A BRIEF OUTLINE
OF
DISCRETE MATHEMATICS

A SHORT COURSE IN DISCRETE MATHEMATICS FOR A COMPUTER SCIENCE UNDERGRADUATE

EDITED BY

DALE FLETTER
D P

The University of California
Davis

2019

PUBLISHER

created using LaTeX via TexWorks © 2007-2017 Jonathan Kew, Stefan Löffler, Charlie Sharpsteen Version 0.6.2 (travis-ci) [r.7ecce17, 4/29/17 9:23 AM]running on a MacBook Pro under macOS Mojave, 10.14.2

Für alle, die die Schönheit von Wissenschaft anderen zeigen wollen.

Contents

List of Figures

List of Tables

Introduction

Discrete math is a survey of many different areas of mathematics not covered by the usual series of calculus classes required of an engineering student. Since it is a survey it uses an overwhelming number of conventional symbols that have been adopted over the centuries by mathematicians. You will be reading a paper that was set using a popular free typesetting tool called Latex that is nearly universally used by scientists, engineers and mathematicians to publish papers. At times in this paper you will see references to backslashed sequences like $\forall$, $\in$, etc. These are references to the plaintext markups that are used in Latex in case you want to begin using that tool. For anyone who aspires to graduate school it is a requirement.

While the intent of this material is to be as purely formal as possible, it is necessary to make the course easier to make some assumptions of prior knowledge even before it is formally introduced. For example the section on logic assumes the reader is already familiar with all secondary school math including the concepts of integer versus real numbers and the basic rules of algebra.

In order to make a course for this material as compact and efficient as possible, this outline makes many assumptions about prerequisite material. It is assumed the student is already familiar with a two column proof form as typically presented in a high school geometry class. At least two courses in programming are assumed with a thorough understanding of the logic of Boolean expressions and their evaluation. The most basic understanding of function and set are taken as a given to avoid introducing more formal material on those topics. Directed graphs are introduced when talking about relations even though the formal treatment is later. And any material which requires calculus is excluded even though the student is expected to have the algebra skills that are needed to complete at least one course in calculus requires.

For an intense 6 week summer course there is always a chance that all of the material in this outline will not be covered. In our opinion the section on discrete probability can be safely skipped when the program will require a full course in statistics and probability. And the section on trees can be skipped since there will be a programming course on data structures and the student who completes this section on graphs should be well prepared to quickly master the new material in that course. An instructor using this material can of course take the chapters in a different order although there was great attention given this outline to avoid introducing material out of this sequence.

Comment About Copyright

This text must be treated as a derivative work of Rosen unless and until a publisher picks up this manuscript. I believe that it can be used under a theory of fair use for classroom use but that is not a legal opinion, only a self-serving moralistic one. I would encourage anyone who wishes to use this consider the policies of their institution before making copies readily available.

Comments About the Order of the Material

There are two ways to approach this material in the courses I have seen, start with logic or start with sets. I believe that starting with logic makes more sense for two reasons. First, a computer science student will have already achieved a level of mastery over Boolean expressions from programming and will be comfortable with the introduction of propositional calculus. Second, for those who wish to take a more formal approach to the material, they can get through the section on logic and may be able to largely skip the section on proofs and achieve that end. To make this text more approachable for those at a teaching college we have chosen to cover logic first to make the discussion of proofs easier.

Many instructors like to introduce relations before functions and in many academic ways it makes more sense. However every student has been well introduced to functions before they take this course but only introduced to relations in a less rigorous way. By giving them material that they think they know but developing a more formal approach to it, the fact that relations can be viewed as a superset of functions should make it easier for them to grasp the formalism.

Many instructors will place the material on Graphs and Trees at the end of the course. We believe this is a mistake. Material covered at the end of the course is often neglected and the applications of graphs and trees are essential in industry and in later computer science courses. We bring it forward to ensure that there is sufficient time to study them in depth. This is done to the detriment of discrete probability. We believe that every computer science program will require a course in probability and statistics and find that putting this material at the end does not impact the flow of presentation while recognizing the value of giving a student some exposure to the discrete form of probability before that undergraduate material.

We use a similar reason to push the material on computation to the end. Many courses do not even take up this subject in favor of covering it as a complete course by itself. We include it here to offer some flexibility to the instructor who may choose to cover some material without trying to cover all the material we provide.

We intend for this text to be a competitor to Schaum's Outline which we have judged to be insufficient for our course offerings. This text has far greater depth than Schaums's requiring less supplementation for the instructor.

Stylistic and typographic quirks

Style manuals specify that a period at the end of a sentence be included within the parenthetic expression when it ends the sentence. We reject that and go with the non-standard usage of placing the period after the closing parethesis. Likewise for quotes, commas, etc. The attempt is to bring the English into alignment with programming practice instead of obedience to an illogical (from a CS perspective) convention.

Some History

Previous to the 1970s, there were no computer science programs to speak of. Those who were the computer science pioneers were mathematicians first and practiced computer science as applied mathematics. Consequently they were well educated in many areas of mathematics that are truncated or absent in many computer science programs as the major tends to cater more toward the needs of software engineers instead of applied mathematicians.

At its best, a course in discrete mathematics for the computer scientist must cover a great deal of material in a very short period of time. This is compounded by its placement in most programs in the first two years of undergraduate study. The need to teach it quickly and thoroughly is a challenge. This particular text was originally created to serve as a reference text to replace dependence upon Schaum's or one of the many texts offered. In our opinion those texts suffered from one of two defects, they were either encyclopedic and expensive or they left out key material that required supplementation by the instructor. This text at least answers the second problem and arguably the first in some applications. By reducing the exercises and most of the proofs for the theorems it is far shorter than the scope of material would allow if it attempted to take on the burden of being a teaching text. So it still requires a great deal of supplement for exercises and lectures. However for a student reference work to use during and after the course we believe it is superior to any alternative. And of course it is being offered for free to fellow instructors of this material. The assumption is that some instructors, and perhaps even some students, will offer constructive criticism on how the mission of this text can be improved for future instructors and students.

It may seem a bit pompous to call this a forward to the first edition. But that is simply done to acknowledge that as a first edition this text is surely imperfect in many ways. Most obviously there are inevitable typographical errors from transcription or flaws in the language. But it is also a hope that we will return to this and offer other volumes connected to this which include the much needed exercises and even some courseware to make the job of an instructor charged with teaching this material slightly easier.

Chapter 1

Logic

Logic is foundational for all of science and math. For computer science it is important to recall that most computers are logic machines. While logic can be studied using language, as was the liberal arts tradition, we abstract away the language and deal with a pure symbolic logic and that will be our focus. Since you are already well familar with how to construct Boolean expressions in some programming language we expect you will easily understand the development. Your instructor introduce the fascinating subject of natural language to the strict symbolic language forms in lecture and/or exercises. We will use this symbolic logic to prove things about any domain that can be represented using propositions and predicates.

1.1 Propositional Logic

Def 1.1.1 (Proposition)**.** A proposition is a declarative sentence which is true or false, but not both. Also called a statement. In symbolic logic we use single lowercase letters beginning with p to represent propositions. These are called *propositional variables*.

Notes. There are two **literal values** in this logic, one that represents true and the other that represents false. Most commonly T is used for true and F for false. It is also common to use 1 for true and 0 for

A sentence with variables are not propositions. Pronouns function as variables since the antecedent of the pronoun is ambiguous.

false. Many computer implementations do this, sometimes with unintended consequences. Questions (interogatives) and commands (statements in the imperative mood) are not propositions since no truth value can be associated with them. The **truth value** of a proposition is true, denoted by the logical literal T, if it is a true proposition, and the truth value of a proposition is false, denoted by F, it it is a false proposition. The area of logic that deals with propositions is called the **propositonal calculus** or **propositional logic**.

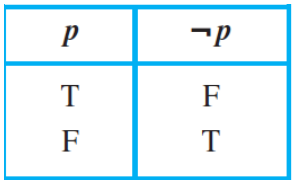
1.1.1 Atomic and Compound Propositions and Logical Connectives

Def 1.1.2 (Logical Connectives and Compound Propositions)**.** A **compound proposition** is a proposition formed from atomic propositions with logical connectives or logical operators. Also called a logical or Boolean expression. An atomic propositions is one that is not a compound propositions.

Note Schaum's represents an abstract compound proposition P with atomic propositions p,q,r, etc as P(p,q,r,...).
Def 1.1.3 (Negation)**.** Let p be a proposition. The *negation of* p, denoted by $\neg p$ (also denoted by $\overline{p}$, is the statement *"It is not the case that p."*
The proposition $\neg p$ is read "not p". The truth value of the negation of p, $\neg p$, is the opposite of the truth value of p. Other notations for logical negation include $\neg$ and ~. Many computer science students will use the ! symbol as well.

Note how you can always negate a natural language proposition by prepending the phrase, "it is not the case that..." followed by the original proposition.
Def 1.1.4 (Truth Table)**.** Compound propositions are often presented in a table which shows the truth value associated with the expression for every possible combination of the atomic propositions in that compound proposition. The order of the rows most commonly starts with true for all atomic propositions and proceeds by alternating the rightmost atomic proposition between true and false

with changes in the next leftmost proposition until all possible combinations have been listed. Table 1.1 displays the **truth table** for the negation of a proposition. The negation symbol is a unary operator taking as the single operand the proposition and giving a new propositon which is the negation of the given proposition.



p	$\neg p$
T	F
F	T

Table 1.1: The Truth Table for the Negation of a Proposition

Def 1.1.5 (Conjunction)**.** Let p and q be propositions. The *conjunction* of p and q, denoted by $p \wedge q$, is the proposition p *and* q. The conjunction $p \wedge q$ is true when both p and q are true and is false otherwise.
Def 1.1.6 (Disjunction)**.** Let p and q be propositIons. The disjunction of p and q, denoted by $p \vee q$, is the proposition p *or* q. The disjunction $p \vee q$ is false when both p and q are false and true otherwise.

Logical disjunction captures part of the meaning of the natural language word "or." But logical disjunction must be carefully observed to have a distinct formal meaning. Contrast this with what is called the exclusive or.
Def 1.1.7 (Exclusive OR)**.** Let p and q b propsition. The *exclusive or* of p and q, denoted by $p \oplus q$, is the proposition that is true when exactly one of p and q is true and is false otherwise.

1.1.2 Conditional Statements or Logical Implication

Def 1.1.8. Let p and q be propositions. The conditional statement p →q is the proposition "if p then q". The conditional statement $p \rightarrow q$ is false when p is true and q is false, and true otherwise. In the conditional statement $p \rightarrow q$, p is called the **hypothesis** (or **antecedent** or **premise**)(and q is called the **conclusion**, **consequent**. There are many ways in which the implication is set in type. Alternatives to what is defined here include $\Longrightarrow$ and $\supset$. It is

acceptable to use -> when typing.
Notes. logical implication is not causality. When the antecedent is false the expression is always true. The only way the expression is false is when the antecedent is true and the consequence is false. this is a logical operator and must not be confused with the use of the conditional statement in programmng languages. We will discuss this again in the section on Algorithms.

Conditional statements are a backbone of deductive logic and mathematical reasoning. Yet students often fail to grasp how to understand and use these statements. The lack of understanding here will hamper the ability to avoid many errors in understanding and writing mathematical proofs.

The conditional statement is also called material implication. In a logic class you will also come across this concept as necessary and sufficient conditions. In logic, the necessary condition is the consequent of the conditional statement. We saw that a material implication is only false when the antecedent is true and the consequent is false. So if we find that the consequent is true and the implication is true, the antecedent MUST BE TRUE.

Logic also teaches sufficient conditions. The sufficient condition is the antecedent. But having the antecedent be true and the implication be true does not guarantee us that the consequent will also be true. For example, "If you get an A on your final, then you will get an A for the course." Is it necessary for you to get an A on the midterm to get an A in the course? The material implication can still be true even when you got a B on the final, that is you get an A even though it is not the case that you got an A on the final. Getting an A on the final is sufficient for you to get an A in the course but it is not necessary for you to earn an A on the final to get an A for the course.

It will help you if you take the time to understand necessary and sufficient in the context of the material implication.
Def 1.1.9 (Inverse, Converse and Contrapositive)**.** Given a conditional statement $p \to q$, its **inverse** is the statement $q \to p$, its converse is $\neg p \to \neg p$ and its **contrapositive** is the statement $\neg q \to \neg p$

Def 1.1.10 (The Bi-Conditional)**.** Let p and q be propositions. The *biconditional statement* $p \leftrightarrow q$ is the proposition "p if and only if q." The biconditional statement $p \leftrightarrow q$ is true when p and q have the same truth values, an d is false otherwise. Biconditional statements are also called *bi-implicatons*. We can also read this as "p is logically equivalent to q".

Note that you get the same truth values for the compound expressions $(p \leftrightarrow q) \wedge (q \rightarrow p)$ and $p \leftrightarrow q$.

p	q	$p \vee q$	$p \oplus q$	$p \wedge q$	$p \rightarrow q$	$p \leftrightarrow q$
0	0	0	0	0	1	1
0	1	1	1	0	1	0
1	0	1	1	0	0	0
1	1	1	0	1	1	1

Table 1.2: Bitwise Logic

1.1.3 Evaluation of Compound Propositions

Without parentheses compound logical statements can be ambiguous. But always explicitly including the parentheses leads to large numbers of them and makes the expressions harder to read. To avoid a large number of parentheses we adopt a convention of **operator precedence**. See Table 1.3.

Operator	Precedence
$\neg$	1
$\wedge$	2
$\vee$	3
$\rightarrow$	4
$\leftrightarrow$	5

Table 1.3: Precedence of Logical Operators

Truth Tables can be used to help derive the truth value of complex compound logic statements. Take the innermost binary operations required by the rules of precedence and create a column. Give the truth values for that small compound statement and then build up until you have the entire statement.

1.1.4 Propositional Consistency

consistent statements

1.1.5 Propositional Satisfiability

Def 1.1.11. A compound proposition is **satisfiable** if there is an assignment of truth values to the variables in the compound proposition that makes the statement form true.

1.1.6 Paradox

Some declarative statements defy a truth value. For example, “This statement is false” cannot be given a consistent truth value. If it is true, that the statement is false, then it must be true contradicting the assertion. Such a statement is called a **paradox**.

Def 1.1.12. A bit is a binary digit, typically a zero or a one. A bit string is a sequence of zero or more bits. The length of this string is the number of bits in the string. Given two bit strings of the same length we define bitwise operations:

1. bitwise OR,
2. bitwise AND,
3. bitwise XOR.

1.1.7 Propositional Equivalences

Def 1.1.13 (Tautology and Contradiction)**.** A compound proposition that is always true , no matter what the truth value of the poisitons that occur in it, is called a *tautology*. A compound proposition that is always false is called a *contradiction*. A compound proposition that is neither a tautology nor a contradiction is called a *contingency*.

Def 1.1.14 (Logical Equivalence)**.** The compound proposition p and q are called *logically equivalent* if $p \leftrightarrow q$ is tautology. The notation $p \equiv q$ denotes that p and q are logically equivalent.

Note: The symbol ≡ is not a logical connective and $p \equiv q$ is not a compound proposition but a statement that $p \leftrightarrow q$ is a tautology.

Two distinct propositions may evaluate to the same truth value for each combination of the atomic truth values. These two propositions are said to be logically equivalent or simply equivalent. This can be shown with a truth table and constitutes a valid proof of the equivalence. Logical equivalence leads to the principle that if two compound propositions always have the same truth values regardless of the values of the atomic propositions, then one can be substituted for the other where ever it appears.

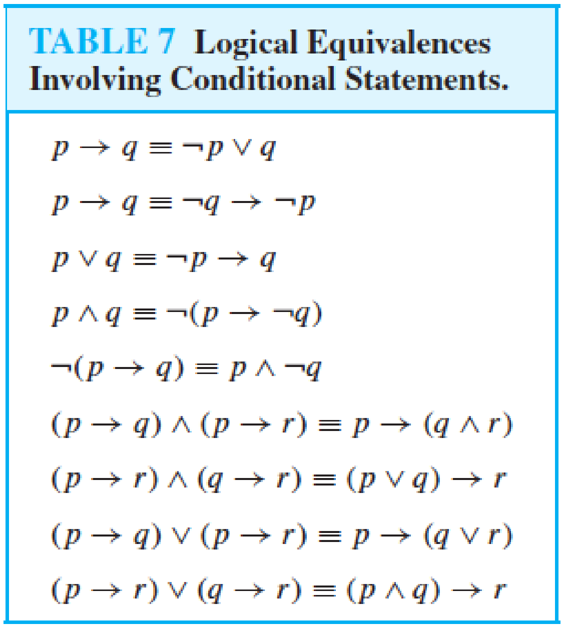


TABLE 7 Logical Equivalences Involving Conditional Statements.
$p \to q \equiv \neg p \vee q$
$p \to q \equiv \neg q \to \neg p$
$p \vee q \equiv \neg p \to q$
$p \wedge q \equiv \neg(p \to \neg q)$
$\neg(p \to q) \equiv p \wedge \neg q$
$(p \to q) \wedge (p \to r) \equiv p \to (q \wedge r)$
$(p \to r) \wedge (q \to r) \equiv (p \vee q) \to r$
$(p \to q) \vee (p \to r) \equiv p \to (q \vee r)$
$(p \to r) \vee (q \to r) \equiv (p \wedge q) \to r$

Table 1.4: Logical Equivalences Involving Conditional Statements

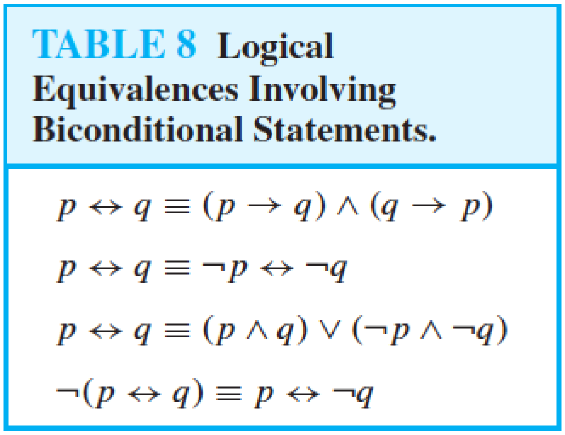


TABLE 8 Logical Equivalences Involving Biconditional Statements.

$p \leftrightarrow q \equiv (p \to q) \wedge (q \to p)$

$p \leftrightarrow q \equiv \neg p \leftrightarrow \neg q$

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

$\neg(p \leftrightarrow q) \equiv p \leftrightarrow \neg q$

Table 1.5: Logical Equivalences Involving Bi-Conditional Statements

1.2 Predicate Logic

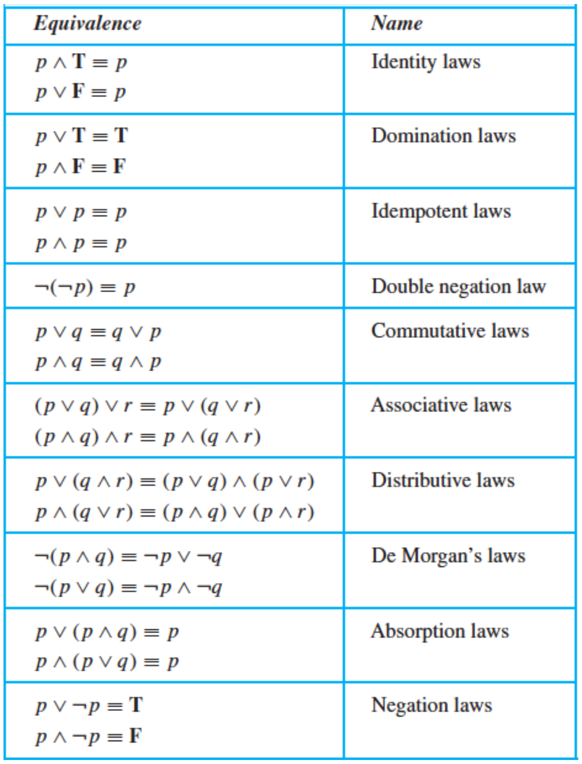
Predicate Logic. First order logic. There are other higher order logics. Propositional logic, zero order logic, cannot help with obviously true statements like, "Socrates is a man, all men are mortal, therefore Socrates is mortal". We need to introduce a way to prove such truths. This is called Predicate Logic or Predicate Calculus.

1.2.1 Predicates

Some objects can have a property. A person may be tall or short. A number may be greater than 100. When we use variables to represent these objects we need a way to test them for the presence or absence of the quality we care about. We call these **propositional functions** and we call the quality being checked for **predicates**.

For example, we may have a variable x which is an integer. We can have a predicate which tests it to see if it is greater than 100. The predicate is "... is greater than 100" and we apply that to the variable x.

Def 1.2.1 (Predicate)**.** A *predicate* is a function which when applied to an object will evaluate to true when some property is present and



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

Table 1.6: Logical Equivalences

false otherwise. We denote the predicate using a capital letter and write the expression using a functional notation. So the predicate P when applied to the variable x is written $P(x)$ and will take on the value of true or false when the value of x has been fixed (bound).

Predicates are not limited to one argument but can have any number of arguments, called *n-place predicates*. For example the predicate S could be "...have the same color" and can accept pieces of fruit as objects. Then the expression $S(p,q,r,s,t)$ will be true if the color of each piece of fruit p,q,r,s,t matches the others and false otherwise.

We often define a predicate using the notation, $P : x+1 > x$.
Notes. More advanced texts may discard the parentheses around the arguments in a propositional function
Def 1.2.2. Value Assignment When we wish to indicate that a variable has been bound to a value, we use an assignment statement. We denote the assignment of the value c to the variable x with this notation: $x := c$. Some authors use $x \leftarrow c$ and many programming languages do this wil an equals sign, $x = c$. It helps to notice that there is an implied change of state from before and after the assignment. For example $t := t+1$ must be understood to refer to the value of t before the assignment while evaluating the expression and changing the value of t with the assignment.

We saw that the statement $x > 2$ was not a proposition since x is a variable. But once the variable is bound to a value it then has a truth value. We can denote this using this notation: $[x := 1, (x > 2)]$ and this will evaluate to false. We can also do this symbolically like this: $G : x > 2$, [x:=3, G(x)] which will evaluate to true.

1.2.2 Multi-place Predicates

Some predicates take more than one variable. For example if we want to compare the color of two pieces of fruit, we can do it like so:
$ASetOfFruit = \{apple, banana, papaya, grape, strawberry\}$,
$SameColor(x,y) : xisthesamecolorasy$,
$SameColor(apple, strawberry) = trie$, assuming the apple and the strawberry are both red.

Note how we specified that the variables are drawn from some set, which is called the **domain of discourse**.

1.2.3 Quantifiers

Variables can be bound to values. But we often wish to assert a proposition over a range of values or to claim that there is an object with some property. This process of creating a proposition over some range of objects is called *quantification*. The two fundamental quantifications are the universal and the existential.

Universal Quantification

Many mathematical statements assert that a property is true for all values of a variable in a partiocular doman, called the **domain of discourse** (or **universe of discourse**, often just referred to s the **domain**. Such a statement is expressed using universal quantification. The universal quantification of $P(x)$ for a particular doman in the proposition that asserts that $P(x)$ is true for all values of x in this domain. Note that the domain specifies the possible values of the variable x. The meaning of the universal quantification of $P(x)$ changes when we change the domain. The domain must always be specified when a universal quantifier is used; without it, the universal quantification of a statement is not defined.

Def 1.2.3 (Universal Quantification)**.** The universal quantification of P(x) is the statement

"$P(x)$ for all values of x in the domain."

The notation $\forall x P(x)$ denotes the universal quantification of $P(x)$. Here $\forall$ is called the universal quantifier. We read $\forall x P(x)$ as "for all x P(x)" or "for every x P(x)". An element for which $P(x)$ is false is called a counterexample of $\forall x P(x)$.

Notes. A domain of discourse must be provided when using universal instantiation. Generally an implicit assumption is made that all domains of discourse for quantifiers are nonempty. Note that if the domain is empty, then $\forall P(x)$ is true for any propositional function $P(X)$ because there are no elements x in the domain for which $P(x)$ is false.

The changing the domain of discourse can change the evaluation of the predicate. For example if $P(x): xispurple$, if I wish to say that all eggplants are purple, I can say that when talking about eggplants, $\forall x P(x)$ is true but when talking about apples it is not true that $\forall x P(x)$. If you were to show me an eggplant that is not purple, you have given me a **counter example** and that will refute my assertion that all eggplants are purple. A single counter example is sufficient to refute a universal claim.

In the domain of natural numbers, what is the truth value of this assertion? $\forall n : n+1 > n$?

Consider $R : x^2 \geq x$ and the assertion $\forall x : R(x)$ in the domain of integers. Now consider it for the domain of reals.

It is sometimes helpful to think of a universal quantification as the conjunction of all elements in the domain when the domain is finite. For example if n is a natural number, $\forall n P(n) \equiv P(0) \wedge P(1) \wedge \cdots \wedge P(n)$.

Translation challenge: All apples are sweet. $A(x): xisanapple$, $S(x): xissweet$. Do you translate as $\forall x A(x) \wedge S(x)$ if the domain of discourse is the set of all fruits? No. It must be $\forall x A(x) \rightarrow S(x)$. Otherwise you are saying all fruits are sweet apples.

1.2.4 Existential Quantifier

Many mathematical statements assert that there is an element with a certain property. For example, for any integer i there is another integer j such that $i + j = 0$. Such statements are expressed using existential quantification. With existential quantification, we for a proposition that is true if and only if $P(x)$ is true for at least one value of x in the domain.

Def 1.2.4. The *existential quantification* of $P(x)$ is the proposition

"There exists an element x in the domain such that $P(x)$."

We use the notation $\exists x P(x)$ for the existential quantification of $P(x)$. Here $\exists$ is called the **existential quantifier**. The domain must always be specified when a statement $\exists P(x)$ is used. The meaning of

$\exists P(x)$ changes when the domain changes. without specifying the domain, the statement $\exists P(x)$ has no meaning. The existential quantifier $\exists P(x)$ is read as, "There is an x such that $P(x)$, "There is at least one x such that $P(x)$", or "For some $xP(x)$".
Notes. Generally, an implicit assumption is made that all domains of discourse for quantifiers are nonempty. If the domain is empty, the $\exists P(x)$ is false whenever $P(X)$ is a propositional function because when the domain is empty, there can be no element in the domain for which $P(x)$ is true.
For finite domains, the existential quantifier can be thought of as a disjunction. For n a natural number, $\exists nP(n)$ can be rewritten as $P(0) \vee P(1) \vee \cdots \vee P(n)$.

1.2.5 Other Quantifiers

You will sometimes see other quantifiers but the only one which occurs often enough to get notice is the uniqueness quantifier:
Def 1.2.5. The *uniqueness quantification* of $P(x)$ is the proposition:

"There exists exactly one element x in the domain such that $P(x)$".

We use the notation $\exists!$ for the uniqueness quantification.

1.2.6 Quantifiers with Restricted Domains

An abbreviated notation is sometimes used to specify some subset of the domain. For example $\forall x < 0(x^2 > 0)$ in the domain of real numbers places the restrictive clause next to the quantifier.

The truth of an existential statement is proven by demonstrating one object that makes the predicate true. We call that object the *witness*. To show that an existential statement is false we must show that no such object from the domain will make the predicate true. We must prove a negative. For example, if I assert that in the reals, $\exists x : x = x + 1$, you cannot produce any such x. You could argue that the predicate is a contradiction since the equation $x = x + 1$ is equivalent to $1 = 0$ which is clearly a contradiction. But many existential statements are not that easily proven false.

Def 1.2.6 (Binding Variables)**.** When a variable has been assigned a value, we say the value has been **bound** to the variable. Any variable that has not yet been bound to a value is said to be **free**. The **scope** of the binding is controlled by the use of parentheses or other marks. The value of a variable is also bound by the use of a quantifier and the variable is either within or outside the scope of that quantifier.

Care must be taken with quantified statements regarding the scope of the binding. When a variable is bound to a quantifier, all instances of that variable within the scope are bound. Any use of that variable outside the scope is unbound, or free. Within the scope of the binding a variable can be changed with no effect on the logic. For example $\forall a(Q(a) \wedge P(a))$, the quantifier binds a. We can rewrite this as $\forall b(Q(b) \wedge P(b))$. But if the expression is $\forall a Q(a) \wedge P(a)$, the first predicate is bound to the universal quantification but the second is not. The second use is *unbound* or *free*. Another example would be $\exists x : (x + y > x)$ the variable x is bound but the variable y is free.

