examples, demonstrating the principle's application in various scenarios, such as proving that among any group of 367 people, there must be at least two with the same birthday, due to there being only 366 possible birthdays.

The Generalized Pigeonhole Principle

The Generalized Pigeonhole Principle allows us to determine a minimum threshold that guarantees a certain distribution of objects across containers: If N objects are placed into k boxes, then there is at least one box containing at least $\lceil \frac{N}{k} \rceil$ objects.

This principle is encapsulated in **Theorem 2: The Generalized Pigeonhole Principle**. Examples provided in the section apply this theorem to practical problems, such as determining the minimum number of students required in a class to guarantee that at least a certain number receive the same grade.

Applications and Elegant Examples

The section also explores elegant applications of the Pigeonhole Principle, such as proving the existence of a subsequence within a sequence of real numbers: **Theorem 3:** Every sequence of $n^2 + 1$ distinct real numbers contains a subsequence of length $n + 1$ that is either strictly increasing or strictly decreasing.

Furthermore, the section touches on Ramsey theory and the existence of certain structures within a graph, as illustrated by the problem of friends and enemies at a party.

Key Takeaways

- The Pigeonhole Principle is a simple yet powerful tool in proving the existence of certain properties in sets and distributions.
- The generalized version extends the principle to scenarios where objects outnumber containers by larger multiples.
- These principles find practical applications in various fields, such as data transmission, coding theory, and graph theory.

The next section that we will cover this week is **Section 6.3 - Permutations And Combinations**.

Section 6.3 - Permutations And Combinations

Overview

The section provides a comprehensive discussion on permutations and combinations, fundamental concepts in combinatorics that deal with the arrangement and selection of objects. The section begins by defining permutations and demonstrating how to calculate them using the product rule.

Permutations

Permutations are introduced as ordered arrangements of objects, with specific notation $P(n, r)$ denoting the number of permutations of r objects from a set of n distinct objects. The formula for permutations is derived as follows:

$$P(n, r) = n(n-1)(n-2)\cdots(n-r+1)$$

This formula is proven using the product rule, which states that the number of ways to perform a sequence of tasks is the product of the number of ways to perform each individual task.

Combinations

Combinations, unlike permutations, are defined as selections where order does not matter. The notation $C(n, r)$, or $\binom{n}{r}$, represents the number of ways to choose r objects from a set of n distinct objects without considering the order. The formula for combinations is given by:

$$C(n, r) = \frac{n!}{r!(n-r)!}$$

This is explained by dividing the number of r-permutations by the $r!$ ways to order the r objects, eliminating the consideration of sequence.

Key Theorems and Corollaries

Important results discussed in the section include:

- **Theorem 1** establishes the formula for the number of permutations.
- **Theorem 2** presents the formula for combinations, also known as the binomial coefficient.
- **Corollary 2** indicates the symmetry in combinations: $C(n, r) = C(n, n-r)$.

Applications

Several applications of permutations and combinations are provided through examples, demonstrating the utility of these concepts in solving practical problems involving arrangement and selection.

Key Takeaways

- Permutations and combinations are crucial for counting the number of possible arrangements or selections in a set.
- The difference between permutations and combinations lies in whether the order of selection is important.
- These concepts have significant applications in probability, statistics, and various fields where combinatorial optimization is required.

The last section that we will cover this week is **Section 6.4 - Binomial Coefficients and Identities**.

Section 6.4 - Binomial Coefficients and Identities

Overview

The section covers the fundamental concepts of binomial coefficients and various identities associated with them. Binomial coefficients, denoted by $\binom{n}{r}$, are the coefficients in the expansion of powers of binomial expressions such as $(a+b)^n$.

The Binomial Theorem

The Binomial Theorem is introduced as a way to express the expansion of powers of binomial expressions. It states that for any nonnegative integer n and any variables x and y, the expansion is given by:

$$(x+y)^n = \sum_{j=0}^{n} \binom{n}{j} x^{n-j} y^j$$

This theorem is proved using a combinatorial argument, which demonstrates the relationship between algebraic expansion and combinatorial counting.

