Ibadan Distance Learning Centre Series

CSC 242

FOUNDATIONS OF COMPUTER SCIENCE

By

Olufade Falade Williams Onifade (PhD)

Department of Computer Science,

University of Ibadan,

Ibadan, Nigeria.

Lesson 1: General Introduction

Computer Science, like every other discipline has its fundamental topics which form the foundation for a solid career in the discipline. A collection of these topics has come to be regarded as The Foundations of Computer Science and these topics are the focus of this manual. Some of these topics include Logic, Program Correctness, Mathematical Induction and Data Structures among others. These topics, although usually overlooked today, form the basis for all advancements in the field. Innovative solutions arising from research and development efforts are often built directly or indirectly from the concepts of these fundamentals. Therefore, this is a course you must give uttermost attention as your success in the Computer Science discipline is hinged on it. Don't just cram these concepts to pass, ensure you gain mastery of them.

To get the best out of this course, endeavour to attend all lectures, study all lessons carefully, attempt the exercises within the text and make sure you answer all questions in the pre-test and post-test. Also, it is important that you promptly submit all your assignments and avoid 'dubbing' assignments in order to avoid implicating yourself and for you to ensure that you've truly learnt what the assignment intends to teach. I wish you the very best.

Lesson 2: Course Objectives

At the end of this course, you should be able to

1. Understand and write propositional and predicate logic statements
2. Know the difference between loops and recursion
3. Find loop invariants and prove program correctness
4. Understand data structures and the algorithms for manipulating them
5. Understand the fundamentals of cryptography

Contents:

1. Logic
2. Sets
3. Mathematical Induction
4. Loops and Recursion I
5. Loops and Recursion II
6. Program Correctness (Loop invariant, Hoare's Axiom)
7. Discrete Probability
8. Randomized Algorithms
9. Lists
10. Graphs
11. Cryptography
12. Revision

Contents

General Introduction 2

Course Objectives 2

Course Outline 3

Lesson 1: Logic 5

Lesson 2: Sets 20

Lesson 3: Mathematical Induction 28

Lesson 4: Loops and Recursion 37

Lesson 5: Program Correctness 46

Lesson 6: Hoare's Axioms .... **Error! Bookmark not defined.**

Lesson 7: Sets .... **Error! Bookmark not defined.**

Lesson 8: Lists 62

Lesson 9: Graphs 67

Lesson 10: Discrete Probability .... **Error! Bookmark not defined.**

Lesson 11: Randomized Algorithms .... **Error! Bookmark not defined.**

Lesson 12: Introduction to Cryptography .... **Error! Bookmark not defined.**

Lesson 13: REVISION .... **Error! Bookmark not defined.**

Lesson 1: Logic

Introduction

In this lesson, we will learn about propositional and predicate logic. We will learn how to interpret and write sound logical statements. Logic forms the basis of all mathematical and automated reasoning with practical applications in the design and specifications of systems, artificial intelligence, computer programming and other areas of Computer Science, as well as many other fields of study.

We will study the basic constituents of each type of logic and learn how to use the logical connectives to create compound statements. Quite importantly, we will also learn how to translate English-like statements into logically correct statements. We will also learn how to quantify over logic statements.

Objectives

At the ends of this lesson, you should

1. Understand propositional and predicate logic
2. Understand logical operations and their respective connectives
3. Understand quantification over logical statements
4. Understand logical equivalences

1.1 Propositional Logic

The rules of logic give precise meaning to mathematical statements and these rules are used to distinguish between valid and invalid mathematical arguments. Besides the importance of logic in understanding mathematical reasoning, logic is also highly applicable in several areas of Computer Science such as the design of computer circuits, the construction of computer programs, the verification of program correctness etc. Let us go on to understand the basic constituents of Propositional logic which are *propositions*.

A proposition is a [declarative] statement that is either true or false, but not both and not neither. This means a propositional statement declares a fact which must have exactly one value – either true or false. Consider the following sentences in Example 1.1:

Example 1.1:

1. Will you come?
2. Nigeria is a country in West Africa
3. Please, take a seat
4. $1 + 3 = 3$
5. Get out of this class now!
6. $x > 1$

Of the six (6) statements, only statements 2 and 4 are propositions because one can answer true or false to those statements. Statement 6 could have been a proposition too but it's not, because the value of x is unknown (we will later see that this can well be turned into propositions via predicate logic). Statement 1 is a question, statement 3 is a request and statement 5 is an instruction. Propositions will often be written as propositional variables using letters of the English alphabet such as *p*, *q*, *r*, *s*, … The value of a proposition, either true or false is referred to as its ***truth value***.

Just as new numbers can be obtained from other numbers using arithmetic operations; new (compound) propositions can be obtained from other propositions using logical operations or connectives represented by arithmetic operators. Table 1.1 gives a list of the logical operators and their meanings.

Table 1.1: Logical connectives and their meanings

Connective	Symbol	Usage example	English translation
Negation	¬ or ~	$\neg p$	***not*** *p*
Disjunction	∨	$p \vee q$	*p* **or** *q*
Conjunction	∧	$p \wedge q$	*p* ***and*** *q*
Implication	→	$p \rightarrow q$	***if*** *p* ***then*** *q*
Bi-conditional	↔	$p \leftrightarrow q$	*p* ***if and only if*** *q*

To better appreciate these connectives, their operations on propositions at all possible combinations of truth values is shown in what is called a Truth table. Table 1.2 shows the truth table for Negation which is the only unary connective and table 1.3 shows the truth table for disjunction, conjunction, implication and bi-conditional. These tables indicate the manner of operation of the logical connectives. Disjunctions are always true except both propositions are false, Conjunctions are always false except both propositions are true bi-conditionals are only true when both propositions have the same truth value. An implication is only false when the condition is true and the conclusion is false. This indicates that the required condition is fulfilled by the expected conclusion from the conclusion is not fulfilled; this is the only time an implication is false. In an implication statement $p \rightarrow q$, p is called condition, hypothesis, premise or antecedent while q is referred to as the conclusion or consequence. The precedence of these operators is as shown in table 1.4 with the negation having the highest precedence and the bi-conditional having the lowest.

Table 1.2: Truth table for negation (¬)

p	**¬ *p***
T	F
F	T

Table 1.3: truth table for Disjunction (∨), Conjunction (∧), Implication (→) and Bi-conditional (↔)

p	***q***	***p* ∨ *q***	***p* ∧ *q***	***p* → *q***	***p* ↔ *q***
T	T	T	T	T	T
T	F	T	F	F	F
F	T	T	F	T	F
F	F	F	F	T	T

Table 1.4: Precedence of logical connectives

Logical Connective	**Precedence**
¬	1
∧	2
∨	3
→	4

$\leftrightarrow$	5

Translating English Statements

A major reason for translating English statements into logical statements is to remove the ambiguity in English (as common to all human language). This may often involve making a set of reasonable assumptions based on the intended meaning of the sentence. However, once we have translated sentences from English into logical expressions, then we can analyse these logical expressions to determine their truth values, we can manipulate them, and we can use rules of inference to reason about them. Examples 1.2 and 1.3 illustrate the process of translating an English sentence into a logical expression.

Examples 1.2: How can this English statement be translated into a logical expression?

"*You can access the Internet from campus only if you are a Computer Science major or you are not a fresher*."

Solution: There are a number of ways to translate this sentence into a logical expression. Although it is possible to represent the sentence by a single propositional variable, such as *p*, this would not be useful when analysing its meaning or reasoning with it. Instead, we will use identify different parts of the sentence and use propositional variables to represent each part then we will determine the appropriate logical connectives to use between them.

Suppose we let *p*, *q*, and *r* represent "*You can access the Internet from campus*," "*You are a Computer Science major*," and "*You are a fresher*," respectively. Note that "only if" is one way a conditional statement can be expressed, this sentence can be represented as

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

Example 1.3: How can this English statement be translated into a logical expression?

"*You cannot ride the roller coaster if you are under 4 feet tall unless you are older than 16 years old*."

Solution: Let *q*, *r*, and *s* represent "*You can ride the roller coaster*," "*You are under 4 feet tall*,"

and "*You are older than 16 years old*," respectively. Then the sentence can be translated to

$$(r \wedge \neg s) \rightarrow \neg q$$

Of course, there are other ways to represent the original sentence as a logical expression, but the one we have used should meet our needs.

Of all the logical connectives, the implication can be particularly difficult to translate from English language because of the many possible ways in which it can be expressed in English language. Some of the common English expressions used to mean $p \rightarrow q$ are:

"if p, then q" "p implies q" "if p, q" "p only if q"
"p is sufficient for q" "a sufficient condition for q is p" "q if p"
"q whenever p" "q when p" "q is necessary for p"
"a necessary condition for p is q" "q follows from p" "q unless $\neg p$"

Compound Logical Expressions

Using the logical connectives, we will now learn how to determine the truth values of compound propositions. Compound propositions are formed from two or more propositions combined with logical connectives. Compound propositions can be classified according to their truth values as ***tautology***, ***contradiction*** or ***contingency***. A compound proposition that is always *true* irrespective of the truth values of the constituent propositions is called a ***tautology***. A compound proposition that is always *false* irrespective of the truth values of the constituent propositions is called a ***contradiction***. A compound proposition that is neither a tautology nor a contradiction is called a ***contingency***. $p \vee \neg p$ and $p \wedge \neg p$ are examples of tautology and contradiction respectively.

Two compound propositions are said to be logically equivalent if they both have the same truth values in all possible cases. Formally, two compound propositions p and q are logically equivalent (written as $p \equiv q$) if $p \leftrightarrow q$ is a tautology. One way to verify the equivalence of two compound propositions is to use the truth table. However, with some common equivalences, the equivalence of any two compound propositions can be verified without the use of truth tables. Some of the common propositional equivalences are shown in tables 1.5, 1.6 and 1.7. Examples 1.4 and 1.5 are used to show how these equivalences can be used to determine the equivalence of any two compound propositions without the use of truth tables.

Example 1.4: Show that $\neg(p \rightarrow q)$ and $p \wedge \neg q$ are logically equivalent.

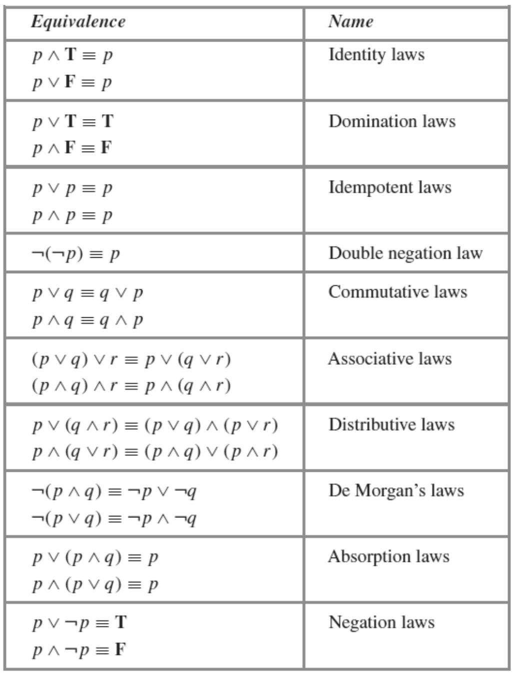
Solution:

$\neg(p \rightarrow q) \equiv \neg(\neg p \vee q)$ (by the 1st equivalence of implications)

$\equiv \neg(\neg p) \land \neg q$ (by De Morgan's law)

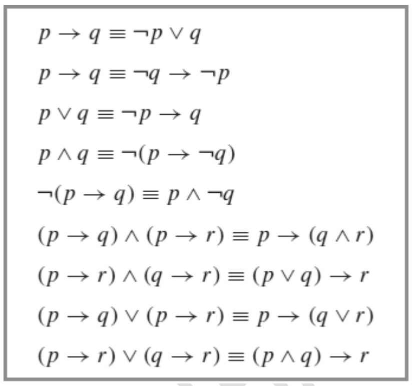
$\equiv p \land \neg q$ (by double negation law)

Table 1.5: Logical equivalences



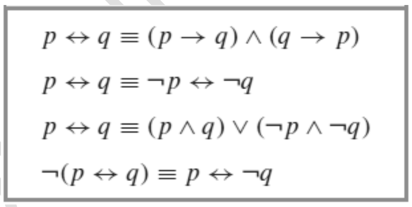
Equivalence	*Name*
$p \land \mathbf{T} \equiv p$ $p \lor \mathbf{F} \equiv p$	Identity laws
$p \lor \mathbf{T} \equiv \mathbf{T}$ $p \land \mathbf{F} \equiv \mathbf{F}$	Domination laws
$p \lor p \equiv p$ $p \land p \equiv p$	Idempotent laws
$\neg(\neg p) \equiv p$	Double negation law
$p \lor q \equiv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

Table 1.6: Logical equivalences of implication



$$p \to q \equiv \neg p \lor q$$
$$p \to q \equiv \neg q \to \neg p$$
$$p \lor q \equiv \neg p \to q$$
$$p \land q \equiv \neg(p \to \neg q)$$
$$\neg(p \to q) \equiv p \land \neg q$$
$$(p \to q) \land (p \to r) \equiv p \to (q \land r)$$
$$(p \to r) \land (q \to r) \equiv (p \lor q) \to r$$
$$(p \to q) \lor (p \to r) \equiv p \to (q \lor r)$$
$$(p \to r) \lor (q \to r) \equiv (p \land q) \to r$$

Table 1.7: Logical equivalences of bi-conditionals



$$p \leftrightarrow q \equiv (p \to q) \land (q \to p)$$
$$p \leftrightarrow q \equiv \neg p \leftrightarrow \neg q$$
$$p \leftrightarrow q \equiv (p \land q) \lor (\neg p \land \neg q)$$
$$\neg(p \leftrightarrow q) \equiv p \leftrightarrow \neg q$$

Example 1.5: Show that $\neg(p \lor (\neg p \land q))$ and $\neg p \land \neg q$ are logically equivalent

Solution:

$\neg(p \lor (\neg p \land q)) \equiv \neg p \land \neg(\neg p \land q)$ (by De Morgan's law)

$\equiv \neg p \land (\neg(\neg p) \lor \neg q)$ (by De Morgan's law)

$\equiv \neg p \land (p \lor \neg q)$ (by double negation law)

$\equiv (\neg p \land p) \lor (\neg p \land \neg q)$ (by distributive law)

$\equiv \boldsymbol{F} \lor (\neg p \land \neg q)$ (by the 2nd negation law)

$\equiv (\neg p \land \neg q) \lor \boldsymbol{F}$ (by commutative law)

$\equiv (\neg p \land \neg q)$ (by the 2nd identity law)

1.2 Predicate Logic

Propositional logic, studied so far, cannot adequately express the meaning of all statements in Mathematics and natural language. For example, suppose that we know that

"Every computer connected to the university network is functioning properly."

