Problem Solving and Programming with Python

As per Anna University syllabus - GE8151

SECOND EDITION

REEMA THAREJA

Assistant Professor
Department of Computer Science
Shyama Prasad Mukherji College for Women
University of Delhi

OXFORD
UNIVERSITY PRESS

OXFORD
UNIVERSITY PRESS

Oxford University Press is a department of the University of Oxford.
It furthers the University's objective of excellence in research, scholarship,
and education by publishing worldwide. Oxford is a registered trade mark of
Oxford University Press in the UK and in certain other countries.

Published in India by
Oxford University Press
Ground Floor, 2/11, Ansari Road, Daryaganj, New Delhi 110002, India

First Edition published in 2018
Second Edition published in 2019

ISBN-13: 978-0-19-012093-1
ISBN-10: 0-19-012093-2

Typeset in Times New Roman
by Ideal Publishing Solutions, Delhi
Printed in India by

Cover image: ©andriano.cz / Shutterstock

Preface

Computers are so widely used in our day-to-day lives that imagining a life without them has become almost impossible. They are not only used by professionals but also by children for interactively learning lessons, playing games, and doing their homework. Applications of the computer and its users are increasing by the day. Learning computer and programming basics is a stepping stone to having an insight into how the machines work. In-depth knowledge of basic computing terminologies and problem solving strategies such as algorithm, flowchart, and pseudocode are very essential to develop efficient and effective computer programs that may help solve a user's problems.

Since computers cannot understand human languages, special programming languages are designed for this purpose. Python is one such language. It is an open-source, easy, high-level, interpreted, interactive, object-oriented and reliable language that uses English-like words. It can run on almost all platforms including Windows, Mac OS X, and Linux. Python is also a versatile language that supports development of a wide range of applications ranging from simple text processing to WWW browsers to games. Moreover, programmers can embed Python within their C, C++, COM, ActiveX, CORBA, and Java programs to give 'scripting' capabilities to the users.

Python uses easy syntax and short codes as well as supports multiple programming paradigms, including object oriented programming, functional Python programming, and parallel programming models. Hence, it has become an ideal choice for the programmers and even the novices in computer programming field find it easy to learn and implement. It has encompassed a huge user base that is constantly growing and this strength of Python can be understood from the fact that it is the most preferred programming language in companies such as Nokia, Google, YouTube, and even NASA for its easy syntax and short codes.

About the Book

This book is designed as a textbook to cater to the requirements of the Python programming course offered to the first year engineering students of Anna University. The objective of this book is to introduce the students to the fundamentals of problem solving strategies and the concepts of Python programming language, and enable them to apply these concepts for solving real-world problems.

The book is organized into 8 chapters that provide comprehensive coverage of all the relevant topics using simple language. It also contains useful annexures to various chapters including for additional information. Case studies and appendices are also provided to supplement the text.

Programming skill is best developed by rigorous practice. Keeping this in mind, the book provides a number of programming examples that would help the reader learn how to write efficient programs. These programming examples have already been complied and tested using Python 3.4.1 version and can be also executed on Python 3.5 and 3.6 versions. To further enhance the understanding of the subject, there are numerous chapter-end exercises provided in the form of objective-type questions, review questions, and programming problems.

Key Features of the Book

The following are the important features of the book:

- **Complete coverage** of the Problem Solving and Python Programming syllabus offered by Anna University
- Offers **simple** and **lucid** treatment of concepts supported with illustrations for easy understanding
- Contains **separate chapters** on Strings, Files, Classes, and Exception Handling

- Provides **numerous programming and illustrative examples** along with their outputs to help students master the art of writing efficient Python programs
- Includes **notes** and **programming tips** to highlight the important concepts and help readers avoid common programming errors
- Offers **rich chapter-end pedagogy** including plenty of objective-type questions (with answers), review questions, programming and debugging exercises to facilitate revision and practice of concepts learnt
- Includes **6 annexures** and **4 appendices** covering differences between Python 2.x and 3.x, installing Python, debugging and testing, Turtle graphics, plotting graphs, lab exercises, and GUI Programming provided to supplement the text. Exercises are also added at the end of several annexures and appendices
- Includes **lab exercises** explained through algorithms and flowcharts to help readers hone their logical and programming abilities
- Provides **case studies** on creating calculator, calendar, hash files, compressing strings and files, finding resolution of an image, and mail merge that are linked to various chapters to demonstrate the application of concepts
- Contains **2 solved previous years' question papers** and **2 solved model question papers** included to help readers prepare for the semester-end university examinations
- Point-wise **summary** and **glossary** of key terms to aid quick recapitulation to concepts

Organization of the Book

The book contains 8 chapters, 6 annexures, 7 case studies, and 4 appendices. The details of the book are presented as follows.

Chapter 1 discusses the various strategies used for problem solving. Topics such as algorithms, flowcharts, and pseudocodes supported with some illustrative problems are covered in this chapter.

Annexure 1 discusses about programming languages and their evolution through generations. It describes different programming paradigms, features of OOP, and merits and demerits of object oriented programming languages. The chapter also gives a comparative study Python and other OOP languages, and highlights the applications of OOP paradigm.

Chapter 2 details the history, important features, and applications of Python. It also presents the various building blocks (such as keywords, identifiers, constants variables, operators, expressions, statements, and naming conventions) supported by the language. It discusses functions and modules, and Python interpreter and interactive mode too.

The chapter is followed by 3 annexures—***Annexure 2*** provides instructions for installing Python, ***Annexure 3*** provides the comparison between Python 2.x and Python 3.x versions, and ***Annexure 4*** discusses testing and debugging of Python programs using IDLE.

Chapter 3 deals with the different types of decision control statements such as selection/conditional branching, iterative, break, continue, pass, and else statements.

Case studies 1** and **2 on simple calculator and generating a calendar show the implementation of concepts discussed in Chapters 2 and 3.

Chapter 4 provides a detailed explanation of defining and calling functions. It also explains the important concepts such as variable length arguments, fruitful functions, recursive functions, and function composition in Python.

Annexure 5 explains how functions are objects in Python. ***Case study 3*** on shuffling a deck of cards demonstrates the concepts of functions as well as recursion.

Chapter 5 unleashes the concept of strings. The chapter lays special focus on the operators used with strings, slicing operation, built-in string methods and functions, comparing and iterating through strings, and the `string` module.

Chapter 6 details the different data structures (such as list, tuple, dictionary, etc.) that are extensively used in Python. It deals with creating, accessing, cloning, ad updating of lists as well as list methods and functions. It also describes functional programming and creating, accessing, and updating tuples. It also includes the concepts related to dictionaries, nested lists, aliasing, list parameters, lists as arrays, nested tuples, nested dictionaries, advanced list processing, and dictionary comprehensions.

Chapter 7 discusses how data can be stored in files. The chapter deals with opening, processing (like reading, writing, appending, etc.), and closing of files though a Python program. These files are handled in text mode as well as binary mode for better clarity of the concepts. The chapter also explains the concept of file, directory, and the osmodule. It also discusses % (string formatting operator) and command line arguments.

Case studies 4, 5, and ***6*** on creating a hash file, mail merge, and finding the resolution of an image demonstrate the applications of concepts related strings and file handling.

Chapter 8 elucidates the concepts of exception handling that can be used to make your programs robust. Concepts such as try, except, and finally blocks, raising and re-raising exceptions, built-in and user-defined exceptions, assertions, and handling invoked functions, used for handling exceptions are demonstrated in this chapter. It also discusses the concept of modules and packages.

Annexure 6 discusses classes and objects.

Case study 7 shows how to compress strings and files using exception handling concepts.

The **4 appendices** included in the book discuss about lab exercises, GUI programming, usage of Turtle graphics, and plotting graphs.

Online Resources

For the benefit of faculty and students reading this book, additional resources available online include:

For Faculty

- Solutions manual (for programming exercises)
- Chapter-wise PPTs
- Chapters on Inheritance and Operator Overloading
- Additional Material

For Students

- Lab exercises
- Test generator
- Projects
- Solutions to find the output and error exercises
- Extra reading material on unit testing in Python, sorting and searching methods, multi-threaded programming, network programming, event-driven programming and accessing databases using Python, and additional information on some more topics
- Additional algorithms, pseudocodes, and flowcharts

Acknowledgements

The writing of this textbook was a mammoth task for which a lot of help was required from many people. Fortunately, I have had wholehearted support of my family, friends, and fellow members of the teaching staff and students at Shyama Prasad Mukherji College, New Delhi.

My special thanks would always go to my parents, Mr Janak Raj Thareja and Mrs Usha Thareja, and my siblings, Pallav, Kimi, and Rashi, who were a source of abiding inspiration and divine blessings for me. I am especially thankful to my son, Goransh, who has been very patient and cooperative in letting me realize my dreams. My sincere thanks go to my uncle, Mr B.L. Theraja, for his inspiration and guidance in writing this book.

I would like to acknowledge the technical assistance provided to me by Mr Mitul Kapoor. I would like to thank him for sparing out his precious time to help me to design and test the programs.