1.2.7 Precedence of Quantifiers

We must update the precedence of the operators. Parentheses are still the highest. But quantifiers come before propositional operators. The order of precedence places all propositional operators ahead of quantifiers. All quantifiers are equal precendence.

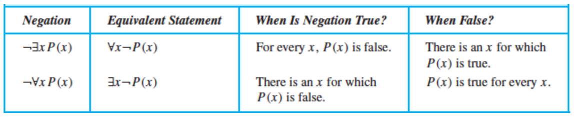
1.2.8 Logical Equivalences Involving Quantifiers

Def 1.2.7. Statements involving predicates and quantifier are *logically equivalent* if and only if they have trhe same truth value no matter which predicates are substitued into these staeent and which domain of discourse is used for the variables in these propositionan functions. We use the notation $S \equiv T$ to indicate that two statements S and T involving predicates and quantifiers are logically equivalent.

$\forall x(P(x) \wedge Q(x)) \equiv \forall x P(x) \wedge \forall x Q(x)$ regardless of the domain.

1.2.9 Negating Quantified Expressions

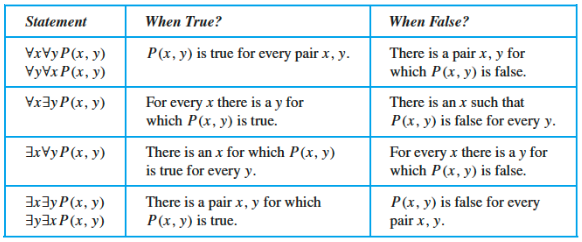
$\neg\forall x P(x) \equiv \exists x \neg P(x)$ is a logical equivalence. $\neg\exists x P(x) \equiv \forall x \neg P(x)$ is a logical equivalence. These are DeMorgan's laws again, this time for quantified statements.



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

Table 1.7: DeMorgansForQuantifiedExpressions

1.2.10 Nested Quantifiers and their Order



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

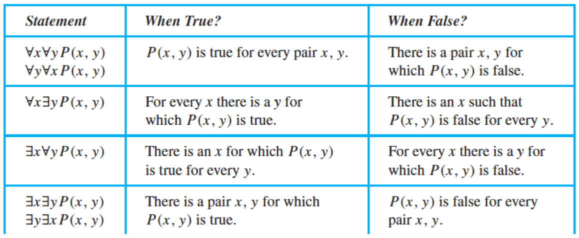
Table 1.8: Quantification Of Two Variables

It is important to note that the order of the quantifiers can make a difference. For example is this statement true or false for the domain of integers? $\forall x \exists y : (x + y = 10)$. This is easily proven true with a bit of algebra. But what about $\exists y \forall x : (x + y = 100)$. This is false since there is no such integer that will always give 10 when added to any other integer.

Universal quantifiers at the outermost level can be omitted, i.e., free variables are interpreted as universally quantified at the outermost level. Quantifiers can be applied to more than one variable at once (e.g., forall x,y). The infix equality sign (e.g., x = y) can be used as a shorthand for the equality predicate (e.g., equals(x,y)).

1.2.11 Negating Nested Quantifiers

Table-1-5-1-QuantificationsOfTwoVariables



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

Table 1.9: Quant Of 2 Var

1.2.12 Thinking of Quantification as Loops

THINKING OF QUANTIFICATION AS LOOPS In working with quantifications of more than one variable, it is sometimes helpful to think in terms of nested loops. (Of course, if there are infinitely many elements in the domain of some variable, we cannot actually loop through all values. Nevertheless, this way of thinking is helpful in understanding nested quantifiers.) For example, to see whether $\forall x \forall y P(x, y)$ is true, we loop through the values for x, and for each x we loop through the values for y. If we find that $P(x, y)$ is true for all values for x and y, we have determined that $\forall x \forall y P(x, y)$ is true. If we ever hit a value x for which we hit a value y for which $P(x, y)$ is false, we have shown that $\forall x \forall y P(x, y)$ is false. Similarly, to determine whether $\forall x \exists y P(x, y)$ is true, we loop through the values for x. For each x we loop through the values for y until we find a y for which $P(x, y)$ is true. If for every x we hit such a y, then $\forall x \exists y P(x, y)$ is true; if for some x we never hit such a y, then $\forall x \exists y P(x, y)$ is false.

To see whether $\exists x \forall y P(x, y)$ is true, we loop through the values for x until we find an x for which $P(x, y)$ is always true when we loop through all values for y. Once we find such an x, we knowthat $\exists x \forall y P(x, y)$ is true. If we never hit such an x, then we know that $\exists x \forall y P(x, y)$ is false.

Finally, to see whether $\exists x \exists y P(x,y)$ is true, we loop through the values for x, where for each x we loop through the values for y until we hit an x for which we hit a y for which $P(x,y)$ is true. The statement $\exists x \exists y P(x,y)$ is false only if we never hit an x for which we hit a y such that $P(x,y)$ is true.

$S(x) : xisaman$, $M(x) : xismortal$. Express "all men are mortal, Socrates is a man, therefore Socrates is mortal". $S(Socrates) \forall x S(x) \rightarrow M(x)$ (move to proofs where inference rule is discussed.)

1.2.13 Logic Programming

An important type of programming language is designed to reason using the rules of predicate logic. Prolog (from Programming in Logic), developed in the 1970s by computer scientists working in the area of artificial intelligence, is an example of such a language. Prolog programs include a set of declarations consisting of two types of statements, Prolog facts and Prolog rules. Prolog facts define predicates by specifying the elements that satisfy these predicates. Prolog rules are used to define new predicates using those already defined by Prolog facts.

Chapter 2

Introduction to Proofs

For those who are already comfortable writing formal arguments and mathematical proofs, this chapter can be skipped. But many students approach the subject of proofs with trepidation and this chapter is included to connect the prior subject of formal logic and all the chapters that follow which develop their material with more formal presentations including proofs.

To introduce proofs we will depend upon a few formal definitions that you learned in high school and prove some other properties using those definitions.
Def 2.0.1 (Even and Odd Integers)**.** An even integer is one that is equal to twice some other integer. An odd integer is one that is equal to twice some other integer plus 1.

But we must reiterate this text is not designed to teach but to provide a support to the learning and later reference to the material within our scope. This is seen here in that other than a few examples needed to briefly illustrate some points, one will never learn to prove anything from this chapter. Instead it is a brief survey of basic terminology and cartoons of essential arguments to quickly remind the reader how to structure a proof of that type. We make some attempt in later chapters to give specific examples of the more interesting ways we reason about material using some of the techniques we present in its cartoon form in this chapter.

2.1 Inference

2.1.1 Valid Arguments in Propositional Logic

Def 2.1.1 (Argument)**.** We define an *argument* as some series of statements meant to convince the reader of the truth of some proposition we call the *conclusion*.

2.1.2 Rules of Inference for Propositional Logic

We already saw how we can show the logical equivalence of two compound propositions using truth tables. However this becomes impractical as the number of atomic propositions grows. It also becomes difficult to follow the argument if this is the only way used to demonstrate the logical equivalence. In a high school geometry course you have already seen a different way that depends upon a series of statements and certain established rules of inference. We show how these are built from the logic.

One of the most basic is represented by this equivalence:

$$p \wedge (p \rightarrow q) \equiv q$$

This statement can be read as "Given that p is true with the the conditional statement if p is true that q must be true then it is logically equivalent to the value of q." When you construct the truth table you see this logical equivalence. This essential point of logic is called Modus Ponens, which is short for the Latin phrase "*modus ponendo ponens*" which means "mode that affirms by affirming" and goes back to antiquity. It is also called the law of detachment. In more advanced material on logic this is considered beyond the logic presented up to now but we are not going to let that detail trouble us. It is the first of many argument forms that are used in argumentation.

In antiquity logic came from rhetoric and the recognition that it was the rhetorical form that sometimes gave the truth or falsity of an argument and not the meaning of the propositions. That is why we jumped over natural language and the translations back and forth between symbolic logic and natural language. While an argument may sound persuasive, sometimes when reduced to its symbolic equivalent we can find a flaw in the argument.

The first thing you will notice is that this presentation of these rules of valid argumentation, or rules of inference, are not laid out like the compound propositions we saw before. In fact they are laid out in a way that mimics natural language by mimicing a sentence structure.
For example I can state modus ponens in words as:
It is the case that whenever p is true that q is also true
It is the case that p is true
Therefore it is the case that q must be true.
It is customary to place a horizontal rule between the argument and the conclusion which is stated following the word, "therefore". Mathematicians have a symbol to mean therefore which is $\therefore$.

2.1.3 Using Rules of Inference to Build Arguments

Def 2.1.2 (Argument Form)**.** An argument in propositional logic is a sequence of propositions. All but the final proposition in the argument are called premises and the final proposition is called the conclusion. An argument is valid if the truth of all its premises implies that the conclusion is true.

An argument form in proposition logic is a sequence of compoutn propositions involving propositional variables. An arguent form is valid if no matter which particular propositions are substituted for the proposition variables in its premise, th econclusion is true if the premises are all true. (valid v sound??)

If you look you will note that a great many properties in mathematics are stated as either conditional or bi-conditonal statements. "If i is an even integer, then $i+1$ is an odd integer." If you want to mount an argument as to why this assertion must be true, you must demonstrate a series of equivalent statements that cannot be refuted until you reach the conclusion. In this case you will already have some definitions or other propositions that are not refuted, in this

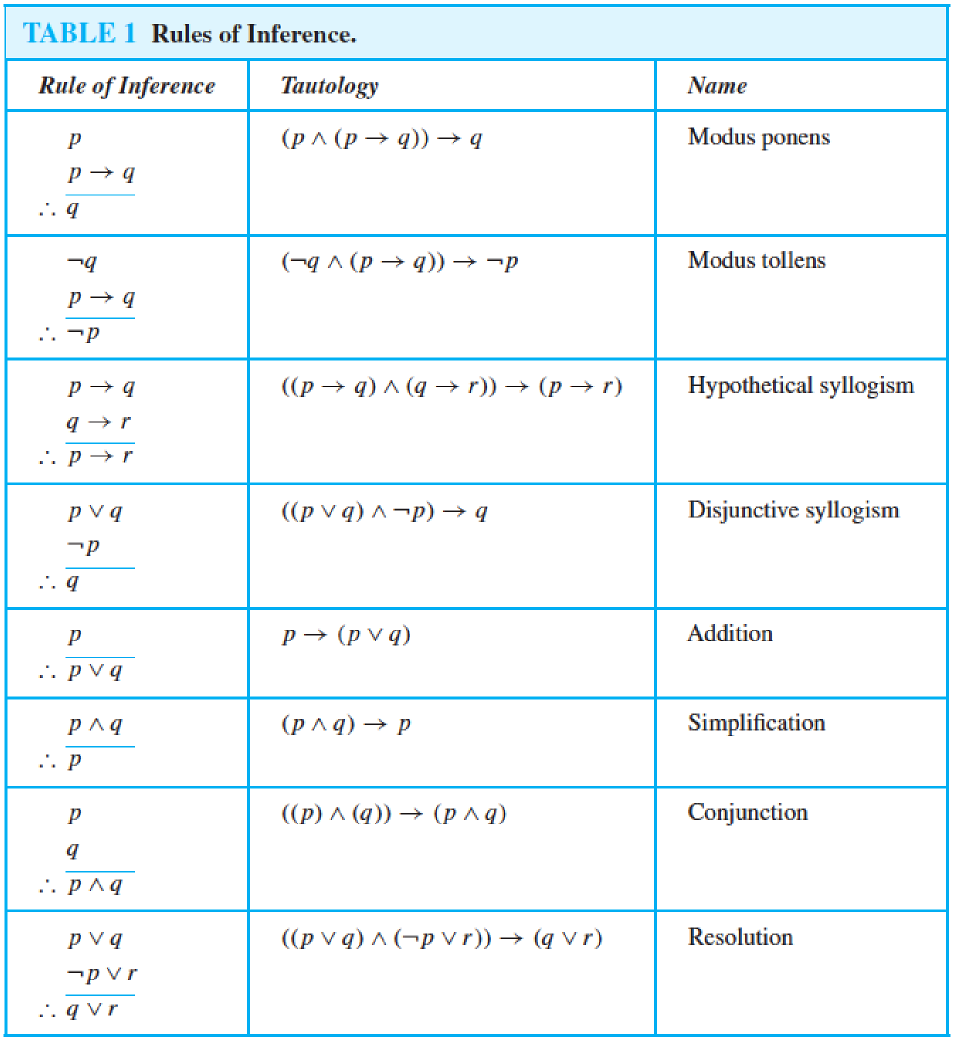


TABLE 1 **Rules of Inference.**

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

Table 2.1: Common Rules of Inference

case the definiton of what it means to be even or odd. You must then offer the next assertion and be prepared to defend it. For example, "well if i is an even integer then there is some other integer which I'll call k such that $i = 2k$ by the definition of what it means to be an even integer." Note that I have given both a new assertion, "there is some integer k such that $i = 2k$" and a justification, "... by the definition of what it means to be an even integer." In a classic two column presentation of the argument you would have one column which has the assert and another which has the justification for that assertion. You may have been taught this form in your high school geometry class. It helps you when you are first learning this style of formal argumentation since it forces you to support your argument (and makes grading so much easier!!!). Continuing our example, my next statement can be "$i + 1 = 2k + 1$, equals plus equals gives equals." We then can observe that "$2k + 1$ is odd, by the definition of odd." Finally we conclude with "Therefore $i + 1$ is an odd integer since it is equal to $2k + 1$" and we have now completed the argument. A classic flourish in the presentation of such an argument is to put Q.E.D. which is the abbreviation for *Quod Erat Demonstrandum*, a Latin phrase that means "that which was to be demonstrated." In textbooks this is often replace with a blackened or open square to signify that we reached the conclusion and the argument is over (■ or □). typographicallycalled a tombstone or Halmos.

Formal arguments like this are called proofs.

Note an important logical point about the prior argument. We claim it is application of the modus ponens inference. But where is the truth of the antecedent? Reconsider the implication, if p then q. We already know that when the antecedent is false we evaluate the implication as true. That is to say in the last example that if it is the case that i is NOT an even integer, that we make no claim about $i + 1$. Once you ignore the two cases where the antecedent is false you only have two cases where it is true. For the conditional statement to be true we must argue that the case where the consequent is true is necessarily true and that the consequent must necessarily never be false. This allows us to accept the antecedent as true since we are only interested in conditionals with a true antecedent. If this does not make sense on the first reading, don't fret. But sooner or later it will occur to you that many conditional statements apply modus ponens without dealing with false antecedents.

You will rarely see advanced mathematics texts with proofs set in a two column format. After your first introduction you are expected to be able to use a more natural prose method of presenting your formal arguments with no loss of precision in your logic. As you progress your need to justify each inference is reduced as you are writing for a more sophisticated reader who can make the most basic leaps of inference without aid. But like all non-fiction writing, you must know your audience. For a class like this one the expected reader needs to be a fellow student who may be slightly ahead or behind you. If you have any doubt that the rules of inference used are not obvious you need to put in remarks that do not interrupt the flow of the argument but support why the next assertion is valid.

From this point on in these notes where arguments are needed, you will see them presented in this fashion and we have tried to give you good examples of how to both structure your argument as well as lay them out on a page in a way that is consistent with tradition so as to make them easily understood by another mathematician.

2.1.4 Resolution

Computer systems have come a long way in being able to do proofs. While it is not possible to do it in the general, they have been successful in many applicaitons. It should be noted that they make heavy use of one rule of interence called **resolution**. That rule of interence uses this tautology:

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

The consequent is called the resolvent.

2.1.5 Fallacies

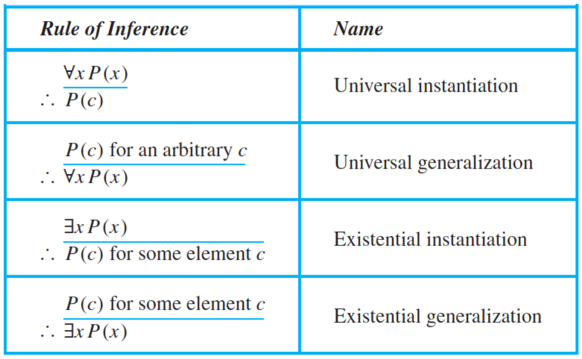
Fallacies are important in rhetoric. Some mentioned in a rhetoric course are logical fallacies and not just rhetorical fallacies. It is important that you recognize the logical fallacies when they are presented. In discussing conditional statements we saw one which is mistaking the converse for the contrapositive. While $[(p \rightarrow q) \wedge p] \rightarrow q$ is a valid inference, it is NOT the case that $[(p \rightarrow q) \wedge q] \rightarrow p$. This second statement is the converse of the conditional assertion and we saw that the truth tables do not match. In rhetoric this is called the **fallacy of affirmig the conclusion**.

In a very similar way, given the conditional statement $(p \rightarrow q)$ and $\neg p$ does NOT allow you to assert q. This is called the **fallcy of denying the hypothesis**. One of the strengths of symbolic logic over rhetoric is that seeing these fallacies is much easier in their symbolic form than when they are made in natural language.

One other common fallacy is that of **affirming the disjunct**. Recall that for the logical *or*, one or both of the terms may be true to give a true disjunction. But some people falsely argue that if one of the disjuncts is true the other must be false. This confuses the disjunction with the exclusive or.

2.1.6 Rules of Inference for Quantified Statements

Up to now we have only looked at the rules of valid inference for propositional statements. Including predicate logic adds a few more. They are summarized in Table 2.2



Rule of Inference	*Name*
$\forall x P(x)$ $\therefore P(c)$	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

Table 2.2: RulesOfInferenceForQuantifiedStatements

2.1.7 Combining Rules of Inference for Propositions and Quantified Statements

Here is one example of how the rules of inference for propositional logic can be applied with rules of inference for quantified statements as well.

Given: For all positive integers n, if n is greater than 4, then n^2 is less than 2^n, Prove:

$$100^2 < 2^{100}$$

One inference rule that uses both propositional and predicate logic is **universal modus ponens**. This rule tells us that if $\forall x(P(x) \to Q(x))$ is true, and if $P(a)$ is true where a is some specific member of the domain, then $Q(a)$ will also be true. Here is an example:
Example 1. Given that for all positive integers n, if n is greater than 4, then n^2 is less than 2^n, prove that $100^2 < 2^{100}$.

Proof. Let $P(n)$ denote "$n > 4$" and let $Q(n)$ denote "n^2". The statement "For all positive integers n, if n is greater than 4, then n^2 is less than 2^n can be represented by $\forall n(P(n) \to Q(n)$, where the domain consists of all positive integers. We start by assuming that $\forall n(P(n) \to Q(n)$ is true. Note that $P(100)$ is true because $100 > 4$. It follows by universal modus ponens that $Q(n)$ is true, namely that $100^2 < 2^{100}$. □

Note that this proof proceeded from its premises to its conclusion by applying one rule of inference or logic sequentially until the conclusion was affirmed. This is an example of a **Direct Proof**. Here is another example.
Example 2. Prove that if n is an odd integer, then n^2 must be odd.

Proof. Note that this problem states $\forall n(P(n) \to Q(n))$, where "$P(n)$ is an odd integer" and "$Q(n)$ is n^2 is odd". The proof procedes by proving the implication. We begin the proof by assuming the antecedent of the implicaiton, that $P(n)$ is true and that n is an odd integer. By the definition of an odd integer we have $\exists k : n = 2k + 1$, where k is an integer. If we square both sides of the equation

$n = 2k+1$ we get $n^2 = (2k+1)^2$ which gives us $4k^2+4k+1$. Factoring out 2 gives us $2(2k^2+2k)+1$. We know that k is an integer, that squaring an integer gives us an integer and that $2k$ will also be an integer and that adding an integer to an integer gives us an integer. Therefore we know that $(2k^2+2k)$ is an integer. This shows that the right side of the equation has the form twice some integer plus one which we know to satisfy the defintion of what it means to be odd. This proves that n^2 must be an integer satisfying the proof obligation to show that squaring an odd number must necessarily give us an odd integer. □

There are times when the attempt to perform a direct proof of an implication seems to lead to a dead end. When this happens it is sometimes easier to prove the contrapositive of the implication. This form of proof is called **Proof by Contraposition** as we demonstrate here:

Example 3. Prove that if n is an integer and $3n+2$ is odd, then n is odd.

Proof. We might proceed with the usual practice of assuming the antecedent and trying to derive the conclusion. This gives us that $3n+2$ is odd which means that $3n+2 = k+1$ where k is some integer. We want to prove that n is odd but there does not seem to be an easy way to proceed. We will try proving the contraposition instead. First we must state the contrapositive of the implication which we state as, if it is not the case that n is odd, then it is not the case that $3n+2$ is odd.$(\neg q \to \neg p)$ We can use the fact that if something is not odd it must be even and restate the contraposition as, if n is even, then $3n+2$ is even. If n is even it can be restated as $2k$ where k is some integer. Substituting we can restate the conclusion as $3(2k)+2$ or $6k+2$. We can factor out the 2 giving $2(3k+1)$ which fits the definition of an even number proving the consequent and concluding the proof. □

Example 4. Prove: If pigs can fly, then I sing better than Justin Bieber.

Proof. We are asked to prove another implication. In this case the antecedent of the implication is clearly false since we know that pigs cannot fly. Recall that an implication with a false antecedent is always true. A false premise can be used to "prove" anything. □

This is an example what is called a vacuuous proof. Here is a more interesting example.
Example 5. Prove that if $n = ab$, where a and b are positive integers, then $a \leq \sqrt{n}$ or $b \leq \sqrt{n}$

Proof. Because there is no obvious way of showing that $a \leq \sqrt{n}$ or $b \leq \sqrt{n}$ directly from the equation $n = ab$, where a and b are positive integers, we attempt a proof by contraposition. The first step in a proof by contraposition is to assume that the conclusion of the conditional statement “If $n = ab$, where a and b are positive integers, then $a \leq \sqrt{n}$ or $b \leq \sqrt{n}$” is false. That is, we assume that the statement $(a \leq \sqrt{n}) \vee (b \leq \sqrt{n})$ is false. Using the meaning of disjunction together with De Morgan’s law, we see that this implies that both $(a \leq \sqrt{n})$ and $(b \leq \sqrt{n})$ are false.

This implies that $a > \sqrt{n}$ and $b > \sqrt{n}$. We can multiply these inequalities together (using the fact that if $0 < s < t$ and $0 < u < v$, then $su < tv$) to obtain $ab > \sqrt{n} \ldots \sqrt{n} = n$. This shows that $ab \neq n$, which contradicts the statement $n = ab$. Because the negation of the conclusion of the conditional statement implies that the hypothesis is false, the original conditional statement is true. Our proof by contraposition succeeded; we have proved that if $n = ab$, where a and b are positive integers, then $a \leq \sqrt{n}$ or $b \leq \sqrt{n}$. □

The last section introduced a great deal of terminology and basic logical structure. But many feel that real mathematics, including logic, does not begin until one begins to reason about the material and attempt to determine new truths from the truths that are either accepted as true or which can be arrived at through reason. This chapter is a fast introduction to this mode of reasoning.

The chapter introduces the basic terminology of deductive reasoning and its presentation in the form of formal proofs. Various proof methods and strategies are summarized with classic examples. The remainder of the book will build upon this basis and demonstrate good proof style.

2.2 Introduction to Proofs

2.2.1 Why Write Proofs?

Mathematics can be distinguished from other scientists by the kind of reasoning that we do. For reasons that this course explains, computer science is mostly a form of applied mathematics and therefore learning how to do mathematical proofs is a needed skill for a true computer scientist. Reasoning is often split into two types, inductive and deductive. Inductive reasoning is that which tries to generalize from a series of observations while deductive uses logic to conclude that an assertion must be true based on other statements that are already accepted as true. When someone gives a series of assertions that each build on what has come before and using accepted reasoning, we say they have proven their assertion and that the argument is a valid proof of the assertion. This has been a part of logic since at least Euclid's geometry and continues to be a cornerstone of undergraduate education. This chapter will be a poor substitute for a full semester course on the subject but should at least prepare the student for more formal work.

2.2.2 What Am I Allowed to Assume for this Proof?

Given the emphasis on using accepted truths as premises, the student quickly finds themselves asking, what am I allowed to assume as either a premise or a rule of inference? The last chapter gave the most common ones from logic but often the student must use some high school algebra. The simple answer is that anything you could do in high school as valid algebra can be done in a proof for this course. More formally it must be something that can be justified by some argument to authority, which will be some previously published property. And of course it needs to be applied correctly. What cannot be done is to introduce a truth that in any way assumes the conclusion one is trying to make. We deal with that in this chapter when talking about fallacies.

Some students are uncomfortable with the handwaving of allowing that which was admitted in high school math. For reasons beyond the scope of this book it is not possible to put all of arithmetic onto a firm axiomatic basis. But to help the nervous here is a brief list of high school algebra:

- Closure Laws for Addition and Multiplication
- Associative Laws for Addition and Multiplication
- Commutative Laws for Addition and Multiplication
- Additive and Multiplicative Identity Laws
- Identity Elements Axiom
- Inverse Laws for Addition and Multiplication
- Distributive Laws
- Trichotomy Law
- Transitivity Law
- Additive Compatibility Law
- Multiplicative Compatibility Law

For this text we take one more axiom that may not always be covered in a high school class but which we will need in the section on induction. That is called the *completeness property*. This includes the definition of the upper bound and least upper bound.
Def 2.2.1 (Completeness Property)**.** Every nonempty set of real numbers that is bounded above has a least upper bound.

informal proofs, theorem, propositions, facts, results, proof, axioms, postulates, lemma, corollary, conjecture

2.2.3 Understanding How Theorems Are Stated

Often there is an **assertion** we want to make, some **proposition** which must either be true or false. Assertions often start as **conjectures**, propositions which we do not yet know are either true or false and which we would like to have a proof but do not yet have one. We often start with basic definitions which are essentially **axioms** that we accept as true without question. We must then construct a series of statements that each make new assertions by using the prior axioms and the **accepted rules of inference**. If we are successful, the final assertion is the proposition we wanted to prove which we call the **conclusion** of the argument. When we are presenting the assertion with the list of infered assertions that lead to the conclusion, we call this a **proof**. **Theorems** are nothing more than assertions we find helpful presented with their arguments as to why the assertion must be true.

It is not uncommon that something accepted as an axiom is later found to be untrue, that is false. Or that a proof which depended upon a prior theorem which is found to contain a flaw. This leads to an important point about proofs, one can present a **valid** argument, one which uses only valid rules of inference, yet is found to depend upon one or more premises that are false. The argument is still said to be valid but the argument unsound. To be a **sound** argument one must have valid reasoning plus true premises.

2.3 Common Argument Forms

2.3.1 Direct Proofs

In the chapter on Logic, there was a table of logical equivalences, Table 1.5. The equivalence was demonstrated by showing that no matter the truth condition of the propositions, both expressions would evaluate to the same truth value. That is, the truth tables for both columns under the expressions matched on every row. This is a proof by truth table. However the number of rows will grow exponentially with the number of propositions so this kind of proof cannot be done by hand and often not even by machine for large numbers of propositions.

This leads to the more common style of proof which is the series of statements grounded on the prior truths but making a new and equivalent statement. We stated the rule of inference known as Modus Ponens which says that given a true conditional statement and the fact that the antecedent of the conditional is known to be true, that the consequent of the conditional must be true.

$$p \wedge (p \to q) \equiv q \text{ (Modus Ponens)}$$

2.3.2 Direct Proofs

Theorem 2.3.1. The sum of two even numbers is even

Two-Column Formal. Let n,m be the two even numbers (premise given)
Then n=2*k and m=2*j where k and j are integers (def of even)
Then n+m=2*k + 2*j (basic law of arithmetic)
Then n+m=2*(k+j) (associative law of arithmetic)
Then n+m is even (def of even)

□

Theorem 2.3.2. The sum of two even numbers is even.