No rules of propositional logic allow us to conclude the truth of the statement

"MATH3 is functioning properly,"

Where MATH3 is one of the computers connected to the university network. Likewise, we cannot use the rules of propositional logic to conclude from the statement

"CS2 is under attack by an intruder,"

where CS2 is a computer on the university network, to conclude the truth of

"There is a computer on the university network that is under attack by an intruder."

In this section we will introduce a more powerful type of logic called Predicate logic. We will see how predicate logic can be used to express the meaning of a wide range of statements in Mathematics and Computer Science in ways that permit us to reason and explore relationships between objects. To understand predicate logic, we first need to introduce the concept of a predicate. Afterward, we will introduce the notion of quantifiers, which enable us to reason with statements that assert that a certain property holds for all objects of a certain type and with statements that assert the existence of an object with a particular property.

Predicates

Statements involving variables such as

$x \leq 4$, $x = y - 2$, $x > 3$, "computer *x* is faulty". "computer *y* is functioning properly"

often occur in mathematical assertions, computer programs and system specifications. However, it is impossible to establish the truth values of such statements without knowing the values of the variables in them. This kind of statements can be made into propositions and given truth values by using predicates.

Consider the statement

"*x* is greater than 3"

The statement has two parts. The first part, the variable *x*, is the subject of the statement. The second part—the predicate, "is greater than 3"—refers to a property that the subject of the statement can have. We can denote the statement "x is greater than 3" by *P(x),* where *P* denotes the predicate "is greater than 3" and *x* is the variable. The statement *P(x)* is also said to be the value of the propositional function *P* at *x*. Once a value has been assigned to the variable *x*, the statement *P(x)* becomes a proposition and has a truth value. Consider Examples 1.6 and 1.7. A predicate is also known as a propositional function.

Example 1.6:

Let *P(x)* denote the statement "$x > 3$." What are the truth values of *P(4)* and *P(2)*?

Solution: We obtain the statement *P(4)* by setting $x = 4$ in the statement "$x > 3$." Hence, *P(4)*, which is the statement "$4 > 3$," is true. However, *P(2)*, which is the statement "$2 > 3$," is false.

Example 1.7:

Let *Q(x, y)* denote the statement "$x = y + 3$." What are the truth values of the propositions *Q(1, 2)* and *Q(3, 0)*?

Solution: To obtain *Q(1, 2)*, set $x = 1$ and $y = 2$ in the statement *Q(x, y)*. Hence, *Q(1, 2)* is the statement "$1 = 2 + 3$," which is false. The statement *Q(3, 0)* is the proposition "$3 = 0 + 3$," which is true.

In general, a statement involving *n* variables $x_1, x_2, \ldots, x_n$ can be denoted by $P(x_1, x_2, \ldots, x_n)$. A statement of the form $P(x_1, x_2, \ldots, x_n)$ is the value of the propositional function *P* at the *n*-tuple $(x_1, x_2, \ldots, x_n)$, and *P* is also called an *n*-place predicate or a *n*-ary predicate.

Quantifiers

When the variables in a propositional function are assigned values, the resulting statement becomes a proposition with a certain truth value. However, there is another important way, called quantification, to create a proposition from a propositional function. Quantification expresses the extent to which a predicate is true over a range of elements. In English, the words all, some, many, none, and few are used in quantifications. We will focus on two types of quantification: *universal quantification*, which tells us that a predicate is true for every element under consideration, and *existential quantification*, which tells us that there is one or more element under consideration for which the predicate is true. The area of logic that deals with predicates and quantifiers is called Predicate Calculus.

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. $\forall x P(x)$ is read 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 $\forall x P(x)$.

Example 1.8:

Let $P(x)$ be the statement "$x + 1 > x$." What is the truth value of the quantification $\forall x P(x)$, where the domain consists of all real numbers?

Solution: Because $P(x)$ is true for all real numbers x, the quantification $\forall x P(x)$ is true.

Note: The truth value of $\forall x P(x)$ always depends on the domain, so it is important to know the domain or universe of discourse under consideration and to treat the proposition under the specified domain. Apart from "for all" and "for every", other ways of representing universal quantification include "for each", "all of", "given any", "for arbitrary", "for any".

Example 1.9: Suppose that $P(x)$ is "$x^2 > 0$." To show that the statement $\forall x P(x)$ is false where the universe of discourse consists of all integers, we will need to find a counterexample. We see that $x = 0$ is a counterexample because $x^2 = 0$ when $x = 0$, so that x^2 is not greater than 0 when $x = 0$.

Looking for counterexamples to universally quantified statements is an important activity in the study of mathematics. When all the elements in the domain can be listed—say, $x_1, x_2, \ldots, x_n$—it follows that the universal quantification $\forall x P(x)$ is the same as the conjunction

$$P(x_1) \wedge P(x_2) \wedge \cdots \wedge P(x_n),$$

because this conjunction is true if and only if $P(x_1), P(x_2), \ldots, P(x_n)$ are all true.

Example 1.10: What is the truth value of $\forall x P(x)$, where $P(x)$ is the statement "$x^2 < 10$" and the domain consists of the positive integers not exceeding 4?

Solution: The statement $\forall x P(x)$ is the same as the conjunction

$$P(1) \wedge P(2) \wedge P(3) \wedge P(4),$$

because the domain consists of the integers 1, 2, 3, and 4. Because $P(4)$, which is the statement

"$4^2 < 10$," is false, it follows that $\forall x P(x)$ is false.

Apart from the universal quantifier is the existential quantifier which tells us that there is at least one element in the domain under consideration for which a given predicate is true.

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 $\exists xP(x)$ for the existential quantification of *P(x)* and $\exists$ is called the existential quantifier.

A domain must always be specified when a statement $\exists xP(x)$ is used, as the meaning of $\exists$xP(x) changes when the domain changes. Thus, without specifying the domain, the statement $\exists$xP(x) has no meaning.

Besides the phrase “there exists,” we existential quantification can also be expressed using phrases like “for some,” “for at least one,” or “there is.” The existential quantification $\exists xP(x)$ 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)*”.

The next three examples illustrate the interpretation of the existential quantifier and a summary of bot quantifiers discussed so far is shown in table 1.8.

Example 1.11: Let *P(x)* denote the statement “$x > 3$.” What is the truth value of the quantification $\exists xP(x)$, where the domain consists of all real numbers?

Solution: Because “$x > 3$” is sometimes true—for instance, when $x = 4$—the existential quantification of *P(x)*, which is $\exists xP(x)$, is true.

Example 1.12: Let *Q(x)* denote the statement “$x = x + 1$.” What is the truth value of the quantification $\exists xQ(x)$, where the domain consists of all real numbers?

Solution: Because *Q(x)* is false for every real number *x*, the existential quantification of *Q(x)*, which is $\exists xQ(x)$, is false.

When all elements in the domain can be listed—say, $x_1, x_2, . . . , x_n$—the existential quantification $\exists xP(x)$ is the same as the disjunction

$$P(x_1) \vee P(x_2) \vee \cdot \cdot \cdot \vee P(x_n),$$

because this disjunction is true if and only if at least one of $P(x_1), P(x_2), . . . , P(x_n)$ is true

Example 1.13: What is the truth value of $\exists x P(x)$, where $P(x)$ is the statement "$x^2 > 10$" and the universe of discourse consists of the positive integers not exceeding 4?

Solution: Because the domain is {1, 2, 3, 4}, the proposition $\exists x P(x)$ is the same as the disjunction

$$P(1) \vee P(2) \vee P(3) \vee P(4).$$

Because $P(4)$, which is the statement "$4^2 > 10$," is true, it follows that $\exists x P(x)$ is true.

Table 1.8: Summary of the Quantifiers

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

Translating English Statements

Earlier in this lesson, we learnt how to translate English language statements to logic statements and vice versa using propositional logic. However, now that we understand predicate logic and that it is more expressive than propositional logic statements, let's learn how to translate English statements into predicate logic statements. There are no hard and fast rules for doing this as there can even be more than one way of translating a particular natural language sentence into a logical statement. However, with the aid of the simple examples provided in this section, you should be able to handle some other such tasks.

Example 1.14: Express the statement "Every student in this class has studied calculus" using predicates and quantifiers.

Solution: First, we rewrite the statement so that we can clearly identify the appropriate quantifiers to use. Thus, we have

"For every student in this class, that student has studied calculus."

Next, we introduce a variable x so that our statement becomes

"For every student x in this class, x has studied calculus."

Then, we introduce a predicate $C(x)$, which is the statement "x has studied calculus." Consequently, if the domain for x consists of the students in the class, we can translate our statement as $\forall x C(x)$. However, there are other correct approaches; different domains of discourse and other predicates can be used. The approach we select depends on the subsequent reasoning we want to carry out. For example, we may be interested in a wider group of people than only those in this class. If we change the domain to consist of all people, we will need to express our statement as

"For every person *x*, if person *x* is a student in this class then *x* has studied calculus."

If $S(x)$ represents the statement that person *x* is in this class, we see that our statement can be expressed as $\forall x(S(x) \rightarrow C(x))$. [Caution! Our statement cannot be expressed as $\forall x(S(x) \wedge C(x))$ because this statement says that all people are students in this class and have studied calculus!]

Finally, when we are interested in the background of people in subjects besides calculus, we may prefer to use the two-variable quantifier $Q(x, y)$ for the statement "student *x* has studied subject *y*." Then we would replace $C(x)$ by $Q(x, calculus)$ in both approaches to obtain

$$\forall x Q(x, calculus) \text{ or } \forall x(S(x) \rightarrow Q(x, calculus)).$$

Example 1.15: Express the statements "Some student in this class has visited Mexico" and "Every student in this class has visited either Canada or Mexico" using predicates and quantifiers.

Solution: The statement "Some student in this class has visited Mexico" means that

"There is a student in this class with the property that the student has visited Mexico."

We can introduce a variable *x*, so that our statement becomes

"There is a student *x* in this class having the property that *x* has visited Mexico."

We introduce $M(x)$, which is the statement "*x* has visited Mexico." If the domain for *x* consists

of the students in this class, we can translate this first statement as $\exists x M(x)$.

However, if we are interested in people other than those in this class, we look at the statement

a little differently. Our statement can be expressed as

"There is a person *x* having the properties that *x* is a student in this class and *x* has visited

Mexico."

In this case, the domain for the variable x consists of all people. We introduce $S(x)$ to represent "x is a student in this class." Our solution becomes $\exists x(S(x) \land M(x))$ because the statement is that there is a person x who is a student in this class and who has visited Mexico. [Caution! Our statement cannot be expressed as $\exists x(S(x) \rightarrow M(x))$, which is true when there is someone not in the class because, in that case, for such a person x, $S(x) \rightarrow M(x)$ becomes either $F \rightarrow T$ or $F \rightarrow F$, both of which are true.]

Similarly, the second statement can be expressed as

"For every x in this class, x has the property that x has visited Mexico or x has visited

Canada."

(Note that we are assuming the inclusive, rather than the exclusive, OR here.) We let $C(x)$ be "x has visited Canada." Following our earlier reasoning, we see that if the domain for x consists of the students in this class, this second statement can be expressed as $\forall x(C(x) \lor M(x))$. However, if the domain for x consists of all people, our statement can be expressed as

"For every person x, if x is a student in this class, then x has visited Mexico or x has visited Canada."

In this case, the statement can be expressed as $\forall x(S(x) \rightarrow (C(x) \lor M(x)))$. Instead of using $M(x)$ and $C(x)$ to represent that x has visited Mexico and x has visited Canada, respectively, we could use a two-place predicate $V(x, y)$ to represent "x has visited country y." In this case, $V(x, Mexico)$ and $V(x, Canada)$ would have the same meaning as $M(x)$ and $C(x)$ and could replace them in our answers. If we are working with many statements that involve people visiting different countries, we might prefer to use this two-variable approach. Otherwise, for simplicity, we would stick with the one-variable predicates $M(x)$ and $C(x)$.