Brief Contents

Preface v
Brief Contents ix
Road Map to Syllabus xiv

1. Algorithmic Problem Solving 1
   Annexure 1 —Introduction to Object Oriented Programming (OOP) 25
2. Basics of Python Programming 37
   Annexure 2 — Installing Python 83
   Annexure 3 — Comparison between Python 2.x and Python 3.x Versions 85
   Annexure 4 — Testing and Debugging 88
3. Control Flow Statements 95
   Case Study 1 — Simple Calculator 140
   Case Study 2 — Generating a Calendar 143
4. Functions 145
   Annexure 5 — Functions as Objects 187
   Case Study 3 — Shuffling a Deck of Cards 189
5. Strings 190
6. Lists, Tuples, and Dictionaries 233
7. File Handling 303
   Case Study 4 — Creating a Hash File (or a message digest of a file) 334
   Case Study 5 — Mail Merge Program 336
   Case Study 6 — Finding Resolution of an Image 338
8. Error and Exception Handling, Modules, and Packages 339
   Annexure 6 — Classes and Objects 369
   Case Study 7 — Compressing String and Files 381

Appendix A — Lab Exercises 384
Appendix B — GUI Programming with `tkinter` *Package 407*
Appendix C — Simple Graphics using Turtle 417
Appendix D — Plotting Graphs in Python 423
Solved Question Paper (2018/19) 429
Solved Question Paper (2017/18) 444
Solved Model Question Paper - 1 461
Solved Model Question Paper - 2 463
About the Author 465

Detailed Contents

Preface v
Brief Contents ix
Road Map to Syllabus xiv

1. Algorithmic Problem Solving 1

1.1 Introduction *1*
1.2 Algorithms *1*
1.3 Building Blocks of Algorithm (Instructions, State, Control flow, Functions) *2*
1.3.1 Subcharts/Subroutine/Predefined Process *2*
1.4 Algorithmic Problem Solving Steps *2*
1.5 Simple Strategies and Notations for Developing Algorithms *3*
1.5.1 Control Structures Used in Algorithms *4*
1.5.2 Flowcharts *8*
1.5.3 Pseudocodes *10*
1.5.4 Programming Languages *12*
1.6 Illustrative Problems *13*
1.6.1 Find Minimum in a List *13*
1.6.2 Insert a Card in a List of Sorted Cards *15*
1.6.3 Guess an Integer Number in a Range *17*
1.6.4 Tower of Hanoi *19*
Annexure 1 — Introduction to Object Oriented Programming (OOP) 25

2. Basics of Python Programming 37

2.1 Features of Python *37*
2.2 History of Python *39*
2.3 The Future of Python *41*
2.4 Python Interpreter and Interactive Mode *41*
2.5 Writing and Executing First Python Program *42*
2.6 Value and Types *43*
2.7 Numbers *43*
2.8 Boolean *46*
2.9 Lists *46*
2.10 Strings *47*
2.11 Variables and Identifiers *50*
2.12 Data Types *51*
2.13 Statements *51*
2.13.1 Assignment Statement *51*
2.13.2 Input Statement *54*
2.13.3 Indentation *54*
2.14 Reserved Words *55*
2.15 Tuple Statement *55*
2.16 Dictionary *56*
2.17 Operators and Expressions *56*
2.17.1 Arithmetic Operators *56*
2.17.2 Comparison Operators *57*
2.17.3 Assignment and In-place or Shortcut Operators *58*
2.17.4 Unary Operators *59*
2.17.5 Bitwise Operators *59*
2.17.6 Shift Operators *60*
2.17.7 Logical Operators *60*
2.17.8 Membership Operators *61*
2.17.9 Identity Operators *61*
2.17.10 Operators Precedence and Associativity *62*
2.18 Expressions in Python *63*
2.19 Operations on Strings *63*
2.19.1 Concatenation *64*
2.19.2 Multiplication (or String Repetition) 64
2.19.3 Slice a String *65*
2.20 Type Conversion *66*
2.21 Comments *67*
2.22 Function and Modules *68*
2.22.1 Function Definition and Use *68*
2.22.2 Flow of Execution *68*
2.22.3 Parameters and Arguments *69*
Annexure 2 — Installing Python 83

Annexure 3 — Comparison between Python 2.x and Python 3.x Versions 85
Annexure 4 — Testing and Debugging 88

3. Control Flow Statements 95

3.1 Introduction to Decision Control Statements *95*
3.2 Selection/Conditional Branching Statements *96*
3.2.1 `if` Statement *96*
3.2.2 `if-else` Statement *97*
3.2.3 Nested `if` Statements *100*
3.2.4 `if-elif-else` Statement *100*
3.3 Basic Loop Structures/ Iterative Statements *105*
3.3.1 `while` loop *106*
3.3.2 `for` Loop *113*
3.3.3 Selecting an appropriate loop *115*
3.4 Nested Loops *122*
3.5 The `break` Statement *125*
3.6 The `continue` Statement *126*
3.7 The `pass` Statement *129*
3.8 The `else` Statement Used with Loops *130*
Case Study 1 — Simple Calculator 140
Case Study 2 — Generating a Calendar 143

4. Functions 145

4.1 Introduction *145*
4.1.1 Need for Functions *146*
4.2 Defining a Function *147*
4.3 Function Call *148*
4.3.1 Function Parameters *148*
4.4 Variable Scope and Lifetime *151*
4.4.1 Local and Global Variables *151*
4.4.2 Using the Global Statement *152*
4.4.3 Resolution of Names *155*
4.5 Fruitful Functions *155*
4.5.1 The `return` Statement *155*
4.5.2 Parameters *157*
4.6 Lambda Functions or Anonymous Functions *161*
4.7 Function Composition in Python *164*
4.8 Documentation Strings *165*
4.9 Good Programming Practices *166*
4.10 Recursive Functions *171*
4.10.1 Greatest Common Divisor *173*
4.10.2 Finding Exponents *173*
4.10.3 The Fibonacci Series *174*
4.10.4 Recursion vs Iteration *177*
4.11 Function Redefinition *179*
Annexure 5 — Functions as Objects 187
Case Study 3 — Shuffling a Deck of Cards 189

5. Strings 190

5.1 Concatenating, Appending, and Multiplying Strings *191*
5.2 Strings are Immutable *193*
5.3 String Formatting Operator *194*
5.4 Built-in String Methods and Functions *197*
5.5 Slice Operation *202*
5.5.1 Specifying Stride While Slicing Strings *204*
5.6 `ord()` and `chr()` Functions *205*
5.7 `in` and `not in` operators *205*
5.8 Comparing Strings *206*
5.9 Iterating String *207*
5.10 The String Module *213*
5.11 Metacharacters in Regular Expression *216*
5.11.1 Character Classes *218*
5.11.2 Groups *220*
5.11.3 Application of Regular Expression to Extract Email *221*

6. Lists, Tuples, and Dictionaries 233

6.1 Sequence *233*
6.2 Lists *233*
6.2.1 Access Values in Lists *234*
6.2.2 Updating Values in Lists *234*
6.2.3 Nested Lists *236*
6.2.4 Aliasing *237*
6.2.5 Cloning Lists *237*
6.2.6 List Parameters *238*
6.2.7 Basic List Operations *238*
6.2.8 List Methods *239*
6.2.9 Lists as Arrays *242*

6.2.10 Advanced List Processing 244
6.2.11 Looping in Lists 245
6.3 Functional Programming 247
6.3.1 `filter()` Function 247
6.3.2 `map()` Function 248
6.3.3 `reduce()` Function 249
6.4 Tuple 261
6.4.1 Creating Tuple 261
6.4.2 Utility of Tuples 262
6.4.3 Accessing Values in a Tuple 263
6.4.4 Updating Tuple 263
6.4.5 Deleting Elements in Tuple 264
6.4.6 Basic Tuple Operations 264
6.4.7 Tuple Assignment 265
6.4.8 Tuples for Returning Multiple Values 266
6.4.9 Nested Tuples 266
6.4.10 Checking the Index: `index()` method 267
6.4.11 Counting the Elements: `count()` Method 268
6.4.12 List Comprehension and Tuples 268
6.4.13 Variable-length Argument Tuples 269
6.4.14 The `zip()` Function 270
6.4.15 Advantages of Tuple over List 272
6.5 Dictionaries 275
6.5.1 Creating a Dictionary 275
6.5.2 Accessing Values 277
6.5.3 Adding and Modifying an Item in a Dictionary 277
6.5.4 Modifying an Entry 278
6.5.5 Deleting Items 278
6.5.6 Sorting Items in a Dictionary 281
6.5.7 Looping over a Dictionary 281
6.5.8 Nested Dictionaries 282
6.5.9 Built-in Dictionary Functions and Methods 282
6.5.10 Difference between a List and a Dictionary 285
6.5.11 String Formatting with Dictionaries 285
6.5.12 When to use which Data Structure? 286
6.5.13 List vs Tuple vs Dictionary vs Set 286

7. File Handling 303

7.1 Introduction 303
7.2 File Path 303
7.3 Types of Files 304
7.3.1 ASCII Text Files 305
7.3.2 Binary Files 305
7.4 Opening and Closing Files 306
7.4.1 The `open()` Function 306
7.4.2 The File Object Attributes 307
7.4.3 The `close()` Method 308
7.5 Reading and Writing Files 309
7.5.1 `write()` and `writelines()` Methods 309
7.5.2 `append()` Method 309
7.5.3 The `read()` and `readline()` Methods 310
7.5.4 Opening Files using `with` Keyword 312
7.5.5 Splitting Words 313
7.5.6 Some Other Useful File Methods 314
7.6 File Positions 314
7.7 % (String Formatting Operator) 317
7.8 Command Line Arguments 319
7.9 Renaming and Deleting Files 320
7.10 Directory Methods 321
7.10.1 Methods from the os Module 324
Case Study 4 — Creating a Hash File (or a message digest of a file) 334
Case Study 5 — Mail Merge Program 336
Case Study 6 — Finding Resolution of an Image 338

8. Error and Exception Handling, Modules, and Packages 339

8.1 Introduction to Errors and Exceptions 339
8.1.1 Syntax Errors 339
8.1.2 Logic Error 340
8.1.3 Exceptions 340
8.2 Handling Exceptions 341
8.3 Multiple Except Blocks 342

8.4 Multiple Exceptions in a Single Block *343*
8.5 Except Block Without Exception *343*
8.6 The `else` clause *344*
8.7 Raising Exceptions *345*
8.8 Instantiating Exceptions *345*
8.9 Handling Exceptions in Invoked Functions *346*
8.10 Built-in and User-Defined Exceptions *348*
8.11 The `try...` finally Block *349*
8.12 Pre-defined Clean–up Action *351*
8.13 Modules *353*
8.13.1 The from…import statement *354*
8.13.2 Name of Module *355*
8.13.3 Making your own Modules *355*
8.13.4 The `dir()` function *357*
8.13.5 The Python Module *358*
8.13.6 Modules and Namespaces *358*
8.14 Packages in Python *360*
8.15 Standard Library Modules *362*
Annexure 6 — Classes and Objects 369
Case Study 7 — Compressing String and Files 381

Appendix A — Lab Exercises 384
Appendix B — GUI Programming with `tkinter` *Package 407*
Appendix C — Simple Graphics using Turtle 417
Appendix D — Plotting Graphs in Python 423
Solved Question Paper (2018/19) 429
Solved Question Paper (2017/18) 444
Solved Model Question Paper - 1 461
Solved Model Question Paper - 2 463
About the Author 465

CHAPTER 1

Algorithmic Problem Solving

KEY Concepts

• Algorithm • Flowchart • Pseudocode • Iteration • Recursion
• Illustrative Examples

1.1 INTRODUCTION

A *program* is a set of instructions that tells the computer how to solve a particular problem. Various program design tools like algorithms, pseudocdes and flowcharts are used to design the blueprint of the solution (or the program to be written). Computer programming goes a step further in problem solving process. *Programming* means writing computer programs. While programming, the programmers take an algorithm and code the instructions in a particular programming language so that it can be executed by a computer. These days, there are many programming languages available in the market. The programmer can choose any language depending on his expertise and the problem domain.

