DESIGN AND ANALYSIS OF ALGORITHMS

S. Sridhar
Associate Professor
Department of Information Science and Technology
College of Engineering, Guindy Campus
Anna University, Chennai

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
YMCA Library Building, 1 Jai Singh Road, New Delhi 110001, India

First published in 2014

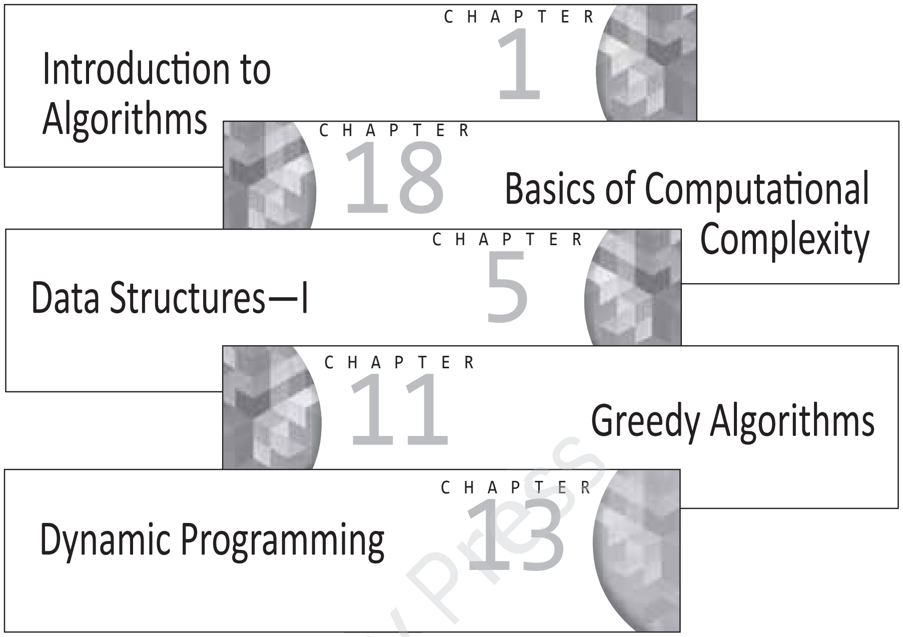
ISBN-13: 978-0-19-809369-5
ISBN-10: 0-19-809369-1

Typeset in Times New Roman
by Ideal Publishing Solutions, Delhi
Printed in India by Radha Press, New Delhi 110031

Third-party website addresses mentioned in this book are provided
by Oxford University Press in good faith and for information only.
Oxford University Press disclaims any responsibility for the material contained therein.

Features of the Book

Topical Coverage
The book provides extensive coverage of topics starting from algorithm basics and data structures, moving on to different algorithm design techniques, followed by discussions on computational complexities and PRAM algorithms.



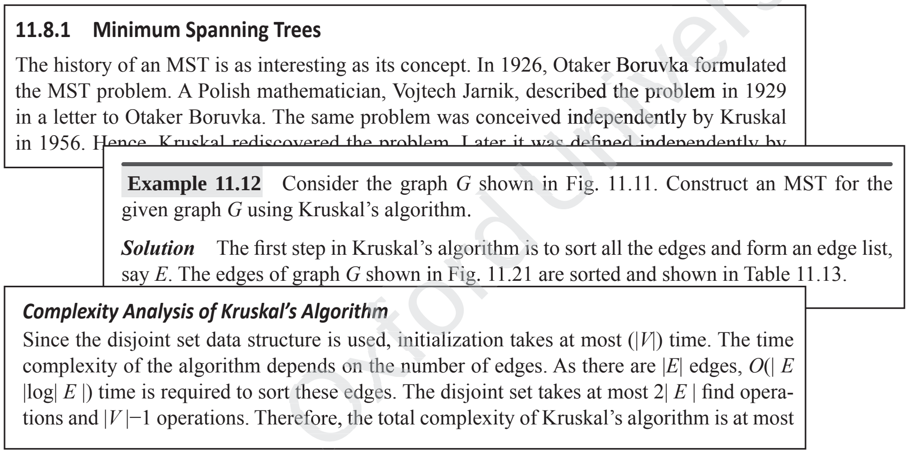
CHAPTER 1
Introduction to Algorithms

CHAPTER 18
Basics of Computational Complexity

CHAPTER 5
Data Structures—I

CHAPTER 11
Greedy Algorithms

CHAPTER 13
Dynamic Programming



11.8.1 Minimum Spanning Trees

The history of an MST is as interesting as its concept. In 1926, Otaker Boruvka formulated the MST problem. A Polish mathematician, Vojtech Jarnik, described the problem in 1929 in a letter to Otaker Boruvka. The same problem was conceived independently by Kruskal in 1956. Hence, Kruskal rediscovered the problem. Later it was defined independently by