Prose Style. Let a and b be the two even integers. We know by the definition of even that both can be expressed as twice two other integers c and d such that $a = 2c$ and $b = 2d$. Then the sum of $a + b$ can be expressed as $2c + 2d$. After factoring out the 2 we see that the sum fits the definition of even. □

2.3.3 Proof by Contraposition

indirect proofs, vacuous and trivial proofs

2.3.4 Proofs by Contradiction

A proposition must be either true or false (rule of excluded middle). If it can be shown that a proposition cannot be false, then it must be true. To do this assume the opposite of what is to be proven and then show that it leads to a contradiction. Recall that a contradiction is false regardless of the truth values of the propositions. Once this is done you can assert that the contradiction is false and then conclude that the hypothesis must be true.

2.3.5 Exhaustive Proof and Proof by Cases

without loss of generality (WLOG)

2.3.6 Mistakes in Proofs

begging the question, circular reasoning

2.3.7 Looking for Counterexamples

2.3.8 Proof Strategy in Action

2.3.9 Additional Proof Methods

2.4 Proof Styles

2.4.1 Proof by Truth Table

If you can show that two statements always have the same truth value regardless of the truth value of the variables in the expression, you have demonstrated logical equivalence and one can always be substituted for the other where ever it appears. This can be done by creating a truth table with each expression at the top of a column and a row for every combination of truth values of the atomic propositions and show that the two columns have the same values on every row. But since the number of rows in a truth table grows exponentially with the number of atomic propositions, this method is only useful for small equivalences.

2.4.2 Two-Column Proofs

High-school geometry teaches a two-column method. Some algebra classes use a similar approach to evaluating algebraic expressions. Most introductions to proofs begin with this form of proof presentation. Each line is a statement that is derived deductively from the lines above it. This is less tedious than a truth table proof and seems to correspond with how humans can follow detailed logic. In the most rigorous approach each line must list the inference explicitly in the right hand column to justify the statement just made. This has the advantage of making a student carefully understand how they are using logic to make the statements and easier for a grader to see that valid logic has always been applied. But the insistence on listing every rule applied makes the proof tedious for longer proofs as trivial points of logic clutter the work.

2.4.3 Prose Style Proofs

Traditional mathematicians used a prose style of proof. The best of this technique applies all the rules of non-fiction writing. You make reasonable assumptions of what your reader will easily follow and which leaps of inference require some parenthetic remark to aid the reader. Good proof presentations in this form use natural language in a rigorous way yet will still suffer from an occassional slip in inference. Yet by the end of the first introduction to proofs this is the way undergrads are expected to present their proofs. The vital point to recall is that any proof in this form could be reduced to a two-column proof if demanded. Experienced graders will look for large leaps and critically examine the inference to see if it is valid. If you feel it is correct but cannot justify it to yourself, take the time to break it into two or more steps that you can see the rule application.

2.4.4 Other Proof Styles

There is an example of proof by combinatorial argument and a couple others in this and other texts. After the student becomes comfortable with these basics of formal argumentation we believe she will be well prepared to expand her ability to learn new techniques of mounting formal arguments.

Machine proofs are an area of research and many more rigorous proof styles have been described in the past 100 years. This is beyond the scope of an introductory proof class but we mention some of the notational forms used to prepare you for more advanced work.

Note that many proofs are presented in this fashion and will use a notation called a turnstile $\vdash$:

2.4.5 Proof Strategies

2.4.6 Looking for Counterexamples

Any assertion using a universal quantification can be refuted by citing a single counterexample.

2.5 Formal Methods

An important application of proof techniques is found in the study of what is called formal methods which includes proofs of program correctness

2.5.1 Program Verification

A program is said to be correcrt if it produces the correct output for every possible input. A proof of correctness for a program consists of two parts. The first part shows that the correct answer is obtained if the program terminates. This part is said to establish the partial correctness of the program. The second part proves that the program always terminates. When working with proofs of program correctness we use two propositions. The first, called the pre-conditions, gives a proposition that all input values must satisfy. In addition the second proposition is called the post-condition and if the program has correctly computed the value it will evaluate to true. The pre- and post-conditions are sometimes called the initial and final assertions.

Def 2.5.1. A program, or program segment, S is said to be partially correct with respect to the initial assertion p and the final assertion q if whenever p is true for the input values of S and S terminates, then q is true for the ouptut values of S. The notation $p[S]q$ indicates that the program, or program segment, S is partially correct with respect to the initial assertion p and the final assertion q. The notation $p[S]q$ is known as a *Hoare triple*.

Rules of Inference composition rule Conditional Statements Loop Invariants

Chapter 3

Sets

3.1 Set Definition

Def 3.1.1. A *set* is an unordered but well defined collection of objects which are called the *elements/members* of the set. The objects in a set are called the *elements* or *members*, of the set. A set is said to *contain* its elements. We write $a \in A$ and say, “a is an element of the set A” to mean that A contains a and $a \notin A$ to mean that the element a is not contained by the set A.
Notes. Sets are unordered. {a, b} is the same as {b, a}. The number of times an object is enumerated makes no difference, it is still one element.

3.2 Set Specification

3.2.1 Set Enumeration

The easiest way to describe small sets is to enumerate (list) the elements. This is done by writing the elements between braces with a comma between them. For example let V be the set of English vowels. We can write $V = \{a, e, i, o, u\}$ to define the set V. If we want to talk about the positive odd integers less than 10 as the set O, we can define it as the set $\{1, 3, 5, 7, 9\}$. The set

$M = \{1, "1", \text{my dog Rover}, \text{red-head}\}$ can be a set. The notation $a_1, a_2, a_3 \in A$ is the same as $a_1 \in A, a_2 \in A, a_3 \in A$. We can start a pattern and use the ellipses symbol to indicate that the reader should infer the pattern. R={3,6,9,12,15, . . . }. $\{0, 1, 2, 3, \ldots, 100\}$ The enumeration can be a description of the elements. {addresses on Pine Street} defines a set of addresses that are on Pine Street. $O = \{\text{positive odd integers less than 10}\}$.

3.2.2 Set Builder Notation

Set comprehension, set intension. Three parts, a variable, a colon or vertical bar and a logical predicate.

$\{a|$ a is a positive integer }

or

E : is even, $A = \{a|E(a)\}$
We read this as set A is defined as the set of all a such that a is even. The vertical bar is read "such that". To the left are the variables that represent the set members and to the right is the condition that all members must satisfy.

Example: $A = \{x|\neg E(x), x < 10\}$
Notes. Predicates separated by commas are implied conjunction.

Sometimes we restrict the domain of a quantified statement explicitly by making use of a particular notation. For example, $\forall x \in S(P(x))$ denotes the universal quantification of $P(x)$ over all elements in the set S. In other words, $\forall x \in S(P(x))$ is shorthand for $\forall x(x \in S(P(x)) \rightarrow P(x))$ Similarly, $\exists x \in S(P(x))$ denotes the existential quantification of $P(x)$ over all elements in S. That is, $\exists x \in S(P(x))$ is shorthand for $\exists x(x \in S \wedge P(x))$.

3.3 Common sets and their Notation in Mathematics and Computer Science

$\mathbb{N} = \{1, 2, 3, \ldots\}$the set of **natural numbers**
$\mathbb{Z} = \{\cdots - 3, -2, -1, 0, 1, 2, 3, \ldots\}$The set of **integers**

$\mathbb{Q} = \{\frac{p}{q}|p \in \mathbb{Z}, q \in \mathbb{Z}, q \neq 0\}$The set of **rational numbers**
$\mathbb{R} = \{$the set of **real numbers**$\}$
$\mathbb{C} = \{$the set of **complex numbers**$\}$
$\mathbb{B} = \{0, 1\}$ the set of **a set of binary symbols**
Notes. We can restrict the universe from which the objects are drawn to the right of the "|" like so:
$O = \{\mathbb{Z}^+|x$ is odd and $x < 10\}$
This is read as "The set O is equal to the set of all x drawn from the set of positive integers such that x is less than 10." The superscript indicates that only the positive members of the domain are to be considered. Zero may or may not be considered a natural number depending upon the author. For this course we accept zero as a natural number and use the notation $\mathbb{Z}^+$ for the set of positive integers. The special font used for these special sets is called Blackboard Bold. Z is used for integers because the German word for numbers is Zahlen.

A set is an object. Therefore we can have a set which contains other sets:

$$\text{Basic data types} = \{\mathbb{B}, \mathbb{N}, \mathbb{Z}, \mathbb{R}\}$$

This fact leads to interesting paradoxes which mark the difference between what is known as Naive Set Theory and more advanced forms.
Notes. Sets and data types in programming languages are related. There is a shortcut used sometimes to assert a universal truth for a set. For example $\forall x \in S(P(x))$ which asserts that all elements of the set S have property P. It can also be stated as $\forall x(x \in S \rightarrow P(X))$

3.4 Truth Sets

Def 3.4.1 (Truth Sets)**.** Given a predicate P, and a domain D, we define the **truth set** of P to be the set of elements x in D for which $P(x)$ is true. The truth set of $P(x)$ is denoted by$\{x \in D|P(x)\}$.

3.5 Set Equality

Def 3.5.1 (Set Equality)**.** Two sets are *equal* if and only if they have the same elements. That is, if A and B are sets, then A and B are equal if and only if $\forall x(x \in A \leftrightarrow x \in B)$. We write $A = B$ if A and B are equal sets.
Notes. Some authors will use $\subset$ specifically to designate a proper subset and use $\subseteq$ for subsets that could be equal. We only use $\subset$ in these notes to indicate either a proper subset or an equal set.
Def 3.5.2 (Empty Set)**.** A set that contains no elements is called *null set* or *empty set*and is denoted by $\emptyset$. The empty set can also be denoted by {}

3.6 Subsets

If a set is composed of elements of another set, we call the new set a **subset**. The subset may have all the same elements as the first set. If it has fewer elements, we may call it a **proper sub-set**.
Def 3.6.1 (Subset)**.** A set A is said to be a *subset* of B if and only if every element of A is also an element of B. We write $A \subset B$ to indicate that A is a subset of B and $A \not\subset B$ to indicate that A is not a subset of B. $\forall x(x \in A \rightarrow x \in B)$ is true whenever A is a subset of B.
Theorem 3.6.1. *$A = B$ if and only if $A \subset B$ and $B \subset A$*

3.6.1 Intervals of Reals

We frequently talk about subsets of the reals or integers as intervals. There are conflicting notations.

$$\begin{aligned}
(a,b) =]a,b[&= \{x \in \mathbb{R} | a < x < b\}, \\
[a,b) = [a,b[&= \{x \in \mathbb{R} | a \le x < b\}, \\
(a,b] =]a,b] &= \{x \in \mathbb{R} | a < x \le b\}, \\
[a,b] = [a,b] &= \{x \in \mathbb{R} | a \le x \le b\}.
\end{aligned}$$

For intervals that extend to infinity toward the negative or positive infinity, we substitute $-\infty$ for a or $+\infty$ for b.

3.6.2 Integer Intervals

The notation $[a \ldots b]$ when a and b are integers, or $\{a \ldots b\}$, or just $a \ldots b$ is sometimes used to indicate the interval of all integers between a and b, including both. This notation is used in some programming languages; in Pascal, for example, it is used to formally define a subrange type, most frequently used to specify lower and upper bounds of valid indices of an array.

An integer interval that has a finite lower or upper endpoint always includes that endpoint. Therefore, the exclusion of endpoints can be explicitly denoted by writing $a \ldots b-1$,$a+1 \ldots b$, or $a+1 \ldots b-1$. Alternate-bracket notations like $[a \ldots b)$ or $[a \ldots b[$ are rarely used for integer intervals.
Def 3.6.2. We denote the set $\{0,1,\ldots m-1\}$ with the notation $\mathbb{Z}_m$.

3.7 Universal and Empty Sets

Def 3.7.1 (Universal and Empty Sets)**.** The *universal set* or universe is a set that represents some domain of discourse. There is no symbol that everyone accepts as the symbol for the universal set. We adopt $\mathbb{U}$ to designate the universal set. The set which contains no elements is the *null* or *empty set* and is disignated by $\emptyset$.
Notes. Never confuse $\emptyset$ with $\{\emptyset\}$. The first is a set which contains no elements. The second is a set which contains one element and that element is a set which contains no elements. Think of file folders on a disk as sets and files as elements.
Theorem 3.7.1. For any set S, the following are true:
$\emptyset \subset S$,
$S \subset \mathbb{U}$,
$\emptyset$ is unique
For any set $S, \emptyset \subset S \subset \mathbb{U}$

3.8 Set Operators

Def 3.8.1 (Set Union)**.** Let A and B be sets. The **union** of the sets A and B, denoted by $A \cup B$, is the set that contains those elements that are either in A or in B, or in both. $A \cup B = \{x : x \in A \vee x \in B\}$

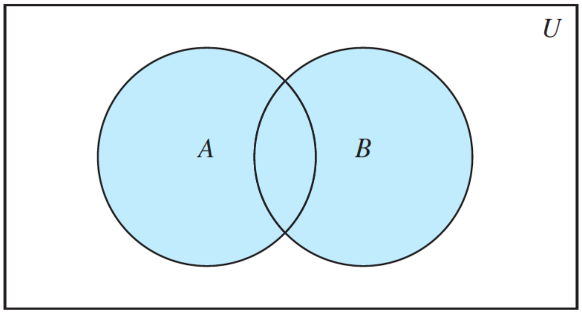


Figure 3.1: Venn Diagram Of A Union B

Def 3.8.2 (Set Intersection)**.** Let A and B be sets. The **intersection** of the sets A and B, denoted by $A \cap B$, is the set containing those elements in both A and B. $A \cap B = \{x : x \in A \wedge x \in B\}$

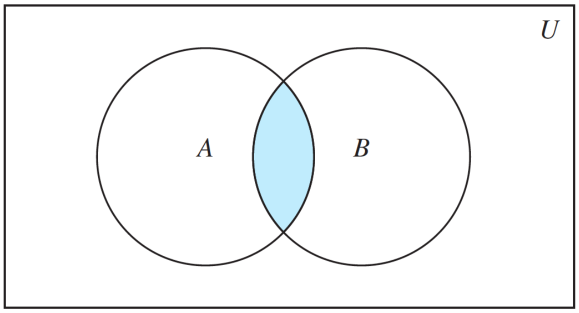


Figure 3.2: Venn Diagram Of The Intersection Of A and B

Def 3.8.3 (Disjoint Sets)**.** Two sets are disjoint if and only if $A \cup B = \emptyset$

Def 3.8.4 (Set Difference)**.** Let A and B be sets. The *difference* of A and B denoted by $A - B$ and sometimes by $A \setminus B$ according to the ISO 31-11 standard, is the set containing those elements that are in A but not in B. The difference of A and B is also called the *complement of B with respect to A*. $A - B = \{x | x \in A \wedge x \notin B\}$.

It is sometimes written $B - A$, but this notation is ambiguous, as in some contexts it can be interpreted as the set of all elements $b - a$, where b is taken from B and a from A.
Def 3.8.5 (Symmetric Difference)**.** The symmetric difference of A and B, denoted by $A \oplus B$, is the set containing those elements in either A or B, but not in both A and B.
Def 3.8.6 (Set Complement)**.** A^C or $\bar{A}$ is called a set complement. It is all elements of the universal set which are not contained in the set A.

3.9 Identities of Set Algebra

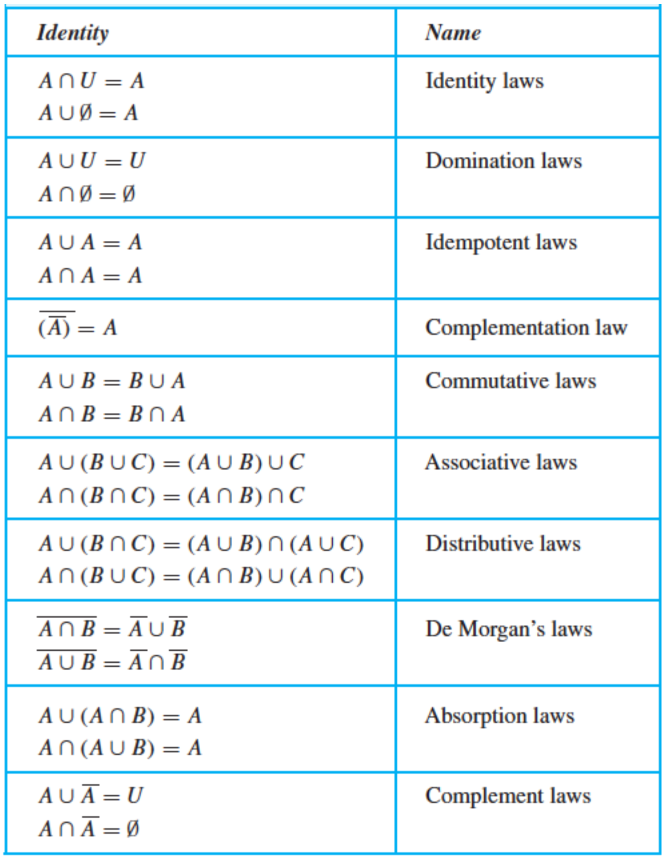
Set operators give identities that can be proven. The following are a list of the most basic identifies and the names they are given. Any of these can be proven using the formal definition of the operator and the rules of equivalence and inference from logic.
Notes. Note the similarity to Identities of Logic

3.10 Venn Diagrams

Diagrams that represent sets as ovals within a square box with or without labeled elements are called Venn diagrams. The outer box represents the universe of discourse, U, for the sets. In a Venn diagram, the universal set is th einterior of a rectangle. If $A \subset B$, then the area of A will be contained within the area of B. If A and B are disjoint, then the area of A will overlap art of the area of B.
Def 3.10.1 (Set Cross Product or Cartesian Cross Product)**.** The ordered n-tuple $(a_1, a_2, ..., a_n)$ is the ordered collection that has a_1 as its first element, a_2 as its second element, . . . , and a_n as its nth element. Two tuples are considered equal if and only if all corresponding elements of the two tuples match.

$$A \times B = \{(a,b) \mid a \in A \wedge b \in B\}$$

Def 3.10.2. Let A and B be sets. The Cartesian product of A and B, denoted by $A \times B$, is the set of all ordered pairs (a,b), where $a \in A$



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

Table 3.1: Set Identities

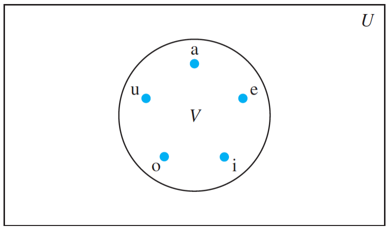


Table 3.2: VennDiagramOfVowels

and $b \in B$. Hence, $A \times B = \{(a,b)|a \in A \wedge b \in B\}$.

$A \times B$
Notes. The elements of a tuple are ordered. (a,b) is not the same as (b,a).
Cross products of sets to themselves can be represented with superscripts. $A^2 = A \times A$.
The cross product is a SET of TUPLES.
Def 3.10.3. The Cartesian product of the sets $A_1, A_2, ..., A_n$, denoted by $A_1 \times A_2 \times \cdots \times A_n$, is the set of ordered n-tuples $(a_1, a_2, ..., a_n)$, where a_i belongs to A_i for i = 1, 2, . . . , n. In other words,
$A_1 \times A_2 \times \cdots \times A_n = \{(a_1, a_2, ..., a_n)|a_i \in A_i \text{ for } i = 1, 2, ..., n\}$.

3.11 Set Cardinality

Def 3.11.1 (Set Cardinality)**.** The size of the set is the number of elements in the set for sets with a finite number of elements. We will defer infinite sets until later. This is called the set *cardinality*. It is denoted by **card**(S) or $|S|$.
Def 3.11.2 (Powerset)**.** A special set called the power set is that set which includes every possible and unique subset of some set S. Note: the easiest way to enumerate the subset is by the number of elements in the set. List all the subsets with zero members (only one, the null set). Then all the subsets of size 1, 2, etc.
Theorem 3.11.1. If the size of set S is m, there are 2^m posible subsets.

Special symbol $\mathscr{P}(A)$ designates the power set of set A.
Notes. the powerset IS A SET of SETS

3.12 Indexed Sets/Indexed Classes of Sets and Generalized Set Operations

Let I be any nonempty set (not necessarily a numeric set), and let S be a collection of sets. An indexing function from I to S is a function

$f: I \to S$. For an $i \in I$, we denote the image $f(i)$ by A_i. Thus we can say:

$\{A_i : i \in I\} or \{A_i\} \in I$, or simply $\{A_i\}$

The set I is called the **indexing set** and the elements of I the **indices**.

When many sets are joined or intersected, we introduce an indexed notation:

$\cup_{i=1}^{n} a_i$

$\cap_{i=1}^{n} a_i$

(give Venn diagram)

Representations of sets, set membership tables.
Def 3.12.1 (Fundamental Product of Sets)**.** Consider a set of sets A_1, A_2, etc that are all unique. Now let A_i^* mean either A_i or A_i^c, that the notation A_i^* either means A_i or it means A_i^c. The fundamental product of the set S is that is the union of all sets denoted by A^*
Notes. there are m sets, 2^n such fundamental products (why?) any two fundamental products are disjoint the union of all the fundamental products is the universal
Def 3.12.2 (Partitions of a Set)**.** A **partition** of a set S is a collection of disjoint nonempty subsets of S that have S as their union. Equivalently we can say the collection of subsets A_i, $i \in I$ where I is an index set) forms a partition if and only if

$$A_i \neq \emptyset \text{ for } i \in I$$

$$A_i \cup A_j = \emptyset \text{ when } i \neq j$$

$$\bigcup_{i \in I} A_i = S$$

3.13 Multisets and Bags

A *multiset* is a set that does not require each object to be unique. It can be represented by a set of pairs with the first element of the pair representing the object and the second representing the *multiplicity* of that object in the multiset. These are sometimes called *bags*

Sometimes the number of times that an element occurs in an unordered collection matters. Multisets are unordered collections of elements where an element can occur as a member more than once. The notation $\{m_1 \cdot a_1, m_2 \cdot a_2, \ldots, m_r \cdot a_r\}$ denotes the multiset with element a_1 occurring m_1 times, element a_2 occurring m_2 times, and so on. The numbers $m_i, i = 1,2,3, \ldots r$ are called the multiplicies of the elements $a_i, i = 1,2,3, \ldots, r$.

3.14 Strings

We noted that a set cross product may not have a numeric sets but may map to some arbitrary set of symbols. The set of English letters can be the set. If we let the symbol Σ represent a set of unique symbols then we can represent the set of all pairs as Σ^2, all triples as Σ^3, etc and we call Σ an **alphabet**. We can extend this concept to a set of strings that do not all need to be the same length. Since members of Σ are distinct and unique, there is no ambiguity in dropping the parentheses when representing the set of strings.

Strings have some special notation and definitions. The string that contains no elements is the empty string, represented by a lower case lambda, λ. A string may be called a word.

3.14.1 Kleene Star notation

Sometimes we wish to discuss all the possible strings that could be constructed We can construct exactly one sequence of length zero from a alphabet Σ which we represent as λ. If the domain of the sequence has m elements, it is possible to construct m strings (words) of length 1, m^2 of length 2, m^3 of length 3, etc. We will often want to talk about strings that can be of any length but only contain elements drawn from the codomain. This will be the union of the set of length zero, the sets of length 1,2,3, ... to infinity. This set of all possible strings drawn from the set is called the Kleene closure and designated K_*

We can talk about binary numbers as strings. $\mathbb{B}^2$ designates the set of binary strings of two bits. {00,01,10,11}
$\mathbb{B}^8$ designates the set of 8 binary strings
$\{00000000, 00000001, 00000010, \ldots, 11111111\}$
$\mathbb{B}^*$ represents the set of all possible binary strings.

3.15 From Paradox to Types

A point that is much better addressed in an advanced class is the fact that the naive theory of sets is bounded by Russell's paradox. To avoid that paradox in programming languages there is a theory of types which permeates programming languages. We make the notational observation here which is used for the remainder of the text. Each variable is drawn from some set. The convention which is largely adopted is to place that set, which we call type, following the declaration of the variable like so: *operand1:real* to designate that the variable *operand1* is understood to contain a number drawn from $\mathbb{R}$. However we quickly get beyond mathematical sets in this text to define variables like $S : graph$ where $graph$ is understood to be a set/type of mathematical graph.

Chapter 4

Functions

Sets are collections. Functions take you from one set to another.

Def 4.0.1 (Function)**.** Let A and B be non-empty sets. A *function* f from A to B is an assignment of exactly one element in B to each element in A. The set A is called the domain of the function and the set B is called the co-domain. The element from the domain, a is called the pre-image and the element b from the co-domain is called the target or image. The set of all images is called the range (note difference from co-domain). Functions are sometimes called **mappings** or **transformations**. The notation $a \mapsto b$ is used to denote the association of a pre-image a to an image b.

Def 4.0.2 (Domain, codomain, preimage, image, range)**.** If f is a function from A to B, we say that A is the *domain* of f and B is the *codomain* of f. If $f(a) = b$, we say that b is the *image* of a and a is the *preimage* of b. The *range*,or *image*, of f is the set of all images of elements of A. Also, if f is a function from A to B,we say that f *maps* A to B.

We denote the image of S by $f(S)$, so

$$f(S) = \{t \mid \exists x \in S(t = f(s))\}$$

We also use the shorthand $\{f(s) \mid s \in S\}$ to denote this set.

Def 4.0.3 (function equality)**.** Two functions are **equal** when they have the same domain, have the same codomain, and map each element of their common domain to the same element in their common codomain.

Note that if we change either the domain or codomain they are different functions.

Note that $f_1 + f_2$) and $f_1 f_2$ are defined for real and integer valued functions.

4.0.1 Set Builder Notation for Functions

4.0.2 Function signature, Function definition

The signature of the function gives the function name, the domain and the co-domain and is written as:

$$f : A \to B$$

where f is the function name, A is the domain and B is the co-domain and is read "the function f from A to B".
assigned_grade: {students} → {grades}
The function definition gives the information needed to determine which element from the co-domain is the image of the element from the domain. This is called the function definition.

$$f(a) = b$$

In pseudocode function signature is represented in the header for a function

$$floor(someValue : real) : integer$$

The function definition is usually given in the block which follows but sometimes the signature can be used by itself as is done with C. Note that the type definitions are equivalent to the domain and codomains of the function.

4.0.3 Function Equality

4.0.4 Functions are subsets of set cross products, maplets

All functions are subsets of the cross product of the domain and co-domain. A listing of all pairs is a valid function definition as is a table. Some authors will note specific mappings from an element a in the domain to the element b in the co-domain with the maplet notation:

$$a \mapsto b$$

Sometimes we want to

domain, co-domain, image, target, maplet, scope function signature versus definition graphic representation, arrow diagrams well defined/proper function, partial functions functional equality graphic representation, analytic geometry (recognizing incomplete functions and non-functions) Function versus operator

$f : S \rightarrow T$ The function f is a function that takes an argument from the set S and gives a result from the set T. S is the domain and T is the co-domain.

For function application we typically write $f(s) = t$ where t is some expression based on s and a method by which we can determine the unique element t that s maps to and can define the function using this notation. For example the function f might double the argument and add one which can be expressed as:

$f(x) = 2 * x + 1$

When x is bound to a value, it results in a maplet from a member of the domain to some member in the codomain.

$f(2) = 5$ or maplet $2 \mapsto 5$

For finite functions note that a function can be fully defined just by listing all the maplets of the function.

4.1 Properties of Functions

Def 4.1.1 (Onto or Surjective)**.** A function f from A to B is called *onto*, or a *surjection*, if and only if for every element $b \in B$ there is an element $a \in A$ with $f(a) = b$. A function f is called *surjective* if it is onto.
Notes. A function f is onto if $\forall y \exists x(f(x) = y)$, where the domain for x is the domain of the function and the doman for y is the codomain of the function.
Def 4.1.2 (One-to-One, Into, Injective)**.** A function f from A to B is said to be *one-to-one*, *into*, or *injective*, if and only if $f(a) = f(a)$ implies that $a = b$ for all a and b in the domain of f. A function is said to be an *injection* if it is one-to-one.
Def 4.1.3 (One-to-One-Correspondence, One-to-One-Mapping, Bijective)**.** A function f from A to B is said to be *one-to-one correspondence*, or *bijection*, if it is both one-to-one and onto.