Summary

In this lesson, you have learnt

1. That a proposition is a statement that is either true or false but not neither
2. About logical connectives and their usage/operations
3. how to translate English statements to propositional logic statements

4. About predicates and quantifiers and how they are used to realize more expressive logic statements

Post-Test

1. Use truth tables to show whether each of the following compound proposition is a tautology, contradiction or contingency
   a. $p \vee \neg p$
   b. $p \wedge \neg p$
   c. $(p \wedge q) \rightarrow p$
   d. $(p \rightarrow q) \rightarrow \neg q$
2. Use truth tables to verify the logical equivalences in tables 1.5, 1.6 and 1.7
3. Translate the following English statements into propositional logical expressions using logical connectives
   a. If it doesn't rain, then I will go to the party
   b. I supplied the correct password, but was still denied access
   c. In the university, you are either a student or a staff
   d. I go to the library whenever I have to read
4. Given the predicate $P(x)$ means "$x^3 > 0$". What is the truth value of the following quantified predicates?
   a. $\forall x P(x)$ where $x \in Z$ (the set of all integers)
   b. $\forall x P(x)$ where $x \in Z^+$ (the set of all positive integers)
   c. $\exists x P(x)$ where x belongs the set of all real numbers between 0 and 1
   d. $\exists x P(x)$ where x belongs the set of all real numbers between -1 and 0

References

1. Rosen, K. Discrete Mathematics and its Applications. 7th Edition, McGraw-Hill, 2012.
2. Martin, J. C. Introduction to Languages and the Theory of Computation. 4th Edition, McGraw-Hill, 2011.

Lesson 2: Sets

Introduction

In this lesson, we will study sets as a data structure and understand how data is stored in sets. A set is an unordered collection of elements and we shall see that it is perhaps the most basic data structure.

The important concepts that we shall study include set membership, set operations such as union, intersection and complement, the cardinality of sets, subsets, supersets, power set and set identities. We shall also learn how to represent sets with the Venn diagram and how to use the inclusion and exclusion principle to derive set cardinalities.

Objectives

At the end of this lesson, you should know

1. what sets are and how they are built
2. the various special sets and their properties
3. the various set operations and how to use them to derive new sets
4. how to determine the power set of a set
5. how to represent a set using the Venn diagram

2.1 Introduction

The set is a fundamental discrete (data) structure on which most other structures are built. A set is an unordered collection of items (called members or elements) which may or may not be related (have similar properties). For instance, all students taking ***CISXXX*** make up a set of related objects, while the set {*a, true, 1, car*} is a set of unrelated items, but it's still a set. A set is said to ***contain*** its elements while the elements of a set are said to ***belong*** to the set. We write $a \in A$ and to denote that an element *a* belongs to the set *A* and that write $a \notin A$ to denote that an element *a* ***does not*** belong to the set *A* respectively. Usually, the name of the set is written in upper case while the name of its elements is written in lower case.

Two popular notations for writing sets are the ***roster*** and ***set-builder*** notations. The roster notation lists the elements of the set and uses ellipses (…) where the pattern of set elements is obvious, while the set-builder notation describes a set by stating the property(ies) that every element of such set must have.

The following sets are written in roster notation: $\{a, b, c, \ldots, z\}$, $\{a, e, i, o, u\}$, $\{2, 4, 6, 8, \ldots\}$, $\{1, 4, 9, 16, 25\}$.

The following sets are written in set-builder notation: $\{x \mid x \text{ is even and } x \in Z^+\}$, $\{x \mid x \text{ is a perfect square}\}$.

We list the following special sets which we shall find useful in this course:

N = $\{1, 2, 3, \ldots\}$, the set of natural numbers

Z = $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$, the set of integers

Z$^+$ = $\{1, 2, 3, \ldots\}$, the set of positive integers

Q = $\{p/q \mid p \in Z, q \in Z, \text{ and } q \neq 0\}$, the set of rational numbers

R, the set of real numbers

R$^+$, the set of positive real numbers

C, the set of complex numbers.

Note that sets form what is referred to as data types in programming languages. For instance, most programming languages define a data type for integers which is essentially the set **Z** defined above, the data type Boolean is also the set $\{0, 1\}$ or the {true, false}.

Set Equality: Two sets are said to be equal if and only if they both contain the same elements. Therefore, suppose *A* and *B* are sets, from our knowledge of logic, we can say that *A* and *B* are equal if the logic statement $\forall x(x \in A \leftrightarrow x \in B)$ is true.

Thus, the sets {*1, 2, 3*} and {*3, 1, 2*} are equal. Since sets are unordered, the ordering of elements does not matter, as long as the sets contain the same elements, they are equal. Also, the sets {*a, b, c, d*} and the sets {*c, a, c, b, a, d*} are the same since they contain the same elements irrespective of the repeated ones.

The Empty Set: The empty set is a special set which contains no elements. It is denoted with the symbol Ø or {}. Note that the empty set is quite different from the sets { Ø } or { *0* } which are both singleton sets (i.e. a set with one element).

Set Cardinality: The size of a set *A* is referred to as its cardinality and is written as |*A*|. Therefore, given a set *S* = {*1, 4, 9, 16*}, |*S*| = *4*, and given the set *B* = {*a, e, i, o, u*}, |*B*| = *5*. For the set *C* = {*a, b, a, c*}, |*C*| = *3*, because repeated set elements are only counted once.

2.2 Set Operations

Just as we could perform logical operations on propositions, we can perform set operations on sets. We will look at three set operations namely the *union*, *intersection* and *complement* operations.

Union: The union of two sets *A* and *B*, written as *A* ∪ *B* is the set containing elements that belong to ***either*** in *A* ***or*** in *B* – thus, elements contained in both sets are only listed once. We can write this as

$$A \cup B = \{x \mid (x \in A) \vee (x \in B)\}$$

Example 2.1: The union of the sets *A* = {*a, b, c, d*} and *B* = {*1,2,3*} is *A* ∪ *B* = {*a, b, c, d, 1, 2, 3*}

Intersection: The intersection of two sets *A* and *B*, written as *A* ∪ *B* is the set containing elements that belong to ***both*** in *A* ***and*** in *B*. We can write this as

$$A \cap B = \{x \mid (x \in A) \wedge (x \in B)\}$$

Example 2.2: The intersection of the sets *A* = {*a, 2, b, c*} and *B* = {*a, 1, 2, 3*} is *A* ∩ *B* = {*a, 2*}

Note: Two sets *A* and *B* with *A* ∩ *B* = Ø are said to be disjoint

Complement: Let A and B be sets, the difference of A and B written as $A – B$ (or sometimes written as $A \setminus B$) is the set of elements belonging to A but not to B. The difference of A and B is also called the complement of B with respect to A. We can write this as

$$A - B = \{x \mid (x \in A) \wedge ((x \notin B))\}$$

When a universal set U is defined, then we can determine the complement of a set A, written as A', this is simply the set of elements in the universal set U but not in A. and in terms of set difference, this is $U – A$.

Example 2.3: Given two sets $A = \{1, 2, 3, 4, 5\}$, $B = \{2, 4, 6\}$ and a universal set $U = \{1, 2, 3, \ldots, 10\}$.

$A – B = \{1, 3, 5\}$ $\qquad$ $B – A = \{6\}$

$A' = \{6, 7, 8, 9, 10\}$ $\qquad$ $B' = \{1, 3, 5, 7, 8, 9, 10\}$

From the above, it is obvious that $A – B = A \cap B'$

2.3 Subsets, Supersets and Power Set

Now we discuss subsets, supersets and power set. A set can contain other sets, and when this happens, the contained set is called a subset of the containing set while the containing set is called a superset of the contained set. This also means that every element of the contained set is also a member of the containing set.

Let A and B be two sets such that $B \in A$, then we write

$B \subseteq A$, meaning B is a subset of A (i.e. B is a set contained in A)

and write $A \supseteq B$, meaning A is a superset of B (i.e. A is a set containing B).

Intuitively, this means that every element in B is also contained in A. Thus, we can define this using predicate logic as follows:

$$\forall x(x \in B \rightarrow x \in A)$$

Example 2.4: Given the set $A = \{a, b, \{b, c\}, d, \{a\}\}$. The set A contains two subsets namely, $\{b, c\}$ and $\{a\}$. Thus, we can write $\{b, c\} \subseteq A$ and $\{a\} \subseteq A$. Note that there are 5 elements in the set A, that is $|A| = 5$.

A typical example of subsets and supersets is seen with the special sets discussed earlier. A careful observation of the sets reveals that the following relationships hold:

$$\mathbf{N} \subseteq \mathbf{Z} \subseteq \mathbf{Q} \subseteq \mathbf{R} \subseteq \mathbf{C}$$

When we wish to emphasize that a set B is a subset of a set A but that $A \neq B$, we write $B \subset A$ and say that B is a ***proper subset*** of A. For $B \subset A$ to be true, it must be the case that $B \subseteq A$ and there must exist an element x of A that is not an element of B. That is, B is a proper subset of A if and only if the following proposition is true

$$\forall x(x \in B \rightarrow x \in A) \wedge \exists x(x \in A \wedge\ x \notin B)$$

For any two sets A, B, if $B \subseteq A$ holds, the $A \supseteq B$ holds and is read "A superset B", which means the set A contains the set B. It's just another way of saying that a set is a subset of another set.

Power Set: The power set of a set A, denoted $p(A)$, is the set of all subsets of A.

Example 2.5: What is the power set of the set $A = \{1, 2, 3\}$

Solution: $p(A) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$

Example 2.6: What is the power set of the set $A = \{a, b\}$

Solution: $p(A) = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$

From the power set, we see that the empty set is a subset of every set and every set is also a subset of itself. Also, the cardinality of the power set of a set A can be obtained with the formula 2^n, where n is the cardinality of A. Therefore, a set with cardinality 4, will have power set with cardinality $2^4 = 16$.

2.4 Venn Diagram

The Venn Diagram, developed by John Venn, is a graphical illustration of sets and set operations. In a Venn diagram, a set is represented as a circle drawn inside a rectangle which stands for the Universal set. Figure 2.1a shows the Venn diagram representation of a set A and the universal set U and Figure 2.1b shows the representation of subsets. The shaded portions in Figure 2.2a and Figure 2.2b represent the union and intersection respectively of two sets A and B. Likewise, the shaded portions in Figure 2.3a and Figure 2.3b represent the set difference of two sets A and B and complement of a set A respectively.

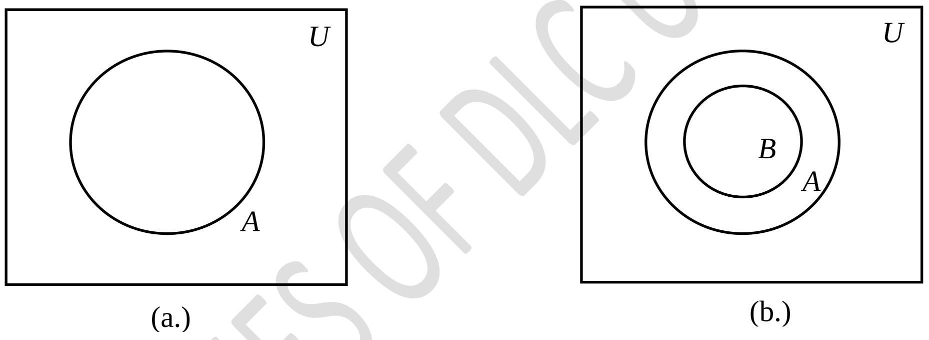


Figure 2.1: (a.) Venn Diagram Representation of a set (b.) Subset representation ($B \subseteq A$)

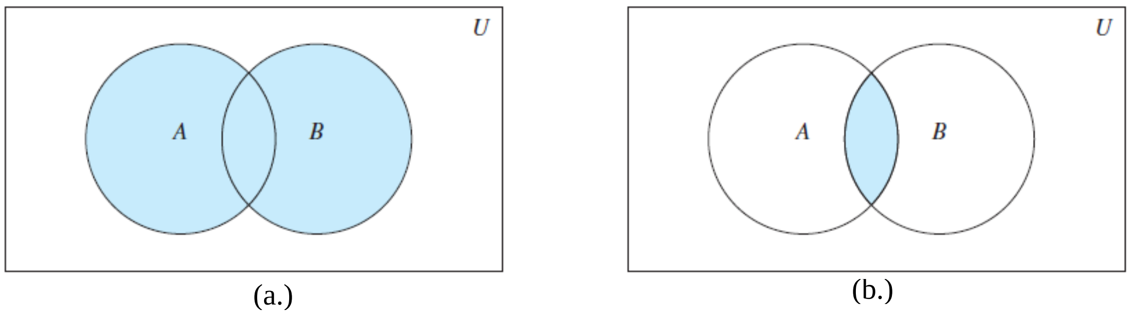
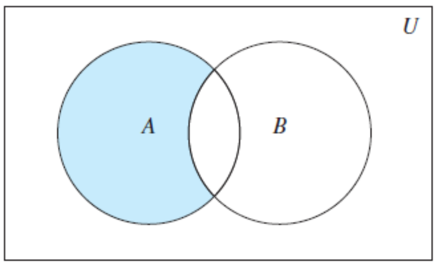
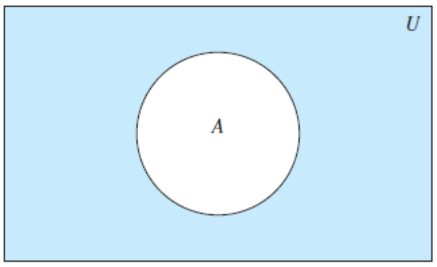


Figure 2.2: (a.) Set Union ($A \cup B$) (b.) Set Intersection ($A \cap B$)



(a.)