Identities Involving Binomial Coefficients

Several identities involving binomial coefficients are presented, including Pascal's Identity, which relates the coefficients in the context of Pascal's Triangle:

$$\binom{n+1}{k} = \binom{n}{k-1} + \binom{n}{k}$$

Other identities include Vandermonde's Identity, which provides a combinatorial method for calculating certain sums of binomial coefficients:

$$\binom{m+n}{r} = \sum_{k=0}^{r} \binom{m}{r-k} \binom{n}{k}$$

Applications and Examples

Applications and examples provided in the section include using the Binomial Theorem for expanding algebraic expressions and calculating specific coefficients, demonstrating the practical utility of these mathematical tools.

Key Takeaways

- Binomial coefficients play a critical role in combinatorics, probability, and algebra.
- The Binomial Theorem and associated identities offer a systematic way to work with polynomials and combinatorial quantities.
- These concepts provide a deep understanding of the structural properties of binomial expressions and their coefficients.

Exam 2

Exam 2

12.0.1 Piazza

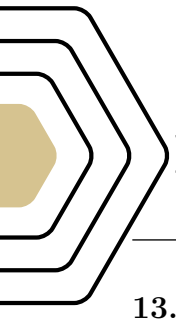
Must post / respond to at least **two** Piazza posts this week.

12.0.2 Quiz

The quiz's for this week can be found at:

- Quiz 11 - Ethics and Binary Applications

Probability



Probability

13.0.1 Assigned Reading

The reading assignments for this week is from, :

- Chapter 7.1 - An Introduction To Discrete Probability
- Chapter 7.2 - Probability Theory
- Chapter 7.3 - Bayes' Theorem
- Chapter 7.4 - Expected Value And Variance

13.0.2 Piazza

Must post / respond to at least **two** Piazza posts this week.

13.0.3 Lectures

The lectures for this week and their links can be found below:

- Intro To Discrete Probability $\approx$ 32 min.
- Probability Theory 2 $\approx$ 22 min.
- Conditional Probability And Independence $\approx$ 44 min.
- Bayes' Theorem And The LTP $\approx$ 41 min.
- Bayesian Spam Filters $\approx$ 20 min.

Below is a list of lecture notes for this week:

- Bayes Lecture Notes
- Introduction To Discrete Probability Lecture Notes
- Introduction To Probability Theory Lecture Notes
- Conditional Probability And Independence Lecture Notes
- Bayes' Theorem And The Law Of Total Probability Lecture Notes

13.0.4 Assignments

The assignment for this week is:

- Assignment 10 - Probability

13.0.5 Quiz

The quiz's for this week can be found at:

- Quiz 12 - Probability

13.0.6 Chapter Summary

The first section that we are covering this week is **Section 7.1 - An Introduction To Discrete Probability**.

Section 7.1 - An Introduction To Discrete Probability

Overview

This section introduces the basic concepts of discrete probability.

Key Concepts

- **Probability Definition**: Probability is defined for experiments with a finite number of equally likely outcomes. The probability of an event E is given by the formula:

$$p(E) = \frac{|E|}{|S|}$$

  where $|E|$ represents the number of outcomes in event E and $|S|$ is the total number of outcomes in the sample space.

- **Probability Range**: The probability of any event is a value between 0 and 1, inclusive.
- **Illustrative Examples and Theorems**: The section includes various examples and theorems to illustrate probability calculations. Classic problems like drawing cards and rolling dice are used to demonstrate these concepts.
- **Monty Hall Problem**: A famous problem in probability, demonstrating counterintuitive results in probability theory, is discussed to elucidate the concepts.

The next section that we will cover this week is **Section 7.2 - Probability Theory**.

Section 7.2 - Probability Theory

Overview

Probability theory is a branch of mathematics concerned with analyzing random events. The probabilities of different outcomes are calculated to understand the likelihood of various events occurring.