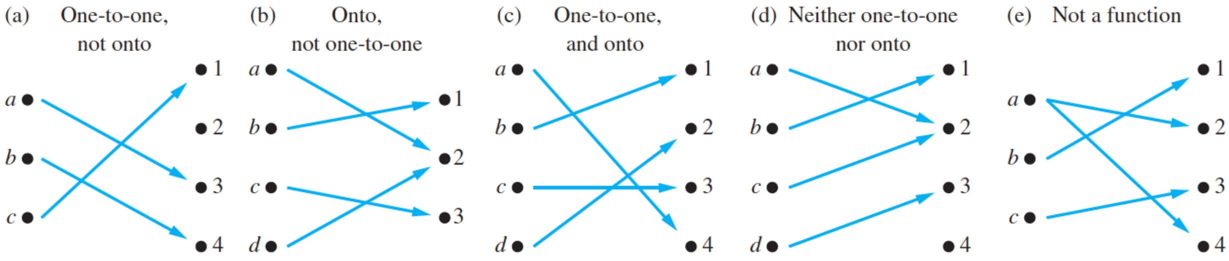


Figure 4.1: Types Of Correspondences

4.1.1 Inverse of a Function

Def 4.1.4 (Inverse of a Function)**.** Let f be a One-to-One-Correspondence from the set A to the set B. The inverse function of f is the function that assigns to an element b belonging to B the unique element a in A such that $f(a) = b$. The inverse function of f is denoted by f^{-1}. Hence, $f^{-1}(b) = a$ when $f(a) = b$.
Def 4.1.5 (Increasing and Strictly Increasing Functions)**.** A function f whose domain and codomain are subsets of the set of real numbers is called *increasing* if $f(x) \le f(y)$, and *strictly increasing* if $f(x) < f(y)$,

whenever $x < y$ and x and y are in the domain of f. (The word *strictly* in this definition indicates a strict inequality.)

4.2 Composition of Functions

Def 4.2.1 (Composition of Functions)**.** Let g be a function from the set A to the set B and let f be a function from the set B to the set C. The *composition* of the functions f and g, denoted by $f \circ g$, id defined by

$$(f \circ g)(a) = f(g(a)).$$

Notes. Note that function composition is right-associative and that $f \circ g \neq g \circ f$. Note that the composition operator is a function which takes other functions as operands and returns a function. This is called treating functions as first-class objects in object oriented languags. Not all programming languages offer this ability.

4.3 Common functions in computer science

We are used to the functional notation of algebra

$$g(x, y)$$

We are also accustomed to operators from programming languages

$$x + y$$

If the function g is defined as the sum of the two arguments, the two notations mean the same thing. The second form is called operation notation and works well for both binary and unary operators. But for arguments of more than 2 it is difficult to use. Note that the usual form is infix. But if we adopt a different convention, putting the operator after all the arguments, it is now possible to have functions of any number of arguments written in operator notation with no loss of precision. If the operator follows the operands, the notation is called post-fix. If the operator preceeds the operands, it is called prefix. Thus the function notation, prefix, infix, and suffix notation for addition are

$$+(a,b)$$

$$ab+$$

$$a+b$$

$$+ab$$

Note that the C language has a way of converting a symbol into an infix operator for a binary function.

We make special note of the important distinctions between common programming languages and the mathematical functions they implement. First, it is obvious that the mathematical convention of juxtaposing operands as implied multiplication does not work for computer languages which are not constrained to single character variables. Therefore the usual convention of an explicit multiplication operation, most always *, is adopted. Next is the challenge of differentiating between regular division and integer division. Several operators are employed in various languages and texts and sometimes is only implied by the type of the returned value. In many languages the type of the value to be returned value is infered by the types of the operands. Therefore an integer divided by an integer returns an integer, a case of integer division. To overcome this we see the apparently odd truth that $4/3 \neq 4.0/3$ in many languages. A more rigorous approach is delayed until the chapter on number theory but we define integer division in this chapter to connect the material more closely to contemporary coding practice.

4.3.1 Functions between Natural Numbers or Integers

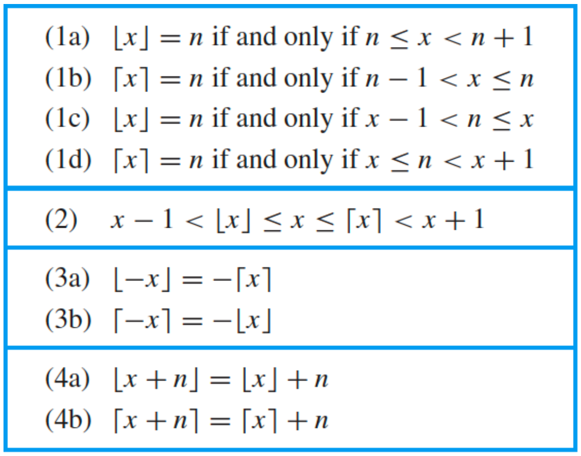
$$\text{Factorial}(n) = \prod_{i=1}^{n} i$$

The remainder function is that function which returns the remainder when doing integer division. We will formally define this in the unit on Integers.

4.3.2 Functions of Real to Integer

Def 4.3.1 (Floor and Ceiling Functions)**.** The *floor function* assigns to the real number x the largest integer that is less than or equal to x. The value of the floor function at x is denoted by $\lfloor x \rfloor$. The *ceiling function* assigns to the ral number x the smallest integre that is greater than or equal to x. The value of the celing function at x is denoted by $\lceil x \rceil$.

The floor function is sometimes called the *greates integer function*. It may be denoted by $[x]$.



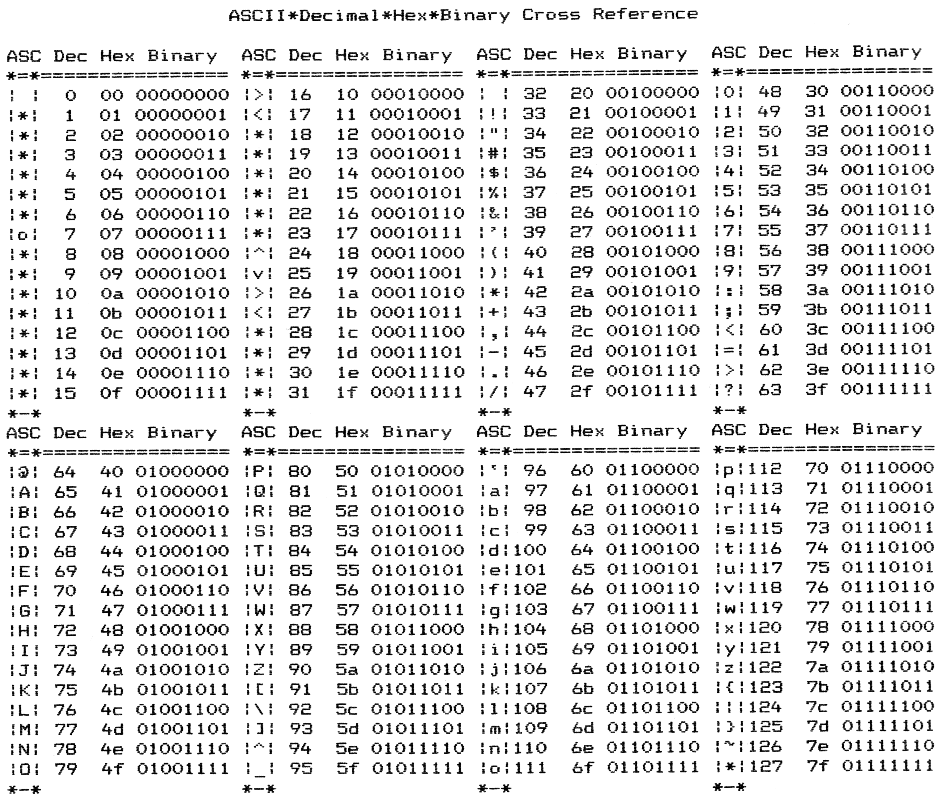
(1a)	$\lfloor x \rfloor = n$ if and only if $n \le x < n + 1$
(1b)	$\lceil x \rceil = n$ if and only if $n - 1 < x \le n$
(1c)	$\lfloor x \rfloor = n$ if and only if $x - 1 < n \le x$
(1d)	$\lceil x \rceil = n$ if and only if $x \le n < x + 1$
(2)	$x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1$
(3a)	$\lfloor -x \rfloor = -\lceil x \rceil$
(3b)	$\lceil -x \rceil = -\lfloor x \rfloor$
(4a)	$\lfloor x + n \rfloor = \lfloor x \rfloor + n$
(4b)	$\lceil x + n \rceil = \lceil x \rceil + n$

Table 4.1: Useful Properties of the Floor and Ceiling Functions

4.3.3 Functions between Boolean and Integer

ASCII is a 7-bit character set containing 128 characters. It contains the numbers from 0-9, the upper and lower case English letters from A to Z, and some special characters. The character sets used in modern computers, in HTML, and on the Internet, are all based on ASCII.

Boolean ↔ Integer



```
                         ASCII*Decimal*Hex*Binary Cross Reference

ASC Dec Hex Binary   ASC Dec Hex Binary   ASC Dec Hex Binary   ASC Dec Hex Binary
*=*================= *=*================= *=*================= *=*=================
¦ ¦  0   00 00000000 ¦>¦ 16   10 00010000 ¦ ¦ 32   20 00100000 ¦0¦ 48   30 00110000
¦*¦  1   01 00000001 ¦<¦ 17   11 00010001 ¦!¦ 33   21 00100001 ¦1¦ 49   31 00110001
¦*¦  2   02 00000010 ¦*¦ 18   12 00010010 ¦"¦ 34   22 00100010 ¦2¦ 50   32 00110010
¦*¦  3   03 00000011 ¦*¦ 19   13 00010011 ¦#¦ 35   23 00100011 ¦3¦ 51   33 00110011
¦*¦  4   04 00000100 ¦*¦ 20   14 00010100 ¦$¦ 36   24 00100100 ¦4¦ 52   34 00110100
¦*¦  5   05 00000101 ¦*¦ 21   15 00010101 ¦%¦ 37   25 00100101 ¦5¦ 53   35 00110101
¦*¦  6   06 00000110 ¦*¦ 22   16 00010110 ¦&¦ 38   26 00100110 ¦6¦ 54   36 00110110
¦o¦  7   07 00000111 ¦*¦ 23   17 00010111 ¦'¦ 39   27 00100111 ¦7¦ 55   37 00110111
¦*¦  8   08 00001000 ¦^¦ 24   18 00011000 ¦(¦ 40   28 00101000 ¦8¦ 56   38 00111000
¦*¦  9   09 00001001 ¦v¦ 25   19 00011001 ¦)¦ 41   29 00101001 ¦9¦ 57   39 00111001
¦*¦ 10   0a 00001010 ¦>¦ 26   1a 00011010 ¦*¦ 42   2a 00101010 ¦:¦ 58   3a 00111010
¦*¦ 11   0b 00001011 ¦<¦ 27   1b 00011011 ¦+¦ 43   2b 00101011 ¦;¦ 59   3b 00111011
¦*¦ 12   0c 00001100 ¦*¦ 28   1c 00011100 ¦,¦ 44   2c 00101100 ¦<¦ 60   3c 00111100
¦*¦ 13   0d 00001101 ¦*¦ 29   1d 00011101 ¦-¦ 45   2d 00101101 ¦=¦ 61   3d 00111101
¦*¦ 14   0e 00001110 ¦*¦ 30   1e 00011110 ¦.¦ 46   2e 00101110 ¦>¦ 62   3e 00111110
¦*¦ 15   0f 00001111 ¦*¦ 31   1f 00011111 ¦/¦ 47   2f 00101111 ¦?¦ 63   3f 00111111
*-*                  *-*                  *-*                  *-*
ASC Dec Hex Binary   ASC Dec Hex Binary   ASC Dec Hex Binary   ASC Dec Hex Binary
*=*================= *=*================= *=*================= *=*=================
¦@¦ 64   40 01000000 ¦P¦ 80   50 01010000 ¦`¦ 96   60 01100000 ¦p¦112   70 01110000
¦A¦ 65   41 01000001 ¦Q¦ 81   51 01010001 ¦a¦ 97   61 01100001 ¦q¦113   71 01110001
¦B¦ 66   42 01000010 ¦R¦ 82   52 01010010 ¦b¦ 98   62 01100010 ¦r¦114   72 01110010
¦C¦ 67   43 01000011 ¦S¦ 83   53 01010011 ¦c¦ 99   63 01100011 ¦s¦115   73 01110011
¦D¦ 68   44 01000100 ¦T¦ 84   54 01010100 ¦d¦100   64 01100100 ¦t¦116   74 01110100
¦E¦ 69   45 01000101 ¦U¦ 85   55 01010101 ¦e¦101   65 01100101 ¦u¦117   75 01110101
¦F¦ 70   46 01000110 ¦V¦ 86   56 01010110 ¦f¦102   66 01100110 ¦v¦118   76 01110110
¦G¦ 71   47 01000111 ¦W¦ 87   57 01010111 ¦g¦103   67 01100111 ¦w¦119   77 01110111
¦H¦ 72   48 01001000 ¦X¦ 88   58 01011000 ¦h¦104   68 01101000 ¦x¦120   78 01111000
¦I¦ 73   49 01001001 ¦Y¦ 89   59 01011001 ¦i¦105   69 01101001 ¦y¦121   79 01111001
¦J¦ 74   4a 01001010 ¦Z¦ 90   5a 01011010 ¦j¦106   6a 01101010 ¦z¦122   7a 01111010
¦K¦ 75   4b 01001011 ¦[¦ 91   5b 01011011 ¦k¦107   6b 01101011 ¦{¦123   7b 01111011
¦L¦ 76   4c 01001100 ¦\¦ 92   5c 01011100 ¦l¦108   6c 01101100 ¦¦¦124   7c 01111100
¦M¦ 77   4d 01001101 ¦]¦ 93   5d 01011101 ¦m¦109   6d 01101101 ¦}¦125   7d 01111101
¦N¦ 78   4e 01001110 ¦^¦ 94   5e 01011110 ¦n¦110   6e 01101110 ¦~¦126   7e 01111110
¦O¦ 79   4f 01001111 ¦_¦ 95   5f 01011111 ¦o¦111   6f 01101111 ¦*¦127   7f 01111111
*-*                  *-*                  *-*                  *-*
```

Table 4.2: ASCII: American Standard Code for Information Interchange, 7-bit

4.3.4 Functions Between Boolean and Real

Boolean ↔ $\mathbb{R}$

4.3.5 Functions of Real to Real

polynomial, log, log-linear, exponential increasing/decreasing versus strictly increasing and decreasing cipher function permutation function

4.3.6 Other Useful Functions

Def 4.3.2 (Characteristic Function)**.** Let S be a subset of a universal set U. The **characteristic function** f_S of S is the function from U to the set $\mathbb{B}$ such that $f_S(x) = 0$ if x does not belong to S and $f_S(x) = 1$ when x does belong to S.

4.4 Composition of Functions

Properties of Functions: Onto, One-to-One, One-to-One Correspondence, Invertible An invertible function is called a permutation

Elementary Functions Used in Computer Science and Engineering exponential and logarithmic, floor and ceiling, integer and absolute value, remainder function (MOD operator in programming), integer to binary and binary to integer, from natural numbers to integers.

It helps to note that when specifying function composition we begin to speak of the function like an operand. For example we can define a function h as the composition of two other functions f and g using the composition operator. In functional notation we call this treating a function as a first class object, that is, the function is an object no different than other operands but simply drawn from the set of functions.

subsection Total and Partial Functions A total function is one that has a maplet for every element in the domain. These are called well

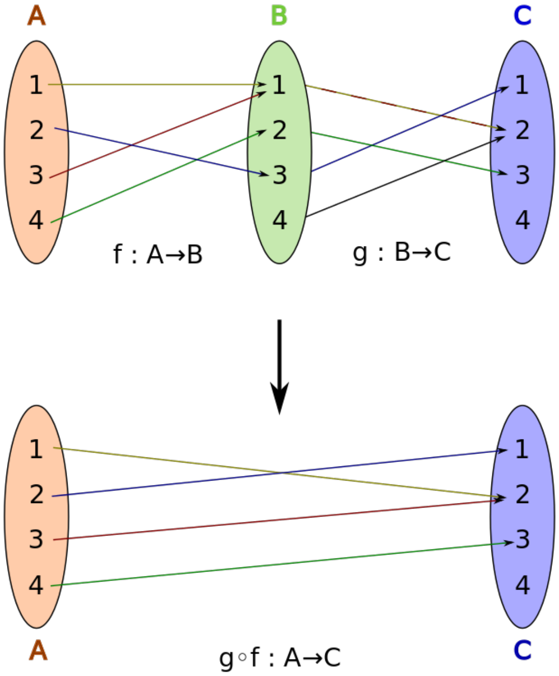


Figure 4.2: Function Composition

defined functions. Some functions are undefined for some elements in the domain and these are called partial functions. Note that division on two numbers is a partial function. It is represented by an line through the arrow from domain to co-domain.

Not every element in the co-domain may be the image of an element in the domain. The set of all images is called the scope of the function.

Def 4.4.1 (Partial Function)**.** A *partial function* f from a set A to a set B is an assignment to each element a in a subset of A, called the *domain of definition* of f , of a unique element b in B. The sets A and B are called the *domain* and *codomain* of f , respectively. We say that f is *undefined* for elements in A that are not in the domain of definition of f. When the domain of definition of f equals A, we say that f is a *total function*. The notation $h : A \nrightarrow B$ is often used as a notation to designate a partial function h that maps the set A to B where the set A has undefined elements for the function h. However some authors use the same notation for partial functions as they do for partial functions.

4.5 Visual Representations of Functions: Set mappings and analytic geometry

In analytic geometry we speak of the graph of a function when we talk about ploting it against two or more axes. Note that in discrete math we will use the term *graph* in a different way which does not mean the same thing.

4.6 Sequences

Formally, a sequence can be defined as a function whose domain is either the set of the natural numbers (for infinite sequences) or the set of the first n natural numbers (for a sequence of finite length n). The position of an element in a sequence is its rank or index; it is the natural number from which the element is the image. It depends on the context or a specific convention, if the first element has index 0 or

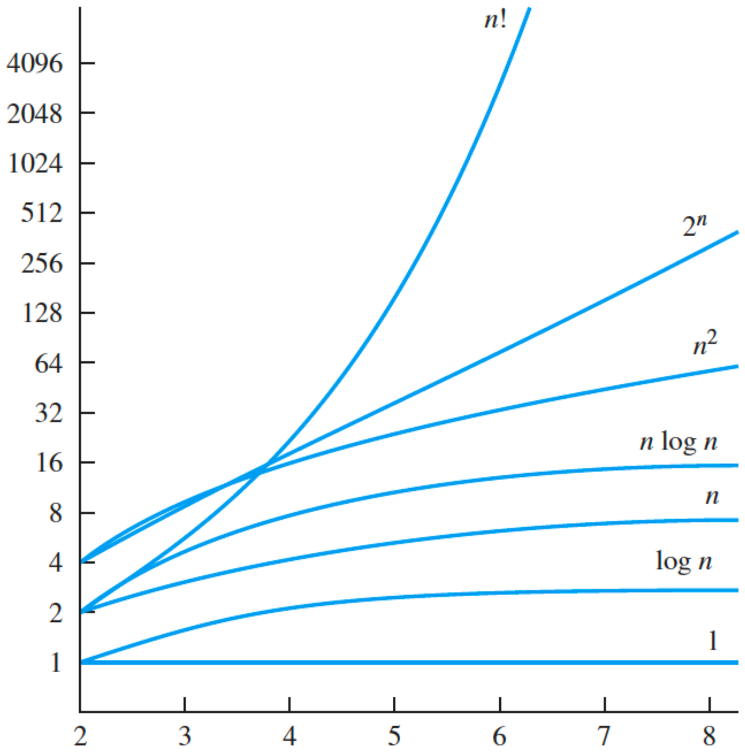


Figure 4.3: Graphs of Common Functions Needed In Computer Science

1. When a symbol has been chosen for denoting a sequence, the nth element of the sequence is denoted by this symbol with n as subscript; for example, the nth element of the Fibonacci sequence is generally denoted F_n.

Like tuples, these are ordered lists. Unlike tuples, they vary in length and notation.
Def 4.6.1 (Sequence)**.** A *sequence* is a function from a subset of the integers to a set S. We use the notation a_n to denote the image of the integer n. We call a_n a *term* of the sequence.

We use the notation $\{a_n\}$ to describe the sequence. Note how this conflicts with set notation. Note how there is no requirement that the set S be numeric. We often describe sequences by listing the terms of the sequence in order of increasing subscripts.
Def 4.6.2 (Geometric Progression)**.** A *geometric progression* is a sequence of the form

$$ar^0, ar^1, ar^2, \ldots, ar^n, \ldots$$

where the *initial term a* and the common ratio *r* are real numbers.

Sequences of the form $a_1, a_2, ..., a_n$ where the terms are drawn from some set of symbols rather than numbers are often used in computer science. These finite sequences are also called *strings*. This string is also denoted by $a_1a_2...a_n$. (Recall that bit strings, which are finite sequences of bits, were introduced in Section 1.1.) The length of a string is the number of terms in this string. The empty string, denoted by λ, is the string that has no terms. The empty string has length zero.

4.6.1 Special Integer Sequences

A common technique needed in programming is finding an algorithm that will generate a particular sequence. This is the same as finding a function to do the same substituting the index variable for the loop counter.

Trying to guess the function that can be used to create the sequence is a common test item. Note that an infinite number of sequences can be created that begin with the same few terms that are given. Still it is important to be able to quickly recognize the most common sequences and functions such as arithmetic and geometric progressions.

In mathematics a *finite sequence* is usually defined as any function f whose domain is a finite initial set $\{1,2,3,\dots,n\}$ of positive integers. The number n of integers in the domain of f is the *length* of f. When considering f as a finite sequence it is customery to write

$$f_1, f_2, \dots$$

rather than

$$f(1), f(2), \dots$$

in designating the value of f at 1,2, $\dots$.

Along with this subscript notation there is the *n-tuple* notation

$$(f_1, f_2, \dots f_n)$$

f_i is the i_{th} *entry* or *term* of f. This *n-tuple* notation indicates how explicit short finite sequences can be defined and pictured. For example, the equation

$$x = (5,1,4,0)$$

tells us that x is the finite sequence of length 4 whose values on its domain {1,2,3,4} are given by

$$x_1 = x(1) = 5$$

$$x_2 = x(2) = 1$$

$$x_3 = x(3) = 4$$

$$x_4 = x(4) = 0$$

The most common domain for a sequence is $\mathbb{N}$. However it is frequently more convenient to use natural numbers including zero. There is an edge case where sometimes the domain is the null set in which case the sequence is empty and can be shown using a pair of empty parentheses to emphasize that there are no terms in the sequence

$$()$$

Consider a function $\sigma : \mathbb{N} \to \mathbb{Z}$ For example sigma(x)=2*x this gives maplets 1|->2, 2|->4, etc Since the domain is understood to be natural numbers it is easier to just write the images as a list: 2,4,6,8, etc We can use subscript notation and state this more abstractly:

A sequence A is a(1),a(2),a(3),...a(i) or $\{a(n)n \in N\}$ or even more compactly as $\{a(n)\}$ when it is understood we intend a sequence. It is common in sequences to base not strictly on natural numbers but natural numbers plus the member zero.

Note that this definition of sequence allows for non-numeric sequences.
Def 4.6.3 (Geometric Progression)**.** A *geometric progression* is a sequence of the form

$$ar^0, ar^1, ar^2, \ldots, ar^n, \ldots$$

where the *initial term* a and the *common ratio* r are real numbers.
Def 4.6.4 (Arithmetic Progression)**.** An *arithmetic progression* is a sequence of the form

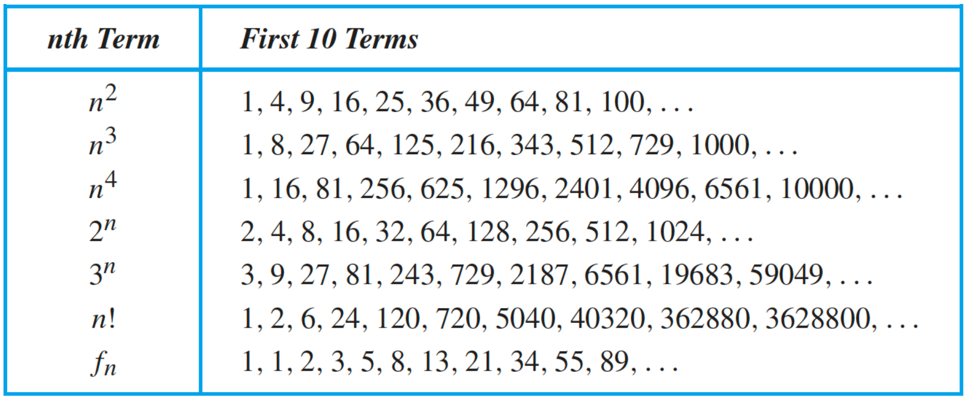
$$a + 0d, a + 1d, a + 2d, \ldots, a + nd, \ldots$$

where the *initial term* a and the *common difference* d are real numbers.

The reader interested in integer sequences will want to look at the Online Encyclopedia of Integer Sequences originally created by Neil Sloane and available online.

4.6.2 Strings

Sequences sometimes have a co-domain of symbols that can be listed in a sequence such that each symbol is a distinct symbol. This allows sequences of symbols to be represented without commas without



nth Term	*First 10 Terms*
n^2	1, 4, 9, 16, 25, 36, 49, 64, 81, 100, ...
n^3	1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, ...
n^4	1, 16, 81, 256, 625, 1296, 2401, 4096, 6561, 10000, ...
2^n	2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, ...
3^n	3, 9, 27, 81, 243, 729, 2187, 6561, 19683, 59049, ...
$n!$	1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, ...
f_n	1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Table 4.3: Some Useful Sequences

confusion. The set of symbols is usually called the alphabet and commonly represented with the capital sigma, Σ. Finite equences of symbols are called strings and represented by $a_1a_2a_3\ldots a_n$

4.6.3 Evaluation of Functions

Note that functions have one notation that just gives the function a name and identifies the domain and codomain but does not define how one maps to the image given the pre-image. Some functions are trivially evaluated. The other notation defines how that is done and it is often a mathematical expression. But it need not be. It can be any procedure that will ensure a value from the codomain is returned when requested.

This has a historical point. Prior to the middle of the 20c the word computer was a job description not a machine. It was the job of the mathematician to give the computers the instructions on how to compute the functions that were needed. More often than not these values were compiled into books and used as reference when the function needed to be evaluated. Turing was one of the first to ask what limits a machine would have if the mathematician gave instructions that could be executed by a machine. We now call these procedures the algorithms that are used to evaluate a function given it aguments.

4.7 Sigma Notation, Summations and Open Form Formulas

Given a function which produces a sequence, one operation we frequently wish to do is sum the terms. We define notation for this.

Def 4.7.1 (Summation)**.** Given a sequence $\{a_n\}$, we use the notation $\sum_{j=m}^{n} a_j$ to represent $a_m + a_{m+1} + ... + a_n$. The variable j is called the *index of summation*, the value m the *lower limit* and the value n the *upper limit*.

Shift the index of summation. Ex: What is the evaluation of $\sum_{j=1}^{5} j^2$ with an index of summation that runs from 0 to 4 instead of 1 to 5? Substitute $k = j - 1$ so the old index of summation would run from 1 to 5 but the new index of summation will run from 0 to 4 and the new summand will be $(k+1)^2$. Sol:

4.7.1 Properties of Summation

$$\sum_{i=m}^{n} ca = c \sum_{i=m}^{n} a \tag{4.1}$$

$$\sum_{i=m}^{n} (a \pm b) = \sum_{i=m}^{n} a \pm \sum_{i=m}^{n} b \tag{4.2}$$

$$\sum_{i=m}^{n} a = \sum_{i=m+p}^{n+p} a \tag{4.3}$$

It may start at something other than 1 and may sum to infinity. The letter j is called the *index of summation*, and the choise of the letter j as variable is arbitrary; that is, we could have used any other letter, such as i or k. Or, in notation,

$$\sum_{j=m}^{n} a_j = \sum_{i=m}^{n} a_i = \sum_{k=m}^{n} a_k$$

where m is the **lower limit** and n is the *upper limit*.