(b.)

Figure 2.3: (a.) Set Difference (*A* – *B*) (b.) Set Complement (*A′*)

Summary

In this lesson, we have learnt that

1. A set is an unordered collection of not necessarily related items
2. Sets form the fundamental discrete/data structure upon which all others are built
3. Sets are said to contain their elements and an element is said to belong to a set.
4. Set operations include union, intersection and difference (complement)
5. Sets can contain other sets called subsets.
6. The empty set Ø, is a subset of every set and every set is a subset of itself
7. The power set of a set is the et of all its subsets.
8. Venn diagrams are graphical representation of sets and their operations.

Post-Test

1. Given the set $A = \{x^2 \mid x \in Z^+ \wedge x \leq 10\}$. Use the roster notation to list the elements of *A* and find $|A|$
2. Given the set $B = \{x \mid x \in Z^+ \wedge x \leq 100 \wedge x \text{ is prime}\}$ and the set *A* as defined in 1 above.

   a. Find $A \cup B$

   b. Find $A \cap B$

   c. Draw the Venn diagram to represent the intersection and union of the two sets.
3. What is the power set of the set $V = \{a, e, i, o, u\}$

References

1. Rosen, K. Discrete Mathematics and its Applications. 7th Edition, McGraw-Hill, 2012.

Lesson 3: Mathematical Induction

Introduction

Mathematical Induction is a very important technique used to prove mathematical assertions, program correctness, the complexity of algorithms as well as some identities and inequalities. Generally speaking, the Principle of Mathematical Induction (PMI) is used to prove results already obtained by some means and not to derive new formulae or theorem. The idea of PMI is to prove that a particular assertion is true for all positive integers n, by showing that it is true for $n = 1$, and show that if it holds for any integer k, then it holds for the next integer $k + 1$. This two parts of the PMI are what we shall use to prove assertions and results all through this chapter. We will also study a more complex form of the PMI which is called the Strong Induction which is suitable for proving results that cannot be easily proven by mathematical induction.

Objectives

At the end of this lesson, you should be able to

1. Understand the Principle of Mathematical Induction (PMI)
2. Know how to use PMI to prove summation theorems
3. Know how to use PMI to prove inequality theorems
4. Know how to use PMI to prove divisibility theorems

3.1 Introduction

Suppose that we have an infinite ladder and we want to know whether we can reach every step on this ladder. We know two things:

1. We can reach the first rung of the ladder.
2. If we can reach a particular rung of the ladder, then we can reach the next rung.

Can we conclude that we can reach every rung? By (1), we know that we can reach the first rung of the ladder. Moreover, because we can reach the first rung, by (2), we can also reach the second rung, since it is the next rung after the first rung. Applying (2) again, because we can reach the second rung, we can also reach the third rung. Continuing in this way, we can show that we can reach the fourth rung, fifth rung and so on. But can we conclude that we are able to reach every rung of this infinite ladder? The answer is yes, and we can verify this using an important proof technique called mathematical induction. That is, we can show that $P(n)$ is true for every positive integer n, where $P(n)$ is the statement that we can reach the n^{th} rung of the ladder. Mathematical induction is an extremely important proof technique that can be used to prove assertions of this type.

Generally, mathematical induction can be used to prove statements that assert that $P(n)$ is true for all positive integers n, where $P(n)$ is a propositional function. A proof by mathematical induction has two parts, a ***basis step***, where we show that $P(1)$ is true, and an ***inductive step***, where we show that for all positive integers k, if $P(k)$ is true, then $P(k + 1)$ is true. The Principle of Mathematical Induction (PMI) is stated as follows:

To prove that a propositional function $P(n)$ is true for all positive integers n, we must complete two steps – the basis step and the inductive step – defined as follows:

1. *The Basis step*: show that $P(1)$ is true.
2. *The Inductive Step*: show that the proposition $P(k) \rightarrow P(k + 1)$ is true for every positive integer k.

To use PMI to prove that an assertion is true, we first show that the basis step holds i.e. $P(1)$ is true, then we prove that the inductive step holds by assuming that $P(k)$ is true and under this assumption we show that $P(k + 1)$ must also be true. The assumption that $P(k)$ is true is called the *inductive hypothesis*. Note that the inductive hypothesis does not assume that $P(k)$ is true for all positive integers, rather it only shows that if $P(k)$ is true, then $P(k + 1)$ is also true for all positive integers. Once we complete both steps in a proof by mathematical induction, we have shown that $P(n)$ is true for all positive integers, that is, we have shown that $\forall n P(n)$ is true where the quantification is over the set of positive integers. Expressed as a rule of inference, this proof technique can be stated as

$$\forall n \in \mathbf{Z}^{+}(\ (P(1) \wedge \forall k(P(k) \rightarrow P(k+1))) \rightarrow \forall n P(n)\ )$$

3.2 Example Proofs

We will now see how to use PMI to prove theorems using a number of examples. Remember that mathematical induction can be used to prove assertions that are said to be true for all positive integers. Note that most theorem and mathematical results, implicitly contain the statement "for all …", but it is not usually stated explicitly. Remember also that Mathematical Induction is used to prove results, theorems or formulae already obtained by some means, but it cannot be used to obtain new formulae. Thus, let us look at the following examples

Example 3.1: Show that if n is a positive integer, then

$$1+2+\cdots+n=\frac{n(n+1)}{2}$$

Solution: The above is a closed formula for the summation of the first n positive integers. To prove by mathematical induction that this formula holds, we must show that the basis step as well as the inductive step hold.

Let P(n) be the proposition $1+2+\cdots+n=\frac{n(n+1)}{2}$

The Basis step: To show that $P(1)$ holds, we substitute 1 for n in the equation:

$$1=\frac{1(1+1)}{2}$$

The Inductive step: We start with the inductive hypothesis, assuming that $P(k)$ holds, which means

$$1+2+\cdots+k=\frac{k(k+1)}{2}$$

Then if $P(k)$ is true, then $P(k+1)$ must be true as well. This means

$$1+2+\cdots+k+(k+1)=\frac{(k+1)[(k+1)+1]}{2}=\frac{(k+1)(k+2)}{2}$$

To verify that $P(k + 1)$ is true, we add $(k + 1)$ to both sides of $P(k)$.

$$1 + 2 + \cdots + k + (k + 1) = \frac{k(k+1)}{2} + (k + 1)$$

$$= \frac{k(k+1) + 2(k+1)}{2}$$

$$= \frac{(k+1)(k+2)}{2}$$

From this last equation, we see that under the assumption that $P(k)$ is true, $P(k + 1)$ is also true and this completes the inductive step of the proof by mathematical induction. Thus, we have shown by mathematical induction that the theorem $1 + 2 + \cdots + n = \frac{n(n+1)}{2}$ is true for every positive integer n.

Example 3.2: Use PMI to show that

$$1 + 2 + 2^2 + \cdot \cdot \cdot + 2^n = 2^{n+1} - 1$$

for every non-negative integer n.

Solution: First, note that the condition attached to the theorem says it holds for all non-negative integers, which means the set of positive integers including zero (0). Thus, the basis step becomes $P(0)$.

Let $P(n)$ be the proposition $1 + 2 + 2^2 + \cdot \cdot \cdot + 2^n = 2^{n+1} - 1$

Basis step: We know that $P(0)$ is true because $2^0 = 1 = 2^{0+1} - 1$, which shows $P(0)$ is true.

Inductive Step: For the inductive hypothesis, we assume that $P(k)$ is true for an arbitrary nonnegative integer, k, so that we have

$$1 + 2 + 2^2 + \cdot \cdot \cdot + 2^k = 2^{k+1} - 1$$

If $P(k)$ is true, the we know that $P(k + 1)$ must be true as well. Thus, we need to verify that $P(k + 1)$, that is

$$1 + 2 + 2^2 + \cdots + 2^k + 2^{k+1} = 2^{(k+1)+1} - 1 = 2^{k+2} - 1$$

is true under the assumption that $P(k)$ is true. Therefore, under the assumption that $P(k)$ is true, we add the $(k + 1)$th term to both sides of $P(k)$ and have

$$1 + 2 + 2^2 + \cdots + 2^k + 2^{k+1} = (2^{k+1} - 1) + 2^{k+1}$$

$$= 2^{k+1} + 2^{k+1} - 1$$

$$= 2(2^{k+1}) - 1$$

$$= 2^1 \times 2^{k+1} - 1$$

$$= 2^{1+k+1} - 1$$

$$= 2^{k+2} - 1$$

Having completed the inductive step, we know by Mathematical induction that $P(n)$ is true for all nonnegative integers, that is $1 + 2 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$.

The two examples above have used PMI to prove summation theorems. We will now study how PMI is used to prove other theorems that are not necessarily summations. The next two examples show how PMI is used to prove inequalities.

Example 3.3: Use PMI to prove the inequality $n < 2^n$.

Solution: Let $n < 2^n$ the proposition $P(n)$.

Basis step: We know that $P(1)$ is true because $1 < 2^1$

Inductive step: We first assume the inductive hypothesis that $P(k)$ is true for an arbitrary positive integer k. That is, the inductive hypothesis $P(k)$ is the statement that $k < 2^k$. To complete the inductive step, we need to show that if $P(k)$ is true, then $P(k + 1)$, which is the statement that $k + 1 < 2^{k+1}$, is true. That is, we need to show that if $k < 2^k$, then $k + 1 < 2^{k+1}$.

To show that this conditional statement is true for the positive integer k, we first add 1 to both sides of $k < 2^k$, and then note that $1 \leq 2^k$. This tells us that

$$k + 1 < 2^{k+1} \leq 2^k + 2^k = 2 \cdot 2^k = 2^{k+1}.$$

This shows that $P(k + 1)$ is true, namely, that $k + 1 < 2^{k+1}$, based on the assumption that $P(k)$ is true. The induction step is complete.

Therefore, because we have completed both the basis step and the inductive step, by the principle of mathematical induction we have shown that $n < 2^n$ is true for all positive integers n.

Example 3.4: Use mathematical induction to prove that $2^n < n!$ for every integer n with $n \geq 4$. (Note that this inequality is false for $n = 1, 2$, and 3.)

Solution: Let $P(n)$ be the proposition $2^n < n!$.

Basis step: The basis step is $P(4)$ since the proposition only holds for integers greater than or equal to 4. It is obvious that $P(4)$ is true, because

$2^4 < 4!$

$= 16 < 24$.

Inductive step: For the inductive step, we assume that $P(k)$ is true for an arbitrary integer k with $k \geq 4$. That is, we assume that $2^k < k!$ for the positive integer k with $k \geq 4$. We must show that under this hypothesis, $P(k + 1)$ is also true. That is, we must show that if $2k < k!$ for an arbitrary positive integer k where $k \geq 4$, then $2^{k+1} < (k + 1)!$. We have

$2^{k+1} = 2 \cdot 2k$ (by definition of exponent)

$< 2 \cdot k!$ (by the inductive hypothesis)

$< (k + 1)k!$ (because $2 < k + 1$)

$= (k + 1)!$ (by definition of factorial function)

This shows that $P(k + 1)$ is true when $P(k)$ is true. This completes the inductive step of the proof. We have completed the basis step and the inductive step. Hence, by mathematical induction *P(n)* is true for all integers n with $n \geq 4$. That is, we have proved that $2n < n!$ is true for all integers n with $n \geq 4$.

The final class of assertion that we will learn to prove with mathematical induction involves the results of divisibility of integers and we shall illustrate this with the next example as well. For proofs of this kind, keep in mind, the following divisibility theorem, which we shall find useful:

Theorem 1: Let a, b, and c be integers and $a \neq 0$:

i.) if $a \mid b$ and $a \mid c$, then $a \mid (b + c)$

ii.) if $a \mid b$ and $a \mid c$, then $a \mid bc$

iii.) if $a \mid b$ and $b \mid c$, then $a \mid c$

Note: $a \mid b$ means a divides b **OR** b is divisible by a.

Example 3.5: Use mathematical induction to prove that $n^3 - n$ is divisible by 3 whenever n is a positive integer.

Solution: Let $P(n)$ be the proposition: "$n^3 - n$ is divisible by 3"

Basis step: $P(1)$ is true because $1^3 - 1 = 0$ is divisible by 3.

Inductive step: For the inductive hypothesis we assume that P(k) is true; that is, we assume that $k^3 - k$ is divisible by 3 for an arbitrary positive integer k. To complete the inductive step, we must show that when we assume the inductive hypothesis, it follows that $P(k + 1)$, the statement that $(k + 1)^3 - (k + 1)$ is divisible by 3, is also true. That is, we must show that

$(k + 1)^3 - (k + 1)$ is divisible by 3.

Note that

$(k + 1)^3 - (k + 1) = (k^3 + 3k^2 + 3k + 1) - (k + 1)$

$= (k^3 - k) + 3(k^2 + k)$.

Using the inductive hypothesis, we conclude that the first term $k^3 - k$ is divisible by 3. The second term is divisible by 3 because it is 3 times an integer. So, by part (*i*) of Theorem 1 above, we know that $(k + 1)^3 - (k + 1)$ is also divisible by 3. This completes the inductive step. Because we have completed both the basis step and the inductive step, by the principle of mathematical induction we know that $n^3 - n$ is divisible by 3 whenever n is a positive integer.

3.3 Guidelines for constructing correct proofs by Mathematical Induction

The kind of theorems and results that can be proven by mathematical induction is not limited to the types proven in the examples given above. In other to be able to prove other theorem types, following guidelines will be very helpful.