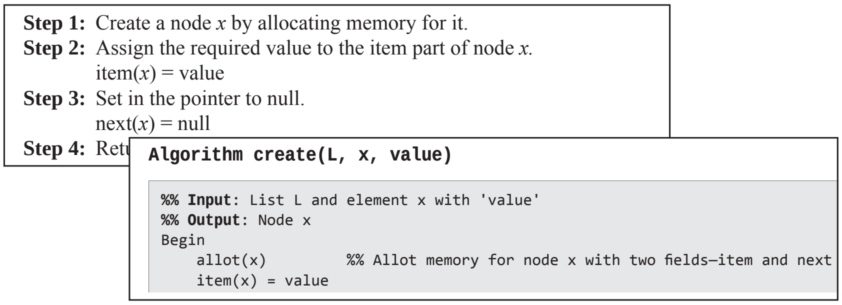
Example 11.12 Consider the graph G shown in Fig. 11.11. Construct an MST for the given graph G using Kruskal's algorithm.

Solution The first step in Kruskal's algorithm is to sort all the edges and form an edge list, say E. The edges of graph G shown in Fig. 11.21 are sorted and shown in Table 11.13.

Complexity Analysis of Kruskal's Algorithm
Since the disjoint set data structure is used, initialization takes at most ($|V|$) time. The time complexity of the algorithm depends on the number of edges. As there are $|E|$ edges, $O(|E| \log|E|)$ time is required to sort these edges. The disjoint set takes at most $2|E|$ find operations and $|V|-1$ operations. Therefore, the total complexity of Kruskal's algorithm is at most

Treatment of Concepts
Simple and logical explanations of concepts are provided using essential mathematical expressions, illustrations, and numerous solved examples. Different types of problems along with their algorithms, examples, and complexity analyses are given.

Algorithm Presentation
Algorithms are presented in two ways, that is, *step-wise approach* (informal representation) and *pseudocode approach* (formal representation) for providing a better understanding of the logic behind solving a given problem without using any specific programming language.



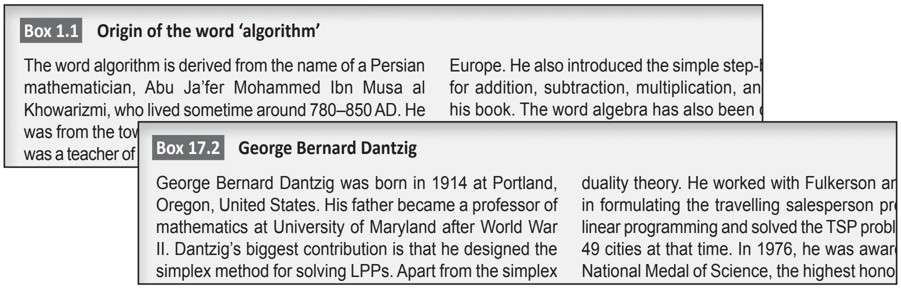
```
Step 1: Create a node x by allocating memory for it.
Step 2: Assign the required value to the item part of node x.
        item(x) = value
Step 3: Set in the pointer to null.
        next(x) = null
Step 4: Retu
```

Algorithm create(L, x, value)

```
%% Input: List L and element x with 'value'
%% Output: Node x
Begin
    allot(x)          %% Allot memory for node x with two fields—item and next
    item(x) = value
```

Historical Notes

Historical notes are provided throughout the book as separate boxes. They provide additional information related to the concepts discussed.



Box 1.1 **Origin of the word 'algorithm'**

The word algorithm is derived from the name of a Persian mathematician, Abu Ja'fer Mohammed Ibn Musa al Khowarizmi, who lived sometime around 780–850 AD. He was from the tov was a teacher of

Europe. He also introduced the simple step- for addition, subtraction, multiplication, an his book. The word algebra has also been

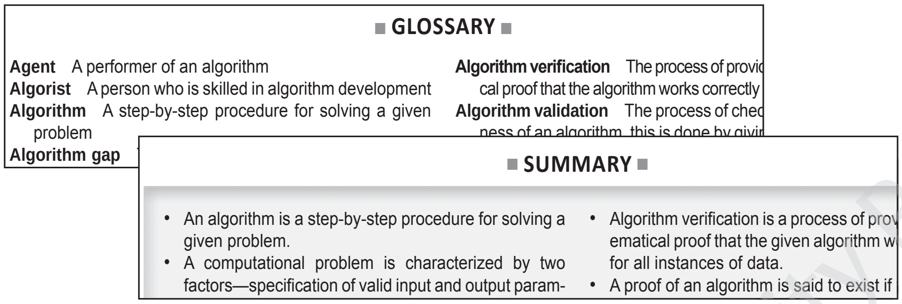
Box 17.2 **George Bernard Dantzig**

George Bernard Dantzig was born in 1914 at Portland, Oregon, United States. His father became a professor of mathematics at University of Maryland after World War II. Dantzig's biggest contribution is that he designed the simplex method for solving LPPs. Apart from the simplex

duality theory. He worked with Fulkerson ar in formulating the travelling salesperson pr linear programming and solved the TSP probl 49 cities at that time. In 1976, he was awar National Medal of Science, the highest hono

Glossary and Summary

A glossary of key terms along with definitions and a point-wise summary is given at the end of each chapter to help readers recapitulate the important concepts explained.