The following sections will deal with different program design tools, knowledge of which is compulsory to develop programming skills.

1.2 ALGORITHMS

In computing, we focus on the type of problems categorically known as *algorithmic problems*, where their solutions are expressible in the form of algorithms. The term 'algorithm' was derived from the name of Mohammed al-Khwarizmi, a Persian mathematician in the nineth century (Al-Khwarizmi → Algorism (in Latin) → Algorithm). The typical meaning of an algorithm is a formally defined procedure for performing some calculation. If a procedure is formally defined, then it must be implemented using some formal language, and such languages are known as *programming languages.* The algorithm gives the logic of the program, that is, a step-by-step description of how to arrive at a solution.

In general terms, an algorithm provides a blueprint to writing a program to solve a particular problem. It is considered to be an effective procedure for solving a problem in a finite number of steps. That is, a well-defined algorithm always provides an answer, and is guaranteed to terminate.

Algorithms are mainly used to achieve *software reuse.* Once we have an idea or a blueprint of a solution, we can implement it in any language, such as C, C++, Java, and so on. In order to qualify as an algorithm, a sequence of instructions must possess the following characteristics:

- **Precision:** The instructions should be written in a precise manner.
- **Uniqueness:** The outputs of each step should be unambiguous, i.e., they should be unique and only depend on the input and the output of the preceding steps.
- **Finiteness:** Not even a single instruction must be repeated infinitely.

- **Effectiveness**: The algorithm should designed in such a way that it should be the most effective among many different ways to solve a problem.
- **Input:** The algorithm must receive an input.
- **Output:** After the algorithm gets terminated, the desired result must be obtained.
- **Generality:** The algorithm can be applied to various set of inputs.

1.3 BUILDING BLOCKS OF ALGORITHM (INSTRUCTIONS, STATE, CONTROL FLOW, FUNCTIONS)

An algorithm starts from an *initial state* with some input. The *instructions/statements* describe the processing that must be done on the input to produce the *final output* (the final state). Note that an *instruction* is a single operation which when executed converts one state to other.

In the course of processing, data is read from an input device, stored in computer's memory for further processing, and then the result of the processing is written to an output device.

The data is stored in the computer's memory in the form of variables or constants. The *state* of an algorithm is defined as its condition regarding current values or contents of the stored data.

An algorithm is a list of precise steps and the order of steps determines the functioning of the algorithm. The *flow of control* (or the control flow) of an algorithm can be specified as top-down or bottom-up approach. Thus, the flow of control specifies the order in which individual instructions of an algorithm are executed.

1.3.1 Subcharts/Subroutine/Predefined Process

A subroutine (or procedure or function or routine) is a sequence of instructions that performs a specific task. These instructions are packaged as a single unit and can be used (or invoked or called) wherever that particular task needs to be performed. After performing its defined task, the sub-routine branches back (or *returns*) to the next instruction after the one that invoked it.

A subroutine may be designed to accept one or more data values (also known as parameters) from the calling code. It may also return a value to its caller. A subroutine can also be written in such a way that it calls itself repeatedly.

The subroutine symbol is used to write steps for procedures. These procedures can be called from anywhere in the code. This means that once the flowchart for a process is drawn, it can be referenced and used from anywhere in the code.

1.4 ALGORITHMIC PROBLEM SOLVING STEPS

As mentioned earlier, algorithms are solutions to problems. They are not solutions themselves. They just list specific instructions that need to be performed for getting the solution. In computer science, emphasis is laid on writing a good and effective algorithm and this emphasis makes computer science distinct from other disciplines. For example, computer science is distinct from theoretical mathematics because those practitioners are typically satisfied with just proving the existence of a solution to a problem but in computer science, the problem is not solved until the algorithm is used to implement the solution.

We will now discuss about the sequence of steps one must typically follow for designing an effective algorithm.

1. Understanding the problem
2. Determining the capabilities of the computational device
3. Exact/approximate solution
4. Select the appropriate data structure
5. Algorithm design techniques
6. Methods of specifying an algorithm

7. Proving an algorithms correctness
8. Analysing the performance of an algorithm

Understanding the problem The problem given should be clearly and completely understood. It is compared with earlier problems that have already been solved to check if it is similar to them and a known algorithm exists. If the algorithm is available, it is used, otherwise a new one has to be developed.

Determining the capabilities of the computational device After understanding the problem, the capabilities of the computing device should be known. For this, the type of the architecture, speed and memory availability of the device are noted.

Exact/approximate solution The next step is to develop the algorithm. The algorithm must compute correct output for all possible and legitimate inputs. This solution can be an exact solution or an approximate solution. For example, you can only have an approximate solution in case of finding square root of number or finding the solutions of non-linear equations.

Select the appropriate data structure A *data type* is a well-defined collection of data with a well-defined set of operations on it. A *data structure* is basically a group of data elements that are put together under one name, and which defines a particular way of storing and organizing data in a computer so that it can be used efficiently. The elementary data structures are as follows.

- **List:** Allows fast access of data.
- **Sets:** Treats data as elements of a set. Allows application of operations such as intersection, union, and equivalence.
- **Dictionaries:** Allows data to be stored as a key-value pair.

Algorithm design techniques Developing an algorithm is an art which may never be fully automated. By mastering the design techniques, it will become easier for you to develop new and useful algorithms. Examples of algorithm design techniques include dynamic programming.

Methods of specifying an algorithm An algorithm is just a sequence of steps or instructions that can be used to implement a solution. After writing the algorithm, it is specified either using a natural language or with the help of pseudocode and flowcharts. We will read about them in the next section.

Proving algorithms correctness Writing an algorithm is not just enough. You need to prove that it computes solutions for all the possible valid inputs. This process is often referred to as algorithm validation. Algorithm validation ensures that the algorithm will work correctly irrespective of the programming language in which it will be implemented.

Analysing the performance of algorithms When an algorithm is executed, it uses the computer's resources like the Central Processing Unit (CPU) to perform its operation and to hold the program and data respectively. An algorithm is analysed to measure its performance in terms of CPU time and memory space required to execute that algorithm. This is a challenging task and is often used to compare different algorithms for a particular problem. The result of the comparison helps us to choose the best solution from all possible solutions. Analysis of the algorithm also helps us to determine whether the algorithm will be able to meet any efficiency constraint that exits or not.

1.5 SIMPLE STRATEGIES AND NOTATIONS FOR DEVELOPING ALGORITHMS

An algorithm is a *step–by–step procedure* for solving a task or problem. However, these steps must be ordered, unambiguous and finite in number. Basically, an algorithm is nothing but English-like representation of logic which is used to solve the problem.

For accomplishing a particular task, different algorithms can be written. The different algorithms differ in their requirements of CPU time and memory space. The programmer selects the best suited algorithm for the given task to be solved.

Various strategies and notations used for developing and designing algorithms are discussed in the following sections.

1.5.1 Control Structures Used in Algorithms

An algorithm has a finite number of steps and some steps may involve decision making and repetition. Broadly speaking, an algorithm may employ three control structures, namely, sequence, decision, and repetition.

Sequence Sequence means that each step of the algorithm is executed in the specified order. An algorithm to add two numbers is given as follows. This algorithm performs the steps in a purely sequential order.

Example 1.1 Algorithm to add two numbers

```
Step 1 : Start
Step 2 : Input first number as A
Step 3 : Input second number as B
Step 4 : Set Sum = A + B
Step 5 : Print Sum
Step 6 : End
```

Decision Decision statements are used when the outcome of the process depends on some condition. For example, if x = y, then print "EQUAL". Hence, the general form of the if construct can be given as follows:

```
IF condition then process
```

A condition in this context is any statement that may evaluate either to a true value or a false value. In the preceding example, the variable *x* can either be equal or not equal to *y*. However, it cannot be both true and false. If the condition is true then the process is executed.

A decision statement can also be stated in the following manner:

```
IF condition
   then process1
ELSE process2
```

This form is commonly known as the if-else construct. Here, if the condition is true then process1 is executed, else process2 is executed. An algorithm to check the equality of two numbers is shown below.

Example 1.2 Algorithm to test the quality of two numbers

```
Step 1 : Start
Step 2 : Input first number as A
Step 3 : Input second number as B
Step 4 : IF A = B
           Print "Equal"
         ELSE
           Print "Not equal" [END of IF]
Step 5 : End
```

Let us look at the following two simple algorithms to find the greatest among three numbers.

Example 1.3 Algorithm to find the greatest of three numbers

```
Step 1: Start
Step 2: Read the three numbers A,B,C
Step 3: Compare A and B. If A is greater perform step 4 else perform step 5.
Step 4: Compare A and C. If A is greater, output "A is greatest" else output "C is
        greatest"
Step 5: Compare B and C. If B is greater, output "B is greatest" else output "C is
        greatest"
Step 6: End
```

Example 1.4 Algorithm to find the greatest of three numbers using an additional variable MAX

```
Step 1: Start
Step 2: Read the three numbers A,B,C
Step 3: Compare A and B. If A is greater, store Ain MAX, else store B in MAX
Step 4: Compare MAX and C. If MAX is greater, output "MAX is greater" else output
        "C is greater"
Step 5: End
```

Both the algorithms given in Examples 1.3 and 1.4 accomplish same goal, but in different ways. The programmer selects the algorithm based on the advantages and disadvantages of each algorithm. For example, the first algorithm has more number of comparisons, whereas in the second algorithm an additional variable MAX is used to do the comparison.

Iteration or repetition which involves executing one or more steps for a number of times, can be implemented using constructs such as the while loops and for loops. These loops execute one or more steps until some condition is true.