1. Express the statement that is to be proved in the form "for all $n \geq b$, $P(n)$" for a fixed integer b.

2. For the Basis step, show that $P(b)$ is true, taking care that the correct value of b is used. This completes the first part of the proof.

3. For the Inductive Step, state, and clearly identify, the inductive hypothesis, in the form "assume that $P(k)$ is true for an arbitrary fixed integer $k \geq b$."

5. State what needs to be proved under the assumption that the inductive hypothesis is true. That is, write out what $P(k + 1)$ is.

6. Prove the statement $P(k + 1)$ making use of the assumption $P(k)$. Be sure that your proof is valid for all integers k with $k \geq b$, taking care that the proof works for small values of k, including $k = b$.

7. Clearly identify the conclusion of the inductive step, such as by saying "this completes the inductive step."

8. After completing the basis step and the inductive step, state the conclusion, namely that by mathematical induction, $P(n)$ is true for all integers n with $n \geq b$.

Summary

In this lesson, we have learnt

1. The Principle of Mathematical induction is used to prove theorems or assertions to know whether they are true.
2. Mathematical induction cannot be used to derive new theorems but to prove already obtained results or existing theorems.
3. Mathematical induction can be used to prove a wide variety of theorems and results including summations and inequalities

Post-Test

1. Use PMI to prove the formula for the sum of a finite number of terms of a Geometric Progression with initial term a and common ratio, r. *It's your responsibility to find out and write the formula*.
2. The harmonic numbers H_j for $j = 1, 2, 3, \ldots$ are defined by the series

$$H_j = 1 + \frac{1}{2} + \frac{1}{3} + \cdots \frac{1}{j}$$

   For instance,

$$H_5 = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} = \frac{137}{60}$$

   Use PMI to show that

   $H_{2^n} = 1 + \frac{n}{2}$, whenever n is a nonnegative integer.

3. Use mathematical induction to prove that $7^{n+2} + 8^{2n+1}$ is divisible by 57 for every nonnegative integer n.

References

1. Rosen, K. Discrete Mathematics and its Applications. 7th Edition, McGraw-Hill, 2012.

Lesson 4: Loops and Recursion I

Introduction

In this lesson, we will discuss recursive definitions of data/discrete structures such as functions, sequences and sets. A structure is recursive if is defined in terms of itself. Very similar to but more concise than loops, recursive definitions help define structures in very succinct terms that aid the understanding of how to generate new items belonging to such structures. We also discuss recursive algorithms and study them alongside their iterative (loop) versions. As we shall see, the concise nature of recursive algorithms does not necessarily translate to their efficiency. In fact, more often than not, recursive algorithms are less efficient that their iterative versions.

Objectives

At the end of this lesson, you should learn

1. Recursive definition of structures such as sequences, functions and sets.
2. To use mathematical induction to prove recursive definitions

4.1 Recursive Definitions

Sometimes it is difficult to define an object explicitly. However, it may be easy to define this object in terms of itself. This process is called recursion. We can use recursion to define sequences, functions and sets. For instance, the sequence of powers of 2 is given by $a_n = 2^n$ for $n = 0, 1, 2, \dots$ The same sequence can also be defined by specifying the first term $a_0 = 1$, and a rule for finding a term of the sequence from the previous one, i.e., $a_{n+1} = 2a_n$. When a sequence is defined in this manner, we can use induction to prove results about the sequence, specifically, *Structural Induction*.

Structural induction: We define a set recursively by specifying some initial elements in a basis step and provide a rule for constructing new elements from those already in the recursive step

Use two steps to define a function with the set of non-negative integers as its domain

- **Basis step**: specify the value for the function at zero
- **Recursive step**: give a rule for finding its value at an integer from its values at smaller integers

Such a definition is called a recursive or inductive definition.

Example 4.1: Suppose f is defined recursively by

$f(0) = 3$

$f(n+1) = 2f(n) + 3$

Find $f(1)$, $f(2)$, $f(3)$, and $f(4)$

Solution: From the recursive definition, it follows that

$f(1) = 2f(0) + 3 = 2 \times 3 + 3 = 9$

$f(2) = 2f(1) + 3 = 2 \times 9 + 3 = 21$

$f(3) = 2f(2) + 3 = 2 \times 21 + 3 = 45$

$f(4) = 2f(3) + 3 = 2 \times 45 + 3 = 93$

Example 4.2: Give an inductive definition of the factorial function $f(n) = n!$

Solution:

Note that $(n+1)! = (n+1) \cdot n!$

We can define $f(0) = 1$ and $f(n+1) = (n+1) \cdot f(n)$

To determine a value, e.g., $f(5) = 5!$, we can use the recursive function

$f(5) = 5 \cdot f(4) = 5 \cdot 4 \cdot f(3) = 5 \cdot 4 \cdot 3 \cdot f(2) = 5 \cdot 4 \cdot 3 \cdot 2 \cdot f(1) = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot f(0) = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 = 120$

4.2 Recursive Functions

Recursively defined functions are well defined such that for every positive integer, the value of the function is determined in an unambiguous way. Given any positive integer, we can use the two parts of the definition to find the value of the function at that integer and we will always obtain the same value no matter how we apply two parts of the definition.

Example 4.3: Given a recursive definition of a^n, where a is a non-zero real number and n is a non-negative integer.

Solution:

Note that $a^{n+1} = a \cdot a^n$ and $a^0 = 1$

These two equations uniquely define a^n for all nonnegative integer n.

Example 4.4: Given a recursive definition of $\sum_{k=0}^{n} a^k$

Solution: The first part of the recursive definition is $\sum_{k=0}^{0} a^k = a^0$

The second part is $\sum_{k=0}^{n+1} a_k = (\sum_{k=0}^{n} a_k) + a_{n+1}$

The Fibonacci Numbers: Fibonacci numbers $f_0, f_1, f_2,$ are defined by the equations, $f_0 = 0$, $f_1 = 1$, and $f_n = f_{n-1} + f_{n-2}$ for n = 2, 3, 4, …

By definition

$f_2 = f_1 + f_0 = 1 + 0 = 1$

$f_3 = f_2 + f_1 = 1 + 1 = 2$

$f_4 = f_3 + f_2 = 2 + 1 = 3$

$f_5 = f_4 + f_3 = 3 + 2 = 5$

$f_6 = f_5 + f_4 = 5 + 3 = 8$

Summary

In this lesson, you have learnt that

1. Sequences, functions and sets can be defined recursively
2. Recursive definitions can be proven with mathematical induction.

Post-Test

1. Show that whenever $n \geq 3$, $f_n > \alpha^{n-2}$, where $\alpha = (1 + \sqrt{5})/2$.

References

1. Rosen, K. Discrete Mathematics and its Applications. 7th Edition, McGraw-Hill, 2012.

Lesson 5: Loops and Recursion II

Introduction

In this lesson, we will discuss recursive definitions of data/discrete structures such as functions, sequences and sets. A structure is recursive if is defined in terms of itself. Very similar to but more concise than loops, recursive definitions help define structures in very succinct terms that aid the understanding of how to generate new items belonging to such structures. We also discuss recursive algorithms and study them alongside their iterative (loop) versions. As we shall see, the concise nature of recursive algorithms does not necessarily translate to their efficiency. In fact, more often than not, recursive algorithms are less efficient that their iterative versions.

Objectives

At the end of this lesson, you should learn

1. About recursive and iterative algorithms

5.1 Recursively Defined Sets and Structures

Consider the subset S of the set of integers defined by

- Basis step: $3 \in S$
- Recursive step: if $x \in S$ and $y \in S$, then $x + y \in S$

The new elements formed by this are 3 + 3 = 6, 3 + 6 = 9, 6 + 6 = 12, … Thus, S is the set of all positive multiples of 3.

Strings can be defined recursively.

The set $\sum^*$ of strings over the alphabet $\sum$ can be defined recursively by

- Basis step: $\lambda \in \sum^*$ (where λ is the empty string containing no symbols)
- Recursive step: if $w \in \sum^*$ and $x \in \sum$ then $wx \in \sum^*$

The basis step defines that the empty string belongs to the set of strings. The recursive step states new strings are produced by adding a symbol from ∑ to the end of stings in ∑*. At each application of the recursive step, strings containing one additional symbol are generated.

Example 4.1: If ∑ = {0, 1}, the strings found to be in ∑*, the set of all bit strings, are

- λ, specified to be in ∑* in the basis step
- 0 and 1 found in the 1st recursive step
- 00, 01, 10, and 11 are found in the 2nd recursive step, and so on

5.2 Recursive Algorithms

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

Example 4.2: Give a recursive algorithm for computing $n!$ where n is a non-negative integer

Solution: We can compute $n! = n \cdot (n - 1)!$ Where n is a positive integer, and that $0! = 1$ for a particular integer. We use the recursive step n times

```
procedure factorial(n: non-negative integer)
    if n=0 then
            factorial(n):=1
    else
            factorial(n):= n * factorial(n-1)
```

Example 4.3: Give a recursive algorithm for computing a^n where a is a non-zero real number and n is a non-negative integer.

Solution: We can base a recursive algorithm on the recursive definition of a^n. This definition states that $a^{n+1} = a \cdot a^n$ for $n > 0$ and the initial condition $a^0 = 1$. To find a^n, successively use the recursive step to reduce the exponent until it becomes zero.

```
procedure power(a: nonzero real number, n: non-negative integer)
    if n=0 then
```

```
            power(a,n):=1
    else
            power(a,n):=a · power(a, n-1)
```

Recursion and Iteration:

A recursive definition expresses the value of a function at a positive integer in terms of the values of the function at smaller integers. This means that we can devise a recursive algorithm to evaluate a recursively defined function at a positive integer. Instead of successively reducing the computation to the evaluation of the function at smaller integers, we can start with the value of the function at one or more integers, the base cases, and successively apply the recursive definition to find the values of the function at successive larger integers. Such a procedure is called iterative. Often iterative algorithms are more efficient than their recursive versions (unless special-purpose machines are used). Consider the recursive algorithm for the Fibonacci numbers and its iterative version.

Recursive Fibonacci algorithm

```
procedure fibonacci(n: nonzero integer)
    if n=0 then
            fibonacci(n):=0
    else if n=1 then
            fibonacci(1):=1
    else
            fibonacci(n)=fibonacci(n-1)+fibonacci(n-2)
```

Iterative Fibonacci algorithm

```
procedure iterative_fibonacci(n: nonzero integer)
      if n=0 then
            y:=0
      else
        begin
          x:=0
          y:=1
          for i:=1 to n-1
                  z:=x+y
                  x:=y
                  y:=z
         end
```

```
end
{y is the n-th Fibonacci number}
```

When we use a recursive procedure to find f_n, we first express f_n as $f_{n-1} + f_{n-2}$. Then we replace both of these Fibonacci numbers by the sum of two previous Fibonacci numbers, and so on. When f_1 or f_0 arises, it is replaced by its value. Note that at each stage of the recursion, until f_1 or f_0 is obtained, the number of Fibonacci numbers to be evaluated has doubled. For instance, when we find f_4 using this recursive algorithm, we must carry out all the computations illustrated in the tree diagram in Figure 4.1.

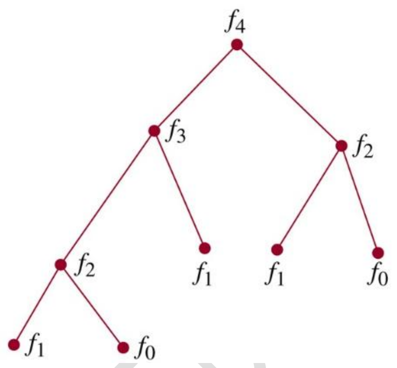


Figure 4.1: Computation of f_4 in the recursive Fibonacci algorithm

The iterative algorithm initializes x as $f_0 = 0$, and y as $f_1 = 1$. When the loop is traversed, the sum of x and y is assigned to the auxiliary variable z. Then x is assigned the value of y and y is assigned the value of the auxiliary variable z. Thus, after going through the loop the first time, it follows x equals f_1 and y equals $f_0 + f_1 = f_2$

Next, $f_1 + f_2 = f_3, \ldots$

After going through the algorithm n-1 times, x equals f_{n-1} and y equals f_n. Only n-1 additions have been used to find f_n.

Summary

In this lesson, you have learnt that

1. How to write recursive algorithms

2. Iterative algorithms are often more effective than recursive algorithms

Post-Test

1. Write an iterative algorithm for computing the factorial of a non-negative integer *n*.

References

Rosen, K. Discrete Mathematics and its Applications. 7th Edition, McGraw-Hill, 2012.

Lesson 6: Program Correctness

Introduction

It is not enough to write programs, there is need to verify their correctness. A very obvious way to do this might be to test the program with sample inputs, but this can never suffice as the set of possible inputs is usually infinite and cannot all be accounted for. We therefore to resort to mathematical proofs to verify the correctness of a program.

This lesson will introduce you to the assertions, inferences and invariants used to prove program segments and with worked examples, you should be able to learn the steps of proving program correctness. While all the examples in this lesson proved the programs correct, it must also be noted that the same procedure is followed in an attempt to verify a wrong program in which case, the proof fails.

Objectives

At the end of this lesson, you should

1. Know how to prove programs correct
2. Understand the rules of inference for proving conditional statements
3. Understand how to derive loop invariants to prove loops

6.1 Introduction

Suppose we have written a program to solve a problem, after debugging, how can we be sure that the program always gets the correct answer? We need a proof to show that the program always gives the correct answer.