Basic Probability Assignment

- The formula $p(E) = \frac{|E|}{|S|}$ is used to calculate the probability of an event E occurring within a finite sample space S. This formula assumes each outcome is equally likely.
- Probabilities are always between 0 and 1, inclusive, where 0 indicates impossibility and 1 indicates certainty.

Conditional Probability and Independence

- Conditional probability is used when the probability of an event depends on another event. The formula $p(E|F) = \frac{p(E \cap F)}{p(F)}$ quantifies this dependence.
- Two events are independent if the occurrence of one does not affect the probability of the other. This is mathematically expressed as $p(E \cap F) = p(E)p(F)$.

Random Variables

- A random variable translates outcomes of random processes into numerical values, facilitating probability calculations.
- The distribution of a random variable describes the probabilities of its different possible values.

Bernoulli Trials and Binomial Distribution

- Bernoulli trials are experiments with two possible outcomes, typically "success" and "failure". The binomial distribution calculates the probability of a fixed number of successes in a set number of these trials.

Monte Carlo Algorithms

- Monte Carlo algorithms use random sampling to obtain numerical results, often used in scenarios where deterministic methods are impractical.

The Probabilistic Method

- This method uses probability to prove the existence of certain structures or properties when direct construction is difficult. It's a powerful tool in combinatorics and theoretical computer science.

The next section that we will cover this week is **Section 7.3 - Bayes' Theorem**.

Section 7.3 - Bayes' Theorem

Overview

Bayes' Theorem is a crucial result in probability theory and is used for calculating the likelihood of an event based on prior knowledge of conditions that might be related to the event.

Introduction to Bayes' Theorem

- The theorem is named after Thomas Bayes and allows for the revision of predictions or hypotheses based on new evidence.
- It is widely used in various fields like medicine, law, and machine learning for decision making under uncertainty.

The Mathematical Formulation

- The theorem mathematically expresses how a subjective degree of belief should rationally change to account for evidence: $P(A|B) = \frac{P(B|A)P(A)}{P(B)}$.
- Here, $P(A|B)$ is the probability of event A given that B is true, $P(B|A)$ is the probability of event B given that A is true, $P(A)$ is the independent probability of A, and $P(B)$ is the independent probability of B.

Applications of Bayes' Theorem

- The theorem is used in various real-world applications, such as medical diagnosis, where it helps in determining the probability of a disease given a specific test result.
- Another application is in the field of spam filtering in emails, where it helps to determine the probability of an email being spam based on its content.

Generalized Bayes' Theorem

- The generalized form of Bayes' Theorem can handle situations with multiple events and is used in more complex scenarios.
- This generalization allows the incorporation of multiple pieces of evidence to refine the probability estimates of an event.

The last section that we will cover this week is **Section 7.4 - Expected Value And Variance**.

Section 7.4 - Expected Value And Variance

Overview

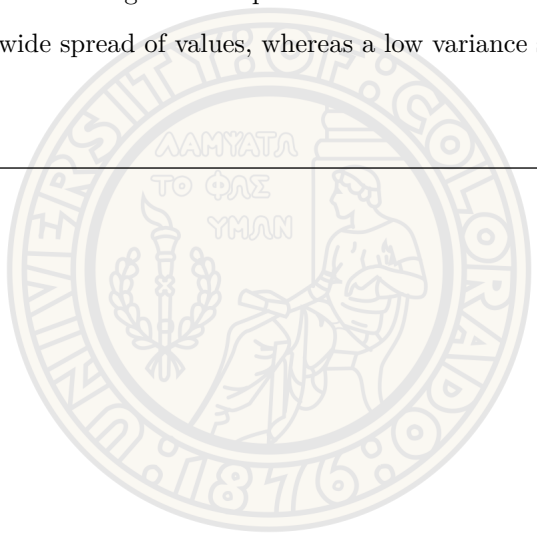
This section delves into the concepts of expected value and variance, which are pivotal in understanding the behavior of random variables in probability and statistics.