4.7.2 Closed form solution

For finite sequences, it is always possible to add the terms to calculate the sum. But these sequences can be very long making the summation an expensive operaton. And it is impossible if the sequence is infinite. Another way must be found and this brings us to closed form solutions. Many summations can be restated as formulas that are functions of the upper limit of summation. These are less burdensome computationally and are possible for infinite sequences with calculus. A closed form soluiton will have a finite number of operations and does not require the articulation of every term of the sequence.

Theorem 4.7.1. If a and r are real numbers with $r \neq 0$, then

$$\sum_{j=0}^{n} ar^j = \begin{cases} \frac{ar^{n+1}-a}{r-1}, & \text{if } r \neq 1, \\ \text{(n+1)a}, & \text{if } r = 1. \end{cases}$$

telescoping

4.7.3 Useful Closed Form Solutions

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} \tag{4.4}$$

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} \tag{4.5}$$

$$\sum_{k-1}^{n} k^3 = \frac{n^2(n+1)^2}{4} \tag{4.6}$$

$$\sum_{i=0}^{n} 2^i = 2^0 + 2^1 + \ldots 2^{n-1} + 2^n = 2^{n-1} - 1 \tag{4.7}$$

4.8 Growth of Functions and Asymptotic Notation

As we saw when we talked about the graphs of common graphs (Figure 4.3) we see that there is a difference among these functions in how quickly the function values climb as the argument, n, grows.

For example you know that a quadratic polynomial will overtake a linear polynomial. But many linear functions will evaluate to higher numbers for small values. We want some way to express the concept that a quadratic is in some sense bigger. This leads to a new notation, the Big-O or asymptotic notation or **Bachmann-Landau notation** and the O is sometimes called the **Landau** symbol. Knuth extended this notation in his work. Characterize functions according to their growth rates. different functions with the same growtwth rate may be represented using the same O notaiotn. the order of the function. O is used to specify an upper growth bound. Since a slower growing function may give a higher value for low values of the argument but pull ahead and forever remain ahead for large values, it is not sufficient to offer a few values to demonstrate.

Def 4.8.1. Let f and g be functions fromthe set of integers or reals to the set of real numbers. We say that $f(n)$ is $O(g(n))$ if there are constants C and k such that

$$|f(x)| \leq C|g(n)|$$

whenever $n > k$. We read this as "$f(n)$ is big-oh of $g(n)$". The constants C and k are called the **witnesses** to the relationship $f(n)$ is to $O(g(n))$. To demonstrate that $f(n)$ is big-oh we only need to produce one set of witnesses. Note that it is an abuse of the notation to say $f(n) = O(g(n))$ since no equality is established, only an assertion that the functions belong to the same class of functions.

$$f(n) = O(g(n)) \text{ as } n \to \infty$$

if and only if for all sufficiently large values of n, the absolute value of $f(n)$ is at most a positive constant multiple of another function $g(n)$. That is $f(n) =)(g(n))$ if and only if there existss a positive realy number C and a real number n_0 such that

$$|f(n)| \leq cg(n)\text{for all } n \geq n_0$$

Instead of looking at at the function itself, it is helpful to look at a function as a family of functions. For a linear function $f(n) = n$ we can define a set of functions based on that by introducing constants

c_0 and c_1 like this: $f(n) = c_1 n + c_0$. This can be done for all the common functions giving us this list:

log	$O(\log n)$	
log linear	$O(n \log n)$	
linear	$O(n)$	$c_1 n + c_0$
quadratic	On^2	$c_2 n^2 + c_1 n + c_0$
cubic	$O(n^3)$	$c_3 n^3 + c_2 n^2 + c_1 n + c_0$
exponential	$O(2^n)$	
factorial	$O(n!)$	

If there are constants such that a function can be restated as one of these families of functions then we say it is *of the same order of magnitude* as

Given two linear polynomial equations, we can always do this by changing the values of the constants. But no matter how you change the constants of a linear polynomial, a quadratic polynomial will always overtake it. This shows that a quadratic will ALWAYS beat a linear function, that they are in some sense in different categories regardless of the choice of constants. This gives us the concept of the category of the function or the order or magnitude designated by Big O. We can change any linear function into another linear polynomial function by changing the constants. We group all of these together and call them linear polynomial functions or polynomial functions on n, written O(n). We can prove that there are at least 7 categories that are important to the study of computer science, constant O(1), linear O(n), log O(log n), log-linear O(n log n), quadratic O(n**2), exponential O(2**n), and factorial O(n!). Note that each of these is a grouping of functions that include all the variations of different constants. By choosing the constants, we can always find some n sub 0 such that a quadratic will beat a linear. This gives us an ordering of these categories

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$

For any two specific functions of different categories one can find the n sub 0 at which the larger function overtakes the smaller and forever remains ahead. This way of viewing the relative size of functions will be used when we study the complexity of algorithms later.

Theorem 4.8.1. Let

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n^1 + a_0$$

be a polynomial in n of degree k, where each a_i is nonnegative. Then

$$p(n) = \Theta(n^k).$$

Proof. □

Def 4.8.2 (Big-Omega)**.** Let f and g be functions from the set of integers or reals to the set of reals. We say that $f(n)$ is $\Omega(g(n))$ if there are positive constants C and k such that

$$|f(n)| \geq C|g(n)|$$

whenever $x > k$. Read this as "$f(n)$ is big-Omega of $g(n)$".

Def 4.8.3 (Big Theta)**.**

Chapter 5

Relations

Relations support comparisons or connections between elements within a set or between two or more sets, or between members of the same set. For example “less than or equal”, “is perpendicular to”, “is a cousin to” are relations on a set of numbers, lines, or people.

Relations are closely associated with functions but we looked at those first since you are familiar with them from before. Relations are a superset of functions, all functions are relations but not all relations are functions.

The fundamental restriction to a function is that it needed to evaluate to exactly one value. A relation can be a many-to-many mapping from the domain to the codomain. In its most abstract statement, all valid subsets of the cross product of the domain and codomain are relations.

A fundamental paradigm is one called objects/relations and is found in programming languages today. We saw that sets are collections of objects and we saw that predicates take on the value true when a particular property is found in that object. Objects can have relationships among them and the most basic is the binary relationship. In a binary relationship we way that the two objects either are in or not in that relationship with each other. This chapter explores the consequences of this and its many applications.

5.1 Relations

Def 5.1.1 (Binary Relations)**.** Let A and B be sets. A *binary relation R between A and B* is a subset of the Cartesian Product $A \times B$. We can denote the binary relation on two elements a and b by $a\,R\,b$, or by $(a,b) \in R$. We can denote the lack of relationship R between the two elements with the notation $a \not{R}\, b$ or $(a,b) \notin R$. The sets A and B may be the same set in which case we say R is a relation on A^2 or on the set A. We read "a is in relation to b" or "a is related to b" to indicate they are related.
Relations are not limited to relations on two sets but can be applied to any number of sets. The relation is still defined as a subset of the n-tuples formed from the cross product of the sets.

Relations are sets of tuples of the cross product and set operators can be used on them.

Given a set A, the relation $\{(a,a)|a \in A\}$ is the equality or identity relation and is denoted by Δ_A.

Note that functions are a subset of relations. Note that plotting graphs of functions uses the Cartesian plane which is a relation of $\mathbb{R}^2$.

5.1.1 Inverse Relation

The *inverse of a relation R* from set A to B is the relation denoted by $R^{-1} = \{(b,a)|(a,b) \in R\}$.

5.1.2 Relational operators

Equality, inequality, greater than, greater than or equal, less than, less than or equals are all relational operators that every programmer learns. To these we introduce one that is not often seen in programming, the divides relation.
Def 5.1.2 (Divides)**.** If a and b are integers with $a \neq 0$, we say that a *divides* b if there is an integer c such that $b = ac$, or equivalently $\frac{b}{a}$ is an integer. When a divides b we say that a is a factor or divisor of b, and that b is a multiple of a. The notation $a \mid b$ denotes that a

divides b. We write $a \nmid b$ when a does not divide b.

5.2 Properties of Relations

Def 5.2.1 (reflexive relation)**.** A relation R on a set A is called *reflexive* if $(a,a) \in R$ for every element $a \in A$. The relation R is reflexive if $\forall a((a,a) \in R)$.
Def 5.2.2 (Symmetric Relation)**.** A relation R on a set A is alled *symmetric* if $(b,a) \in R$ for all $a,b \in A$.
Def 5.2.3 (Antisymmetric)**.** A relation R on set A such that for all $a,b \in A$, if $(a,b) \in R$ and $(b,a) \in R$, then $a = b$ is called *antisymmetric*
Def 5.2.4. A relation R on a set A is called *transitive* if whenever $(a,b) \in R$ and $(b,c) \in R$, then $(a,c) \in R$. The relation R on a set A is transitive if we have $\forall a \forall b \forall c(((a,b) \in R \wedge (b,c) \in R) \rightarrow (a,c) \in R)$
Def 5.2.5 (Anti-Reflexive Relation)**.**

5.3 Composition of Relations

Combining Relations in Rosen
Def 5.3.1 (Composition of Relations)**.** Let R be a relation from set A to set B and S a relation from B to a set C. The *composite* of R and S is the relation consisting of ordered pairs (a,c) such that $(a,b) \in R$ and $(b,c) \in S$. We denote the composite of R and S by $S \circ R$.
Notes. Carefully note the order of the operand in the composition operation.

5.4 Graphic Representation of Relations

For relations onto themselves, this gives what we will later call a directed graph. We cover graphs more formally in a later chapter.

5.4.1 Digraphs

A convenient pictorial representation of relations is to represent the domain and co-domain as ovals with the elements labeled. A pair that is in the relation is then represented by an arc from the element in the domain to the element in the co-domain. When the co-domain is the same as the domain the oval is sometimes omitted. We call this a *directed graph*. We will cover directed graphs more formally in the chapter on Graphs.

5.5 Closure Property of Relations

5.5.1 Transitive Closure of Relations

Def 5.5.1. Let R be a relation on a set A. The *connectivity relation* R^* consists of the pairs (a,b) such that there is a path of length one from a to b in R.

5.5.2 Kleene Closure and Order

5.6 N-ary Relations

We have only spoken of binary relations up to now. But more than two elements can be in a relation.

Def 5.6.1. Let $A_1, A_2, \ldots, A_n$ be sets. An *n-ary relation* on these sets is a subset of $A_1 \times A_2 \times \cdots \times A_n$. The sets $A_1, A_2, \ldots, A_n$ are called the *domains* of the relation, and n is called its *degree*.

Def 5.6.2. Let R be an n-ary relation and C a condition that elements in R may satisfy. Then the *selection operator* s_C maps the n-ary relation R to the n-ary relation of all n-tuples from R that satisfy the condition C.

Def 5.6.3. The *projection* $P_{i_1 i_2, \ldots, i_m}$ where $i_1 < i_2 < \ldots i_m$, maps the n-tuple $(a_1, a_2, \ldots, a_n)$ to the m-tuple $(a_{i_1}, a_{i_2}, \ldots, a_{i_m})$, where $m \leq n$.

5.6.1 Application to Databases

5.7 Matrices and Linear Algebra

Many include matrices in Discrete Mathematics. We exclude this since it is offered as a separate course.

5.8 Representations of Relations

The definition of relations are as a subset of the cross product $A \times B$. But two alternate representations of relations are useful.

5.8.1 Representing Relations Using Matrices

Given a subset of $A \times B$, one can represent the relation by creating a matrix with the elements of the sets A and B along the two axes of the matrix and entering a 1 when that pair is in the relation and 0 if not.

5.8.2 Representing Relations Using Digraphs

5.8.3 Paths in Directed Graphs

5.9 Closures on Relations

5.9.1 Closures

Sometimes an operation will return something that is outside the set from which the argument was drawn. For example, you can perform division on two integers but not get an integer as a return value. You can solve an equation like $2 \cdot 2x + 1 = 0$ with 2 and 1 both being valid rational numbers yet find you have a number that is outside the set of rational numbers. We say that the set is not closed under these operation s.

When we look at the properties of relations sometimes we see that a given relation is closed, that is every pair needed to satisfy that relation are found in the relation. When we find that to be true, we say the set is closed under that relation.

Let R be a relation on a set A. R may or may not have some property **P**, say reflexivity. If the set A does not possess the property, it is possible to add pairs to the relation such that the union of the two relations (which after all are just sets of pairs) will possess that property. We are now ready to offer a formal definition.

Def 5.9.1 (Closure)**.** A set that is closed under an operation or collection of operations is said to satisfy a closure property. When a set S is not closed under some operations, one can usually find the smallest set containing S that is closed. This smallest closed set is called the closure of S (with respect to these operations). For example, the closure under subtraction of the set of natural numbers, viewed as a subset of the real numbers, is the set of integers. The closure of a set S which does not posses the closure property under the reflexion relation can be created by summing the relation with all pairs that make the relation reflexive.

The reflexive closure of a relation$R$can be formed by adding to R all pairs of the form $(a,a)$with$a \in A$,not already in$R$. The addition of these pairs produces a new relation that is reflexive, contains$R$,and is contained within any reflexive relation containing R. We see that the reflexive closure of R equals $R \cup \Delta$, where $\Delta = \{(a,a) | a \in A\}$is the diagonal relation on$A$.

5.9.2 Transitive Closures

Def 5.9.2. Let R be a relation on a set A. The *connectivity relation* R^* consists of the pairs (a,b) such that there is a path of length at least one from a to b in R.

Theorem 5.9.1. The transitive closure of a relation R equals the connectivity relation R^*.

Lemma 5.9.1. Let A be a set with n elements, and let R be a relation on A. If there is a path of length at least one in R from a to b, then there is such a path with length not exceeding n. Moreover, when $a \neq b$, if there is a path of length at lteast one from R from a to b, then there is such a path with length not exceeding $n-1$.

5.9.3 Computing Transitive Closures

The most well known algorithm to compute the transitive closure of a relation is Warshall's Algorithm.

5.10 Equivalence Relations

reflective, symmetric, transitive remainder function partitions

5.10.1 Equivalence Classes and Set Partitions

Def 5.10.1. The elements a and b that are related by an equivalence relatin are called *equivalent*. The notation $a \sim b$ is often used to denote that a and b are equivalent elements with respect to a particular equivalence relation.
Def 5.10.2. Let R be an equivalence relation on a set A. The set of all elements that are related to an element a of A is called the *equivalence class* of a. The equivalence class of a with respect to R is denoted by $[a]_R$. When only one relation is under consideration, we can delete the subscript R and write $[a]$ for this equivalence class.

If $b \in [a]_R$, then b is called a **representative** of this equivalence class.
Theorem 5.10.1. Let R be an equivalence relation on a set A. These statements for elements a and b of A are equivalent:

(i) aRb

(ii) $[a] = [b]$

(iii) $[a] \cap [b] \neq \emptyset$

Theorem 5.10.2. Let R be an equivalence relation on a set S. Then the equivalence classes of R form a partition of S. Conversely, given a partition $\{A_i | i \in I\}$ of the set S, there is an equivalence relation R that has the sets $A_i, i \in I$, as its equivalnece classes.

5.10.2 Partial Ordering Relations

Def 5.10.3 (Posets)**.** A relation R on a set S is called a *partial ordering* or *partial order* if it is reflexive, antisymemetric, and transitive. A set S together with a partial ordering R is called a *partially ordered set* or *poset*, denoted by (S,R). Members of S are called *elements* of the poset.

Many different relations on sets create posets such as $\geq$, evenly divides ($\mid$), and $\subset$. We create a new relation symbol $\preceq$.
Def 5.10.4. Given a poset (S,R) and two elements a, b, we use the notation $a \preceq b$ to indicate that $(a,b) \in R$. We use $\preceq$ to designate any relation of a poset. The elements a and b of a poset $(S,\preceq)$ are called *comparable* if either $a \preceq b$ or $b \preceq a$. When a and b are selements of S such that neither $a \preceq b$ nor $b \preceq a$, a and b are called *incomparable*. If $a \preceq b$ and $a \neq b$, we use the notation $a \prec b$.
Notes. Object oriented programming languages allow you to code relations. Note that since two elements of a set may be incomparable given the relation $\preceq$, we call it a partial ordering.
Def 5.10.5. If $(S,\preceq)$ is a poset and every two elements of S are comparable, S is called a *totally ordered* or *linearly ordered set*, and $\preceq$ is called a *total order* or a *linear order*. A totally ordered set is also called a *chain*.
Def 5.10.6 (Well-Ordered Set)**.** The poset $(S,\preceq)$ is called a *well-ordered set* if it is a poset such that $\preceq$ is a total ordering and every nonempty subset of S has a least element, (something defined in a few sections).
Theorem 5.10.3 (The Principle of Well-Ordered Induction)**.** Suppose that S is a well-ordered set. Then $P(x)$ is true for all $x \in S$, if INDUCTIVE STEP: For every $y \in S$, if $P(x)$ is true for all $x \in S$ with $x \prec y$, then $P(y)$ is true.

Proof. Suppose it is not the case that $P(x)$ is true for all $x \in S$. Then there is an element $y \in S$ such that $P(y)$ is false. Consequently, the set $A = \{x \in S | P(x) is false\}$ is nonempty. Because S is well ordered, A has a least element a. By the choice of a as a least element of A, we know that $P(x)$ is true for all $x \in S$ with $x \prec a$. This implies by the inductive step $P(a)$ is true. This contradiction shows that $P(x)$ must be true for all $x \in S$. □

Notes. We do not need a basis step in a proof using the principle of well-ordered induction because if x_0 is th eleast element of a well ordered set, the inductive step tells us that $P(x_0)$ is true. This follows because there are no elements $x \in S$ with $x \prec x_0$.

5.10.3 Lexicographic Order

You may have already noticed that the way words get arranged in a dictionary are different than the way they would be sorted if they were numbers. This ordering of strings is called a lexicographic ordering. Consider two posets $(A_1, \preceq_1)$ and $(A_2, \preceq_2)$. The **lexicographic ordering** $\preceq$ on $A_1 \times A_2$ is defined by specifing how one pair, (a_1, a_2) is less than another (b_1, b_2). We say $(a_1, a_2) \prec (b_1, b_2)$ if either $a_1 \prec_1 b_1$ or if both $a_1 = b_1$ and $a_2 \prec b_2$.
Notes. This came from Rosen. why are there two different relationships in this presentation?
Def 5.10.7 (Lexicographic Ordering)**.** Given strings $a_1a_2 \dots a_m$ and $b_1b_2 \dots b_n$ on a partially ordered set s. Suppose these strings are not equal. Let t minimim of m and n. We define the string A as less than B if and only if

$$(a_1, a_2, \dots, a_t) \prec (b_1, b_2, \dots, b_t),$$

or

$$(a_1, a_2, \dots, a_t) = (b_1, b_2, \dots, b_t) \wedge m < n$$

5.10.4 Haase Diagrams

Consider Figure 9-6-3a of a small poset $(\{1, 2, 3, 4, 6, 8, 12\}, |)$. This Recall that a poset is a partial ordering and that means it is reflexive, anti-symmetric and transitive. If we know that the di-graph is of a poset we can then remove the reflexive edges. This gives us Figure 9-6-3b which has the reflexive edges removed. Since we know it must also be transitive we can safely remove those edges as well. If we use the convention that vertices that are "less than" in the ordering be placed lower on the graph we can also remove the directed edges and replace them with undirected edges. This gives Figure 9-6-3c which is known as a Haase Diagram or ordering diagram and represents the poset in an economical visual fashion.

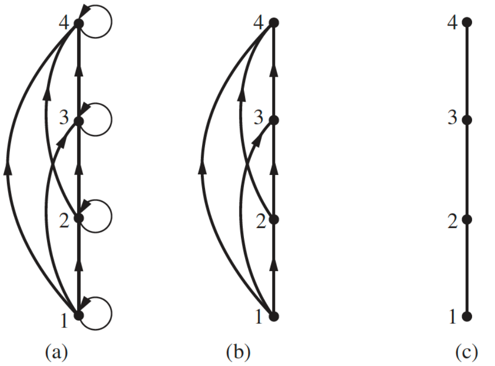


Table 5.1: Construction of a Hasse Diagram

5.10.5 Maximal and Minimal Elements

When discussing the basic axioms of arithmetic that a student can assume in proofs, we introduced the Completeness Property (Definition 2.2.1). We now add to that.

Posets have properties that are useful to some applications, in particular maximal and minimal elements.
Def 5.10.8 (Upper and Lower Bounds)**.** An element of a poset is called maximal if it is not less than any element o f the poset. That is, a is **maximal** in the poset $(S, \preceq)$ if there is no $b \in S$ such that $a \prec b$. Similarly, an element of a poset is called minimal if it is not greater than any element of the poset. That is, a is **minimal** if there is no element $b \in S$ such that $b \prec a$. Maximal and minimal elements are the top and bottom elements in a Haase Diagram.

If an element a is called the **greatest element** of the poset S if $b \preceq a$ for all $b \in S$. The greatest element is unique when it exists. Likewise, an element is called the least element if it is less than all the other elements in the poset. The element a of a poset is the **least element** of $(S, \preceq)$ if $a \preceq b$ for all $b \in S$. The least element is unique when it exists.

Def 5.10.9 (Least Upper Bound)**.** The element x is called the **least upper bound** of the subset A if x is an upper bound that is less than every other upper bound of A. Because there is onlyu one such element, if it exists, it makes sense to call this element *the* least upper bound. That is, x is the least upper boundf of A if $a \preceq x$ whenever $a \in A$, and $x \preceq z$ whenever z is an upper bound of A. Similarly, the ... We denote these as glb(A) and lub(A).

5.10.6 Lattices

Def 5.10.10 (Lattice)**.** A partially ordered set in which every pair of elements has both a least upper bound and a greatest lower bound is called a **lattice**.

5.10.7 Topological Sorting

Def 5.10.11 (Topological Sort)**.** A total ordering $\preceq$ is said to be **compatible** with the partial ordering R if $a \preceq b$ whenever aRb. Constructing a compatible total ordering from a partial ordering is called **topological sorting** or as some mathematicians will say, a linearization of a partial ordering.
Lemma 5.10.1. Every finite nonempty poset $(S, \preceq)$ has at least one minimal element.

5.11 n-ary Relations and Their Applicaiton for Databases

An innovation of database technology was the application of relations.
Def 5.11.1. Let $A_1, A_2, \ldots, A_n$ be sets and an n-ary relation on these sets as a subset of $A_1 \times A_2 \times \ldots \times A_n$. The sets $A_1, A_2, \ldots, A_n$ are called the domains of the relation, and n is called its *degree*.

Let the terms of the n-tuple represent datafields. Then each domain is a set of values that the datafield can take on. Each tuple in the relation can be treated like a record in a data processing system and the collection of them can be presented in a table where each column represents each of the elements drawn from the domains. By putting them into an n-tuple, we establish a relationship among those elements. We call this part of a relational data model.

Often the fields have a functional relation to one or more other elements. That is, given the key field, the other element in the domain is uniquely determined. When such a functional relationship exists from one of the domains to another, we say the pre-image is the key to the table. If this holds for all the terms in the tuple we call this field the primary key for the table.

We define an operation on a table called selection:
Def 5.11.2. Let R be an n-ary relation and C a predicate that elements of R may satisfy. Then the selection operator s_c maps the n-ary relation R to the n-ary relation from R of all n-tuples from R that satisfy the predicate C.

Chapter 6

Algorithms

Relative to many texts on discrete math, this chapter is sparse because most of the material overlaps with a standard undergraduate course that will focus on algorithms and their analysis. The emphasis for these notes is simply the mathematical concept and its relationship to the structures needed to approach computation from a theoretical perspective.

Function definitions must determine a unique value for the argument(s) given. While mathematical formulas are common, they are not required. What IS required is a definite process by which one finds the value in the co-domain to return as the function.

Consider that the argument does not need to be a single value. It can be a structure such as a sequence. The function can be on that sequence and return another sequence that possess some property. The classic example is a sort of that sequence. The input is some perhaps unsorted sequence and what is returned has the property that each element of the sequence is less than or equal to the ones later in the sequence. This is the beginning point for the study of those processes which defines functions like sort.

A pre-requisite to this class is at least one programming course so the student is expected to understand the conventions of a procedural language used to describe an algorithm. There is no universal way to describe an algorithm since programming languages will often use different ways of doing the same thing. The conventions of this book come from Rosen which represents a common notation that uses conventions from mathematics as well as programming languages.

Def 6.0.1 (Definition of Algorithm from Rosen)**.** An *algorithm* is a finite set of precise instructions for performing a computation or for solving a problem.

Def 6.0.2 (Definition of Algorithm from Schaums)**.** An algorithm M is a finite step-by-step list of well-defined instructions for solving a particular problem, for instance to evaluate a function f given an argument X where X may be any mathematical structure. Frequently there may be more than one algorithm to computer $f(X)$ and we need tools to analyze the differences between the different algorithms.

Def 6.0.3 (PseudoCode)**.** Pseudo code is a form of non-fiction prose that attempts to provide a rigorous definition of an algorithm using a natural language. Computer languages such as C or Algol are formal languages that conform to a syntax that allows them to be translated into computer code. But when discussing algorithms the overhead of those formal languages does not help understanding.

The assignment statement will perform a binding of a value to a variable, called a state transition. Most computer languages use the equal sign (=) to indicate assignment. That is unfortunate since it is easy to confuse it with the equality relationship operator which must use a different operator (typically == in many popular languages). Historically there are two other symbols used to indicate assignment including ":=" and "←".

Originally algorithms only needed to be understood by a human who had the job title of computer. The other statements should be easily understood by a student who has taken a programming course.

Algorithm 1 Finding the Maximum Element in a Finite Sequence

```
procedure MAX($a_1, a_2, \dots a_n$: integers)
    $max \leftarrow a_1$
    for $i \leftarrow 2, n$ do                    ▷ We have the answer if r is 0
        if $max < a_i$ then
            $max \leftarrow a_i$
        end if
    end for
return $max$
end procedure
```

Algorithm 2 Constructing Base *b* Expansions

```
procedure BASE B EXPANSION($n$: positive integer)    ▷ The g.c.d.
of a and b
    $q \leftarrow n$
    $k \leftarrow 0$
    while $q \neq 0$ do                           ▷ We have the answer if r is 0
        $a_k \leftarrow q \bmod b$
        $q \leftarrow \lfloor q/b \rfloor$
        $k \leftarrow k + 1$
    end while
return $(a_{k-1} \dots a_1 a_0)_b$
end procedure
```

6.1 Searching Algorithms

6.2 Sorting Algorithms

6.3 Topological Sort Algorithm

Algorithm 3 Topological Sorting

```
procedure TOPOLOGICAL SORT(($S, \preceq$): a finite poset)    ▷
    $k \leftarrow 1$
    while ( do $S \neq \emptyset$)
        $a_k \leftarrow$ a minimal element of $S$    ▷ guaranteed to exist by Lemma 5.10.1
        $S \leftarrow S - \{a_k\}$
        $k \leftarrow k + 1$
    end whilereturn $a_1, a_2, \ldots, a_n$    ▷ a compatible total ordering of $S$
end procedure
```

6.4 The Halting Problem

As a practical matter we know that programs that do not halt are of little use when it comes to computing functions. At a theorectical level it is interesting to consider whether it would be possible to construct a compiler that could recognize a program that would not halt. This leads us to this famous theorem of computer science.
Theorem 6.4.1 (Halting Problem)**.** There is no algorithm that accepts an algorithm and some input data set as input and determines whether it will halt.

Every function has a method by which a value can be calculated. This often comes from calculus for many of the functions used in engineering but not always. For example we have a way of calculating the factorial function that uses only basic arithmetic. Regardless of how it is calculated you will note that they all have very specific steps that must be executed in some order that are well defined. This is called an algorithm. Computer science began with the proof that any well defined algorithm could be mechanically executed by a machine instead of a human. The term computer originally meant a human using computation tools like an adding machine to execute the steps needed to determine the values. These values of the functions were published in books and the function was evaluated by a lookup in a table instead of on-the-spot calculation.

Not all functions take in single parameters. Some, like the permutation function, will take in a large number of values and produce some permutation of those numbers. Sorting is just such a function. Clearly the work done by the sorting function will depend on the size of the input set which we call n.