■ **GLOSSARY** ■

Agent A performer of an algorithm

Algorist A person who is skilled in algorithm development

Algorithm A step-by-step procedure for solving a given problem

Algorithm gap

Algorithm verification The process of provi cal proof that the algorithm works correctly

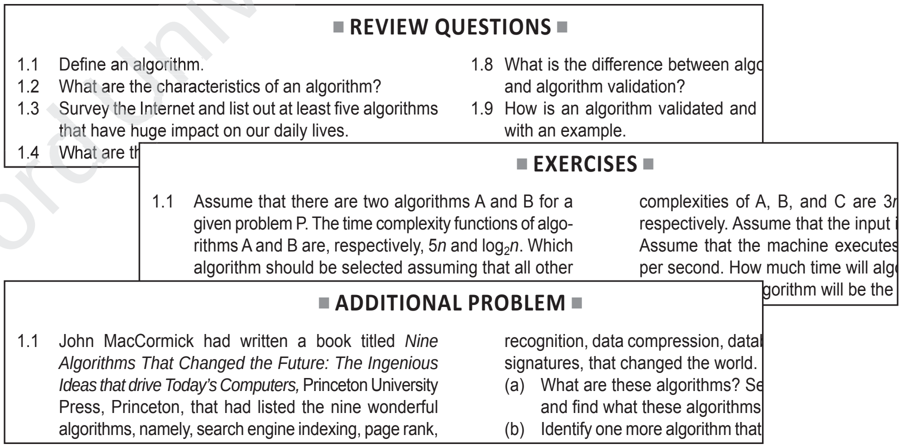
Algorithm validation The process of chec ness of an algorithm, this is done by givin

■ **SUMMARY** ■

- An algorithm is a step-by-step procedure for solving a given problem.
- A computational problem is characterized by two factors—specification of valid input and output param-
- Algorithm verification is a process of prov ematical proof that the given algorithm w for all instances of data.
- A proof of an algorithm is said to exist if

Review Questions, Exercises, and Additional Problems

Numerous review questions, exercises, and additional concept-based problems are given at the end of every chapter to test the readers' conceptual knowledge and also enhance their algorithm writing skills.



■ **REVIEW QUESTIONS** ■

1.1 Define an algorithm.

1.2 What are the characteristics of an algorithm?

1.3 Survey the Internet and list out at least five algorithms that have huge impact on our daily lives.

1.4 What are th

1.8 What is the difference between algo and algorithm validation?

1.9 How is an algorithm validated and with an example.

■ **EXERCISES** ■

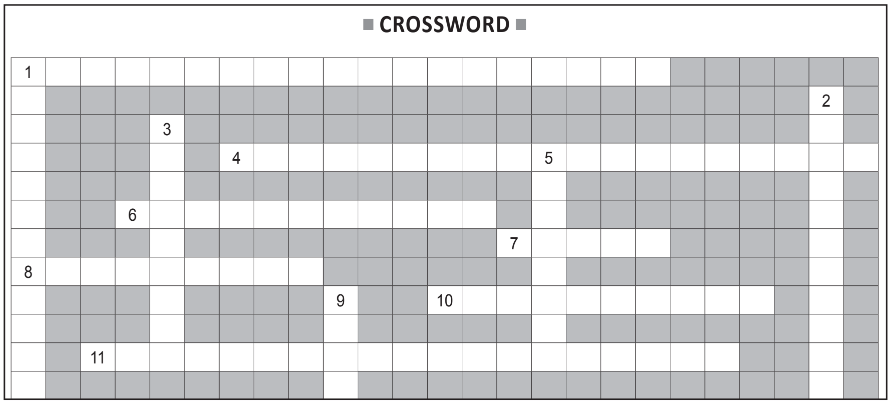
1.1 Assume that there are two algorithms A and B for a given problem P. The time complexity functions of algorithms A and B are, respectively, 5*n* and $\log_2 n$. Which algorithm should be selected assuming that all other complexities of A, B, and C are 3*n* respectively. Assume that the input i Assume that the machine executes per second. How much time will alg gorithm will be the

■ **ADDITIONAL PROBLEM** ■

1.1 John MacCormick had written a book titled *Nine Algorithms That Changed the Future: The Ingenious Ideas that drive Today's Computers,* Princeton University Press, Princeton, that had listed the nine wonderful algorithms, namely, search engine indexing, page rank, recognition, data compression, datal signatures, that changed the world.

(a) What are these algorithms? Se and find what these algorithms

(b) Identify one more algorithm that



■ **CROSSWORD** ■

Crossword Puzzles

Crossword puzzles, given as an exciting and interactive learning exercise at the end of each chapter, motivate and aid readers to self-check their understanding.