Expected Value

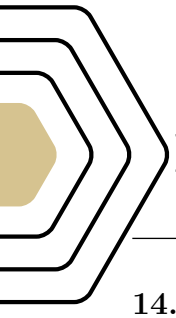
- The expected value, often denoted as $E(X)$, represents the average or mean value of outcomes for a random variable X.
- It is calculated as the sum of all possible values of X, each weighted by its probability of occurrence.
- The concept is crucial in various applications, including decision-making and risk assessment.

Variance

- Variance measures the spread or dispersion of the random variable's values around the expected value.
- Denoted as $V(X)$ or σ^2, it is the average of the squared differences from the mean.
- A high variance indicates a wide spread of values, whereas a low variance suggests a tight clustering around the mean.



Relations



Relations

14.0.1 Assigned Reading

The reading assignments for this week is from, :

- Chapter 9.1 - Relations And Their Properties
- Chapter 9.2 - n-ary Relations And Their Applications
- Chapter 9.3 - Representing Relations
- Chapter 9.4 - Closures Of Relations
- Chapter 9.5 - Equivalence Relations

14.0.2 Piazza

Must post / respond to at least **two** Piazza posts this week.

14.0.3 Lectures

The lectures for this week and their links can be found below:

- Basic Relations ≈ 17 min.
- Properties Of Relations ≈ 20 min.
- Closures Of Relations ≈ 26 min.
- Equivalence Relations: Part 1 ≈ 16 min.
- Equivalence Classes ≈ 17 min.
- Partial Orders ≈ 18 min.
- Posets And Total Orders ≈ 16 min.
- Relations And Databases ≈ 12 min.
- Operations On Relations And Querying Databases ≈ 25 min.

Below is a list of lecture notes for this week:

- Basic Relations And Their Representations Lecture Notes
- Properties Of Relations Lecture Notes
- Closures Of Relations Lecture Notes

14.0.4 Assignments

The assignment for this week is:

- Assignment 11 - Relations

14.0.5 Quiz

The quiz's for this week can be found at:

- Quiz 13 - Relations

14.0.6 Chapter Summary

The first section that we are covering this week is **Section 9.1 - Relations And Their Properties**.

Section 9.1 - Relations And Their Properties

Overview

Binary relations are foundational concepts in discrete mathematics, representing complex relationships in a simple, structured form. They are defined as sets of ordered pairs, where each pair consists of elements from two different sets. This section introduces the basic idea of binary relations through practical examples, such as the relationships between students and courses they are enrolled in, or cities and their corresponding states.

- Binary relations are sets of ordered pairs representing relationships between elements of two sets.
- Real-world examples, such as student-course enrollments, illustrate various binary relations.

Functions as Relations

Functions are a special case of binary relations with unique characteristics. In this subsection, the textbook explains how every function is a relation but not every relation is a function. This distinction is critical and is illustrated through the concept of mapping, where each element in the first set is uniquely paired with an element in the second set.

- Functions are special cases of relations where each element of one set is paired with exactly one element of another set.

Properties of Relations

This part of the textbook explores key properties of relations: reflexivity, symmetry, antisymmetry, and transitivity. These properties are important for understanding how different types of relations behave and interact. The subsection provides various examples to illustrate these properties, helping to clarify complex abstract concepts with tangible scenarios.

- Key properties of relations include reflexivity, symmetry, antisymmetry, and transitivity.
- Examples demonstrate the application of these properties in different contexts.

Combining Relations

Relations can be combined or modified through various operations like union, intersection, and composition. This subsection explains these operations and their significance in the study of relations. It uses practical examples to demonstrate how these operations are applied, showing how complex relations can be formed from simpler ones.

- Relations can be combined using operations like union, intersection, and composition.
- Practical examples demonstrate the application of these operations.

Theorem and Exercises

The final part of this section includes a theorem regarding the powers of transitive relations, a vital concept in understanding relation properties. To reinforce learning, the subsection offers exercises that provide practical application of the theorem and the concepts discussed in the chapter.

- Includes a theorem related to transitive relations, reinforcing the concepts with exercises for practical understanding.

The next section that we will be covering this week is **Section 9.2 - n-ary Relations And Their Applications**.