For practical reasons, we are interested in how long an algorithm will take given a particular set of numbers as input. This becomes a different function T for this algorithm which has as input the size of the input set, n. The time a specific algorithm will work on a particular input set before giving a result is the functionT(n). You might naively assume that T(n) will always result in the same value for any size input but you would be wrong. The T function will vary with both the size but also the specific input. For example some sort algorithms will have T(n) in the category of O(n) for sorted input but O(n**2) for out of order input. The big O of the algorithm will vary with the algorithm and its input.

While we are sometimes interested in the best-case scenario for the algorithm and its input set, we most often are more interested in the worst case scenario. What will be the big O for this algorithm over all the possible inputs it could get? This defines an upper bound for the complexity of this algorithm. Since the worst and best case for an algorithm may have functions from different categories, the best case may be in a different big O category. When analyzing algorithms, we call this the lower bound for that function on T(n). We give this the name Omega. When we find one category that contains both the upper and lower bound, we call this the Theta of that function. When we find a Theta for an algorithm we say we have found the tight upper bound

We defer the further study of algorithmic complexity to your course on Algorithms in your undergrad program.

6.5 Analysis of Algorithms

This section overlaps with a course usually required on the analysis of algorithms. We attempt to limit the material here to only the most basic concepts.

6.5.1 The Time Function of an Algorithm

We want to analyze how long an algorithm takes, called the time complexity of the algorithm. First note that many algorithms will use different numbers of steps depending upon the input they are given. So this is not a fixed function from algorithm to time but instead a range of values. The time might be computable for specific algorithm and a specific input but in general what we want is the range of possible times for this algorithm. Of all the possible ways we can analyze the performance of an algorithm the most basic is to look at the relationship between the “size” of the input and the time ignoring all other considerations. We represent this “size” as N. We call the time function T so we want to perform asymptotic analysis on the function $T(N)$. You might naively expect that a given algorithm will have a fixed relationship to the size of the input but that is wrong. Sorting algorithms in particular will have different time functions for the same size input but input in different orders. This complicates the analysis of the time function of an algorithm and gives us a range of functions.

We know that a computer does not perform all instructions with the same amount of time. A division function will be slower than a move instruction. But the relationship between the time and the instruction can be set at a max and all instructions treated as the same length. This is not useful for predicting how long the algorithm will take but it can be used to determine the order of magnitude since the fixed multiple will not impact the growth. With this in mind, we simply count how many times each instruction is executed by the algorithm with no consideration for the actual time the computer will take with each one.

Given this range, we might be interested in the best possible time, the worst possible time and/or the average time this algorithm will use over some set of input values. The most frequently analyzed is the worst case time and we present how that is described. We will use asymptotic function analysis (chapter on functions) to do this.

While it is true that the time required to execute an instruction varies with the machine and even within a machine the instruction to be executed. For example a divide instruction is longer than a move. But this relationship is fixed, for the most part, and can be ignored for the purposes of asymptotic analysis.

The most common analysis is relative to the economic reality of time. But a comparable analysis can be done for space requirements too. In fact any resource needed to run the algorithm can be subjected to analysis in a comparable way.

Since time functions will vary in their order based on the input, we are often asked to consider the best case and worst case performance of these algorithms when presented with input. It is common with sorting algorithms that the best case and worst case will be of different orders. It is often trivial to say that a given function g is of a higher order, for example a quartic polynomial will beat a quadratic, but we often want to know the LEAST upper bound function. To do the analysis we must have a concept of a lower bound as well. We use **Big-Omega (big-Ω) notation**. If, and only if, we find a single function that serves as both an upper and lower bound, we say that the function is Big Theta.

As a practical matter we usually state big-O, big-Omega and big-Theta in terms of a limited number of reference functions that come up frequently in algorithms.

Determining the dominant factor of a polynomial which determines the order is easy.

We stop our discussion of algorithmic analysis here and leave it to a course devoted to this subject.

Chapter 7

Integers

A math major will be expected to complete a course in Number Theory but few computer science programs require it. Yet there are a few concepts from number theory that are important for a computer science or software engineering student to understand to be prepared for either a career in software engineering or the pursuit of graduate school in computer science. This chapter seeks to limit the material to those two objectives only.

7.1 Division

We introduced the division relationship in a prior chapter. Here it is again.
Def 7.1.1. If a and b are integers with $a \neq 0$, we say that a *divides* b if there is an integer c such that $b = ac$. When a divides b we say that a is a *factor* of b and that b is a *multiple* of a. The notation $a|b$ denotes that a divides b. We write $a \not| b$ when a does not divide b.

When the domain of discourse is the set of integers, we can express $a|b$ as $\exists c(ac = b)$.
Theorem 7.1.1. Let a, b, and c be integers. Then

1. if $a|b$ and $a|c$, then $a|(b + c)$;

2. if $a|b$, then $a|bc$ for all integers c;

3. if $a|b$ and $b|c$, then $a|c$.

Corollary 7.1.1. If a, b, and c are integers such that $a|b$ and $a|c$, then $a|mb+nc$ whenever m and n are integers.

Theorem 7.1.2 (The Division "Algorithm")**.** Let a be an integer and d a positive integer. Then there are unique integers q and r, with $0 \le r < d$, such that $a = dq + r$.

Def 7.1.2. In the equality given in the divison algorithm d is called the *divisor*, a is called the *dividend*, q is called the *quotient* and r is called the *remainder*. This notation is used to express the quotient and remainder:

$$q = a \,\mathbf{div}\, d, r = a \;\mathbf{mod}\;\; d.$$

Notes. The binary operator **div** in this definition is the same as a programming integer divide. Some languages use the percentage sign ("%") as the remainder operator. This definition of mod defines a function since r is uniquely determined.

Note how this is calculated for negative integers.

Note how some programming languages had not always calculated this properly.

7.2 Modular Arithmetic

In some applications, we only care about the remainder. Consider time or degrees. We have special notation for this concept.

Def 7.2.1. If a and b are integers and m is a positive integers, then a is *congruent to b modulo m* if m divides $a-b$. We use the notation $a \equiv b$ (modulo m) or $a \equiv b(modm)$

Notes. Be careful not to confuse **mod**, a function with (mod m). The first is a binary function used as an infix operator while the second is a qualification for the entire equivalence. Authors use different notations to mean one or the other. The (modulo m) equivalence is NOT a function since one integer can be in equivalence to multiple integers, modulo m.

Theorem 7.2.1. Let a and b be integers, and let m be a positive integer. Then $a \equiv b$ (mod m) if and only if $a\mathbf{mod}m = b\mathbf{mod}m$

Theorem 7.2.2. Let m be a positive integer. The integers a and b are congruent modulo m if and only if there is an integer k such that $a = b + km$.
Theorem 7.2.3. Let m be a positive integer. Then:
for any integer a, we have $a \equiv a \pmod{m}$,
if $a \equiv b \pmod{m}$, then $b \equiv a \pmod{m}$,
if $a \equiv b \pmod{m}$ and $b \equiv c \pmod{m}$, then $a \equiv c \pmod{m}$.
Notes. Any integer a is congruent modulo m to a unique integer in the set

$$\{0, 1, 2, \ldots, m-1\}$$

The uniqueness comes from the fact that m cannot divide the difference of two such integers.
Any two integers a and b are cngruent module m if and oly if they have the same remainder when divided by m.

7.2.1 Residue Classes

Congruence modulo m is an equivalence relation and therefore partitions $\mathbb{Z}$ into disjoint equivalence classes called the *residue classes modulo m*. There are m such classes and each contains exactly one of the elements of the set $\{0, 1, \ldots m-1\}$. A set of m integers $\{a_1, a_2, \ldots, a_m\}$ is called a *complete residue system modulo m* if each a_i comes from a distinct residue class. In such a case, each a_i is called a *representative* of its equivalnece class. We denote such a class as $[x]_m$, or sometimes $[x]$for those integers.

$$[x] = \{a \in \mathbb{Z} | a \equiv x \pmod{m}\}$$

The residue classes can be denoted by

$$[0], [1], \ldots, [m-1]$$

or any other integers drawn from a complete residue system.

Addition and multiplication of residue classes modulo m behaves like equality:

$$[a] + [b] = [a + b]$$

$$[a] \cdot [b] = [ab]$$

Theorem 7.2.4 (Modified Cancellation Law for Congruences)**.** Suppose $ab \equiv ac \pmod{m}$ and gcd$a, m = d$, then $b \equiv c \pmod{m/d}$ (where the division m/d is integer division).
Corollary 7.2.1. Suppose p is prime and $ab \equiv ac \pmod{p}$ and $a \neq 0$ (mod p), then $b \equiv c \pmod{p}$.
Theorem 7.2.5 (Congruence Arithmetic)**.** Let m be a positive integer. If $a \equiv b$ (mod m) and $c \equiv d$ (mod m), then $a + c \equiv b + d$ (mod m) and $ac \equiv bd$ (mod m).
Def 7.2.2 (Congruence Classes)**.** The set of all integers congruent to an integer a modulo m is called the **congruence class** of a modulo b or the **congruence classes modulo** m.

Under the congruence relation, addition and multiplication behave like addition.
Corollary 7.2.2 (Congruence Arithmetic)**.** Let m be a positive integer and let a and b be integers. Then

$$(a+b)(\mod m) = ((a \mod m) + (b \mod m)) \mod m$$

and

$$ab \mod m \equiv ((a \mod m)(b \mod m) \mod m$$

Notes. Suppose $p(x)$ is a polynomial with integral coefficients, repeated use of this corollary will prove that if $s \equiv t \pmod{m}$ then $p(s) \equiv p(t) \pmod{m}$.
Def 7.2.3. The number of residue classes relatively prime to m or, equivalently, the number of integers between 1 and m(inclusive) which are relatively prime to m is denoted by $\phi(m)$. Any list of $\phi(m)$ numbers which are coprime to m is called the *reduced residue system modulo m*.

7.3 Cryptology

7.4 Primes and Greatest Common Divisors

Def 7.4.1 (Prime Numbers)**.** A positive integer p greater than 1 is called *prime* if the only positive factors of p are 1 and p. A positive integer than is greater than 1 and is not prime is called *composite*.

Notes. The integer n is composite if and only if there exists an integer a such that $a|n$ and $1 < a < n$.
Theorem 7.4.1 (The Fundamental Theorem of Arithmetic)**.** Every positive integer greater than 1 can be written uniquely as a prime or as the product of two or more primes where the prime factors are written in order of nondecreasing size.
Theorem 7.4.2 (Composite Number)**.** If n is a composite integer, then n has a prime divisor less than or equal to $\sqrt{n}$
Theorem 7.4.3 (The Infinitude of Primes)**.** There are infinitely many primes.
Def 7.4.2. The number of primes not exceeding some real number x is defined as the function $\pi(x)$.
Theorem 7.4.4 (The Prime Number Theorem)**.** The ratio of $\pi(x)$, the number of primes not exceeding x, and the expression $\frac{x}{\ln x}$ approaches 1 as x grows without bound.
$\frac{\pi(x)}{x/lnx}$

7.4.1 Greatest Common Divisors and Least Common Multiples

Def 7.4.3 (GCD, Greatest Common Divisor)**.** Let a and b be integers, not both zero. The largest integer d such that $d|a$ and $d|b$ is called the *greatest common divisor* of a and b. The greatest common divisor of a and b is denoted by gcd(a,b).
Def 7.4.4. The integers a and b are *relatively prime* if their greatest common divisor is 1.
Def 7.4.5. the integers $a_1, a_2, \ldots a_n$ are *pairwise relatively prime* if gcd(a_i, a_j)=a whenever $1 \leq i < j \leq n$.
Def 7.4.6 (Least Common Multiple)**.** The *least common multiple* of the positive integers a and b is the smallest positive integer that is divisible by b oth a and b. The least common multiple of a and b is denoted by lcm(a,b).
Def 7.4.7 (Euler phi Function)**.** The value of the **Euler ϕ-function** at the positive integer n is defined to be the number of positive integers less than or equal to n that are relatively prime to n.

7.5 Integers and Algorithms

7.5.1 Representation of Integers

Theorem 7.5.1 (Base b Expansion of n)**.** Let b be a positive integer greater than 1. Then if n is a positive integer, it can be expressed uniquely in the form

$$n = a_k b^k + a_{k-1} b^{k-1} + \cdots + a_1 b + a_0,$$

where k is a nonnegative integer, $a_0, a_1, \ldots, a_k$ are nonnegative integers less than b, and $a_k \neq 0$. Choosing $b = 1$ gives the conventional decimal expansion and choosing $b = 2$ gives the binary expansion. Choosing $b = 16$ gives the hexadecimal expansion. Choosing $b = 8$ give the octal expansion.
Notes. Each hexadecimal digit is represented by four binary digits. Each octal digit is represented by three binary digits.

Algorithm for the construction of a base b expansion.

Algorithm for the addition of integers.

Algorithm for the multiplication of integers. Algorithm for the computation of div and mod.

7.5.2 Modular Exponentiation

Algorithm for modular exponentiation. You will find that in cryptography it is important to be able to find $b^n \mathbf{mod} m$ efficiently...

7.6 The Euclidean Algorithm

Lemma 7.6.1. Let $a = bq + r$, where a,b,q, and r are integers. Then gcd(a,b)=gcd(b,r)

Algorithm, The Euclidean Algorithm
Theorem 7.6.1 (Bézout's Theorem)**.** If a and b are positive integers, then there exist integers s and t such that gcd(a,b)=$sa + tb$

Algorithm 4 Algorithm for the Addition of Integers

```
procedure INTEGER ADDITION($a, b$: positive integers)
                                        ▷ the binary expansions of $a$ is
                                        ▷ $(a_{n-1}a_{n-2}\ldots a_1a_0)_2$
                                        ▷ and the binary expansion of b is
                                        ▷ $(b_{n-1}b_{n-2}\ldots b_1b_0)_2$.

    $c \leftarrow 0$
    for $j \leftarrow 0, n-1$ do
        $d \leftarrow \lfloor (a_j + b_j + c)/2 \rfloor$
        $s_j \leftarrow a_j + b_j + c - 2d$
        $c \leftarrow d$
    end for
    $s_n \leftarrow c$
return $s_n$
                                        ▷ where $s_ns_{n-1}\ldots s_0)_2$
end procedure
```

Algorithm 5 The Euclidean Algorithm

```
procedure GCD($a, b$: positive integers)
    $x \leftarrow a$
    $y \leftarrow b$
    while $y \neq 0$ do
        $r \leftarrow x$ mod $y$
        $x \leftarrow y$
        $y \leftarrow r$
    end while
    return $x$
end procedure
```

Algorithm, the extended Euclidean algorithm to find linear combinations.
Lemma 7.6.2. If a,b, and c are positive integers such that gcd(a,b)=a and $a|bc$, then $a|c$.
Lemma 7.6.3. If p is a prime and $p|a_1a_2\dots a_n$ where each a_i is an integers, then $p|a_1$ for some i.
Theorem 7.6.2. Let m be a positive integer and let a,b and c be integers. If $ac \equiv bc(\mod m)$ and gcd(c,m)=1, then $a \equiv b(\mod m)$

7.7 Linear Congruences

A congruence of the form

$$ax \equiv b(\mod m),$$

where m is a positive integer, a and b are integers, and x is a variable, is called a **linear congruence**.
Def 7.7.1. The **inverse of** a **modulo** m is that integer $\bar{a}$ such that $\bar{a}a \equiv 1(\mod m)$, when $\bar{a} exists$.
Theorem 7.7.1. If a and m are relatively prime integers and $m > 1$, then an inverse of a modulo m exists. Furthermore, this inverse is unique module m. (That is, there is no other inverse smaller than m).

7.8 The Chinese Remainder Theorem

The Chinese Remainder Theorem has an important application in the arithmetic of large numbers.
Theorem 7.8.1 (Chinese Remainder Theorem)**.** Let $m_1,m_2,\dots,m_n$ be pairwise relatively prime positive integers and $a_1,a_2,\dots,a_n$ be arbitrary integers. Then the system

$$\begin{aligned} x &\equiv a_1(\mod m_1)\\ x &\equiv a_2(\mod m_2)\\ &\vdots\\ x &\equiv a_n(\mod m_n) \end{aligned} \tag{7.1}$$

has a unique solution modulo $m = m_1m_2\dots m_n$.

7.9 Computer Arithmetic with Large Integers

To perform arithmetic with large integers, we select moduli $m_1, m_2, \ldots, m_n$, where each m_i is an integer greater than 2, $\gcd(mi, mj) = 1$ whenever $i = j$, and $m = m_1 m_2 \ldots m_n$ is greater than the results of the arithmetic operations we want to carry out.

Once we have selected our moduli, we carry out arithmetic operations with large integers by performing componentwise operations on the n-tuples representing these integers using their remainders upon division by $m_i, i = 1, 2, ..., n$. Once we have computed the value of each component in the result, we recover its value by solving a system of n congruences modulo $m_i, i = 1, 2, ..., n$. This method of performing arithmetic with large integers has several valuable features. First, it can be used to perform arithmetic with integers larger than can ordinarily be carried out on a computer. Second, computations with respect to the different moduli can be done in parallel, speeding up the arithmetic.

Particularly good choices for moduli for arithmetic with large integers are sets of integers of the form $2k - 1$, where k is a positive integer, because it is easy to do binary arithmetic modulo such integers, and because it is easy to find sets of such integers that are pairwise relatively prime. A second reason is a consequence of the fact that $\gcd(2a - 1, 2b - 1) = 2\gcd(a, b) - 1$.
Theorem 7.9.1 (Fermat's Little Theorem)**.** If p is prime and a is an intege not divisible by p, then $a^{p-1} \equiv 1(\mod p)$. Furthermore, for every integer a we have $a^p \equiv a(\mod p)$

7.9.1 Pseudo Primes

Def 7.9.1. Let b be a positive integer. If n is a composite positive integer, and $b^{n-1} \equiv 1(\mod n)$, then *n is called a pseudoprime to the base b*.
Def 7.9.2. A composite integer n that satisfies the congruence $b^{n-1} \equiv (\mod n)$ for all positive inteers b with $\gcd(b, n) = 1$ is called a *Carmichael number*.

Although there are infinitely many Carmichael numbers, more delicate tests, described in the exercise set, can be devised that can be used as the basis for efficient probabilistic primality tests. Such tests can be used to quickly show that it is almost certainly the case that a given integer is prime. More precisely, if an integer is not prime, then the probability that it passes a series of tests is close to 0. We will describe such a test in Chapter 7 and discuss the notions from probability theory that this test relies on. These probabilistic primality tests can be used, and are used, to find large primes extremely rapidly on computers.

7.10 Cryptography

Number theory plays a key role in cryptography, the subject of transforming information so that it cannot be easily recovered without special knowledge. Number theory is the basis of many classical ciphers, first used thousands of years ago, and used extensively until the 20th century. These ciphers encrypt messages by changing each letter to a different letter, or each block of letters to a different block of letters. We will discuss some classical ciphers, including shift ciphers, which replace each letter by the letter a fixed number of positions later in the alphabet, wrapping around to the beginning of the alphabet when necessary. The classical ciphers we will discuss are examples of private key ciphers where knowing how to encrypt allows someone to also decrypt messages.With a private key cipher, two parties who wish to communicate in secret must share a secret key. The classical ciphers we will discuss are also vulnerable to cryptanalysis, which seeks to recover encrypted information without access to the secret information used to encrypt the message.We will show how to cryptanalyze messages sent using shift ciphers. Number theory is also important in public key cryptography, a type of cryptography invented in the 1970s. In public key cryptography, knowing how to encrypt does not also tell someone howto decrypt. The most widely used public key system, called the RSA cryptosystem, encrypts messages using modular exponentiation, where the modulus is the product of two large primes. Knowing how to encrypt requires that someone know the modulus and an exponent. (It does not require that the two prime factors of the modulus be known.) As far as it is known, knowing how to decrypt requires someone to know how to invert the encryption function, which can only be done in a practical amount of time when someone knows these two large prime factors. In this chapter we will explain how the RSA cryptosystem works, including how to encrypt and decrypt messages. The subject of cryptography also includes the subject of cryptographic protocols, which are exchanges of messages carried out by two or more parties to achieve a specific security goal.We will discuss two important protocols in this chapter. One allows two people to share a common secret key. The other can be used to send signed messages so that a recipient can be sure that they were sent by the purported sender.

7.10.1 Public Key Cryptography

All classical ciphers, including shift ciphers and affine ciphers, are examples of **private key cryptosystems**. In a private key cryptosystem, once you know an encryption key, you can quickly find the decryption key. So, knowing how to encrypt messages using a particular key allows you to decrypt messages that were encrypted using this key. For example, when a shift cipher is used with encryption key k, the plaintext integer p is sent to

$$c = (p + k) \bmod 26$$

Decryption is carried out by shifting by $-k$; that is,

$$p = (c - k) \bmod 26$$

So knowing how to encrypt with a shift cipher also tells you how to decrypt. When a private key cryptosystem is used, two parties who wish to communicate in secret must share a secret key. Because anyone who knows this key can both encrypt and decrypt messages, two people who want to communicate securely need to securely exchange this key. (We will introduce a method for doing this later in this section.) The shift cipher and affine cipher cryptosystems are private key cryptosystems. They are quite simple and are extremely vulnerable to cryptanalysis. However, the same is not true of many modern private key cryptosystems. In particular, the current US government standard for private key cryptography, the Advanced Encryption Standard (AES), is extremely complex and is considered to be highly resistant to cryptanalysis. (See [St06] for details on AES and other modern private key cryptosystems.) AES is widely used in government and commercial communications. However, it still shares the property that for secure communications keys be shared. Furthermore, for extra security, a new key is used for each communication session between two parties, which requires a method for generating keys and securely sharing them. To avoid the need for keys to be shared by every pair of parties that wish to communicate securely, in the 1970s cryptologists introduced the concept of **public key cryptosystems**. When such cryptosystems are used, knowing how to send an encrypted message does not help decrypt messages. In such a system, everyone can have a publicly known encryption key. Only the decryption keys are kept secret, and only the intended recipient of a message can decrypt it, because, as far as it is currently known, knowledge of the encryption key does not let someone recover the plaintext message without an extraordinary amount of work (such as billions of years of computer time).

7.10.2 The RSA Cryptosystem

In 1976, three researchers at the Massachusetts Institute of Technology—Ronald Rivest, Adi Shamir, and Leonard Adleman—introduced to the world a public key cryptosystem, known as the RSA system, from the initials of its inventors. As often happens with cryptographic discoveries, the RSA system had been discovered several years earlier in secret government research in the United Kingdom. Clifford Cocks, working in secrecy at the United Kingdom's Government Communications Headquarters (GCHQ), had discovered this cryptosystem in 1973. However, his invention was unknown to the outside world until the late 1990s, when he was allowed to share classified GCHQ documents from the early 1970s. (An excellent account of this earlier discovery, as well as the work of Rivest, Shamir, and Adleman, can be found in [Si99].) In the RSA cryptosystem, each individual has an encryption key (n,e) where

$n = pq$, the modulus is the product of two large primes p and q, say with 200 digits each, and an exponent e that is relatively prime to $(p-1)(q-1)$.To produce a usable key, two large primes must be found. This can be done quickly on a computer using probabilistic primality tests, referred to earlier in this section. However, the product of these primes $n = pq$, with approximately 400 digits, cannot, as far as is currently known, be factored in a reasonable length of time. As we will see, this is an important reason why decryption cannot, as far as is currently known, be done quickly without a separate decryption key.

To encrypt messages using a particular key (n, e), we first translate a plaintext message M into sequences of integers. To do this, we first translate each plaintext letter into a two-digit number, using the same translation we employed for shift ciphers, with one key difference. That is, we include an initial zero for the letters A through J, so that A is translated into 00, B into 01, . . . , and J into 09. Then, we concatenate these two-digit numbers into strings of digits. Next, we divide this string into equally sized blocks of $2N$ digits, where $2N$ is the largest even number such that the number 2525 . . . 25 with $2N$ digits does not exceed n. (When necessary, we pad the plaintext message with dummy Xs to make the last block the same size as all other blocks.) After these steps, we have translated the plaintext message M into a sequence of integers $m_1, m_2, ..., m_k$ for some integer k. Encryption proceeds by transforming each block m_i to a ciphertext block c_i . This is done using the function

$$C = M^e \mod n.$$

(To perform the encryption, we use an algorithm for fast modular exponentiation, such as Algorithm 5 in Section 4.2.) We leave the encrypted message as blocks of numbers and send these to the intended recipient. Because the RSA cryptosystem encrypts blocks of characters into blocks of characters, it is a block cipher.

The plaintext message can be quickly recovered from a ciphertext message when the decryption key d, an inverse of $emodulo(p-1)(q-1)$, is known. [Such an inverse exists because $\gcd(e, (p-1)(q-1)) = 1$.] To see this, note that if $de \equiv 1($

mod $(p-1)(q-1))$, there is an integer k such that $de = 1 + k(p-1)(q-1)$. It follows that

$$C^d \equiv (M^e)^d = M^{de} = M^{1+k(p-1)(q-1)}(\mod n).$$

By Fermat's little theorem [assuming that $\gcd(M,p) = \gcd(M,q) = 1$, which holds except in rare cases, which we cover in Exercise 28], it follows that $M^{p-1} \equiv 1(\mod p)$ and $M^{q-1} \equiv 1(\mod q)$. Consequently,

$$C^d \equiv M \cdot (M^{p-1})^{k(q-1)} \equiv M \cdot 1 \equiv M(\mod p)$$

and

$$C^d \equiv M \cdot (M^{q-1})^{k(p-1)} \equiv M \cdot 1 = M(\mod q).$$

Because $\gcd(p,q) = 1$, it follows by the Chinese Remainder Theorem that

$$C^d \equiv M(\mod pq).$$

Chapter 8

Induction and Recursion

recursive definitions of sets

Recursion occurs throughout the study of computer science and this chapter must give you the basis to be able to understand the many ways in which it serves our discipline. The first is the concept of proving properties of infinite sets using a recursive argument. All recursive arguments have two parts that have the ladder as a metaphor. Ladders have rungs that are equally spaced. And one you are on a rung, if you can step up to the next one you know you can climb the ladder regardless how tall it may be. But the other vital part of the argument is that you can get onto the ladder in the first place. From this start we can explore the recursive definition of sets, recursive functions and relations and recursive algorithms. proving a truth for an infinite set

Note: Induction versus Deduction Induction has two meanings. In common language induction is a way of generalizing over many observations. But in mathematics we use the word to describe a particular type of deductive reasoning that can prove that some properties of infinite sets must be true. We call this inductive reasoning.

8.1 Mathematical Induction

Notes. Do not confuse mathematical induction with inductive reasoning. In logic deductive reasoning uses rules of inference to draw conclusions from premises, whereas inductive reasong makes conclusions only supported, but not ensured, by evidence. Mathematical proofs, incuding arguments that use mathematical induction, are deductive, not inductive.

Def 8.1.1 (Principle of Mathematical Induction)**.** To prove that $P(n)$ is true for all positive integers n< where $P(n)$ is a propositional function, we complete two steps:
BASIS STEP: Verify that $P(1)$ is true.
INDUCTIVE STEP: Show that the conditional statement $P(k) \to P(k+1)$ is true for all positive integers k.

inductive hypothesis, ASSUME $P(k)$ is true. This is not the same as asserting it is true for all k only that if the assumption holds that $P(k+1)$ must be true. Expressed as a rule of inference it looks like this:

$$[P(1) \wedge \forall k(P(k) \to P(k+1))] \to \forall n P(n),$$

8.1.1 Mathematical Induction

If a set is finite and we want to show some property holds for all elements, an iteration of testing for that property is always possible. But what if the set we examine is infinite? Many properties in mathematics are on infinite sets such as the set of integers or reals. Common examples are proving that for every positive integer n, $n! \leq n^n$. In general we are given some predicate P (in this example that $n! \leq n^n$ and then asked to prove that this is a univeral property

$$\forall P(n), n \in \mathbb{N} \cup 0 | n! \leq n^n$$

For this and many other proofs of this kind we depend upon the Principle of Mathematical Induction. First prove that the proposition is true for the first element of the set. Then prove that if the propertly holds for some arbitrary element, k, that it must hold for the subsequent element, $k+1$. The principle gives you a new rule of inference that allows you to assert that the property must hold for

every element of the set. More formally we state the principle of mathematical induction like this:
Basis Step: Prove $P(1)$
Inductive Step: $P(k) \rightarrow P(k+1)$
Rule of Inference: $[P(1) \wedge \forall P(k) \rightarrow Pk+1] \rightarrow \forall nP(n)$
We will call this the weak form of mathematical induction.