Preface

Computers have become an integral part of our daily life in recent times. They have enormously impacted our personal, professional, as well as social lives. Computers help us in tasks such as document editing, Internet browsing, sending emails, making presentations, performing complex scientific computations, social networking, and playing games. Industries and government offices use computers effectively in production, e-governance, and e-commerce. Considering the increasing demand of computers in society, schools, colleges, and universities have included computer education in their curriculum, to help students become skilled in programming and developing applications which can be used to solve various business, scientific, and social problems.

Programming is a process of converting a given problem into an executable code for the computer. It involves understanding, analysis, and solving problems to create an algorithm. Verification of the algorithm, coding of the algorithm in a specific programming language, testing, debugging, and maintaining the source code are also part of the programming process. Therefore, in order to construct efficient programs, a fine understanding of algorithms is essential. An algorithm is a set of logical instructions for solving a given problem. It is expected to give correct results for valid inputs and should be efficient, consuming less computer resources. An algorithm is implemented as a program using a programming language. A well-designed algorithm runs faster and consumes lesser computer resources, namely time and space. Therefore, expertise in programming is more related to efficiency in problem-solving and effective designing of algorithms, rather than developing codes with the help of programming languages. Though programming languages are important, their role is just limited to the implementation of a well-designed algorithm. For this reason, algorithms are a central theme of computer study.

In fact, history of algorithms is much older than that of computers, dating back to 3000 BC. The ancient people of the Sumerian civilization were aware of basic numeric computation like addition. A Sumerian tablet found in the Euphrates river showed how to partition a given quantity of wheat in a way that each person receives the specified quantity. Such tablets were also used by ancient Babylonians (2000 to 1650 BC). Mathematicians of this period such as Euclid, Al-Khwarizmi, Leonardo Pisano (also known as Fibonacci), and others produced procedures that provided the foundations of the concepts of algorithms. Later developments in the field of algorithms were due to the contributions of Gottfried Leibnitz, David Hilbert, and Alan Turing. The history of modern computers, however, only starts from the 1940s. Thus, algorithms have played a very important role in the development of modern computing.

The study of algorithms, called algorithmics, includes three aspects—algorithm design, analysis, and computational complexity of problems. Algorithm design is a creative activity. It includes various techniques (such as divide-and-conquer, greedy approach, dynamic programming, backtracking) that help in producing outputs at a faster pace by consuming lesser computer resources. Algorithm analysis is the estimation of how much resource is required by the algorithm. Computational complexity deals with the analysis and solvability of problems itself. A compulsory course on algorithms design and analysis is generally offered to computer science and information technology students in most universities. The course aims to help students create efficient algorithms in common engineering design situations and analyse the asymptotic performance of algorithms by using the important algorithmic design paradigms and methods of analysis.

ABOUT THE BOOK

Design and Analysis of Algorithms is designed to serve as a textbook for the first level course in algorithms that discusses all the fundamental and necessary information related to the three important aspects of algorithm study using minimal mathematics and lucid language. This book is suitable for undergraduate students of computer science and engineering (CSE) and information technology (IT), as well as for postgraduate students of computer applications. It is also useful for diploma courses, competitive examinations (like GATE), and recruitment interviews for this subject.

The book begins with an introduction to algorithms and problem-solving concepts followed by an introduction to algorithm writing and analysis of iterative and recursive algorithms. In-depth explanations and designing techniques of various types of algorithms used for problem-solving such as the brute force technique, divide-and conquer-technique, decrease-and-conquer strategy, greedy approach, transform-and-conquer strategy, dynamic programming, branch-and-bound approach, and backtracking, are provided in the book. Subsequent chapters of the book delve into the discussion of string algorithms, iterative improvement, linear programming, computability theory, NP-hard problems, NP- completeness, probability analysis, randomized algorithms, approximation algorithms, and parallel algorithms, with the appendices throwing light on basic mathematics and proof techniques.

The various design techniques have been elucidated with the help of numerous problems, solved examples, and illustrations (including schematics, tables, and cartoons). The algorithms are presented in plain English (informal algorithm presentation) and pseudocode approach (formal algorithm presentation) to make the book programming language-independent and easy-to-comprehend. The book includes a variety of chapter-end pedagogical features such as point-wise summary, glossary, review questions, exercises, and additional problems to help readers assimilate and implement the concepts learnt.

KEY FEATURES

- Provides simple and coherent explanations without using excessive theorems, proofs, and lemmas
- Detailed coverage for topics such as greedy approach, dynamic programming, transform-and-conquer technique, decrease-and-conquer technique, linear programming, and randomized and approximation algorithms
- Dedicated chapters on backtracking and branch-and-bound techniques, string matching algorithms, and parallel algorithms
- Simple and judicious presentation of algorithms throughout the text in both informal and formal forms, followed by the discussion of their complexity analysis
- Numerous review questions, exercises, and additional problems given at the end of each chapter to help readers apply and practise the concepts learnt
- Includes glossary and point-wise summary at the end of each chapter to help readers quickly recapitulate the important concepts
- Provides historical notes on various topics and crossword puzzles at the end of each chapter to elicit learning interest in students

ORGANIZATION OF THE BOOK

The book consists of twenty chapters. A chapter-wise scheme of the book is presented here.