Example 1.5 Algorithm that prints the first 10 natural numbers

```
Step 1: Start
Step 2: [initialize] Set I = 1, N = 10
Step 3: Repeat Steps 3 and 4 while I <= N
Step 4: Print I
Step 5: Set I = I + 1 [END OF LOOP]
Step 6: End
```

Example 1.6 Design an algorithm for adding the test scores given as: 26, 49, 98, 87, 62, 75.

```
Step 1: Start
Step 2: sum = 0
Step 3: Get a value
Step 4: sum = sum + value
Step 5: If next value is present, go to step 3. Otherwise, go to step 6
Step 6: Print the sum
Step 7: End
```

Examples 1.5 and 1.6 demonstrate the application of repetition and iteration logic in an algorithm. Some more examples to depict the concept of control structures in algorithms are given as follows.

Example 1.7 Write an algorithm for interchanging/swapping two values.

```
Step 1: Start
Step 2: Input first number as A
Step 3: Input second number as B
Step 4: Set temp = A
Step 5: Set A = B
Step 6: Set B = temp
Step 7: Print A, B
Step 8: End
```

Example 1.8 Write an algorithm to find the larger of two numbers.

```
Step 1: Start
Step 2: Input first number as A
Step 3: Input second number as B
Step 4: IF A > B
             Print A
        ELSE IF A < B
             Print B
        ELSE
             Print "The numbers are equal"
        [END OF IF]
Step 5: End
```

Example 1.9 Write an algorithm to find whether a number is even or odd.

```
Step 1: Start
Step 2: Input number as A
Step 3: IF A % 2 = 0
          Print "Even"
        ELSE
          Print "Odd"
        [END OF IF]
Step 4: End
```

Example 1.10 Write an algorithm to print the grade obtained by a student using the following rules:

Marks	Grade
Above 75	O
60-75	A
50-60	B
40-50	C
Less than 40	D

```
Step 1: Start
Step 2: Enter the marks obtained as M
Step 3: IF M > 75
      Print "O"
Step 3: IF M >= 60 and M < 75
     Print "A"
Step 4: IF M >= 50 and M < 60
     Print "B"
Step 5: IF M >= 40 and M < 50
     Print "C"
    ELSE
     Print "D"
  [END OF IF]
Step 6: End
```

Example 1.11 Write an algorithm to find the sum of first N natural numbers.

```
Step 1: Start
Step 2: Input N
Step 3: Set I = 1, sum = 0
Step 4: Repeat Steps 4 and 5 while I <= N
Step 5: Set sum = sum + I
Step 6: Set I = I + 1
   [END OF LOOP]
Step 7: Print sum
Step 8: End
```

Recursion It is a technique of solving a problem by breaking it down into smaller and smaller sub-problems until you get to a small enough problem that it can be easily solved. Usually, recursion involves a function calling itself until a specified condition is met. Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function. Every recursive solution has two major cases. They are:

- ***Base case***, in which the problem is simple enough to be solved directly without making any further calls to the same function.

- ***Recursive case***, in which first the problem at hand is divided into simpler sub-parts. Second the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.

Therefore, recursion is defining large and complex problems in terms of smaller and more easily solvable problems. In recursive functions, a complex problem is defined in terms of simpler problems and the simplest problem is given explicitly.

(For a detailed study on Recursion and Iteration, refer to Chapter 4)

Example 1.12 Write a recursive algorithm to find the factorial of a number.

```
Step 1: Start
Step 2: Input number as n
Step 3: Call factorial(n)
Step 4: End

factorial(n)
Step 1: Set f = 1
Step 2: IF n==1 then return 1
        ELSE
         Set f=n*factorial(n-1)
Step 3: Print f
```

1.5.2 Flowcharts

A flowchart is a graphical or symbolic representation of a process. It is basically used to design and document virtually complex processes to help the viewers to visualize the logic of the process, so that they can gain a better understanding of the process and find flaws, bottlenecks, and other less obvious features within it.

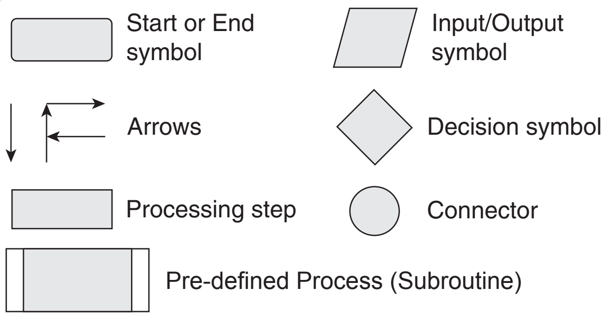


Figure 1.1 Symbols used in a Flowchart

When designing a flowchart, each step in the process is depicted by a different symbol and is associated with a short description. The symbols in the flowchart (refer Figure 1.1) are linked together with arrows to show the flow of logic in the process.

The symbols used in a flowchart include the following:

- *Start and end symbols* are also known as the terminal symbols and are represented as circles, ovals, or rounded rectangles. Terminal symbols are always the first and the last symbols in a flowchart.
- *Arrows* depict the flow of control of the program. They illustrate the exact sequence in which the instructions are executed.
- *Generic processing step,* also called as an activity, is represented using a rectangle. Activities include instructions such as add a to b or save the result. Therefore, a processing symbol represents arithmetic and data movement instructions. When more than one process has to be executed simultaneously, they can be placed in the same processing box. However, their execution will be carried out in the order of their appearance.

- *Input/Output symbols* are represented using a parallelogram and are used to get inputs from the users or display the results to them.
- A *conditional or decision symbol* is represented using a diamond. It is basically used to depict a Yes/No question or a True/False test. The two symbols coming out of it, one from the bottom point and the other from the right point, corresponds to Yes or True, and No or False, respectively. The arrows should always be labelled. A decision symbol in a flowchart can have more than two arrows, which indicates that a complex decision is being taken.
- *Labelled connectors* are represented by an identifying label inside a circle and are used in complex or multi-sheet diagrams to substitute for arrows. For each label, the 'outflow' connector must have one or more 'inflow' connectors. A pair of identically labelled connectors is used to indicate a continued flow when the use of lines becomes confusing.
- A *pre-defined process symbol* is a marker for another process step or series of process flow steps that are formally defined elsewhere. This shape commonly depicts sub-processes (or subroutines in programming flowcharts).

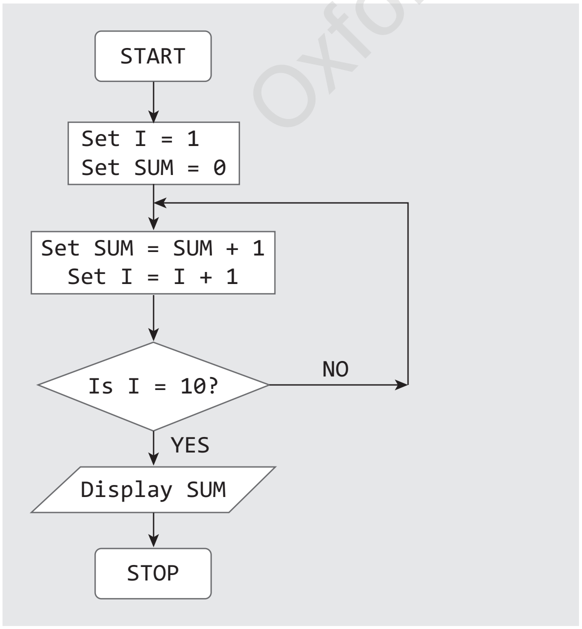
Significance of Flowcharts

A flowchart is a diagrammatic representation that illustrates the sequence of steps that must be performed to solve a problem. It is usually drawn in the early stages of formulating computer solutions. It facilitates communication between programmers and users. Once a flowchart is drawn, programmers can make users understand the solution easily and clearly.

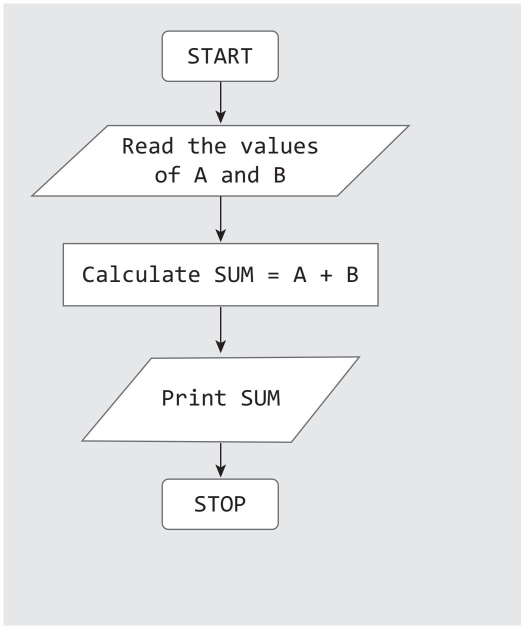
Flowcharts are very important in the programming of a problem as they help the programmers to understand the logic of complicated and lengthy problems. Once a flowchart is drawn, it becomes easy for the programmers to write the program in any high-level language. Hence, the flowchart has become a necessity for better documentation of complex programs.

A flowchart follows the top-down approach in solving problems. Some examples are given as follows.

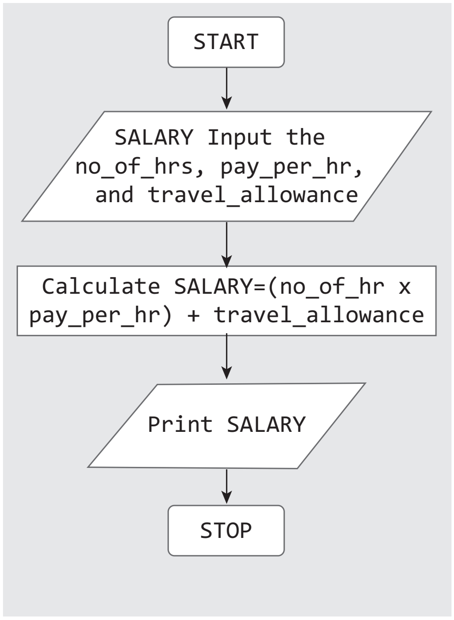
Example 1.13 Draw a flowchart to calculate the sum of the first 10 natural numbers.