Program verification: uses the rules of inference and proof techniques. It is one form of formal verification and can be carried out using a computer. However, only limited progress has been made. Some mathematicians and theoretical computer scientists argue that it will never be realistic to mechanize the proof of correctness of complex programs.

A program is said to be correct if it produces the correct output for every possible input.

A proof that a program is correct consists of two parts

- *Partial correctness: Correct answer is obtained if the program terminates*
- *Shows the program always terminates*.

To specify a program produces the correct output, we must state

- *Initial assertion: the properties that the input values must have*
- *Final assertion: the properties that output of the program should have, if the program did what was intended*

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 output values of S.

The notation $p\{S\}q$, known as the *Hoare Triple*, indicates that the *program*, or *program segment*, S is partially correct with respect to the initial assertion p and the final assertion q.

Example 6.1: Show that the program segment

```
y := 2
z := x + y
```

is correct with respect to the initial assertion p: $x = 1$ and the final assertion q: $z = 3$

Solution:

Suppose that p is true, so that $x = 1$. Then y is assigned the value 2, and z is assigned the sum of the values of x and y, which is 3.

Hence, S is correct with respect to the initial assertion p and the final assertion q. Thus $p\{S\}q$ is true.

6.2 Rules of Inference

Suppose the program S is split into subprograms S_1 and S_2, denote it by $S = S_1;S_2$ that is the program S is made up of S_1 followed by S_2. Suppose the correctness of S_1 w.r.t. the initial assertion p and final assertion q, the correctness of S_2 w.r.t. the initial assertion q and the final assertion r, have been established. It follows that if p is true and S_1 is executed and terminates, then q is true; and if q is true and S_2 executes and terminates, then r is true. Thus, if p is true and $S = S_1;S_2$ is executed and terminates, then r is true. This rule of inference, called the ***composition rule*** can be stated as

$$\begin{array}{c} p\{S_1\}q \\ q\{S_2\}r \\ \hline \therefore\ p\{S_1;S_2\}r \end{array}$$

6.2.1 Conditional Statements

Consider a program statement

```
if condition then
        S
```

To verify this segment is correct w.r.t. initial assertion p and final assertion q, two things must be done

- First, when p is true and condition is true, then q is true after S terminates
- Second, when p is true and condition is false, then q is true (as S is not executed)

This therefore leads to the following rule of inference:

$$\begin{array}{c} (p \wedge condition)\{S\}q \\ (p \wedge \neg condition) \rightarrow q \\ \hline \therefore\ p\{\text{if } condition \text{ then } S\}q \end{array}$$

Example 6.2: Verify the program segment

```
  if x < 0 then
```

```
            abs:=-x
    else
            abs:=x
```

is correct w.r.t. the initial assertion T and the final assertion $abs = |x|$

Solution:

Two things must be demonstrated. First, it must be shown that if the initial condition is true and $x < 0$, then $abs = |x|$. This is correct as when $x < 0$, the assignment statement $abs := -x$ sets $abs = -x$, which is $|x|$ by definition when $x < 0$.

Second, it must be shown that if the initial assertion is true and $x < 0$ is false, so that $x \geq 0$, then $abs = |x|$. This is correct as in this case the program uses the assignment $abs := x$, and x is $|x|$ by definition when $x \geq 0$, so that $abs := x$.

Hence, using the rule of inference for program segments of this type, this segment is correct w.r.t. the given initial and final assertions

6.2.2 Loop Invariants

We will now see how to prove the correctness of while loops. Consider the following program segment.

```
    while condition
            S
```

Note that S is repeatedly executed until *condition* becomes false. An assertion that remains true each time S is executed must be chosen. Such an assertion is called a *loop invariant*. That is, p is a loop invariant if $(p \wedge condition)\{S\}p$ is true.

Suppose that p is a loop invariant, it follows that is p is true before the program segment is executed, p and ¬condition are true after termination, if it occurs. This gives the rule of inference

$$\frac{(p \wedge condition)\{S\}p}{\therefore p\{\text{while } condition\ S\}(\neg condition \wedge p)}$$

Example 6.3: Use a loop invariant to verify that the following program segment terminates with *factorial* = *n*! when *n* is a positive integer

```
    i:=1
    factorial:=1
    while i<n
    begin
          i:=i+1
          factorial:= factorial * i
     end
```

Solution: Let p be the assertion "$factorial = i!$ and $i \le n$". We first prove that p is a loop invariant

Suppose that at the beginning of one execution of the while loop, p is true and the condition holds, i.e., assume that $factorial = i!$ and $i < n$

The new values i_{new} and $factorial_{new}$ of i and $factorial$ are

$i_{new} = i + 1 \le n$ and

$factorial_{new} = factorial \cdot (i + 1) = (i + 1)! = i_{new}!$

Because $i < n$, we also have $i_{new} = i + 1 \le n$

Thus p is true at the end of the execution of the loop. This shows that p is a loop invariant

Just before entering the loop, $i = 1 \le n$ and $factorial = 1 = 1! = i!$ both hold, so p is true

As p is a loop invariant, the rule of inference implies that if the while loop terminates, it terminates with p true and with $i < n$ false.

In this case, at the end, *factorial* = *i*! and $i \leq n$ are true, but $i < n$ is false; in other words, $i = n$ and *factorial* = *i*! = *n*!, as desired

Finally, we need to check that the while loop actually terminates

At the beginning of the program, *i* is assigned the value 1, so that after *n*-1 traversals of the loop, the new value of *i* will be *n*, and the loop terminates at that point.

Summary

In this lesson, you have learnt

1. That programs must be verified for correctness
2. That there are two parts to program to program verification; the *partial correctness* and the proof that the *program always terminates*
3. That to check that a program always produces correct output, two assertions are used – the *initial assertion* (which specifies the properties of the expected input) and the *final assertion* (which checks the properties of the expected output). These assertions together with the program segment to be proven form the *Hoare triple*.
4. About rules of inference for Conditional statements and loop invariants

Post-Test

1. Verify the correctness of the program below:

$procedure(m, n : \text{integers})$

$$S_1 = \left\{ \begin{array}{l} \text{if } \ n < 0 \text{ then } \ a := -n \\ \text{else } \ a := n \end{array} \right\}$$

$$S_2 = \left\{ \begin{array}{l} k := 0 \\ x := 0 \end{array} \right\}$$

$$S_3 = \left\{ \begin{array}{l} \text{while } k < a \\ \text{begin} \\ x := x + m \\ k := k + 1 \\ \text{end} \end{array} \right\}$$

$$S_4 = \left\{ \begin{array}{l} \text{if } \ n < 0 \text{ then } \ product := -x \\ \text{else } \ product := x \end{array} \right\}$$

References

1. Rosen, K. Discrete Mathematics and its Applications. 7th Edition, McGraw-Hill, 2012.

Lesson 7: Discrete Probability

Introduction

The history of Discrete Probability dates back to the 15th – 17th century with the contributions of mathematicians such as Girolamo Cardano, Blaise Pascal and Laplace. Probability is generally concerned with the chances or possibilities associate with a given task.

In this lesson, we will study the probabilities of tasks with finitely many, equally-likely outcomes and learn how they can be combined with set theory operations.

Objectives

At the end of this lesson, you should

1. Know how to find the probability of an event
2. Know how to derived the probabilities of events from other events

7.1 Finite Probability

An **experiment** is a procedure that yields one of a given set of possible outcomes. The **sample space** of the experiment is the set of possible outcomes. An **event** is a subset of the sample space. Laplace' s definition of the probability of an event with finitely many possible outcomes is stated as follows:

If S is a finite nonempty sample space of equally likely outcomes, and E is an event, that

is, a subset of S, then the probability of E is $p(E) = \frac{|E|}{|S|}$

Note that the value of $p(E)$ is always between 0 and 1 i.e. $0 \le p(E) \le 1$.

Example 7.1: A bag contains four blue balls and five red balls. What is the probability that a ball chosen at random from the bag is blue?

Solution: From the question, we know that $|S|$ = 9 (the number of balls in the bag; 4 blue balls + 5 red balls) and $|E|$ = 4 (the number of blue balls in the bag, since the blue balls are under consideration now).

Using the formula above, $p(E) = 4 / 9$

Example 7.2: What is the probability that when two dice are rolled, the sum of the numbers on the two dice is 7?

Solution: There are a total of 36 equally likely possible outcomes when two dice are rolled.

(The product rule can be used to see this; because each die has six possible outcomes, the total number of outcomes when two dice are rolled is 62 = 36.)

There are 6 successful outcomes that can give a sum of 7, namely, (1, 6), (2, 5), (3, 4), (4, 3), (5, 2), and (6, 1), where the values of the first and second dice are represented by an ordered pair. Hence, the probability that a 7 comes up when two fair dice are rolled is 6/36 = 1/6.

Example 7.3: What is the probability that the numbers 11, 4, 17, 39, and 23 are drawn in that order from a bin containing 50 balls labelled with the numbers 1, 2, . . . , 50 if

(a) the ball selected is not returned to the bin before the next ball is selected and

(b) the ball selected is returned to the bin before the next ball is selected?

Solution:

(a) By the product rule, there are $50 \cdot 49 \cdot 48 \cdot 47 \cdot 46$ = 254,251,200 ways to select the balls because each time a ball is drawn there is one fewer ball to choose from. Consequently, the probability that 11, 4, 17, 39, and 23 are drawn in that order is 1/254,251,200. This is an example of sampling without replacement.

(b) By the product rule, there are 505 = 312,500,000 ways to select the balls because there are 50 possible balls to choose from each time a ball is drawn. Consequently, the probability that 11, 4, 17, 39, and 23 are drawn in that order is 1/312,500,000. This is an example of sampling with replacement.

7.2 Probabilities of Complements and Unions of Events

We can use counting techniques to find the probability of events derived from other events. We can find the probabilities of the complement and union of events

Theorem 1: Let E be an event in a sample space S. The probability of the event $\overline{E} = S - E$, the complementary event of E, is given by $p(\overline{E}) = 1 - p(E)$.

Theorem 2: Let $E1$ and $E2$ be events in the sample space S. Then

$p(E1 \cup E2) = p(E1) + p(E2) - p(E1 \cap E2)$.

Example 7.4: A sequence of 10 bits is randomly generated. What is the probability that at least one of these bits is 0?

Solution: Let E be the event that at least one of the 10 bits is 0. Then $\overline{E}$ is the event that all the bits are 1s. Because the sample space S is the set of all bit strings of length 10, it follows that

$$p(E) = 1 - p(\overline{E}) = 1 - \frac{|\overline{E}|}{|S|} = 1 - \frac{1}{2^{10}}$$
$$= 1 - \frac{1}{1024} = \frac{1023}{1024}.$$

Hence, the probability that the bit string will contain at least one 0 bit is 1023/1024. It is quite difficult to find this probability directly without using Theorem 1 above.

Example 7.5: What is the probability that a positive integer selected at random from the set of positive integers not exceeding 100 is divisible by either 2 or 5?

Solution: Let E_1 be the event that the integer selected at random is divisible by 2, and let E_2 be the event that it is divisible by 5. Then $E_1 \cup E_2$ is the event that it is divisible by either 2 or 5.

Also, $E_1 \cap E_2$ is the event that it is divisible by both 2 and 5, or equivalently, that it is divisible by 10. Because $|E_1| = 50$, $|E_2| = 20$, and $|E_1 \cap E_2| = 10$, it follows that

$$\begin{aligned} p(E_1 \cup E_2) &= p(E_1) + p(E_2) - p(E_1 \cap E_2) \\ &= \frac{50}{100} + \frac{20}{100} - \frac{10}{100} = \frac{3}{5}. \end{aligned}$$

Summary

In this lesson, you have learnt that

1. An **experiment** is a procedure that yields one of a given set of possible outcomes.
2. The **sample space** of the experiment is the set of possible outcomes.
3. An **event** is a subset of the sample space
4. We can derive the probabilities of events as a complement of the probabilities of other events or as the union of the probabilities of two events.

Post-Test

1. There are many lotteries now that award enormous prizes to people who correctly choose a set of six numbers out of the first *n* positive integers, where *n* is usually between 30 and 60. What is the probability that a person picks the correct six numbers out of 40?

References

1. Rosen, K. Discrete Mathematics and its Applications. 7th Edition, McGraw-Hill, 2012.

Lesson 8: Randomized Algorithms

Introduction

Objectives

At the end of this lesson, you should

1. Understand randomized algorithms
2. Be introduced to two methods of randomizing algorithm data inputs

8.1 Randomized Algorithms

Suppose we need to hire a new employee and have decided to employ the following pseudocode in figure 8.1. The hire-assistant pseudocode successively interviews candidates and determines a 'best' candidate relative to the previously interviewed ones.

```
HIRE-ASSISTANT(n)
1  best = 0            // candidate 0 is a least-qualified dummy candidate
2  for i = 1 to n
3      interview candidate i
4      if candidate i is better than candidate best
5          best = i
6          hire candidate i
```

Figure 8.1: Pseudocode for hiring a new employee

For certain problems such as the hiring problem, in which it is helpful to assume that all permutations of the input are equally likely, a probabilistic analysis can guide the development of a randomized algorithm. Instead of assuming a distribution of inputs, we impose a distribution. In particular, before running the algorithm, we randomly permute the candidates in order to enforce the property that every permutation is equally likely. Although we have modified the algorithm, we still expect to hire a new office assistant approximately *ln n* times. But now we expect this to be the case for any input, rather than for inputs drawn from a particular distribution.