Chapter 1 provides an overview of algorithms. It introduces all the basic concepts of algorithms and the fundamental stages of problem-solving. The chapter ends with the classification of algorithms.

Chapter 2 starts with the basic tools used for problem-solving. All the guidelines required for presenting the pseudocode and flow charts are provided along with many examples. The focus of this chapter is to provide some practice on writing algorithms. The basics of recursion and algorithm correctness are also covered in this chapter.

Chapter 3 covers the basics of algorithm complexity and analysis of iterative algorithms. Step count and operation count methods used for analysing iterative algorithms are discussed in detail. Asymptotic analysis is also discussed in detail. Finally, the chapter ends with the concept of analysing the efficiency of algorithms.

Chapter 4 discusses the analysis of recursive algorithms. It explains the basics of recurrence equations along with the methods for solving them. Generating functions are also briefly discussed as part of this chapter.

Data structures are an important component of algorithms. *Chapter 5* deals with the fundamentals concepts related to data structures such as stacks, queues, linear lists, and linked lists. Trees and graphs are also explained in this chapter. *Chapter 6* covers advanced data structures used for organizing large amounts of dynamic data. Binary search trees and AVL trees used for organizing dictionaries are discussed in this chapter, in addition to priority queues and heaps. Finally, a discussion on disjoint sets and amortized analysis is provided.

Chapter 7 deals with the brute force techniques, which use no special logic but instead follow an intuitive way of solving problems using the problem statement. Various problems such as sequential search, bubble sort, and selection sort that can be solved using this approach are explained. Some basic computational geometry problems such as closest-pair and convex hull are also covered. The chapter ends with the discussions on exhaustive searching problems such as 15-puzzle problem, 8-queen problem, magic square problem, knapsack problem, container loading problem, and assignment problem.

Chapter 8 discusses the divide-and-conquer design paradigm. Important problems such as quicksort, merge sort, finding maximum and minimum, multiplication of long integers, Strassen matrix multiplication, tiling problem, closest-pair problem, and convex hull are solved using this technique. The chapter ends with a discussion of the Fourier transform problem.

Chapter 9 explains the decrease-and-conquer technique, which is also known as the incremental or inductive approach. Examples problems such as insertion sort, topological sort, generating permutations and subsets, binary search, fake coin detection, and Russian peasant multiplication problem are used to illustrate the decrease by constant and constant factor methods. Finally, discussions on interpolation search, selection, and finding the median problems using the decrease by variable factor method are provided.

Chapter 10 deals with time–space tradeoffs. The selection of one type of efficiency over the other and problems related to linear sorting and Hashing are discussed. The chapter ends with a discussion on B-trees and their operations.

Chapter 11 explains the greedy approach concept. This chapter discusses important problems such as coin change problem, scheduling, knapsack problem, optimal storage of tapes, Huffman code, minimum spanning tree algorithms, and Dijkstra's shortest path algorithm to illustrate the greedy approach.

Chapter 12 discusses the transform-and-conquer approach and its three basic techniques—instance simplification, representation change, and problem reduction. Problems such as Gaussian elimination, decomposition methods, finding determinant and matrix inverses are discussed. Heap sort, Horner's method, binary exponentiation algorithm, and reduction problems are also covered in this chapter.

Chapter 13 deals with dynamic programming. Important example problems such as Fibonacci problem, binomial coefficient multistage graph, Graph algorithms, Floyd-Warshall Algorithm, Bellman–Ford algorithms, travelling salesman problem, chain matrix multiplication, knapsack problem, and optimal binary search tree problem are discussed to illustrate the dynamic programming concept. Finally, the chapter concludes with the flow-shop scheduling algorithms.

Chapter 14 discusses backtracking algorithms. This chapter covers important problems such as *N*-queen problem, Hamiltonian circuit problem, sum of subsets, vertex colouring problem, graph colouring problems, Graham scan, and generating permutations.

Chapter 15 explains branch-and-bound techniques. Search techniques using this concept are discussed. Important problems such as assignment problem and 15-puzzle are covered and the chapter ends with a discussion on traveling salesperson and knapsack problems.

Chapter 16 deals with the string algorithms. Some basic string algorithms such as finding the length of strings, finding substrings, concatenation of two strings, longest common sequence, and pattern recognition algorithms such as Rabin–Karp, Harspool, Knuth–Morris–Pratt and Boyer–Moore algorithms are discussed in this chapter. Finally, approximate string matching algorithm is discussed.

Chapter 17 discusses the iterative approach and basics of linear programming. The linear programming formulation of a problem and simplex method is discussed in this chapter. Minimization problem, principle of duality, and max-flow problems are also explained. Finally, matching algorithms are considered for better understanding of computational complexity.

Chapter 18 explains the basics of computational complexity and the upper and lower bound theory. Decision problems, complexity classes, and reduction concepts are also discussed. This chapter also covers theory of NP-complete problems and examples for proving NP-completeness.