Note: The proofs always use a different variable name in the proof from the principle to be proved. In the example we were asked to prove for all n. But the proof worked with variable k. This is to emphasize that we may not assume that $P(k)$ is true. But we can still use it in a conditional expression that if $P(k)$ is true and we show that the implication that $P(k+1)$ must also be true, it does not matter if the antecedent of the conditional statement is false since we only need to show that in all cases where it is true that the consequent must also be true. Ignoring this subtle point of logic causes many people to commit the fallacy of affirming the consequent, or begging the question.

8.2 Strong Induction and Well-Ordering

Sometimes it is not sufficient to show that the universal is valid for $n = 1$ to establish a base case. Sometimes there are two or more initial conditions that must be proven. This gives us a new form of mathematical induction call the strong form:
Def 8.2.1 (Strong Induction)**.** To prove that $P(n)$ is true for all positiv eintegers n, where $P(n)$ is a propositional function, we complete two steps:
BASIS STEP: We verify that the proposition $P(1)$ is true.
INDUCTIVE STEP: We show that the conditional statement $[P(1) \wedge P(2) \wedge \ldots P(k)] \rightarrow P(k+1)$ is true for all positive integers k.
Def 8.2.2 (Well-Ordering Property)**.** Every nonempty set of nonnegative integers has a least element.

8.3 Recursive Definitions and Structural Induction

8.3.1 Recursively Defined Functions

We can define functions using a recursive definition. Such definitions are called recursive or inductive definitions. Such functions are well defined since the value of the function at every integer is uniquely defined in a nonambiguous way.

Def 8.3.1 (Recurseively Defined Function)**.** We can define a function with the set of nonnegative integers as its domain with two steps:
BASIS STEP: Specify the value of the function at zero (the base case)
RECURSIVE STEP: Give a rule for finding its value at an integer from from its values at smaller integers.

Def 8.3.2. The *Fibonacci numbers* $f_0, f_1, f_2, \dots$, are defined by the equaltions $f_0 = 0$,
$f_1 = 1$, and
$f_n = f_{n-1} + f_{n-2}$ for all $n \geq 2$.

8.3.2 Recursively Defined Sets and Structures

basis step, recursive step, exclusion rule An important use of recurseive definitions for sets is to define **well-formed formulae** of various types.

Def 8.3.3. [Well-Formed Formulae of Compound Propositions] Well-Formed formulae of a simple propositional statements can be defined as follows:
BASIS STEP: T, F, and s, where s is a propositional variable, are well-formed formulae.
RECURSIVE STEP: If E and F are well-formed formulae, then $(\neg E)$, $(E \wedge F)$, $(E \vee F)$, $(E \to F)$, and $(E \leftrightarrow F)$ are well-formed formulae.

Def 8.3.4 (Σ^*)**.** The set if all posible strings over the alphabet Σ can be defined recursively by:
BASIS STEP: $\lambda \in \Sigma^*$ where λ is the empty string containing no symbols).
RECURSIVE STEP: If $w \in \Sigma^*$ and $x \in \Sigma$, then the concatenation $wx \in \Sigma^*$.

A structure cal also be defined using a recursive definition.

Def 8.3.5 (The Set of Rooted Trees)**.** The set of rooted trees, where a rooted tree consists of a set of vertices containing a distinguished vertex called the root, and edges connectinve these vertices, can be defined recursively by these steps:
BASIS STEP: A single vertex r is a rooted tree.
RECURSIVE STEP: Suppose that $T_1, T_2, \ldots T_n$ are disjoint rooted trees with roots $r_1, r_2, \ldots r_n$, respectively. Then the graph formed by starting with a root r, which is not in any of the rooted trees $T_1, T_2, \ldots, T_n$, and adding an edge from r to each of the vertices $r_1, r_2, \ldots r_n$, is also a rooted tree.

8.3.3 Structural Induction

We can prove results about recursively defined sets using a form of mathematical induction. We do this by using a connection between recursively defined sets and mathematical induction.
Theorem 8.3.1. Let S be a set recursively defined by:
BASIS STEP: $3 \in S$
RECURSIVE STEP: If $x \in S$ and $y \in S$, then $x + y \in S$, then all elements of the set S are multiples of 3.
Def 8.3.6 (Proof by Structural Induction)**.** A proof by structural induction consists of two parts. These parts are: BASIS STEP: Show that the result holds for all elements specified in the basis steop of the recursive definition to be in the set.
RECURSIVE STEP: Show that if the statement is true for each of the elements used to construct new elements in the recursive step of the definition, the result holds for these new elements.
Theorem 8.3.2. Show that the set of well-formed formulae in 8.3.3 will always have an equal number of left and right parentheses using structural recursion.
Def 8.3.7. We define the height $h(T)$ of a full binary tree T recursively.
BASIS STEP: The height of the full binary tree T consisting of only a root r is $h(T) = 0$. RECURSIVE STEP: If T_1 and T_2 are full binary trees, then the full binary tree $T = T_1 \cdot T_2$ has height $h(T) = 1 + max(h(T), h(T_2))$.
Theorem 8.3.3. If T is a full binary tree and$n(T)$ defines the

number of nodes in a full binary tree T, then $n(T) \leq 2^{h(T)+1} - 1$

8.3.4 Generalized Induction

Def 8.3.8. The **reversal** of a string w is the string consisting of the symbols of the string in reverse order. The reversal of the string w is denoted w^R.

Ackermann's (ex 47)

8.4 Recursive Algorithms

Def 8.4.1. An algorithm is called *recursive* of iot solves a problem by reducing it to an instance of the same problem with smaller input.

8.4.1 Proving Recursive Algorithms Correct

8.4.2 Recursion and Iteration

A recursive algorithm will compute a value by taking the function evaluation at the requested value and recursively computing smaller values. The function can also be computed by taking the value at the base case and applying the recursive rule until the requested value is determined, changing a recursive algorithm into an interative algorithm.

8.4.3 The Merge Sort

Chapter 9

Counting

We need to count the operations of an algorithm to comput its time complexity.

9.1 The Basics of Counting

9.1.1 Basic Counting Principles

product rule, sum rule. To count means to put the elements in a set into a one-to-one correspondence, a mapping, from the set to the set of natural numbers. That means we find a function $f : \mathbb{N} \rightarrow S$. Any set that has a bijection to the natural numbers is said to be *countable*.

Theorem 9.1.1 (Sum Rule)**.** for two disjoint sets A and B $card(A \cup B) = card(A) + card(B)$

Theorem 9.1.2 (Product Rule)**.** For two disjoint sets A and B $card(A \times B) = card(A) * card(B)$

9.1.2 More Complex Counting Problems

9.1.3 The Inclusion-Exclusion Principle

9.1.4 Tree Diagrams

9.2 The Pigeonhole Pinciple

Theorem 9.2.1 (Pigeonhole Principle)**.** 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.
Corollary 9.2.1. A function f from a set with $k+1$ elements to a set with k elements is not one-to-one.

9.2.1 The Generalized Pigeonhold Principle

Theorem 9.2.2 (The Generalized Pigeonhole Principle)**.** If N objects are placed into k boxes, then there is a least one box containing at least $\lceil N/k \rceil$ objects.

9.2.2 Some Elegant Applications of the Pigeonhold Principle

Theorem 9.2.3. Every sequence of n^2+1 distinct real numbers contains a subsequence of length $n+1$ that is either strictly increasing or trictly decreasing.

9.3 Permutations and Combinations

9.3.1 Permutations

Def 9.3.1. A **permutation** of a set of distinct objects is an ordered arrangement of these objects. An ordered arragement of r elements of a set is called an **r-permutation**. An r-permutation of r objects drawn from a set of n things is denoted as $P(n,r)$.
Theorem 9.3.1. If n is a positive integer and r is an integer with $1 \le r \le n$, then there are

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

r-permutations of a set with n distinct elements.
Corollary 9.3.1. If n and r are integers with $0 \le r \le n$ then $P(n,r) = \frac{n!}{(n-r)!}$.

9.3.2 Combinations

Def 9.3.2. A **combination** of a set of distinct objects is an unorderd arrangement of these objects. An unordered arrangement of r elements of the set is called an **r-combination**. The number of r-combinations of a set with n distinct elements is denoted by $C(n,r)$ also denoted as $\binom{n}{r}$ and is called the **binomial coefficient**.
Theorem 9.3.2. The number of r-combinations of a set with n elements, where n is a nonnegative integer and r is an integer with $0 \le r \le n$, equals

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

Corollary 9.3.2. Let n and r be nonnegative integers with $r \le n$. Then $C(n,r) = C(n,n-r)$.
Def 9.3.3. A *combinatorial proof* of an identity is a proof that uses counting arguments to prove that both sides of the identtity count the same objects but in different ways.

9.4 Binomial Coefficients

9.4.1 The Binomial Theorem

The coefficients of the expansion of a binomial like $(a+b)^n$ are the binomial coefficients. They come up in many different contexts and are the subject of this section.
Theorem 9.4.1 (The Binomial Theorem)**.** Let x and y be variables, and let n be a nonnegative integer. Then

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

$$= \binom{n}{0}x^n + \binom{n}{1}x^{n-1}y^1 + \binom{n}{2}x^{n-2}y^2 + \cdots + \binom{n}{n-1}xy^{n-1} + \binom{n}{n}y^n$$

Corollary 9.4.1. Let n be a nonnegative integer. then

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

.

Corollary 9.4.2. Let n be a positive integer. Then

$$\sum_{k=0}^{n} (-1)^k \binom{n}{k} = 0$$

.

Corollary 9.4.3. Let n be a nonnegative integer. Then

$$\sum_{k=0}^{n} 2^k \binom{n}{k} = 3^n$$

9.4.2 Pascal's Identity and Triangle

Theorem 9.4.2 (Pascal's Identity)**.** Let n and k be positive integers with $n \geq k$. Then

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

.

9.4.3 Some Other Identities of the Binomial Coefficients

Theorem 9.4.3 (Vandermonde's Identity)**.** Let m,n, and r be nonnegative integers with r not exceeding either m or n. Then

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

.

Corollary 9.4.4. If n is a nonnegative integer, then

$$\binom{2n}{n} = \sum_{k=0}^{n} \binom{n}{k}^2$$

Theorem 9.4.4. Let n and r be nonnegative integers with $r \leq n$. Then

$$\binom{n+1}{r+1} = \sum_{j=r}^{n} \binom{j}{r}$$

.

9.5 Generalize Permutations and Combinations

Until now we selected items exactly once from a set. We now relax that and look at other counting problems.

9.5.1 Permutations with Repetition

Consider a cash drawer. There are various denominations of bills that are drawn from a small set of bills, typically singles, fives, tens and twenties. It makes no difference what order the bills are in in the cash box since all bills of the same denomination are interchangable (fungible). We put dividers into the cash box to separate the denominations and the minumum number of dividers will be one less than the number of bills. Since we have 4 different denominations, three dividers suffices to separate them. Now consider an alternative representation of the cash box. Since we keep the bills in a set order going smallest to largest and left to right, we know the denomination by its position relative to the dividers. All the singles are before the first divider, the fives between the first and second dividers, etc. So we can represent a bill by a zero and a divider by one and from this representation show any possible cash box. So this string of zeros and ones represents 5 singles, four fives, three tens and no twenties 000001000010001. Now recognize that if we want to know how many combinations of r objects we can create from a set of n elements allowing for repetition, that is the same as asking how many combinations of r zeros we can create where we have $n-1$ ones, a problem we previously solved.

Theorem 9.5.1. The number of r-permutations of a set of n objects with repetition allowed is n^r.

9.5.2 Combinations with Repetition

Theorem 9.5.2. There are $C(n+r-1,r) = c(n+r-1,n-1)$ r-combinations from a set for n elements when repetition of elements is allowed.

9.5.3 Permutations with Indistinguishable Objects

Up to now we have only drawn the objects from a set. A set does not have duplicates although we allowed multiple draws of the same element. Now we consider bags where some elements are indistinguishable from each other much like ping-pong balls in a ball pit. Consider, how many different words can possibly be constructed from the letters of the word 'SUCCESS'? Which of the S's chosen from the bag makes no difference in the constructed word since they are indistinguishable.

Theorem 9.5.3. [Permutations with Indistinguishable Objects] The number of different permutation of n objects, where there are n_1 indistinguishable objects of type 1, n_2 indistinguishable objects of type 2, …, and n_k indistinguishable objects of type k, is

$$\frac{n!}{n_1!n_2!n_3!\dots n_k!}$$

9.5.4 Counting Combinations when Distributing Objects into Boxes

We now consider the general cases of putting objects into boxes. The objects to be distributed may be distinguishable and the boxes into which they are distributed are distinguishable, say the dealt cards in a card game. Sometimes the objects being distributed are indistinguishable, as the bill in a cash drawer, but the boxes are distinguishable, as the slots in the cash drawer are. Sometimes the objects to be distributed are distinguishable, such as the elements of a set, but the boxes are indistinguishable, cubicles in a large office for example.

Combinations of Distinguishable objects into distinguishable boxes

Theorem 9.5.4. The number of ways to distribute n distinguishable objects into k distinguishable boxes to that n_i objects are placed into box i, $i = 1, 2, ;k$, equals

$$\frac{n!}{n_1!n_2!\dots n_k!}$$

Combinations of Indistinguishable Objects and Distinguishable Boxes

Theorem 9.5.5. There are

$$C(n+r-1, n-1)$$

ways to place r indistinguishable objects into n distinguishable boxes.

Combinations of Distinguishable Objects and Indistinguishable Boxes

For example, many office plans have indistinguishable cubicles that may hold multiple people. Let us say there are four new employees to be placed into three cubicles. How many distinct combinations of people in cubicles can there be? You may be surprised to learn there is no simple closed form solution to problems of this kind even though simple cases can easily be worked out by hand. In this example you can say that there is exactly one way to place all for employees into one cube. Then there are four ways to place three employees in one cube with one in another. There are three ways to place two employees in one cube with two in another. In the last case there are six ways to place two employees in one cubicle with the other two in a cubicle by themselves. In total there are 14 combinations.

Combinations of Indistinguishable Objects and Indistinguishable Boxes

An example of this kind of problem is the task of placing 6 books to be shipped into 4 shipping boxes, assuming the books are all of the same title. The books are indistinguishable and the boxes are also indistinguishable. You can place all six in one box, 5 in one and one in another, 4 and 2, 4 and 1 and 1, 3 and 3, 3 and 2 and 1, 3 and 1 and 1, 2 and 2 and 2, 2 and 2 and 1 and 1.

Observe that distributing n indistinguishable objects into k indistinguishable boxes is the same as writing n as the sum of at most k positive integers in nonincreasing order. If $a_1 + a_2 + \cdots + a_j = n$, where $a_1, a_2, \cdots + a_j$ is a partition of the positive inteers n into j positive integers. We can define a function $p_k(n)$ that gives the number of partitions of n into at most k positive integers. But there is no simple closed form formula for this number.

9.6 Generating Permutations and Combinations

9.6.1 Generating Permutations

9.6.2 Generating Combinations

9.7 other material, schaum's??

The size of a set A is called its *cardinality* and is designated $|A|$. Cardinality is defined as the number of elements in the set for a set with a finite number of elements. We say the cardinality of set A is 3 or simply that set A has three elements. If the number of elements in the set S is finite, we call it a finite set. If not we call it an infinite set.

Set Cardinality, Cardinal Numbers, Cardinality of Infinite Sets The cardinality of a set is defined as the number of elements it contains. Note this implicitly uses the concept of counting which will be more explicitly defined later in the course. When the number of elements in a set is a natural number m or is zero, we say the set is finite in size. Otherwise we say the set has an infinite cardinality or is an infinite set. We define a finite set to be countable. If we can specify a way in which an infinite set can have its members arranged, we call this an infinitely countable set. Later we will see that some sets, such as all the real numbers ≥ 0 and ≤ 1 are uncountably infinite.

Theorem: for two disjoint sets A and B not necessarily disjoint $card(A \cup B)$, the cardinality will be the sum of the cardinality minus the ones double counted because they are in both sets. The ones in both sets are defined as the intersectiion so we have what is called the inclusion, exclusion principle
$card(A \cup B) = card(A) + card(B) - card(A \cap B)$

This can be extended to any number of sets.

The cardinality of infinite sets The natural numbers are an infinite set. There are many other infinite sets that have a bijection to the set of natural numbers. For example we can use this function to map from natural numbers to the set of integers. Any set with a bijection to the set of natural numbers is called countably infinite.

Cantor famously proved that the set of rational numbers is countably infinite using the diagonalization argument. We assign a special symbol to the cardinality of all countably infinite sets, aleph
null.

Real numbers can be shown to have no bijection possible to natural numbers. We assign a special symbol to the cardinality of the set of real numbers and any other set with a bijection to it as aleph
1.

There are many sets that are both infinite yet countable. The set of natural numbers are infinite. Since it has a bijection with the natural numbers we call the set of natural numbers *countably infinite*. Any set that has a bijection with the natural numbers is also countably infinite.

Theorem 9.7.1. The set $\mathbb{Q}$ is countably infinite.

This is proven using Cantor's diagnolization proof.

Chapter 10

Advanced Counting

10.1 Recurrence Relations

Understanding recurrence relations is important to the analysis of recursive algorithms. linear, homogeneous, coefficients

Def 10.1.1. A *recurrence relation* for the sequence $\{a_n\}$ is an equation that expresses a_n in terms of one or more of the previous terms of the sequence, namely, $a_0, a_1, \ldots, a_{n-1}$, for all integers n with $n \geq n_0$, where n_0 is a nonnegative integer. A sequence is called a *solution* of a recurrence relation if its terms satisfy the recurrence relation.

Notes. Carefully note why they are relations and not functions. The initial conditions determine the sequence.

10.1.1 Modeling with Recurrence Relations

Compound interest formulae, Fibonnaci sequence, Catalan numbers

10.2 Solving Linear Recurrence Relations

10.2.1 Types of Recurrence Relations

Def 10.2.1. A *linear homogeneous recurrence relation of degree ke with constant coefficients* is a recurrence relation of the form

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}$$

where $c_1, c_2, \ldots c_k$ are real numbers, and $c_k \neq 0$

Linear recurrence relations have a sum of previous terms multiplied by some function of n.

Homogeneous recurrence relations have no terms that are not multiples of an a_j.

Constant coefficients ensure there are no functions on n as coefficients.

Degree is the k in the definition.

It is **linear** because the RHS is a sum of previous terms of the sequence each multiplied by a function of n and **homogeneous** because no terms occur that are not multiples of the a_js. All the coefficients are constants rather than functions on n. The **degree** is k.

Because linear homogeneous recurrence relations with constant coefficients occur so frequently in the analysis of algorithms, we limit our discussion to them. To see how more advanced recurrence relationships are solved the student should see

10.2.2 Solving Linear Homogeneous Recurrence Relations with Constant Coefficients

Example 6.10 from Schaum's

Consider the following homogeneous recurrence relation:

$$a_n = 2a_{n-1} + 3a_{n-2}$$

The general solution is obtained by first finding its characteristic polynomila $\Delta(x)$ and its roots r_1 and r_2:

$$\Delta(x) = x^2 - 2x - 3 = (x-3)(x+1); \text{roots} r_1 = 3, r_2 = -1$$

Since the roots are distinct, we can use Theorem x to obtain the general solution:

$$a_n = c_1 3^n + c_2(-1)^n$$

Any values for c_1 and c_2 will give a solution to the recurrence relation.

Suppose we are also given the initial condition $a_0 = 1, a_1 = 2$. Using the recurrence relation we can compute the next few terms of the sequence:

$$1, 2, 7, 20, 61, \cdots$$

The unique solution is obtained by finding c_1 and c_2 using the initial conditions. Specifically:

For $n = 0$ and $a_0 = 1$, we get: $c_1 3^0 + c_2(-1)^0 = 1$ or $c_1 + c_2 = 1$

For $n = 1$ and $a_1 = 2$, we get: $c_1 3^1 + c_2(-1)^1 = 2$ or $3c_1 - c_2 = 2$

Solving the system of two equations in the unknowns c_1 and c_2 yields:

$$c_1 = \frac{3}{4} \text{and} c_2 = \frac{1}{4}$$

Thus the following is the unique solution of the given recurrence relation with the given initial conditions $a + 0 = 1, a_1 = 2$:

$$a_n = \frac{3}{4} 3^n + \frac{1}{4}(-1)^n = \frac{3^{n+1} + (-1)^n}{4}$$

10.3 Divide-and-Conquer Algorithms and Recurrence Relations

give one homework with a recursive algorithm and ask for closed form solution for time function complexity.

10.4 Generating Functions

Def 10.4.1. The *generating function for the sequence* $a_0, a_1, \ldots, a_k, \ldots$ of real numbers is the inifinite series

$$G(x) = a_0 + a_1 x + \cdots + a_k x^k + \cdots = \sum_{k=0}^{\infty} a_k x^k$$

10.4.1 Counting Problems and Generating Functions

10.5 Inclusion-Exclusion

10.5.1 The Principle of Inclusion-Exclusion

10.5.2 Applications of Inclusion-Exclusion

10.5.3 An Alternative Form of Inclusion-Exclusion

10.5.4 The Sieve of Eratosthenes

The Sieve of Eratosthenes is an algorithm for making tables of primes. Sequentially write down the integers from 2 to the highest number n you wish to include in the table. Cross out all numbers >2 which are divisible by 2 (every second number). Find the smallest remaining number >2. It is 3. So cross out all numbers >3 which are divisible by 3 (every third number). Find the smallest remaining number >3. It is 5. So cross out all numbers >5 which are divisible by 5 (every fifth number).

Continue until you have crossed out all numbers divisible by $\lfloor \sqrt{n} \rfloor$. The numbers remaining are prime.

The sieve of Eratosthenes can be used to compute the prime counting function as

$$\pi(n) = \lfloor n \rfloor - \sum_i \left\lfloor \frac{n}{p_i} \right\rfloor + \sum_{i<j} \left\lfloor \frac{n}{p_i p_j} \right\rfloor \ldots$$

10.5.5 The Number of Onto Functions

10.5.6 Derangements

10.5.7 Natural Number Functions

10.5.8 Limits on Computability

Not Everything is Countable

There are uncountably many languages over a finite alphabet

10.5.9 Inductively Defined Infinite Sets

The set of natural numbers can be completely defined with the statements that zero is a member of the set, that for all n in the set that succ(n) is a member of the set and that only elements from these two statements exist in the set.

Theorem: The set $\mathbb{R}$ is NOT countably infinite. We call the cardinality of all countably infinite sets $\aleph_0$ pronounced aleph-null. Sets that can be put into a bijection with the set of reals is said to have cardinality $\aleph_1$

The algorithm to calculate a factorial has the simple open form solution of $\Pi 1 * 2 * 3 * ... * n$

This gives us a simple iterative algorithm to calculate n!

factorial(n) fact=1 for i=1 to n fact=fact*i return fact

However there is another algorithm that has appeal.

factorial(n) if n==0 return 1 else return n*(factorial(n-1))

The appeal is the simplicity but the novelty is that the function CALLS ITSELF, a recursion. In general every iterative algorithm can be converted to a recursive algorithm and vice versa. They occur enough in computer science that we need to look at how we look at the time complexity functions they create.

Level Number With recursive algorithms, it is helpful to track how many times the algorithm has called itself. We define a level number to be one the first time it is called and one greater with each recursive call.

Some famous sequences derived from recursive functions Fibonacci is famous because it was created to capture the reproduction pattern of rabbits but is found abundantly in nature.

Fib(n) if n==0 return 0 else if n==1 return 1 else return (Fib(n-1) + Fib(n-2))

When used on the set of natural numbers we get the sequence 0,1,1,2,3,5,8,13,21,34,55, ...

Ackermann Famous for a function that is easily defined but has an explosive growth curve. To evaluate the Ackermann function for even the trivial value of Ackermann(1,3) requires 15 steps.

Ackermann(m,n) if m==0 return n+1 else if m !=0 and n==0 return Ackermann(m-1,1) else (m!=0 and n!=0) return Ackermann((m-1),Ackermann(m, n-1))

Recursive Relations and their Closed Form Solutions

Recursive definition of sets

Chapter 11

Boolean Algebras

This chapter provides the most basic application of propositional logic, logic circuits, and discusses some material that is best motivated by that application. It also covers a brief introduction to abstract algebra that some schools want included within a course on discrete mathematics. It is interesting to note that prior to the introduction of a discrete math course, computer scientists had educations that included abstract algebra. But newer programs no longer require a math course in abstract algebra so some of that material must be introduced via this course.

11.1 Definitions

We begin by defining a new notation to describe Boolean expressions using structural recursion:

Def 11.1.1 (Boolean Variable)**.** Let $\mathbb{B} = \{0,1\}$. Then $\mathbb{B}^n = \{x_1, x_2, \ldots, x_n) \mid x_i$for all $1 \le i \le n\}$ is the set of all possible n-tuples of 0s and 1s. The variable x is called a **Boolean variable** if it assumes values only from $\mathbb{B}$, that is, if its only possible values are 0 and 1. A function from $\mathbb{B}^n$ to $\mathbb{B}$ is called a **Boolean function of degree** n. Boolean functions can be represented using expressions made from variables and Boolean operations. The **Boolean**

expressions in the variables $x_1, x_2, \ldots, x_n$ are defined recurseively as
$0, 1, x_1, x_2, \ldots, x_n$ are Boolean expressions
if E_1 and E_2 are Boolean expressions, then
$\bar{E_1}, \bar{E_2}, (E_1E_2), and(E_1+E_2)$ are Boolean expressions we adopt rules of precedence to avoid large numbers of parentheses

11.2 Boolean Functions

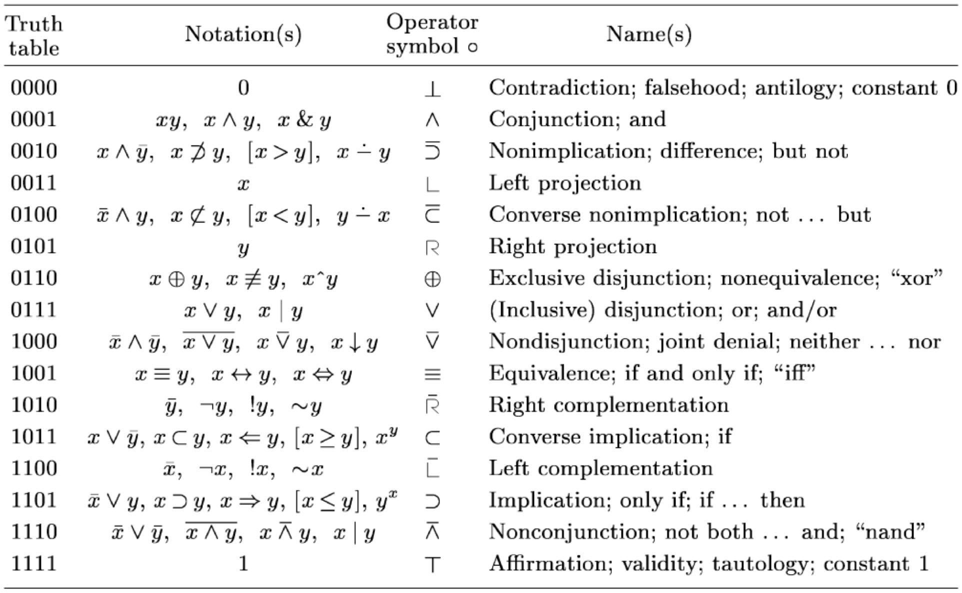
When discussing propositional logic, the only operators discussed were negation, conjuction, disjunction, conditional and bi-conditional. Yet it is possible in the truth table to have 16 distinct functions/propositional operators. (If the number of propositonal variables is 2 there are 4 rows to the truth table and 16 columns each with a unique combination of T/F for the two propositions). What about all those other operators? There are two very famous ones that we need to see in this chapter, one is called the **Sheffer Stroke** denoted by a vertical line ($P \mid Q$) and equivalent to the negation of an and operation, or NAND and the **Peirce Arrow** denoted by a down arrow ($P \downarrow Q$) and logically equivalent to not or or NOR. Note that both of these operators have equivalent representation using the set used when discussing propositional logic. It is the case that all 16 of the potential operators have logical equivalents using just negation, conjunction and disjunction. Such a set of propositional functions is called functionally complete.