Consider, on the other hand, the randomized algorithm that first permutes the candidates and then determines the best candidate. In this case, we randomize in the algorithm, not in the input distribution. Given a particular input, say A_3 above, we cannot say how many times the maximum is updated, because this quantity differs with each run of the algorithm. The first time we run the algorithm on A_3, it may produce the permutation A_1 and perform 10 updates; but the second time we run the algorithm, we may produce the permutation A_2 and perform only one update. The third time we run it, we may perform some other number of updates. Each time we run the algorithm, the execution depends on the random choices made and is likely to differ from the previous execution of the algorithm. For this algorithm and many other randomized algorithms, no particular input elicits its worst-case behaviour. The randomized algorithm performs badly only if the random-number generator produces an "unlucky" permutation. For the hiring problem, the only change needed in the code is to randomly permute the array as shown in figure 8.2. With this simple change, we have created a randomized algorithm whose performance matches that obtained by assuming that the candidates were presented in a random order.

```
RANDOMIZED-HIRE-ASSISTANT(n)
randomly permute the list of candidates
best = 0            // candidate 0 is a least-qualified dummy candidate
for i = 1 to n
    interview candidate i
    if candidate i is better than candidate best
        best = i
        hire candidate i
```

Figure 8.2: A Randomized version of the Hiring pseudocode

Generally speaking, an algorithm is called ***randomized*** if its behaviour is determined not only by its input but also by values produced by a random-number generator.

8.2 Random Permutation of Arrays

Many randomized algorithms randomize the input by permuting the given input array and we shall discuss two methods of doing this. We assume that we are given an array *A* which, contains the elements 1 through *n*. Our goal is to produce a random permutation of the array.

One common method is to assign each element *A*[*i*] of the array a random priority *P*[*i*], and then sort the elements of *A* according to these priorities. For example, if our initial array is *A* = ‹1, 2, 3, 4› and we choose random priorities *P* = ‹36, 3, 62, 19›, we would produce an array *B* = ‹2, 4, 1, 3› since the second priority is the smallest, followed by the fourth, then the first, and finally the third. We call this procedure PERMUTE-BY-SORTING and write the pseudocode in figure 8.3.

```
PERMUTE-BY-SORTING($A$)
1  $n = A.length$
2  let $P[1..n]$ be a new array
3  for $i = 1$ to $n$
4      $P[i] = \text{RANDOM}(1, n^3)$
5  sort $A$, using $P$ as sort keys
```

Figure 8.3: Permute-by-Sorting Pseudocode

Line 4 chooses a random number between 1 and n^3; a range of 1 to n^3 is used to make it likely that all the priorities in *P* are unique. Sorting is done on line 5 and after sorting, if *P*[*i*] is the *j*th smallest priority, then *A*[*i*] lies in position *j* of the output. In this manner we obtain a permutation.

A better method for generating a random permutation is to permute the given array in place. The procedure RANDOMIZE-IN-PLACE (figure 8.4) does so in *O*(*n*) time. In its *i*th iteration, it chooses the element $A[i]$ randomly from among elements $A[i]$ through $A[n]$. Subsequent to the i^{th} iteration, $A[i]$ is never altered.

```
RANDOMIZE-IN-PLACE(A)
1  n = A.length
2  for i = 1 to n
3      swap A[i] with A[RANDOM(i, n)]
```

Figure 8.4: Randomize-In-Place Pseudocode

Summary

In this lesson, you have learnt

1. That randomized algorithms is one whose behaviour is determined not only by its input but also by values produced by a random-number generator
2. Two methods of randomizing data inputs, here referred to as *permute-by-sorting* and *randomize-in-place*.

Post-Test

1. Use loop invariant to show that the Randomize-In-Place algorithm produces a uniform random permutation.

References

1. Cormen T. H., Leiserson C. E., Rivest R. L. and Stein C. Introduction to Algorithms. 3rd Edition, MIT Press, 2009.

Lesson 9: Lists

Introduction

Objectives

At the end of this lesson, you should

1. Know what a list is
2. Understand the two special types of lists – queues and stacks – and the operations defined on them.
3. Understand linked lists and the operations of searching, insertion into and deleting from them.

9.1 Introduction

A list is a finite sequence of data items, i.e., a collection of data items arranged in a certain linear order. The basic operations performed on this data structure are searching for, inserting, and deleting an element.

Two special types of lists, *stacks* and *queues*, are particularly important. A *stack* is a list in which insertions and deletions can be done only at the end. This end is called the top because a stack is usually visualized not horizontally but vertically—akin to a stack of plates whose "operations" it mimics very closely. As a result, elements are added to (pushed onto) a stack and deleted from (popped off) it, the structure operates in a "last-in–first-out" (LIFO) fashion—

exactly like a stack of plates if we can add or remove a plate only from the top. Stacks have a multitude of applications; in particular, they are indispensable for implementing recursive algorithms. A *queue*, on the other hand, is a list from which elements are deleted from one end of the structure, called the front (this operation is called *dequeue*), and new elements are added to the other end, called the rear (this operation is called *enqueue*). Consequently, a queue operates in a “first-in–first-out” (FIFO) fashion—akin to a queue of customers served by a single teller in a bank. Queues also have many important applications, including several algorithms for graph problems.

Many important applications require selection of an item of the highest priority among a dynamically changing set of candidates. A data structure that seeks to satisfy the needs of such applications is called a priority queue. A priority queue is a collection of data items from a totally ordered universe (most often, integer or real numbers). The principal operations on a priority queue are finding its largest element, deleting its largest element, and adding a new element. A priority queue must be implemented so that the last two operations yield another priority queue. Straightforward implementations of this data structure can be based on either an array or a sorted array, but neither of these options yields the most efficient solution possible. A better implementation of a priority queue is based on an ingenious data structure called the heap.

A Special implementation of lists is called linked lists. A linked list is a data structure in which the objects are arranged in a linear order. Unlike an array, however, in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object. A linked list may be either singly-linked or doubly-linked, it may be sorted or not, and it may be circular or not. If a linked list is singly-linked, we omit the *previous* pointer in each element. If a linked list is sorted, the linear order of the list corresponds to the linear order of keys stored in elements of the list; the minimum element is then the *head* of the list, and the maximum element is the *tail*. If the list is unsorted, the elements can appear in any order. In a circular list, the *previous* pointer of the *head* of the list points to the *tail*, and the *next* pointer of the tail of the list points to the head. We can think of a circular list as a ring of elements.

9.2 List Operations

Here, we shall write and study pseudocodes for searching, inserting to and deleting from lists.

Searching a List

The procedure LIST-SEARCH (L, k) (figure 9.2) finds the first element with key k in list L by a simple linear search, returning a pointer to this element. If no object with key k appears in the list, then the procedure returns NIL. For the linked list in Figure 9.1(a), the call LIST-SEARCH (L, 4) returns a pointer to the third element, and the call LIST-SEARCH (L, 7) returns NIL.

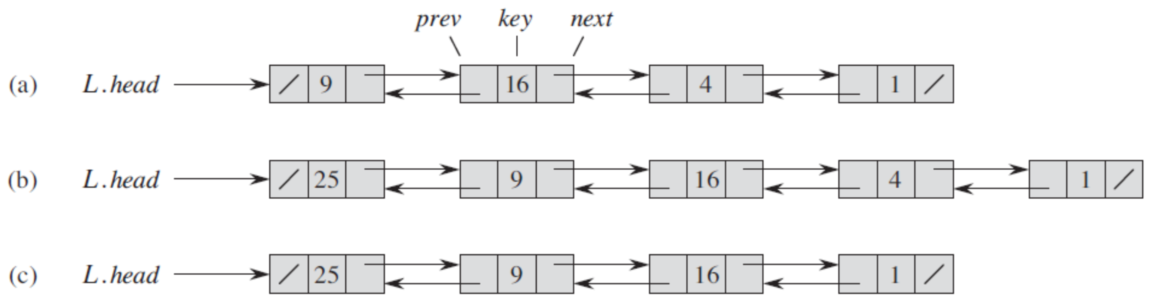


Figure 9.1: Graphical illustration of a list

```
LIST-SEARCH($L, k$)
1  $x = L.head$
2  while $x \neq$ NIL and $x.key \neq k$
3      $x = x.next$
4  return $x$
```

Figure 9.2: LIST-SEARCH algorithm

Inserting into a List

Given an element x whose key attribute has already been set, the LIST-INSERT procedure (figure 9.3) "splices" x onto the front of the linked list, as shown in Figure 9.1(b).

```
LIST-INSERT(L, x)
1  $x.next = L.head$
2  if $L.head \neq$ NIL
3      $L.head.prev = x$
4  $L.head = x$
5  $x.prev =$ NIL
```

Figure 9.3: LIST-INSERT algorithm

Deleting from a List

The procedure LIST-DELETE (figure 9.4) removes an element *x* from a linked list *L*. It must be given a pointer to *x*, and it then "splices" *x* out of the list by updating pointers. If we wish to delete an element with a given key, we must first call LIST-SEARCH to retrieve a pointer to the element. Figure 9.1(c) illustrates the deletion of an item (4) from the list.

```
LIST-DELETE(L, x)
1  if $x.prev \neq$ NIL
2      $x.prev.next = x.next$
3  else $L.head = x.next$
4  if $x.next \neq$ NIL
5      $x.next.prev = x.prev$
```

Figure 9.4: LIST-DELETE algorithm

Summary

In this lesson, you have learnt that

1. A list is a finite sequence of data items
2. Stacks and Queues are special types of lists.
3. The stack operates in a LIFO fashion with its insertion and deletion operations referred to as *push* and *pop* respectively.
4. The queue operates in a FIFO fashion with its insertion and deletion operations referred to as *enqueue* and *dequeue* respectively.

Post-Test

1. Write a pseudocode to perform the following operations
    a. Push element *x* onto stack *S*
    b. Pop stack *S*
    c. Enqueue queue *Q*
    d. Dequeue queue *Q*

References

1. Cormen T. H., Leiserson C. E., Rivest R. L. and Stein C. Introduction to Algorithms. 3rd Edition, MIT Press, 2009.
2. Levitin A. Introduction to the Design and Analysis of Algorithms. 3rd Edition, Pearson Education, 2012.

Lesson 10: Graphs

Introduction

Objectives

At the end of this lesson, you should learn

1 The formal definition of a graph

2 Graph types

3 The different Graph representations

10.1 Overview

As we mentioned in the previous section, a graph is informally thought of as a collection of points in the plane called "vertices" or "nodes," some of them connected by line segments called "edges" or "arcs." Formally, a ***graph*** *G* = ‹*V*,*E*› is defined by a pair of two sets: a finite nonempty set *V* of items called ***vertices*** and a set *E* of pairs of these items called ***edges***. If these pairs of vertices are unordered, i.e., a pair of vertices *(u, v)* is the same as the pair *(v, u)*, we say that the vertices *u* and *v* are ***adjacent*** to each other and that they are connected by the ***undirected edge*** *(u, v)*. We call the vertices *u* and *v* ***endpoints*** of the edge *(u, v)* and say that *u* and *v* are ***incident*** to this edge; we also say that the edge *(u, v)* is incident to its endpoints *u* and *v*. A graph *G* is called ***undirected*** if every edge in it is undirected.

If a pair of vertices *(u, v)* is not the same as the pair *(v, u)*, we say that the edge *(u, v)* is ***directed*** from the vertex *u*, called the edge's ***tail***, to the vertex *v*, called the edge's ***head***. We also say that the edge *(u, v)* leaves *u* and enters *v*. A graph whose every edge is directed is called ***directed***. Directed graphs are also called ***digraphs***. It is normally convenient to label vertices of a graph or a digraph with letters, integer numbers, or, if an application calls for it, character strings (Figure 10.1). The graph depicted in Figure 10.1a has six vertices and seven undirected edges:

$V = \{a, b, c, d, e, f\}$, $E = \{(a, c), (a, d), (b, c), (b, f), (c, e), (d, e), (e, f)\}$.

The digraph depicted in Figure 10.1b has six vertices and eight directed edges:

$V = \{a, b, c, d, e, f\}$, $E = \{(a, c), (b, c), (b, f), (c, e), (d, a), (d, e), (e, c), (e, f)\}$.

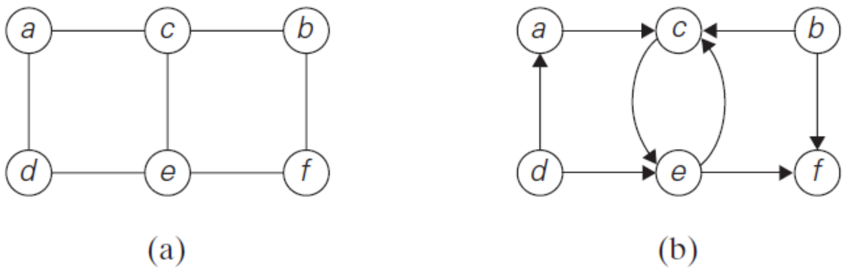


Figure 10.1: Examples of Undirected and Directed Graphs

Our definition of a graph does not forbid ***loops***, or edges connecting vertices to themselves. Unless explicitly stated otherwise, we will consider graphs without loops. Since our definition disallows multiple edges between the same vertices of an undirected graph, we have the following inequality for the number of edges $|E|$ possible in an undirected graph with $|V|$ vertices and no loops:

$$0 \le |E| \le |V|(|V| - 1)/2.$$

(We get the largest number of edges in a graph if there is an edge connecting each of its $|V|$ vertices with all $|V| - 1$ other vertices. We have to divide product $|V|(|V| - 1)$ by 2, however, because it includes every edge twice.) A graph with every pair of its vertices connected by an edge is called ***complete***. A standard notation for the complete graph with $|V|$ vertices is $K_{|V|}$. A graph with relatively few possible edges missing is called ***dense***; a graph with few edges relative to the number of its vertices is called ***sparse***. Whether we are dealing with a dense or sparse graph may influence how we choose to represent the graph and, consequently, the running time of an algorithm being designed or used.