Chapter 19 covers the basic concepts and types of both randomized and approximation algorithms. Randomized algorithms are illustrated through examples such as hiring problem, primality testing, comparison of strings, and randomized quicksort. Approximation algorithms are illustrated through examples based on heuristic, greedy, linear, and dynamic programming approaches.

Chapter 20 begins with an introduction to parallel processing and classification of parallel systems. It then discusses the fundamentals of parallel algorithms and parallel random access machine (PRAM) model. The concept of parallelism is illustrated through examples related to parallel searching, parallel sorting, and graph and matrix multiplication problems.

There are two appendices in this book. *Appendix A* explains the basics of mathematics such as sets, series and sequences, relations, functions, matrix algebra, and probability that are necessary for algorithm study. *Appendix B* deals with mathematical logic and proof techniques.

ONLINE RESOURCES

To aid teachers and students, the book is accompanied by online resources that are available at http://oupinheonline.com/book/sridhar-Design-Analysis-Algorithms/9780198093695. The content for the online resources are as follows:

For Instructors

- PowerPoint slides
- Solutions manual

For Students

- Answers to the crossword puzzles

ACKNOWLEDGEMENTS

This book would not have been possible without the help and encouragement of many friends, colleagues, and well-wishers. I thank all my students who motivated me by asking interesting questions related to the subject. I acknowledge the assistance of my friend, P. Kanniappan, in typing the manuscripts and for all his advice. I also express my gratitude to all my colleagues at the departments of Computer Science and Engineering and Information Science and Technology, Anna University and National Institute of Technology, Tiruchirapalli for reviewing my manuscripts and providing constructive suggestions for improvement. I acknowledge all my reviewers whose feedback has helped making this book better. I am also very thankful to the editorial team at Oxford University Press, India for providing valuable assistance. My sincere thanks are also due to my family members, especially my wife, Dr N. Vasanthy, my mother, Mrs Parameswari, my mother-in-law, Mrs Renuga, and my children, Shobika and Shreevarshika, for their constant support and encouragement during the development of this book.

S. Sridhar

Brief Contents

Detailed Contents

CHAPTER 1

Introduction to Algorithms

"Ideas are the beginning point of all fortunes."

—Napolean Hill

Learning Objectives

This chapter introduces the basics of algorithms. All important definitions and concepts related to the study of algorithms are the focus of this chapter. The reader would be familiar with the following concepts by the end of this chapter:

- Basic terminologies of algorithm study
- Need for algorithms
- Characteristics of algorithms
- Stages of a problem-solving process
- Need for efficient algorithms
- Classification of algorithms

1.1 INTRODUCTION

Computers are powerful tools of computing. One cannot ignore the impact of computers on our modern life. We use computers for personal needs such as typing documents, browsing the Internet, sending emails, playing computer games, performing numeric calculations, and so on. Industries and governments use computers much more effectively to perform complicated tasks to improve productivity and efficiency. Applications of computer systems in airline reservation, video surveillance, biometric recognition, e-governance, and e-commerce are all examples of their usefulness in improving efficiency and productivity. The increasing importance of computers in our lives has prompted schools and universities to introduce computer science as an integral part of our modern education. Informally, everyone is expected to handle computers to accomplish certain basic tasks. This knowledge of using computers to perform our day-to-day activities is often called *computational thinking*. Computational thinking is a necessity to survive in this modern world. However, computer science professionals are expected to accomplish much more than acquiring this basic skill of computer usage. They are required to write specific computer programs to provide computer-automated solutions for problems. Writing a program is slightly more complicated than merely learning to use computers, as it requires 'algorithmic thinking'. *Algorithmic thinking* is an important analytical skill that is required for writing effective programs in order to solve given problems. Algorithmic thinking is not confined to computer science only, but it spreads through all disciplines of study. Computer science is a domain where the skills of algorithmic thinking are taught to the aspiring computer professionals for solving computational problems.

1.2 NEED FOR ALGORITHMIC THINKING

Norman E. Gibbs and Allen B. Tucker[1] proposed a definition for computer science that captures the core truth of computer science study. According to

[1] Gibbs, Norman E., Allen B. Tucker, 'A Model Curriculum for a Liberal Arts Degree in Computer Science', *Communications of ACM*, Vol. 29, Issue 3, 1986.

them, 'Computer science is the systematic study of algorithms and data structures, specifically their formal properties, their mechanical and linguistic realizations, and their applications.' Hence, there cannot be any dispute regarding the fact that study of algorithms is the central theme of computer science.

Let us elaborate this definition further. The formal and mathematical properties of algorithms include the study of algorithm correctness, algorithm design, and algorithm analysis for understanding the behaviour of algorithms. Hardware realizations include the study of computer hardware, which is necessary to run the algorithms in the form of programs. Linguistic realizations include the study of programming languages and their design, translators such as interpreters and compilers, and system software tools such as linkers and loaders, so that the algorithms can be executed by hardware in the form of programs. Applications of algorithms include the study of design and development of efficient software packages and software tools so that these algorithms can be used to solve specific problems.