Section 9.2 - n-ary Relations And Their Applications

Overview

n-ary relations extend binary relations to sets involving more than two elements. This section introduces these relations with practical examples like relationships in student databases and geometric points. These concepts are foundational in understanding complex relationships in various fields including computer science.

- Discusses relationships involving elements from more than two sets, such as student name, major, and GPA.
- n-ary relations are used in databases and examples include relationships in mathematics and geometry.

Databases and Relations

This subsection delves into how databases utilize n-ary relations for efficient data management. Relations are represented as tables, where each tuple corresponds to a record. Examples include student databases, where records are tuples of student details. This part emphasizes the practical application of n-ary relations in storing and managing complex data.

- Explains the role of n-ary relations in databases, with a focus on their efficiency and utility.
- Relations are represented as tables with attributes, illustrated with student record examples.

Operations on n-ary Relations

Various operations such as selection and projection are explored, highlighting how they can be used to extract and manipulate data from n-ary relations. Definitions are provided along with examples showing the application of these operations in database queries. This part is crucial for understanding how complex data is processed and retrieved in relational databases.

- Covers operations like selection and projection to manipulate n-ary relations for specific queries.
- Includes definitions and examples to demonstrate these operations on databases.

SQL and n-ary Relations

The final subsection connects the concepts of n-ary relations with SQL, a standard database query language. It demonstrates how SQL commands are used to execute operations on relations, with practical examples showing the translation of theoretical concepts into real-world database queries. This part bridges the gap between theoretical understanding and practical application in database management.

- Demonstrates the use of SQL in implementing operations on n-ary relations.
- Provides examples to show how SQL commands correspond to relational database operations.

The next section that we will be covering this week is **Section 9.3 - Representing Relations**.

Section 9.3 - Representing Relations

Overview

This subsection introduces zero-one matrices to represent binary relations. It explains how to construct a matrix for a relation R between finite sets A and B. It includes examples to illustrate this representation method.

- Discusses representing binary relations using zero-one matrices.
- For a relation R from set A to B, matrix M_R has 1 in position (i, j) if $(a_i, b_j) \in R$ and 0 otherwise.

Properties of Relations Represented by Matrices

This part delves into how properties of relations like reflexivity, symmetry, and antisymmetry are represented in matrices. It details the criteria for identifying these properties using matrix entries.

- Explores matrix characteristics representing reflexivity, symmetry, and antisymmetry in relations.
- Relation R is reflexive if all diagonal elements of M_R are 1, symmetric if M_R is symmetric, and antisymmetric if $m_{ij} = 1$ implies $m_{ji} = 0$ for $i \neq j$.

Representing Relations Using Digraphs

The final subsection discusses representing relations using directed graphs or digraphs. It describes how elements of a set are represented as vertices and relations as directed edges, providing a visual method for understanding binary relations.

- Introduces directed graphs (digraphs) as a method for representing relations.
- In digraphs, elements are vertices and relations are represented as directed edges.

The next section that we will be covering this week is **Section 9.4 - Closures Of Relations**.

Section 9.4 - Closures Of Relations

Overview

This subsection discusses how to construct closures of relations, which are the smallest relations containing a given relation and having specific properties. The concept of transitive closure is particularly emphasized, with examples showing how to find all pairs in a relation that are indirectly connected.

- Explores how to extend a relation to include certain properties like reflexivity, symmetry, and transitivity.
- Concepts like transitive closure and symmetric closure are introduced with examples.

Reflexive Closures

The reflexive closure of a relation involves adding pairs so that every element is related to itself. The section illustrates this concept with examples, showing how the reflexive closure can be represented mathematically.

- Details the method of adding pairs to make a relation reflexive.
- Reflexive closure is represented as $R \cup \Delta$, where Δ is the diagonal relation.

Symmetric Closures

In symmetric closures, the textbook explains how to add pairs to a relation to ensure that if (a, b) is in the relation, then (b, a) is also included. The symmetric closure of a relation is illustrated with clear examples.

- Discusses adding pairs to a relation to make it symmetric.
- Symmetric closure is represented as $R \cup R^{-1}$, where R^{-1} is the inverse of R.