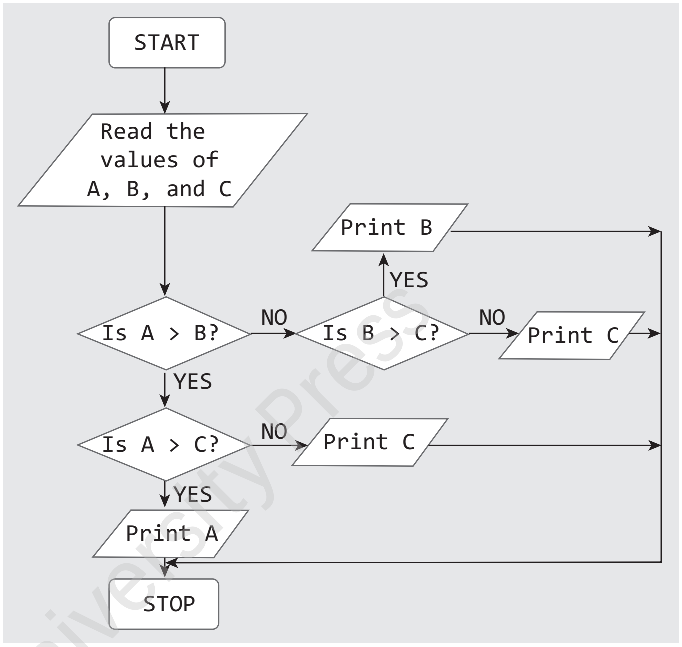
Example 1.14 Draw a flowchart to add two numbers.



Example 1.15 Draw a flowchart to calculate the salary of a daily wager.


START
SALARY Input the no_of_hrs, pay_per_hr, and travel_allowance
Calculate SALARY=(no_of_hr x pay_per_hr) + travel_allowance
Print SALARY
STOP

Example 1.16 Draw a flowchart to determine the largest of three numbers.



Advantages

- They are very good communication tools to explain the logic of a system to all concerned. They help to analyse the problem in a more effective manner.
- They are also used for program documentation. They are even more helpful in the case of complex programs.
- They act as a guide or blueprint for the programmers to code the solution in any programming language. They direct the programmers to go from the starting point of the program to the ending point without missing any step in between. This results in error-free programs.
- They can be used to debug programs that have error(s). They help the programmers to easily detect, locate, and remove mistakes in the program in a systematic manner.

Limitations

- Drawing flowcharts is a laborious and a time-consuming activity. Just imagine the effort required to draw a flowchart of a program having 50,000 statements in it!
- Many a times, the flowchart of a complex program becomes complex and clumsy.
- At times, a little bit of alteration in the solution may require complete redrawing of the flowchart.
- The essentials of what is done may get lost in the technical details of how it is done.
- There are no well-defined standards that limit the details that must be incorporated into a flowchart.

1.5.3 Pseudocodes

Pseudocode is a compact and informal high-level description of an algorithm that uses the structural conventions of a programming language. It facilitates designers to focus on the logic of the algorithm without getting bogged down by the details of language syntax. An ideal pseudocode must be complete, describing the entire logic of the algorithm, so that it can be translated straightaway into a programming language.

It is basically meant for human reading rather than machine reading, so it omits the details that are not essential for humans. Such details include variable declarations, system-specific code, and subroutines.

Pseudocodes are an outline of a program that can easily be converted into programming statements. They consist of short English phrases that explain specific tasks within a program's algorithm. They should not include keywords in any specific computer language.

The sole purpose of pseudocodes is to enhance human understandability of the solution. They are commonly used in textbooks and scientific publications for documenting algorithms, and for sketching out the program structure before the actual coding is done. This helps even non-programmers to understand the logic of the designed solution. There are no standards defined for writing a pseudocode, because a pseudocode is not an executable program. Flowcharts can be considered as graphical alternatives to pseudocodes, but require more space on paper.

Example 1.17 Write a pseudocode for calculating the price of a product after adding the sales tax to its original price.

```
1. Start
2. Read the price of the product
3. Read the sales tax rate
4. Calculate sales tax = price of the item x * sales tax rate
5. Calculate total price = price of the product + sales tax
6. Print total price
7. End

Variables: price of the item, sales tax rate, sales tax, total price
```

Example 1.18 Write a pseudocode to calculate the weekly wages of an employee. The pay depends on wages per hour and the number of hours worked. Moreover, if the employee has worked for more than 30 hours, then he or she gets twice the wages per hour, for every extra hour that he or she has worked.

```
1. Start
2. Read hours worked
3. Read wages per hour
4. Set overtime charges to 0
5. Set overtime hrs to 0
6. IF hours worked > 30 then
   a. Calculate overtime hrs = hours worked - 30
   b. Calculate overtime charges = overtime hrs * (2 * wages per hour)
   c. Set hours worked = hours worked - overtime hrs
   ENDIF
7. Calculate salary = (hours worked x wages per hour) + overtime charges
8. Display salary
9. End
Variables: hours worked, wages per hour, overtime charges, overtime hrs, salary
```

Example 1.19 Write a pseudocode to read the marks of 10 students. If marks is greater than 50, the student passes, else the student fails. Count the number of students passing and failing.

```
1. Start
2. Set pass to 0
3. Set fail to 0
4. Set no of students to 1
5. WHILE no of students < 10
   a. input the marks
   b. IF marks >= 50 then
         Set pass = pass + 1
   ELSE
         Set fail = fail + 1
   ENDIF
   ENDWHILE
6. Display pass
7. Display fail
8. End
Variables: pass, fail, no of students, marks
```

1.5.4 Programming Languages

A *programming language* is a language specifically designed to express computations that can be performed by a computer. Programming languages are used to create programs that control the behaviour of a system, to express algorithms, or as a mode of human communication.

Usually, programming languages have a vocabulary of syntax and semantics for instructing a computer to perform specific tasks. The term *programming language* refers to high-level languages such as BASIC (Beginners' All-purpose Symbolic Instruction Code), C, C++, COBOL (Common Business Oriented Language), FORTRAN (Formula Translator), Python, Ada, and Pascal, to name a few. Each of these languages has a unique set of keywords (words that it understands) and a special syntax for organizing program instructions.

Though high-level programming languages are easy for humans to read and understand, the computer can understand only machine language, which consists of only numbers. Each type of central processing unit (CPU) has its own unique machine language.

In between machine languages and high-level languages, there is another type of language known as assembly language. Assembly languages are similar to machine languages, but they are much easier to program because they allow a programmer to substitute names for numbers.

However, irrespective of the language that a programmer uses, a program written using any programming language has to be converted into machine language so that the computer can understand it. There are two ways to do this: *compile* the program or *interpret* the program.

When planning a software solution, the software development team often faces a common question—which programming language to use? Many programming languages are available today and each one has its own strengths and weaknesses. Python can be used to write an efficient code, whereas a code in BASIC is easy to write and understand; some languages are compiled, whereas others are interpreted; some languages are well known to the programmers, whereas others are completely new. Selecting the perfect language for a particular application at hand is a daunting task.

The selection of language for writing a program depends on the following factors:

- The type of computer hardware and software on which the program is to be executed.
- The type of program.
- The expertise and availability of the programmers.
- Features to write the application.
- The built-in features that support the development of software that are reliable and less prone to crash.
- Lower development and maintenance costs.
- Stability and capability to support even more than the expected simultaneous users.

- Elasticity of a language that implies the ease with which new features (or functions) can be added to the existing program.
- Portability.
- Better speed of development that includes the time it takes to write a code, time taken to find a solution to the problem at hand, time taken to find the bugs, availability of development tools, experience and skill of the programmers, and testing regime.

For example, FORTRAN is a particularly good language for processing numerical data, but it does not lend itself very well to organizing large programs. Pascal can be used for writing well-structured and readable programs, but it is not as flexible as the C programming language. C++ goes one step ahead of C by incorporating powerful object oriented features, but it is complex and difficult to learn. Python, however is a good mix of the best features of all these languages.

1.6 ILLUSTRATIVE PROBLEMS

In this section, we will look into some common problems in computer science and will specify its solution by writing an algorithm and pseudocode, and drawing its corresponding flowchart.

1.6.1 Find Minimum in a List

Consider the following requirement specification.

You are given a list of numbers from which you are supposed to find the minimum value. An algorithm is required for entering the numbers in the list and then calculate the minimum value. The count of numbers to be entered in the list should also be asked from the user.

Before writing the algorithm, just visualize how it works. Take the first number in the list and call it minimum. Compare the minimum's value with all other values in the list one by one. The moment you find a smaller element than the minimum, call it the minimum. Figure 1.2 given below illustrates this concept.