Thus, computer scientists consider the study of algorithms as the core theme of computer science. The art of designing, implementing, and analysing algorithms is called *algorithmics*. Algorithmics is a general word that comprises all aspects of the study of algorithms. Let us now attempt to define algorithms formally. An *algorithm* is a set of unambiguous instructions or procedures used for solving a given problem to provide correct and expected outputs for all valid and legal input data (refer to Box 1.1).

An algorithm can also be referred by other terminologies such as *recipe*, *prescription*, *process*, or *computational procedure.* Box 1.2 provides a brief history of algorithms.

Algorithms thus serve as prescriptions of how a computer should carry out instructions to solve problems. Hence, algorithms can be visualized as strategies for solving a problem. One cannot write a program for a given problem without the necessary analytical skills or a strategy for solving the problem. Thus, the knowledge of how to solve a problem is called *algorithmic thinking*.

One can compare a program construction with the construction of a house. The blueprint of the house incorporating all the planning necessary for the construction of the house can be visualized as an algorithm. A computer program is only an algorithm expressed using a programming language such as C or C++. In addition to possessing the essential skills of using a programming language, the ability to conceive a strategy or apply analytical skills are important for solving a problem or constructing a program.

Box 1.1 Origin of the word 'algorithm'

The word algorithm is derived from the name of a Persian mathematician, Abu Ja'fer Mohammed Ibn Musa al Khowarizmi, who lived sometime around 780–850 AD. He was from the town of Khowarazm, now in Uzbekistan. He was a teacher of mathematics in Baghdad. He wrote a book titled *Kitab al Jabr w'al Muqabala* (*Rules of Restoration and Reduction*) and *Algoritmi Numero Indorum* where he introduced the old Arabic–Indian number systems to Europe. He also introduced the simple step-by-step rules for addition, subtraction, multiplication, and division in his book. The word algebra has also been derived from the title of this book. When his book was translated to Latin, his name was quoted as Algorismus from which the word 'algorithm' emerged. Algorithm as a word thus became famous for referring to procedures that are used by computers for solving problems.

Box 1.2 Short history of computing and algorithms

The history of algorithms dates back to ancient civilizations that used numbers. Ancient civilizations such as Egyptian, Indian, Greek, and Chinese were known to have used algorithms or procedures to carry out tasks like simple calculations. This is in contrast to the history of modern computers that were designed in the 1940s only. Thus, the history of algorithms is much older and more exciting. Euclid, who lived in ancient Greece around 400–300 BC, is credited with writing the first algorithm in history for computing the greatest common divisor (GCD). Archimedes created an algorithm for the approximation of the number pi. Eratosthenes introduced to the world the algorithm for finding prime numbers. Averroes (1126–1198) also used algorithmic methods for calculations.

A number of influences led to the formalization of the theory of algorithms. Some of the noteworthy developments include the introduction of Boolean algebra by George Boole (1847) and set theory by Gottlob Frege (1879). Set theory has become a foundation of modern mathematics. The concept of recursion formulated by Kurt Gödel, and contributions of Giuseppe Peano (1888), Alfred North Whitehead, and Bertrand Russell in mathematical logic led to the formalization of algorithm as an independent field of study.

Alan Turing's Turing machines developed in 1936 play a very important role in computational complexity theory. This Turing machine and Alonzo Church's lambda calculus formed the basis for formalization of the complexity theory. Thus, the history of algorithms is much richer than that of computers.

The history of computing is another interesting story. Abacus and other mechanical devices used for computing have been utilized at various stages of human history. A real attempt was made in the 17th century by Leibniz, a German mathematician and philosopher. He invented infinitesimal calculus, generating the idea of computers. In 1822, Charles Babbage introduced the difference engine, which required changing of gears manually to perform calculations. These ideas finally led to the invention of computers in the 1940s. In 1942, the US government designed a machine called ENIAC (electronic numerical integrator and computer). This was succeeded by EDVAC (electronic discrete variable automatic computer) in 1951. In 1952, IBM introduced its mainframe computer. Developments in the hardware domain and declining costs have shifted computer usage from big research and military environments to homes and educational institutions. In 1981, IBM personal computers were introduced for home and office use.

1.3 OVERVIEW OF ALGORITHMS

Algorithms are generic and not limited to computers alone (refer to Box 1.2). We perform many algorithms in our daily life unknowingly. Consider a few of our daily activities, some of which are listed as follows:

1. Searching for a specific book
2. Arranging books based on titles
3. Using a recipe to cook a new dish
4. Packing items in a suitcase
5. Scheduling daily activities
6. Finding the shortest path to a friend's house
7. Searching for a document on the Internet
8. Preparing a CD of compressed personal data
9. Sending messages via email or SMS

We perform many tasks without being aware of their inherent algorithms. For example, consider the activity of searching a word in a dictionary. How do we search? It can be noted that indexing the words in a dictionary reduces the effort of searching significantly. We just open the book, compare the word with the index given, and accordingly decide on the portions of the books to be searched for finding the meaning of that particular word. This kind of a plan or an idea is called a strategy, which is more formally known as an algorithm.