Transitive Closures

The transitive closure subsection is particularly detailed, explaining how to find all pairs that can be connected through a series of steps within a relation. It provides a comprehensive view of how to construct a transitive closure using paths in directed graphs and matrix representation.

- Addresses the concept of adding pairs to make a relation transitive.
- Transitive closure involves finding all pairs that can be connected through intermediate steps in a relation.

The last section that we will be covering this week is **Section 9.5 - Equivalence Relations**.

Section 9.5 - Equivalence Relations

Overview

This subsection defines equivalence relations and illustrates them with practical examples. It emphasizes their foundational role in mathematics and computer science, particularly in grouping elements based on relationships.

- Introduces equivalence relations as reflexive, symmetric, and transitive.
- Provides examples like congruence modulo and relational concepts in programming.

Properties of Equivalence Relations

The properties subsection delves into the critical aspects that define equivalence relations. It uses examples to show how these properties manifest in real-world scenarios and mathematical contexts.

- Discusses the defining properties: reflexivity, symmetry, and transitivity.
- Examples demonstrate how these properties are applied in various contexts.

Equivalence Classes

This part explains the concept of equivalence classes, which are formed when a set is partitioned based on an equivalence relation. It discusses the significance of these classes in understanding relationships within a set.

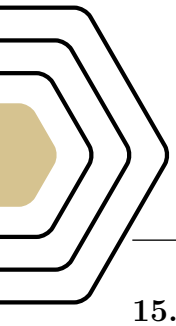
- Describes how equivalence relations partition a set into distinct classes.
- Focuses on the concept of equivalence classes in relation to set theory.

Practical Applications

The final subsection explores practical applications of equivalence relations. It highlights their importance in computer science, particularly in programming, and discusses their use in mathematical fields like modular arithmetic.

- Applies equivalence relations to real-world situations, including computer programming.
- Discusses the use in modular arithmetic and other mathematical applications.

Graph Theory



Graph Theory

15.0.1 Assigned Reading

The reading assignments for this week is from, :

- Chapter 10.1 - Graphs And Graph Models
- Chapter 10.2 - Graph Terminology And Special Types Of Graphs

15.0.2 Piazza

Must post / respond to at least **two** Piazza posts this week.

15.0.3 Lectures

The lectures for this week and their links can be found below:

- Essentials: Intro ≈ 2 min.
- Essentials: Terminology ≈ 15 min.
- Essentials: Degree Sequences ≈ 17 min.
- Essentials: Wrap-Up ≈ 1 min.
- Connectedness And Eulerian Tours ≈ 30 min.
- Graph Coloring ≈ 40 min.

Below is a list of lecture notes for this week:

- Essential Graph Theory Lecture Notes
- Connectedness And Eulerian Tours Lecture Notes
- The Coloring Problem Lecture Notes

15.0.4 Chapter Summary

The first section that we are covering this week is **Section 10.1 - Graphs And Graph Models**.

Section 10.1 - Graphs And Graph Models

Overview:

This section covers the foundational concepts of graph theory. It defines a graph as a collection of vertices and edges, and explores various types of graphs including simple graphs, multigraphs, pseudographs, directed, and mixed graphs. The section delves into graph terminology such as paths, cycles, and vertex degrees. It also highlights the practical applications of graphs in areas like social networks and transportation, and introduces basic graph algorithms. This concise overview encapsulates the key ideas and applications of graph theory presented in the section.

Graph Definition:

- A graph is formally defined as $G = (V, E)$, where 4 is a nonempty set of vertices and E is a set of edges.
- Simple Graphs: Consist of a set of vertices and edges, where each edge connects a pair of distinct vertices without any loops or multiple edges between the same vertices.
- Multigraphs: Permit multiple edges between the same pair of vertices.
- Pseudographs: Allow loops and multiple edges.
- Directed Graphs (Digraphs): Edges have a direction, represented as ordered pairs of vertices.
- Mixed Graphs: Contain both directed and undirected edges.