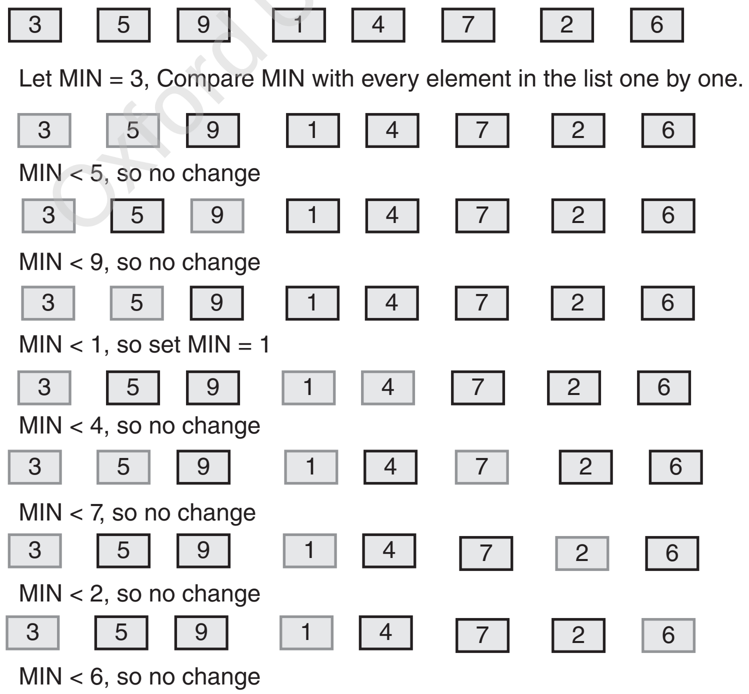


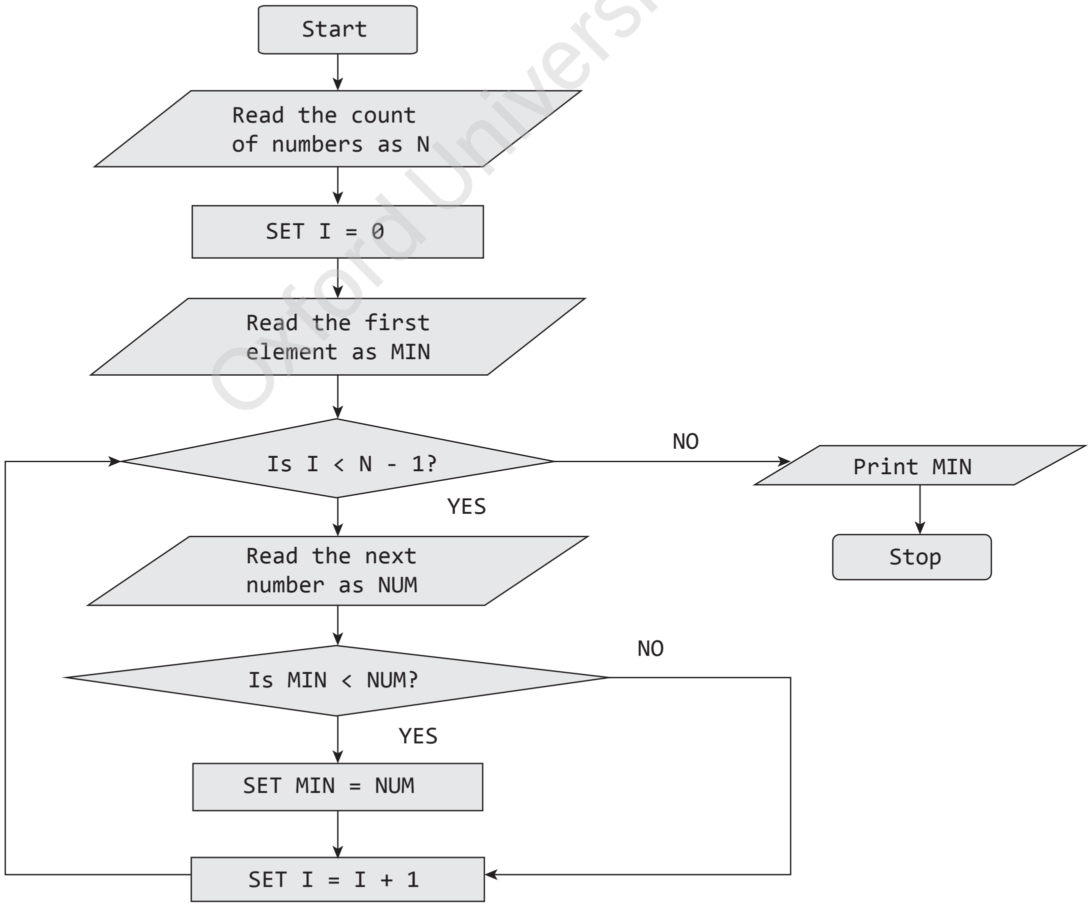
Figure 1.2 Finding minimum value in a list

The step-wise approach for solving the problem of finding the minimum in a list is demonstrated through Algorithm 1.1, Flowchart 1.1, and Pseudocode 1.1.

Algorithm 1.1

```
Step 1:   Start
Step 2:   READ the count of numbers as N
Step 3:   SET I = 0
Step 4:   READ the first element as MIN
Step 5:   REPEAT Steps 6-8 WHILE I < N – 1
Step 6:   READ the next number as NUM
Step 7:   IF MIN < NUM
             SET MIN = NUM
Step 8:   SET I = I + 1
Step 9:   PRINT MIN
Step 10:  End
```

Flowchart 1.1



Pseudocode 1.1

```
Read the count of numbers as N
Set I = 0
Read the first element as MIN
While I < N – 1
   Read the next number as NUM
      IF MIN < NUM
         Set MIN = NUM
      Set I = I + 1
Print MIN
End
Variables: I, N, NUM, MIN
```

1.6.2 Insert a Card in a List of Sorted Cards

Inserting a card in a list of sorted cards is same as inserting an element into a sorted list of numbers. For this, start from the end of the list and compare the new element with the elements of the list to find a suitable position at which the new element has to be inserted. While comparing, also shift the elements one step ahead to create a vacancy for the new element. Figure 1.3 given below illustrates this concept.

Position	0	1	2	3	4	5	Element To Be Inserted
Original List	4	6	9	10	11		7
7>11 ×	4	6	9	10		11	
7>10 ×	4	6	9		10	11	
7>9 ×	4	6		9	10	11	
7>6 √	4	6	7	9	10	11	

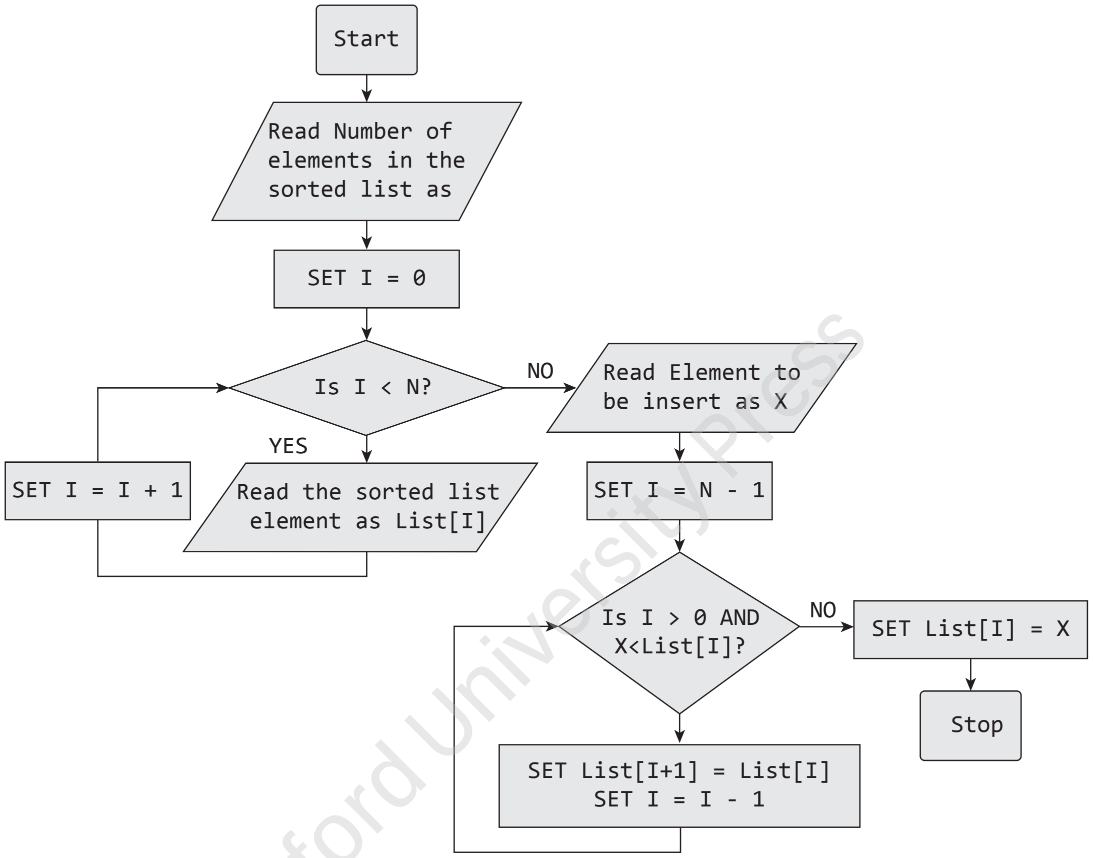
Figure 1.3 Inserting a number in a sorted list

The step-wise approach for solving the problem of inserting a card in a list of sorted cards is represented through Algorithm 1.2, Flowchart 1.2, and Pseudocode 1.2.

Algorithm 1.2

```
Step 1:   Start
Step 2:   READ Number of elements in the sorted list as N
Step 3:   SET I=0
Step 4:   REPEAT Steps 5 and 6 WHILE I < N
Step 5:      READ the Sorted list element as List[I]
Step 6:      SET I = I + 1
Step 7:   READ Element to be inserted as X
Step 8:   SET I = N-1
Step 9:   REPEAT Step 10 and 11 WHILE I >= 0 AND X<List[I]
Step 10:     List[I+1] = List[I]
Step 11:     SET I = I - 1
Step 12:  List[I+1] = X
Step 13:  End
```

Flowchart 1.2



Pseudocode 1.2

```
Start
Read Number of elements in the sorted list as N
Set I=0
WHILE I < N
   Read the Sorted list element as List[I]
   Set I = I + 1
Read element to be inserted as X
Set I = N-1
WHILE I >= 0 AND X<List[I]
   List[I+1] = List[I]
   Set I = I - 1
List[I+1] = X
End
```

1.6.3 Guess an Integer Number in a Range

Let us play a game. While playing this game you will be able to appreciate how two different solutions to the same problem can vary so much in terms of efficiency. The game is that the computer will randomly select an integer from 1 to N and ask you to guess it. To help you guess the number correctly, the computer will tell you if each guess is too high or too low. The good thing is that there is no limit on number of guesses. You can make as many guess as you want to guess that number.

There are two ways to succeed in this game. First is the linear search and second is the binary search. In linear search, you will guess the number as 1, then 2, then 3, then 4, and so on, until you guessed the right number. So you are guessing all the numbers as if they were lined up in a row. This technique is fine but just wonder how many guesses you would have to make. If the computer selects N, you would need N guesses. If N is 1 then, of course, you will make it in the very first guess itself. Even if N is a small number like 5 or 10, then it is still fine but just imagine what will be the number of guesses if N is a large number.

Now let us explore the binary search technique. Since the computer tells you whether a guess is too low, too high, or correct, it is better to start by guessing N/2. If the number selected by the computer is less than N/2, then using the computer's information that the guess is too high, you can eliminate all the numbers from N/2 to N in just one go. If the number selected by the computer is greater than N/2, then all elements from 1 through N/2 are eliminated right away. So with one guess you have narrowed down your possible guesses by half. Isn't that interesting and an intelligent move? Keep on cutting down the set of possible numbers by half with every guess that you make.