10.2 Representation of Graphs

Graphs for computer algorithms are usually represented in one of two ways: the adjacency matrix and adjacency lists. The ***adjacency matrix*** of a graph with n vertices is an $n \times n$ boolean matrix with one row and one column for each of the graph's vertices, in which the element in the ith row and the jth column is equal to 1 if there is an edge from the ith vertex to the jth vertex, and equal to 0 if there is no such edge. For example, the adjacency matrix for the

graph of Figure 10.1a is given in Figure 10.2a. Note that the adjacency matrix of an undirected graph is always symmetric, i.e., $A[i, j] = A[j, i]$ for every $0 \le i, j \le n - 1$ (why?).

The ***adjacency lists*** of a graph or a digraph is a collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list's vertex (i.e., all the vertices connected to it by an edge). Usually, such lists start with a header identifying a vertex for which the list is compiled. For example, Figure 10.2b represents the graph in Figure 10.1a via its adjacency lists. To put it another way, adjacency lists indicate columns of the adjacency matrix that, for a given vertex, contain 1's.

If a graph is sparse, the adjacency list representation may use less space than the corresponding adjacency matrix despite the extra storage consumed by pointers of the linked lists; the situation is exactly opposite for dense graphs. In general, which of the two representations is more convenient depends on the nature of the problem, on the algorithm used for solving it, and, possibly, on the type of input graph (sparse or dense).

$$
\begin{array}{c|cccccc|}
 & a & b & c & d & e & f \\
a & 0 & 0 & 1 & 1 & 0 & 0 \\
b & 0 & 0 & 1 & 0 & 0 & 1 \\
c & 1 & 1 & 0 & 0 & 1 & 0 \\
d & 1 & 0 & 0 & 0 & 1 & 0 \\
e & 0 & 0 & 1 & 1 & 0 & 1 \\
f & 0 & 1 & 0 & 0 & 1 & 0
\end{array}
$$

(a)

$$\begin{tikzcd}
\boxed{a} \arrow[r] & c \arrow[r] & d & \\
\boxed{b} \arrow[r] & c \arrow[r] & f & \\
\boxed{c} \arrow[r] & a \arrow[r] & b \arrow[r] & e \\
\boxed{d} \arrow[r] & a \arrow[r] & e & \\
\boxed{e} \arrow[r] & c \arrow[r] & d \arrow[r] & f \\
\boxed{f} \arrow[r] & b \arrow[r] & e &
\end{tikzcd}$$

(b)

Figure 10.2: Adjacency matrix and adjacency list respectively for the graph in figure 10.1a

Summary

In this lesson, you have learnt

1. A graph is a 2-tuple defined by a pair of two sets – vertices, *V* and edges *E*
2. A graph could be directed or undirected based on the importance or unimportance of the ordering of its vertices pairs, it could also be complete, dense or sparse based on its connectedness.
3. Graphs can be represented in computers either by an adjacency matrix or an adjacency list.

Post-Test

1. A graph contains 100 vertices and each vertex is connected to 99 vertices. How many edges are in the graph?
2. Find out and mention three application areas of Graph theory.

References

1. Cormen T. H., Leiserson C. E., Rivest R. L. and Stein C. Introduction to Algorithms. 3rd Edition, MIT Press, 2009.
2. Levitin A. Introduction to the Design and Analysis of Algorithms. 3rd Edition, Pearson Education, 2012.

3. Rosen, K. Discrete Mathematics and its Applications. 7th Edition, McGraw-Hill, 2012.

Lesson 11: Cryptography

Introduction

Objectives

At the end of this lesson, you should learn

1. What cryptography is about
2. About classical ciphers ad how to use them for encryption and decryption
3. About cryptanalysis and how to perform it

11.1 Introduction

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.

11.2 Classical Ciphers

One of the earliest known uses of cryptography was by Julius Caesar. He made messages secret by shifting each letter three letters forward in the alphabet (sending the last three letters of the alphabet to the first three). For instance, using this scheme the letter B is sent to E and the letter X is sent to A. This is an example of **encryption**, that is, the process of making a message secret.

To express Caesar's encryption process mathematically, first replace each letter by an element of $\mathbf{Z}_{26}$, that is, an integer from 0 to 25 equal to one less than its position in the alphabet. For example, replace A by 0, K by 10, and Z by 25. Caesar's encryption method can be represented by the function f that assigns to the nonnegative integer p, $p \leq 25$, the integer $f(p)$ in the set $\{0, 1, 2, \ldots, 25\}$ with

$$f(p) = (p + 3) \ \mathbf{mod} \ 26.$$

In the encrypted version of the message, the letter represented by p is replaced with the letter represented by $(p + 3) \ \mathbf{mod} \ 26$.

Example 11.1: What is the secret message produced from the message "MEET YOU IN THE PARK" using the Caesar cipher?

Solution: First replace the letters in the message with numbers. This produces

12 4 4 19 24 14 20 8 13 19 7 4 15 0 17 10.

Now replace each of these numbers p by $f(p) = (p + 3) \ \mathbf{mod} \ 26$. This gives

15 7 7 22 1 17 23 11 16 22 10 7 18 3 20 13.

Translating this back to letters produces the encrypted message

"PHHW BRX LQ WKH SDUN."

To recover the original message from a secret message encrypted by the Caesar cipher, the function f^{-1}, the inverse of f, is used. Note that the function f^{-1} sends an integer p from

$\mathbf{Z}_{26}$, to $f^{-1}(p) = (p - 3) \textbf{ mod } 26$. In other words, to find the original message, each letter is

shifted back three letters in the alphabet, with the first three letters sent to the last three letters

of the alphabet. The process of determining the original message from the encrypted message

is called **decryption**.

There are various ways to generalize the Caesar cipher. For example, instead of shifting the

numerical equivalent of each letter by 3, we can shift the numerical equivalent of each letter by k, so that

$$f(p) = (p + k) \textbf{ mod } 26.$$

Such a cipher is called a *shift cipher*. Note that decryption can be carried out using

$$f^{-1}(p) = (p - k) \textbf{ mod } 26.$$

Here the integer k is called a **key**. We illustrate the use of a shift cipher in the next two examples.

Example 11.2: Encrypt the plaintext message "STOP GLOBALWARMING" using the shift cipher with shift $k = 11$.

Solution: To encrypt the message "STOP GLOBAL WARMING" we first translate each letter to the corresponding element of $\mathbf{Z}_{26}$. This produces the string

18 19 14 15 6 11 14 1 0 11 22 0 17 12 8 13 6.

We now apply the shift $f(p) = (p + 11) \textbf{ mod } 26$ to each number in this string. We obtain

3 4 25 0 17 22 25 12 11 22 7 11 2 23 19 24 17.

Translating this last string back to letters, we obtain the ciphertext

"DEZA RWZMLW HLCXTYR."

Example 11.3: Decrypt the ciphertext message "LEWLYPLUJL PZ H NYLHA ALHJOLY" that was encrypted with the shift cipher with shift $k = 7$.

Solution: To decrypt the ciphertext "LEWLYPLUJL PZ H NYLHA ALHJOLY" we first

translate the letters back to elements of $\mathbf{Z}_{26}$. We obtain

11 4 22 11 24 15 11 20 9 11 15 25 7 13 24 11 7 0 0 11 7 9 14 11 24.

Next, we shift each of these numbers by $-k = -7$ modulo 26 to obtain

4 23 15 4 17 8 4 13 2 4 8 18 0 6 17 4 0 19 19 4 0 2 7 4 17.

Finally, we translate these numbers back to letters to obtain the plaintext. We obtain

"EXPERIENCE IS A GREAT TEACHER."

We can generalize shift ciphers further to slightly enhance security by using a function of the form

$$f(p) = (ap + b)\ \mathbf{mod}\ 26,$$

where a and b are integers, chosen so that f is a bijection. (The function $f(p) = (ap + b)\ \mathbf{mod}\ 26$ is a bijection if and only if $\gcd(a, 26) = 1$.) Such a mapping is called an *affine transformation*, and the resulting cipher is called an *affine cipher*.

Example 11.4: What letter replaces the letter K when the function $f(p) = (7p + 3)\ \mathbf{mod}\ 26$ is used for encryption?

Solution: First, note that 10 represents K. Then, using the encryption function specified, it follows that $f(10) = (7 \cdot 10 + 3)\ \mathbf{mod}\ 26 = 21$. Because 21 represents V, K is replaced by V in the encrypted message.

We will now show how to decrypt messages encrypted using an affine cipher. Suppose that $c = (ap + b)\ \mathbf{mod}\ 26$ with $\gcd(a, 26) = 1$. To decrypt we need to show how to express p in terms of c. To do this, we apply the encrypting congruence $c \equiv ap + b \pmod{26}$, and solve it for p. To do this, we first subtract b from both sides, to obtain $c - b \equiv ap \pmod{26}$. Because $\gcd(a, 26) =$

1, we know that there is an inverse $\bar{a}$ of a modulo 26. Multiplying both sides of the last equation by $\bar{a}$ gives us $\bar{a}(c - b) \equiv \bar{a}ap \pmod{26}$. Because $\bar{a}a \equiv 1 \pmod{26}$, this tells us that $p \equiv \bar{a}(c - b) \pmod{26}$. This determines p because p belongs to $\mathbf{Z}_{26}$.

11.3 Cryptanalysis

The process of recovering plaintext from ciphertext without knowledge of both the encryption method and the key is known as **cryptanalysis** or **breaking codes**. In general, cryptanalysis is a difficult process, especially when the encryption method is unknown. We will not discuss cryptanalysis in general, but we will explain how to break messages that were encrypted using a shift cipher.

If we know that a ciphertext message was produced by enciphering a message using a shift cipher, we can try to recover the message by shifting all characters of the ciphertext by each of the 26 possible shifts (including a shift of zero characters). One of these is guaranteed to be the plaintext. However, we can use a more intelligent approach, which we can build upon to cryptanalyze ciphertext resulting from other ciphers. The main tool for cryptanalyzing ciphertext encrypted using a shift cipher is the count of the frequency of letters in the ciphertext. The nine most common letters in English text and their approximate relative frequencies are E (13%,) T (9%), A (8%), O (8%), I (7%,) N (7%,) S (7%,) H (6%), and R (6%). To cryptanalyze ciphertext that we know was produced using a shift cipher, we first find the relative frequencies of letters in the ciphertext. We list the most common letters in the ciphertext in frequency order; we hypothesize that the most common letter in the ciphertext is produced by encrypting E. Then, we determine the value of the shift under this hypothesis, say k. If the message produced by shifting the ciphertext by $-k$ makes sense, we presume that our hypothesis is correct and that we have the correct value of k. If it does not make sense, we next consider the hypothesis that the most common letter in the ciphertext is produced by encrypting T, the second most common letter in English; we find k under this hypothesis, shift the letters of the message by $-k$, and see whether the resulting message makes sense. If it does not, we continue the process working our way through the letters from most common to least common.

Example 11.5: Suppose that we intercepted the ciphertext message

ZNK KGXRE HOXJ MKZY ZNK CUXS

that we know was produced by a shift cipher. What was the original plaintext message?

Solution: Because we know that the intercepted ciphertext message was encrypted using a shift cipher, we begin by calculating the frequency of letters in the ciphertext. We find that the most common letter in the ciphertext is K. So, we hypothesize that the shift cipher sent the plaintext letter E to the ciphertext letter K. If this hypothesis is correct, we know that $10 = 4 + k$ **mod** 26, so $k = 6$. Next, we shift the letters of the message by −6, obtaining

THE EARLY BIRD GETS THE WORM.

Because this message makes sense, we assume that the hypothesis that $k = 6$ is correct.

Summary

In this lesson, you learnt that

1. Cryptography deals with transforming information so that it cannot be easily recovered without special knowledge
2. Caesar's cipher shifts each alphabet forward by 3 positions
3. Shift ciphers shifts alphabets forward by *k* positions; more specifically each alphabet can be encrypted and decrypted with the functions

   $$f(p) = (p + k) \textbf{ mod } 26 \text{ and } f^{-1}(p) = (p - k) \textbf{ mod } 26 \text{ respectively.}$$

4. The process of recovering plaintext from ciphertext without knowledge of both the encryption method and the key is known as **cryptanalysis** or **breaking codes**

Post-Test

1. Encrypt the message DO NOT PASS GO by translating the letters into numbers, applying the given encryption function, and then translating the numbers back into letters.

   a. $f(p) = (p + 3)$ **mod** 26 (the Caesar cipher)

b. $f(p) = (p + 13) \textbf{ mod } 26$

c. $f(p) = (3p + 7) \textbf{ mod } 26$

2. Suppose the ciphertext ERC WYJJMGMIRXPC EHZERGIH XIGLRSPSKC MW MRHMWXMRKYMWLEFPI JVSQ QEKMG was produced by encrypting a plaintext message using a shift cipher. What is the original plaintext?

References

1. Rosen, K. Discrete Mathematics and its Applications. 7th Edition, McGraw-Hill, 2012.

Lesson 12: Revision