Graph Terminology:

- The degree of a vertex, paths, cycles, connected graphs, and subgraphs are explained with examples.
- In directed graphs, concepts of in-degree and out-degree are introduced.

Graph Models:

- Practical applications in various fields like social networks, communication networks, and transportation are highlighted.
- Models for specific applications, like representing a computer network or transportation system, are discussed with examples.

Graph Representations:

- Different methods of representing graphs, such as adjacency lists and adjacency matrices, are detailed.

Graph Algorithms:

- Basic algorithms for graph traversal, such as depth-first search (DFS) and breadth-first search (BFS), are introduced.

The last section that we will cover this week is **Section 10.2 - Graph Terminology And Special Types Of Graphs**.

Section 10.2 - Graph Terminology And Special Types Of Graphs

Overview:

This section elaborates on key terms in graph theory and explores various special graph types. It introduces basic vocabulary essential for solving problems in graph theory, like graph drawing and vertex correspondence. The section covers terminology for both undirected and directed graphs, such as adjacency, degrees, and neighborhood concepts. It also delves into special graph types like complete graphs, cycles, wheels, bipartite graphs, and n-cubes, explaining their characteristics and significance. The section ties these concepts to practical applications, illustrating how graphs model complex systems in various fields.

Graph Foundations:

Clarifies the basic elements of graphs: vertices (nodes) and edges (lines). It distinguishes between simple graphs (no loops or multiple edges), multigraphs (multiple edges allowed), and pseudographs (loops and multiple edges allowed).

Vertex Characteristics:

Focuses on vertex degrees - the number of edges connected to a vertex. It introduces adjacency (when two vertices are connected by an edge) and incidence (relationship between vertices and edges).

Directed Graphs:

In these, edges have directions. The section explains in-degree (number of incoming edges) and out-degree (number of outgoing edges) for vertices in directed graphs.

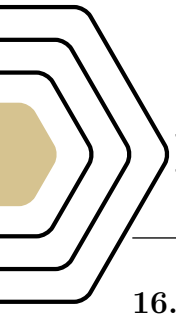
Specific Graph Types:

Describes complete graphs (every pair of distinct vertices is connected by a unique edge), cycles (a path that starts and ends at the same vertex), and bipartite graphs (vertices can be divided into two disjoint sets where each edge connects a vertex from one set to a vertex from the other).

Graph Representation Methods:

Introduces adjacency lists and matrices as ways to represent graph structures in a compact form.

Final Exam



Final Exam

16.0.1 Lectures

The lectures for this week and their links can be found below:

- Final Exam Notes ≈ 2 min.
- Introduction ≈ 16 min.
- Introduction to Lists ≈ 5 min.
- List Operations ≈ 8 min.
- List Manipulations: Iterations ≈ 8 min.
- Python Tutor: Getting Started ≈ 5 min.
- Basics: Defining Functions ≈ 10 min.
- Defining Functions : Multiple Arguments And Larger Functions ≈ 13 min.
- Composition of Functions ≈ 8 min.
- Recursive Functions ≈ 10 min.
- Side Effects: Pure vs. Nonpure Functions ≈ 7 min.
- Polymorphic Functions ≈ 12 min.
- Higher Order Functions ≈ 10 min.
- Lambdas ≈ 7 min.
- Closures ≈ 18 min.
- Map Operation ≈ 17 min.
- Reduce Operation ≈ 12 min.
- Filter ≈ 10 min.

Below is a list of lecture notes for this week:

- Introduction To Functional Programming Lecture Notes

The Python tutor links for this final assignment can be found below:

- Lists
- List Operations
- Iterations
- Introduction
- Defining Functions
- Multiple Argument Functions
- Composition Of Functions
- Recursive Functions
- Pure Functions

- Nonpure Functions
- Polymorphic Functions
- Higher Order Functions
- Lambda Functions
- Closures
- Maps: 1
- Maps: 2
- Maps: 3
- Reduce Operations: 1
- Reduce Operations: 2
- Filters: 1
- Filters: 2

16.0.2 Exam

The optional final programming assignment this week can be found at:

- Final Exam