While linear search technique required N guesses to win the game, binary search technique, on the other hand, can make you win (or find the number) in at most $\log_2 N+1$ guesses. The following table shows the maximum number of guesses for linear search and binary search for a few number sizes:

VALUE OF N	Max Linear Search Guesses	Max Binary Search Guesses
10	10	4
100	100	7
1,000	1,000	10
10,000	10,000	14
100,000	100,000	17
1,000,000	1,000,000	20

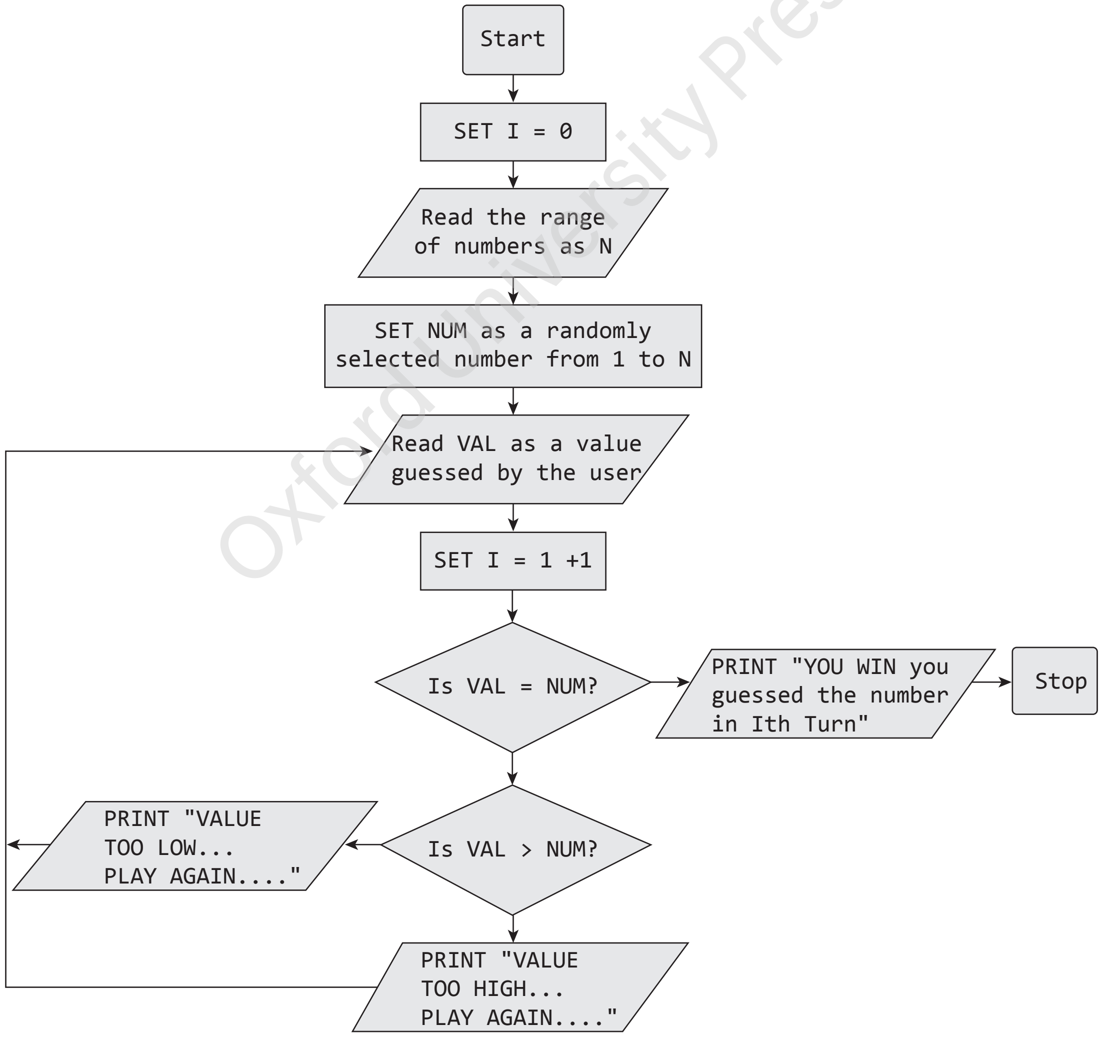
The step-wise approach for solving the problem of guessing an integer number in a given range is demonstrated through Algorithm 1.3, Flowchart 1.3, and Pseudocode 1.3.

Algorithm 1.3

```
Step 1: Start
Step 2: SET I = 0
Step 3: READ the range of numbers as N
Step 4: SET NUM as a randomly selected number from 1 to N
Step 5: READ VAL as a value guessed by the user
Step 6: SET I = I + 1
Step 7: IF VAL = NUM
        THEN PRINT "YOU WIN... You guessed the number in Ith Turn"
        Go to Step 10
```

```
Step 8:   IF VAL > NUM
       THEN PRINT "VALUE TOO HIGH… PLAY AGAIN…."
       Go to Step 5
Step 9: IF VAL < NUM
       THEN PRINT "VALUE TOO LOW… PLAY AGAIN…."
       Go to Step 5
Step 10: End
```

Flowchart 1.3


Start
SET I = 0
Read the range of numbers as N
SET NUM as a randomly selected number from 1 to N
Read VAL as a value guessed by the user
SET I = 1 +1
Is VAL = NUM?
PRINT "YOU WIN you guessed the number in Ith Turn"
Stop
Is VAL > NUM?
PRINT "VALUE TOO LOW... PLAY AGAIN...."
PRINT "VALUE TOO HIGH... PLAY AGAIN...."

Pseudocode 1.3

```
Start
Set I = 0
Read the range of numbers as N
Set NUM as a randomly selected number from 1 to N
Read VAL as a value guessed by the user
Set I = I + 1
While VAL != NUM
IF VAL = NUM
      THEN PRINT "YOU WIN... You guessed the number in Ith Turn"
      Exit
ELSEIF VAL > NUM
      THEN PRINT "VALUE TOO HIGH… PLAY AGAIN…."
   ELSE
      THEN PRINT "VALUE TOO LOW… PLAY AGAIN…."
End
```

1.6.4 Tower of Hanoi

The tower of Hanoi is one of the main applications of recursion. It says, 'if you can solve n–1 cases, then you can easily solve the nth case'.

We will discuss the tower of Hanoi problem using one, then two, and finally three rings. Figures 1.4(a) and (b) explain the working of the tower of Hanoi problem using one and two rings.

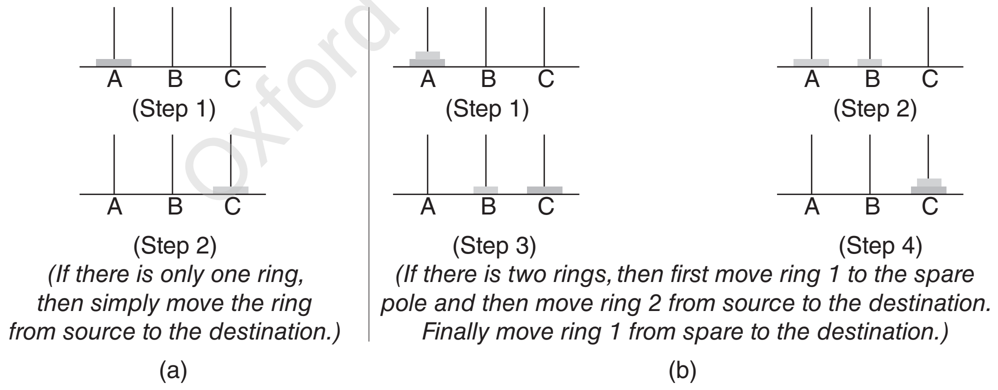

A B C
(Step 1)
A B C
(Step 2)
(If there is only one ring, then simply move the ring from source to the destination.)
(a)
A B C
(Step 1)
A B C
(Step 2)
A B C
(Step 3)
A B C
(Step 4)
(If there is two rings, then first move ring 1 to the spare pole and then move ring 2 from source to the destination. Finally move ring 1 from spare to the destination.)
(b)

Figure 1.4 Working of tower of Hanoi problem using one and two rings

Now, consider at Figure 1.5 which shows three rings mounted on pole A. The problem is to move all these rings from pole A to pole C while maintaining the same order. The main issue is that the smaller disk must always come above the larger disk.

Rules of this problem include

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

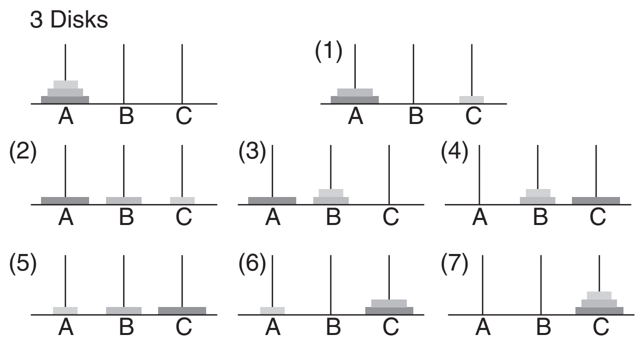


Figure 1.5 Solving the Tower of Hanoi problem

We will be solving this using a spare pole. In our case, A is the source pole, C is the destination pole, and B is the spare pole. To transfer all the three rings from A to C, we will first shift the upper two rings (n–1 rings) from the source pole to the spare pole (refer to Steps 1, 2 and 3 of Figure 1.5). Now that n–1 rings have been removed from pole A, the nth ring can be easily moved from the source pole (A) to the destination pole (C) (refer to Step 4 of Figure 1.5). The final step is to move the n–1 rings from the spare pole (B) as well as pole (A) to the destination pole (C) (refer to Steps 5, 6, and 7 of Figure 1.5).