11.2.1 Boolean Expressions and Boolean Functions

11.2.2 Identities of Boolean Algebra

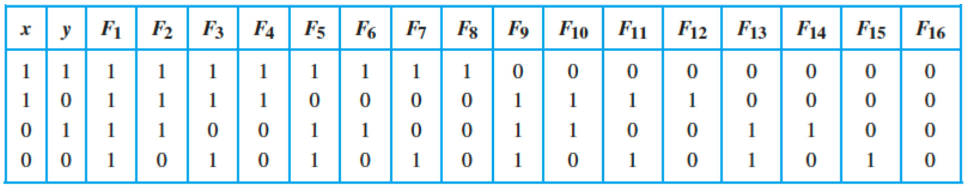
11.2.3 Principle of Duality

Note the similarity of Table 1.1 and 2.1. This is no accident but instead is a fundamental point that is explored in more detail in the math course, Abstract Algebra. One fact that should be noted is the Principle of Duality.



Truth table	Notation(s)	Operator symbol $\circ$	Name(s)
0000	0	$\bot$	Contradiction; falsehood; antilogy; constant 0
0001	$xy,\ \ x \wedge y,\ \ x \mathbin{\&} y$	$\wedge$	Conjunction; and
0010	$x \wedge \bar{y},\ \ x \not\supset y,\ \ [x > y],\ \ x \dot{-} y$	$\bar{\supset}$	Nonimplication; difference; but not
0011	x	L	Left projection
0100	$\bar{x} \wedge y,\ \ x \not\subset y,\ \ [x < y],\ \ y \dot{-} x$	$\bar{\subset}$	Converse nonimplication; not . . . but
0101	y	R	Right projection
0110	$x \oplus y,\ \ x \not\equiv y,\ \ x\char`^y$	$\oplus$	Exclusive disjunction; nonequivalence; “xor”
0111	$x \vee y,\ \ x \mid y$	$\vee$	(Inclusive) disjunction; or; and/or
1000	$\bar{x} \wedge \bar{y},\ \ \overline{x \vee y},\ \ x \mathbin{\bar{\vee}} y,\ \ x \downarrow y$	$\bar{\vee}$	Nondisjunction; joint denial; neither . . . nor
1001	$x \equiv y,\ \ x \leftrightarrow y,\ \ x \Leftrightarrow y$	$\equiv$	Equivalence; if and only if; “iff”
1010	$\bar{y},\ \ \neg y,\ \ !y,\ \ \sim y$	$\bar{\mathsf{R}}$	Right complementation
1011	$x \vee \bar{y},\ x \subset y,\ x \Leftarrow y,\ [x \geq y],\ x^y$	$\subset$	Converse implication; if
1100	$\bar{x},\ \ \neg x,\ \ !x,\ \ \sim x$	$\bar{\mathsf{L}}$	Left complementation
1101	$\bar{x} \vee y,\ x \supset y,\ x \Rightarrow y,\ [x \leq y],\ y^x$	$\supset$	Implication; only if; if . . . then
1110	$\bar{x} \vee \bar{y},\ \ \overline{x \wedge y},\ \ x \mathbin{\bar{\wedge}} y,\ \ x \mid y$	$\bar{\wedge}$	Nonconjunction; not both . . . and; “nand”
1111	1	$\top$	Affirmation; validity; tautology; constant 1

Table 11.1: TableOfPropositionalOperators



x	y	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}
1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
0	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0

Table 11.2: Table of Boolean Functions of Two Variables

11.2.4 Abstract Definition of a Boolean Algebra

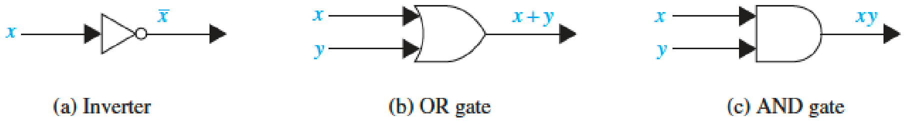
11.3 Representing Boolean Functions

11.3.1 Sum-of-Products Expansions

11.3.2 Functional Completeness

11.4 Logic Gates

The logic of signals on wires can be made to obey the laws of logic. Some voltage is used to represent *true* while another is used to represent *false*. Electrical circuits have been created which implement the laws of the logical operators we have already seen and they are called *gates*. We introduce two notations for representing this type of logic, one the visual one which is often seen in a circuits class and another that is a more algebraic notation.



(a) Inverter

(b) OR gate

(c) AND gate

Figure 11.1: Basic Logic Gages

Def 11.4.1 (Logic gates)**.**

11.4.1 Combinations of Gates

11.4.2 Adders

11.4.3 Minimization of Circuits

11.4.4 combinational circuits versus sequential circuits

11.4.5 Karnaugh Maps and Quine-McCluskey Method

Two meanings: one the notation used by Boole, the other the more general observation that this algebra is the same as for logic and sets. The general topic of other algebras is covered in a course on abstract algebra in the math department.

Sum of Products form for Boolean Algebras well formed formula fundamental product Algorithm for finding sum-of-products form complete sum of products form, midterms, DNF

Homomorphism between Boolean Algebra and basic logic gates

Chapter 12

Graphs

vertices or nodes, edges (directed or undirected), head, tail,
in-degree, out-degree, self-loop (reflexive), was covered in the chapter on functions. Famous algorithms for graph traversal

12.1 Graphs and Graph Models

Def 12.1.1 (Graph)**.** A *graph* $G = (V, E)$ consistes of a non-empty set V of *vertices* or *nodes* and E, a set of *edges*. Each edge has either one or two vertices associated ith it, called its *endpoints*. An edge is said to *connect* its endpoints.

Notes. The number of vertices does not need to be finite. But we will only discuss finite graphs in this outline.

Def 12.1.2 (Simple Graph)**.** A graph in which each edge connects two different vertices and where no two edges connect the same pair of vertices is called a **simple graph**. The edge in a simple graph can be denoted by the set of vertices it connects.

Def 12.1.3 (Multigraph)**.** Graphs that have multiple edges connecting the same vertices are called **multigraphs**. We say that each edge has a multiplicity of edges between the two vertices.

Def 12.1.4 (Directed or Digraph)**.** A *directed graph* (or *digraph* (V, E) consists of a nonempty set of vertices V and a set of *directed edges* or *arcs* E. Each directed edge is associated with an ordereed pair of vertices. The directed edge associated with the ordered pair (u,v) is said to start with u and end at v. When presented in a picture, the directed edge has an arrow that starts at the vertex u and ends at v. A directed graph may also have multiple directed edges.

Graphs model a great many applications in software engineering. They include Acuaintanceship Graphs, Influence Graphs, The Hollywood Graph, Round-Robin Tournaments, Collaboration Graphs, Road Maps, Call Graphs, and the WWW.

12.2 Graph Terminology and Special Types of Graphs

Def 12.2.1 (Edges in an Undirected Graph)**.** Two vertices u and v in an undirected graph G are called *adjacent* (or *neighbors* in G if u and v are endpoints of an edge of G. If e is associated with $\{u,v\}$, the edge e is called *incident with* the vertices u and v. The edge e is also said to *connect* u and v. The vertices u and v ar called *endpoints* of an edge associated with $\{u,v\}$.
Def 12.2.2. The *degree of a vertex in an undirected graph* is the number of edges incident with it, except that a loop at a vertex contributes twice to the degree of that vertex. The degree of the vertex v is denoted by deg(v). A vertex of degree zero is called **isolated** while a vertex with degree one is called **pendant**
Theorem 12.2.1 (The Handshaking Theorem)**.** Let $G = (V,E)$ be an undirected graph with e edges. Then

$$2e = \Sigma_{v \in V} deg(v)$$

.
Notes. This is true even when multiple edges and loops are present.
Theorem 12.2.2. An undirected graph has an even number of vertices of odd degree.
Def 12.2.3. When (u,v) is an edge of the graph G with directed edges, u is said to be *adjacent to* v and v is said to be *adjacent from* u. The vertex u is called the *initial vertex* of (u,v), and v is called the *terminal* or *end vertex* of (u,v). The initial and terminal vertex of a loop are the same.

Def 12.2.4. In a graph with directed edges, the *in-degree of a vertex* v, denoted by $\deg^{-}(v)$, is the number of edges with v as their terminal vertex. The *out-degree of* v, denoted by $\deg^{+}(v)$, is the number of edges with v as their initial vertex. (Note that a loop at a vertex constributes 1 to both the in-degree and the out-degree of this vertex.)

Theorem 12.2.3. Let $G = (V, E)$ be a graph with directed edges. Then

$$\Sigma_{v \in V} \deg^{-}(v) = \Sigma_{v \in V} \deg^{+}(v) = |E|$$

Def 12.2.5. Here are the definitions of some common simple graphs:
A **Complete Graph on n-vertices** is a simple graph that contains exactly one edge between each pair of distinct vertices. It is denoted by K_n.
A **Cycle** of n vertices, $n \geq 3$, $v_1, v_2, \ldots, v_3$ has edges $\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{n-1}, v_n\}, \{v_n, v_1\}$. It is denoted C_n.
A **Wheel** is created by taking a cycle, C_n, adding a single vertex v and adding an edge from the new vertex v to each of the n existing vertices of the cycle. It is designated W_n.
The **n-dimensional hypercube** or **n-cube** is the graph that has vertices representing the 2^n bit strings of length ....blah blah

Def 12.2.6 (Bipartite Graph)**.** A simple graph G is called *bipartite* if its vertex set v can be partititoned into two disjoint sets V_1 and V_2 such that evry edge in the graph connects a vertex in V_1 and a vertex in V_2. For such a graph we call the pair (V_1, V_2) a *bipartition* set V of G.

Theorem 12.2.4. A simple graph is bipartte if an donly if it is possible to assign one of two different colors to each vertex of the graph so that no two adjacent vertices are assigned the same color. This is called a *graph coloring*.

Def 12.2.7 (Complete Bipartite Graphs)**.** The **complete bipartite graph** $K_{m,n}$ is the graph that has its vertex set partitioned into two subsets of m and n vertices, respectively. There is an edge between two vertices if and only if one vertex is in the first subset and the other vertex is in the second subset.

Def 12.2.8. A *subgraph of a graph* $G = (V, E)$ is a graph $H = (W, F)$, where $W \subset V$ and $F \subset E$. A subgraph H of G is a *proper subgraph* of G if $H \neq G$.

Def 12.2.9. The *union* of two simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is the simple graph with vertext set $V_1 \cup V_2$ and edge set $E_1 \cup E_2$. The unions of G_1 and G_2 is denoted by $G_1 \cup G_2$.

12.3 Representing Graphs and Graph Isomorphism

Sometimes we look at graphics that depict two different graphs but we see that while they look different we can make one match to the other by moving vertices around. This leads us to look at how we represent graphs and to define what we mean by saying two graphs are somehow the same.

There are two common way that graphs are represented using tables and these are useful in software engineering. We look at adjacency lists and adjacency matrices.

Def 12.3.1. A simple graph G can be represented by listing each vertex and the vertices that are adjacent to it. This is called an **adjacency list**. This can be extended to include a representation of directed graphs. Note that the size of the lists will be half the size of an undirected graph.

Def 12.3.2. A simple graph G can be represented by a two dimensional matrix with the vertices of the graph on both axes. The presence of an edge between them can be represented by a value in the matrix.

Suppose that $G = (V, E)$ is a simple graph where $|V| = n$. Suppose that the vertices of G are listing arbitraryily as $v_1, v_2, \ldots, v_n$. The **adjacency matrix** A (or A_G) of G, with respect to this listing of the vertices, is the $n \times n$ zero-one matrix with 1 as its (i, j)th entry when v_i and v_j are adjacent, and 0 as its (i, j)th entry when they are not adjacnt. In other words, if its adacency matrix is $A = [a_{ij}]$, then

cases

Def 12.3.3. Graphs can be represented by an **incidence matrix**. Let $G = (V, E)$ be an undirected graph. Suppose that $v_1, v_2, \ldots, v_n$ are the vertices and $e_1, e_2, \ldots, e_m$ are the edges of G. Then the incidence matrix with respect to this ordering of V and E is the $n \times n$ matrix $M = [i, j]$, where cases

Def 12.3.4. The simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic* if there is a one-to-one and onto function (a bijection) f from V_1 to V_2 with the property that a and b are adjacent in G_1 if and only if $f(a)$ and $f(b)$ are adjacent in G_2, for all a and b in V_1. Such a function f is called an *isomorphism*.

Worst case algorithms to determine graph isomorphism have exponential worst case time complexity. However there are algorithms that have linear average time complexity for graph isomorphism.

12.4 Connectivity

12.5 Euler Paths and Circuits

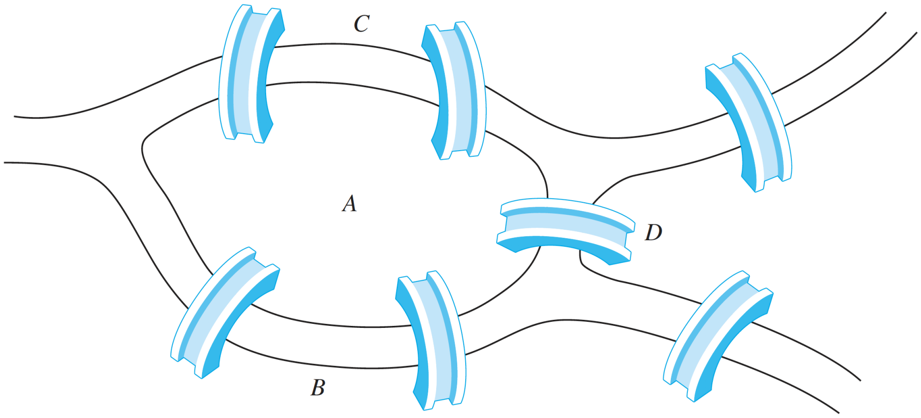


Figure 12.1: The Bridges of Koenigsberg

Def 12.5.1. An *Euler circuit* in a graph G is a simple circuit containing every edge of G. An *Euler path* in G is a simple path containing every edge of G.

Theorem 12.5.1. A connected multigraph with at least two vertices has an Euler curcuit if and only if each of its vertices has even degree.

NECESSARY AND SUFFICIENT CONDITIONS FOR EULER CIRCUITS AND PATHS There are simple criteria for determining whether a multigraph has an Euler circuit or an Euler path. Euler discovered them when he solved the famous Königsberg bridge problem. We will assume that all graphs discussed in this section have a finite number of vertices and edges. What can we say if a connected multigraph has an Euler circuit? What we can show is that every vertex must have even degree. To do this, first note that an Euler circuit begins with a vertex a and continues with an edge incident with a, say a, b. The edge a, b contributes one to deg(a). Each time the circuit passes through a vertex it contributes two to the vertex's degree, because the circuit enters via an edge incident with this vertex and leaves via another such edge. Finally, the circuit terminates where it started, contributing one to deg(a). Therefore, deg(a) must be even, because the circuit contributes one when it begins, one when it ends, and two every time it passes through a (if it ever does). A vertex other than a has even degree because the circuit contributes two to its degree each time it passes through the vertex.We conclude that if a connected graph has an Euler circuit, then every vertex must have even degree. Is this necessary condition for the existence of an Euler circuit also sufficient? That is, must an Euler circuit exist in a connected multigraph if all vertices have even degree? This question can be settled affirmatively with a construction.

Algorithm: Constructing Euler Circuits
Theorem 12.5.2. A connected multigraph has an Euler path but not an Euler circuit if and only if it has exactly two vertices of odd degree.

12.6 Hamilton Paths and Circuits

Def 12.6.1. A simple path in a graph G that passes through every vertex exactly once is called a *Hamilton path*, and a simple circuit in graph G that passes through everty vertex exactly once is called a *Hamilton circuit*. That is, the simple path $x_0, x_1, \ldots, x_{n-1}, x_n$ and $x_i \neq x_j$ for $0 \leq i < j \leq n$, and the simple circuit $x_0, x_1, \ldots, x_{n-1}, x_n, x_0$

(with $n > 0$) is a Hamilton circuit if $x_0, x_1, \ldots x_{n-1}, x_n$ is a Hamilton path.

CONDITIONS FORTHE EXISTENCE OF HAMILTON CIRCUITS
Is there a simple way to determine whether a graph has a Hamilton circuit or path? At first, it might seem that there should be an easy way to determine this, because there is a simple way to answer the similar question of whether a graph has an Euler circuit. Surprisingly, there are no known simple necessary and sufficient criteria for the existence of Hamilton circuits. However, many theorems are known that give sufficient conditions for the existence of Hamilton circuits. Also, certain properties can be used to show that a graph has no Hamilton circuit. For instance, a graph with a vertex of degree one cannot have a Hamilton circuit, because in a Hamilton circuit, each vertex is incident with two edges in the circuit. Moreover, if a vertex in the graph has degree two, then both edges that are incident with this vertex must be part of any Hamilton circuit. Also, note that when a Hamilton circuit is being constructed and this circuit has passed through a vertex, then all remaining edges incident with this vertex, other than the two used in the circuit, can be removed from consideration. Furthermore, a Hamilton circuit cannot contain a smaller circuit within it.
Theorem 12.6.1 (Dirac's Theorem)**.** If G is a simple graph with n vertices with $n \geq 3$ such that the degree of every vertex in G is at least $\frac{n}{2}$, then G has a Hamilton circuit.
Theorem 12.6.2 (Ore's Theorem)**.** If G is a simple graph with n vertices with $n \geq 3$ such that deg(u)+deg(v) $\geq n$ for every pair of nonadjacent vertices u and v in G, then G has a Hamilton circuit.

12.7 Shortest Path Problem

Dijkstra's Algorithm
Theorem 12.7.1. Dijkstra's algorithm finds the length of a shortest path between two vertices in a connected simple undirected weighted graph.
Theorem 12.7.2. Dijkstra's algorithm uses $O(n^2)$ operations (additions and comparisons) to find the length of a shortest path

between two vertices in a connected siple undirected weighted graph with n vertices.

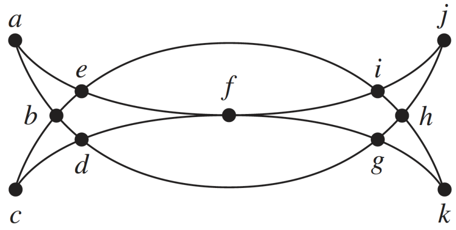


Figure 12.2: Mohammeds Scimitar

12.8 Planar Graphs

Def 12.8.1. A graph is called *planar* if it can be drawn in the plane without any edges crossing (where a crossing of edges is the intersection of the lines or arcs represening them at a point other than their common endpoint). Such a drawing is called a *planar representation* of the graph.

Theorem 12.8.1 (Euler's Formula)**.** Let G be a connected planar simple graph with e edges and v vertices. Let r be the number of regions in a planar representation of G. Then $r - e - v + 2$.

Corollary 12.8.1. If G is a connected planar simple graph with e edges and v vertices, where $v \geq 3$, then $e \leq 3v - 6$

Corollary 12.8.2. If G is a connected planar simple, graph then G has a vertex of degree not exceeding five.

Corollary 12.8.3. If a connected planar simple graph has $e$edges and v vertices with $v \geq 3$ and no circuits of length three, then $e \leq 2v - 4$.

Theorem 12.8.2. A graph is nonplanar if and only if it contains a subgraph homeomorphic to $K_{3,3}$ or K_5.

12.9 Graph Coloring

Def 12.9.1. A *coloring* of a simple graph is the assignment of acolor to each vertex of the graph so that no two adjacent vertices are assigned the same color.

Def 12.9.2. The *chromatic number* of a graph is the least number of colors needed for a coloring of this graph. The chromatic number of a graph G is denoted by $\chi(G)$ (where χ is the Greek letter chi).

Theorem 12.9.1 (The Four Color Theorem)**.** The chromatic number of a planar graph is no greater than four.

Chapter 13

Trees

Recursive def of trees, some famous algorithms for trees
Def 13.0.1 (Tree)**.** A *tree* is a connected undirected graph with no simple circuits.
Theorem 13.0.1. An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.
Def 13.0.2 (Rooted Tree)**.** A *rooted tree* is a tree in hwich one vertex has been designated as the root and every edge is directed away from the root. Suppose T is a rooted tree. If v is a vertex in T other than the root, the **parent** of v is the unique vertex u such that there is a directed edge from u to v. When u is the parent of v, v is called a **child** of u. Vertices with the same parent are called **siblings**. The **ancestors** of a vertex other than the root are the vertices in the path from the root to this vertex, excluding the vertex itself but including the root. The **descendants** of a vertex v are those vetices that have v as an ancestor. A vertex of a tree is called a **leaf** if it has no children. Vertices that have children are called **internal vertices**. The root is an internal vertex unless it is the only vertex in the graph, in which case it is a leaf. If a is a vertexin a tree, the **subtree** with a as its root is the subgraph of the tree consisting of a and its descendants and all edges incident to those descendants.
Def 13.0.3. A rooted tree is called an *m-ary tree* if every internal vertex has no more than m children. The tree is called a "*full m-ary*

tree" if every internal vertex has exactly m children. An m-ary tree with $m = 2$ is called an *binary tree*.
Def 13.0.4. An **ordered rooted tree** is a rooted tree where the children of each internal vertex are ordered. Orered rooted trees are drawn so that the hcildren of each internal vertex are shown in order from left to right. Note that a representation of a rooted tree in the conventional way determines an ordering for its edges.

In an orderedf binary tree (usually called a **binary tree**), if an internal vertex has two children, the first child is called the **left child** and the second child is called the **right child**. The tree rooted at the left child of a vertex is called the **left subtree** of this vertex, and the tree rooted at the right child of a vertex is called the **right subtree** of the vertex.

13.1 Properties of Trees

Theorem 13.1.1. A tree with n vertices has $n-1$ edges.
Theorem 13.1.2. A full m-ary tree with i internal vertices contains $n = mi + 1$ vertices.
Theorem 13.1.3. A full m-ary tree with

(i) n vertices has $i = (n-1)/m$ internal vertices and $l = [(m-1)n+1]/m$ leaves,

(ii) i internal vertices has $n = mi + 1$ vertices and $l = (m-1)i + 1$ leaves,

(iii) l leaves have $n = (ml-1)/(m-1)$ vertices and $i = (l-1)(m-1)$ internal vertices.

Def 13.1.1. The level of a vertex v in a rooted tree is the length of the unique path from the root to this vertex. The level of the root is defined to be zero. The **height** of a rooted tree is the maximum of the levels of vertices. In other words, the height of a rooted tree is the length of the longest path from the root to any vertex. A rooted m-ary tree of hgitht h is **balanced** if all leaves are at levels h or $h-1$.
Theorem 13.1.4. There are at most m^h leaves in an m-ary tree of height h.

Corollary 13.1.1. If an m-ary tree of height h has l leaves, then $h \geq \lceil \log_m l \rceil$. If the m-ary tree is full and balanced, then $h = \lceil \log_m l \rceil$

Chapter 14

Discrete Probability

14.1 An Introduction to Discrete Probability

14.1.1 Finite Probability

14.1.2 The Probability of Combinations of Events

14.1.3 Probabilistic Reasoning

14.2 Probability Theory

14.2.1 Assigning Probabilities

14.2.2 Combinations of Events

14.2.3 Conditional Probability

14.2.4 Independence

14.2.5 Bernoulli Trials and the Binomial Distrubution

14.2.6 The Birthday Problem

14.2.7 Monte Carlo Algorithms

14.2.8 The Probabilistic Method

Let p be the proposition that the sum of the first n odd numbers is n**2. How can we prove such a proposition? Here is the example of how inductive reasoning works.

We can easily evaluate this proposition for small values of n and see that they are true. But since the set of input values is the set of natural numbers we cannot do this for all elements of the set. So we observe this, let us assume that this proposition is true for some value k which is bigger than any value we evaluated manually. If we can prove that the statement MUST be true for the next value, the successor of k, k+1, then we have proven that it must be true for ALL values of n drawn from the natural numbers since we know it is true for the small values and we can continually apply the reasoning that got us from k to k+1 as many times as we need to give us all the values to infinity.

So first we introduce the inductive hypothesis, that it is true for some k: Assume that the first k odd numbers sum to k**2. Now we have the proof obligation to prove that with that assumption that this MUST be true for k+1, that is, the sum of the first k+1 odd numbers will give us (k+1)**2. This requires some clever algebra but nothing you can't follow:

the first k odd number sum to k**2 Sigma(i=0 to k, 2i+1) = Sigma(i=1 to k-1, 2i+1) = k**2 which is equivalent to 1+3+5+ ... +2(k-1) = k**2 we add 2(k+1) to both sides of the equation giving 1+3+5+ ... +2(k-1)+2(k+1) = (k+1)**2 = (k**2 + 2k + 1)=k**2 + (2k+1) Using the premise, we rewrite the LHS 1+3+5+ ... +2(k-1) + 2(k+1) = k**2 + 2(k+1)= k**2 + 2k + 1 showing the left and RHS of the equation are equal QED.

The general principle of weak form of mathematical induction is First, show the proposition is true for one small element from the input. Show that IF the proposition is true for some arbitrary k, that it MUST also be true for the next value after k, k+1. After both parts are proven, you have proven for all value from N.

CAUTION: The inductive assumption looks similar to the thing to be proven but you may not use that in the argument. You must use an arbitrary value k, and then prove that it must also be true for k+1 without again stating the assumption. To do so is the famous logical fallacy of assuming the antecedent or begging the question. This is a common error in inductive proofs.

There are a set of proofs that can be solved inductively but require that more than one small value be proven. This leads to the stronger form of inductive reasoning. In the strong form, you must prove that the proposition holds for some small number of values.

Open Form versus Closed Form Solutions Note that we have proved an equivalence between two expressions, the sum of the first n odd numbers and the expression n**2. The first form has the implied algorithm of summing the first n odd integers, something that is of O(n) while the second has O(1). We call the first version an open form solution while we call the second a closed form solution. The computational advantage is obvious.

An Outline of Discrete Mathematics

(Excluded from UCD course offerings)

Chapter 15

Computation

Association between Automata, Grammar, Language. Difference between syntax and semantics in natural language. Object first or object last in natural language.

15.1 Languages and Grammars

We saw that the notation A_* designates all the possible strings that can be constructed from the set A. When the set A contains symbols that are distinguishable from each other we call that set an *alphabet* and refer to it with the greek letter Σ. Since there is no ambiguity between the strings $(a_1, a_2, \ldots a_n)$ and $a_1 a_2 a_3 \ldots a_n$ for strings of length n, we adopt this notation for string sequences. For short strings then

$$(w, o, r, d) = word$$

We call such strings words, the symbols from the alphabet lettersand the finite sequences in Σ_* as the strings or words generated by the letters of Σ.

Alphabet, Words Operations on words: concatenation Formal Definition of Language Operations on Languages

15.1.1 Phrase-Structured Grammars

Regular Expressions, Regular Languages Generative Grammars, Rules of Production

15.1.2 Context Free and Context Sensitive Grammars

15.1.3 Regular Languages and their Notation

15.1.4 Derivation Trees

parsing

15.1.5 Backus-Naur Form

15.2 Finite State Machines

States, State Transition, Finite State Automata

15.2.1 FSM with no output

Set of Strings

Finite-State Automata

Def 3: A *finite-state automata* $M = (S, I, f, s_0, F)$ consists of a finite set S of *states*, a finite *input alphabet* I, a *transition function* f that assigns a next state to every pair of state and input (so that $f : S \times I \to S$, an *initial* or *start state* s_0, and a subset F of S consisting of *final* (or *accepting states*.

Language Recognition by FSA

Non-deterministic FSA

15.2.2 FSM with output

Mealy and Moore machines

15.3 Language Recognition

15.3.1 Regular Sets

15.3.2 Kleene's Theorem

15.3.3 Regular Sets and Regular Grammars

15.3.4 Beyond Regular Languages and FSMs

A later class will explore grammars beyond regular grammars, the languages they generate and the automata that recognize them.

how do i do a citation?

Index

Bibliography

[1] James Aspnes. Fast deterministic consensus in a noisy environment. *Journal of Algorithms*, 45(1):16–39, October 2002.

[2] James Aspnes. *Wait-Free Consensus*. PhD thesis, Carnegie-Mellon University, 1992.