To summarize, the solution to our problem of moving n rings from A to C using B as spare can be given as:

Base case: If n=1
Move the ring from A to C using B as spare
Recursive case:
Move n – 1 rings from A to B using C as spare
Move the one ring left on A to C using B as spare

We can conclude that the steps to solve the tower of Hanoi problems are:

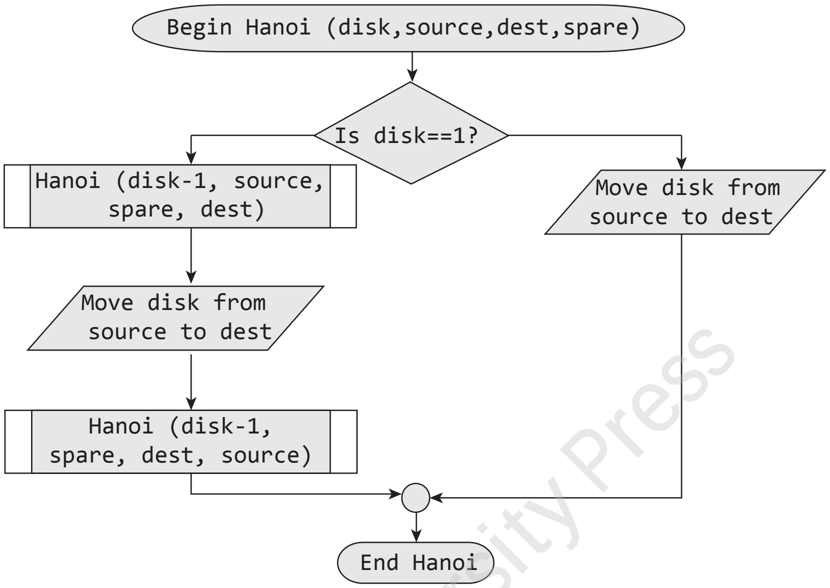
Step 1 − Move n-1 disks from source to the spare pole
Step 2 − Move nth disk from source to destination pole
Step 3 − Move n-1 disks from the spare pole to the destination pole

The step-wise approach for solving the problem of tower of Hanoi is demonstrated through Algorithm 1.4, Flowchart 1.4, and Pseudocode 1.4.

Algorithm 1.4 Hanoi(disk, source, dest, spare)

```
Step 1: Begin
Step 2: IF disk = 1
             THEN Go to step 3
        ELSE
             Go to step 4
Step 3: Move disk from source to dest and then Go to Step 7
Step 4: CALL Hanoi(disk-1, source, spare, dest)
Step 5: Move disk from source to dest
Step 6: CALL Hanoi(disk-1, spare, dest, source)
Step 7: End
```

Flowchart 1.4


Begin Hanoi (disk,source,dest,spare)
Is disk==1?
Hanoi (disk-1, source, spare, dest)
Move disk from source to dest
Hanoi (disk-1, spare, dest, source)
Move disk from source to dest
End Hanoi

Pseudocode 1.4

```
Begin Procedure Hanoi(disk, source, dest, spare)
IF disk = 1
      THEN Move disk from source to dest
ELSE
CALL Hanoi(disk -1, source, spare, dest)
Move disk from source to dest
CALL Hanoi(disk -1, spare, dest, source)
End Procedure Hanoi(disk, source, dest, spare)
```

Note: Refer to Program 4.17 for the code demonstrating the implementation of the solution of the tower of Hanoi problem.

Types of Errors

While writing programs, very often we get errors in our programs. These errors if not removed will either give erroneous output or will not let the compiler to compile the program. These errors are broadly classified under four groups as shown in Figure 1.6.

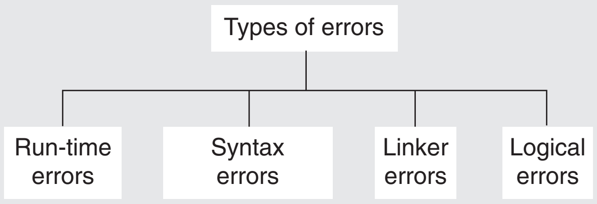


Figure 1.6 Types of errors

Run-time Errors As the name suggests, run-time errors occur when the program is being run executed. Such errors occur when the program performs some illegal operations like

- dividing a number by zero
- opening a file that already exists

- lack of free memory space
- finding square or logarithm of negative numbers

Run-time errors may terminate program execution, so the code must be written in such a way that it handles all sorts of unexpected errors rather terminating it unexpectedly.

Syntax Errors Syntax errors (also known as compile-time errors) are generated when rules of a programming language are violated. Python interprets (executes) each instruction in the program line by line. The moment the interpreter encounters a syntactic error, it stops further execution of the program.

Semantic or Logical Errors Semantic errors are those errors which may comply with rules of the programming language but gives an unexpected and undesirable output which is obviously not correct. For example, if you write a program to add two numbers but instead of writing '+' symbol, you put the '–' symbol. Then Python will subtract the numbers and returns the result. But, actually the output is different from what you expected.

Logical errors are errors in the program code. Such errors are not detected by the compiler, and programmers must check their code line by line or use a debugger to locate and rectify the errors. Logical errors occur due to incorrect statements.

Linker Errors These errors occur when the linker is not able to find the function definition for a given prototype.

Summary

- *Programming* means writing computer programs. While programming, the programmers takes an algorithm and code the instructions in a particular programming language, so that it can be executed by a computer.
- An algorithm provides a blueprint to writing a program to solve a particular problem.
- In the course of processing, data is read from an input device, stored in computer's memory for further processing and then the result of the processing is written to an output device.
- The data is stored in the computer's memory in the form of variables or constants.
- For a complex problem, its algorithm is often divided into smaller units called *functions* (or modules).
- Algorithm validation ensures that the algorithm will work correctly irrespective of the programming language in which it will be implemented.
- An algorithm is analysed to measure its performance in terms of CPU time and memory space required to execute that algorithm.
- Sequence means that each step of the algorithm is executed in the specified order.
- Decision statements are used when the outcome of the process depends on some condition.
- Iteration or Repetition involves executing one or more steps for a number of times, can be implemented using constructs such as the while loops and for loops.
- Programming languages have a vocabulary of syntax and semantics for instructing a computer to perform specific tasks.
- Pseudocode facilitates designers to focus on the logic of the algorithm without getting bogged down by the details of language syntax.

Glossary

Program A set of instructions that tells the computer how to solve a particular problem.
Algorithm A step by step instructions that tells the computer how to solve a particular problem.
Programming The act of writing programs.
Instruction A single operation which when executed convert one state to other.
Modularization The process of dividing an algorithm into modules/functions.
Programming language A language specifically designed to express computations that can be performed by a computer.
Pseudocode A compact and informal high-level description of an algorithm that uses the structural conventions of a programming language.
Flowchart A diagrammatic representation that illustrates the sequence of steps that must be performed to solve a problem.
Recursion A technique of solving a problem by breaking it down into smaller and smaller sub-problems until you get to a small enough problem that it can be easily solved.

Exercises

Fill in the blanks

1. A ________is a set of instructions that tells the computer how to solve a particular problem.
2. Algorithms are mainly used to achieve _________.
3. _____ and _______ statements are used to change the sequence of execution of instructions.
4. ______ is a formally defined procedure for performing some calculation.
5. ______ statements are used when the outcome of the process depends on some condition.
6. Repetition can be implemented using constructs such as ______, ______, and ______.
7. A complex algorithm is often divided into smaller units called ______.
8. The ______ symbol is always the first and the last symbol in a flowchart.
9. ______ is a form of structured English that describes algorithms.
10. ______ is used to express algorithms and as a mode of human communication.
11. The process of dividing an algorithm into modules/ functions is called _________.
12. _________is a technique of solving a problem by breaking it down into smaller and smaller sub-problems until you get to a small enough problem that it can be easily solved.

State True or False

1. An algorithm solves a problem in a finite number of steps.
2. Flowcharts are drawn in the early stages of formulating computer solutions.
3. The main focus of pseudocodes is on the details of the language syntax.
4. Algorithms are implemented using a programming language.
5. Repetition means that each step of the algorithm is executed in a specified order.
6. Terminal symbol depicts the flow of control of the program.
7. Labelled connectors are square in shape.
8. The outputs of each step of an algorithm should be unambiguous. This means that is should be precise.
9. You can have maximum one function in a algorithm.
10. Pseudocode is written using the syntax of a particular programming language.

Multiple Choice Questions

1. Algorithms should be
   (a) precise (b) unambiguous
   (c) clear (d) all of these
2. To check whether a given number is even or odd, you will use which type of control structure?
   (a) sequence (b) decision
   (c) repetition (d) all of these
3. Which one of the following is a graphical or symbolic representation of a process?
   (a) algorithm (b) flowchart
   (c) pseudocode (d) program
4. In a flowchart, which symbol is represented using a rectangle?
   (a) terminal (b) decision
   (c) activity (d) input/output
5. Which of the following details are omitted in pseudocodes?
   (a) variable declaration
   (b) system specific code
   (c) subroutines
   (d) all of these
6. A single operation which when executed convert one state to other is called ________.
   (a) instruction (b) program
   (c) software (d) control
7. Algorithm is validated to
   (a) measure its CPU time
   (b) measure the memory consumed
   (c) check if it is correct
   (d) all of these
8. Programming languages have a vocabulary of _____ for instructing a computer to perform specific tasks.
   (a) syntax (b) semantics
   (c) both of these (d) none of these
9. Syntax and semantics of a language are strictly checked in ________.
   (a) algorithm (b) flowchart
   (c) pseudocode (d) program

Review Questions

1. Define an algorithm. How is it useful in the context of software development?
2. Explain and compare the approaches for designing an algorithm.
3. What is modularization?
4. Explain sequence, repetition, and decision statements. Also give the keywords used in each type of statement.
5. With the help of an example, explain the use of a flowchart.
6. How is a flowchart different from an algorithm? Do we need to have both of them for program development?
7. What do you understand by the term pseudocode?
8. Differentiate between algorithm and pseudocodes.
9. Give the characteristics of an algorithm.
10. What do you understand by the term recursion?
11. Write an algorithm and draw a flowchart that calculates salary of an employee. Prompt the user to enter the Basic Salary, HRA, TA, and DA. Add these components to calculate the Gross Salary. Also deduct 10% salary from the Gross Salary to be paid as tax.
12. Draw a flowchart and write an algorithm and a psuedocode for the following problem statements
    (a) Cook maggi
    (b) Cross road
    (c) Calculate bill of items purchased
    (d) To find out whether a number is positive or negative
    (e) Print "Hello" five times on the screen
    (f) Find area of a rectangle
    (g) Convert meters into centimeters
    (h) Find the sum of first 10 numbers

Answers

Fill in the Blanks

1. Program	4. Algorithm	7. functions or modules	10. Flowcharts
2. software reuse	5. Decision	8. Terminal symbols	11. Modularization
3. Decision, repetition	6. while, do-while, and for loops	9. Psuedocode	12. Recursion

State True or False

1. True 2. True 3. True 4. False 5. False 6. False 7. False 8. False 9. False 10. False

Multiple Choice Questions

1. (d) 2. (b) 3. (b) 4. (c) 5. (d) 6. (a) 7. (c) 8. (c) 9. (d)