The environment of a typical algorithm is shown in Fig. 1.1. Here, *agent*, which can be a human or a computer, is the performer.

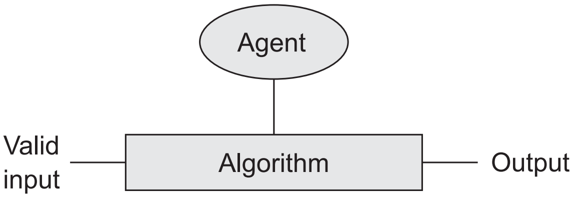


Fig. 1.1 Typical algorithm

To illustrate further, let us consider the example of tea preparation. The hardware used here includes cooking utensils a heater, and a person. To make tea, one needs to have the following ingredients—water, tea powder, and sugar. These ingredients are analogous to the inputs of an algorithm. Here, preparing a cup of tea is called a process. The output of this process is, in this case, tea. This corresponds to the output of an algorithm. The procedure for preparing tea is as follows:

1. Put tea powder in a cup.
2. Boil the water and pour it into the cup.
3. Filter it.
4. Pour milk.
5. Add sugar if necessary.
6. Pour the tea into a cup.

This kind of a procedure can be called an *algorithm*. It can be noted that an algorithm consists of step-by-step instructions that are required to accomplish a certain task.

Humans often perform such procedures intuitively or even mechanically without spending much conscious thought, and hence, they label such actions as habitual activity. Many of our day-to-day activities are not very efficient. However, algorithms that are meant for computers represent a different case altogether. Computer procedures should be efficient as computer resources are scarce. Hence, much thought is given for writing computer procedures that can solve problems. Problems can be classified into two types: computational problems and non-computational problems.

Computational problems can be solved by a computer system. A computational problem is characterized by two factors: (a) the formalization of all legal inputs and expected outputs of a given problem and (b) the characterization of the relationship between problem output and input. Thus, an algorithm is expected to give the expected output for all legal inputs. If an algorithm yields the correct output for a legal input, then it is called an algorithmic solution.

Non-computational problems cannot be solved by a computer system. This classification shows the fundamental differences between computers and humans in solving problems. Computers are more effective than humans in performing calculations and can crunch numbers in a fraction of seconds with more precision and efficiency. However, humans outscore computers in recognition. The ability of recognizing an object by humans is much better than that by machines. Recognition of an object by computer systems requires lots of programming involving images and concepts of image processing. In addition, some tasks are plainly not possible to be carried out by computer systems. Can a computer offer its opinion or show emotion like humans? To put it simply, problems involving more intellectual complexity are much more difficult to solve by computers as computer systems lack intelligence.

Thus, developing algorithms to make computers more intelligent and make them perform tasks like humans becomes much more crucial and challenging. Developing algorithms for computers is both an art and a science. Algorithm design is an art that involves a lot of creative ideas, novelty, and even adventurous strategies and knowledge. Algorithm design is also a science because its construction usually involves the application of some set of principles. The review and knowledge of these principles can facilitate the development of better algorithms, based on sound mathematical and scientific principles.

1.3.1 Computational Problems, Instance, and Size

One encounters many types of computational problems in the domain of computer science. Some of the problems are as follows:

Structuring problems In structuring problems, the input is restructured based on certain conditions or properties. For example, sorting a list in an ascending or a descending order is a structuring problem.

Search problems A search problem involves searching for a target in a list of all possibilities. All potential solutions may be generated and represented in the form of a list or graph. Based on a property or condition, the best solution for the given problem is searched. Puzzles are good examples of search problems where the target or goal is searched in a huge list of possible solutions.

Construction problems These problems involve the construction of solutions based on the constraints associated with the problem.

Decision problems Decision problems are yes/no type of problems where the algorithm output is restricted to answering yes or no. Let us assume that a road network map of a city is given. The problem, say, 'Is there any road connectivity between two cities, say Hyderabad and Chennai?', can be called as a decision problem as the output of this algorithm is restricted to either yes or no.

Optimization problems Optimization problems constitute a very important set of problems that are often encountered in computer science domain. The decision problem about road connectivity between Chennai and Hyderabad can also be posed as an optimization problem as follows: What is the shortest distance between Chennai and Hyderabad? This is an optimization problem, as the problem involves finding the shortest path. Thus, optimization problems involve a certain objective function that is typically of the following form: maximize (say profit) or minimize (say effort) based on a set of constraints that are associated with the problem.

Once the problem is recognized, the input and output of an algorithm should be identified. Consider, for example, a problem of finding the factorial of a number. The factorial of a positive number N can be written as follows: $N! = N \times (N - 1) \times (N - 2) \times \ldots \times 1$. A valid input can be called an *instance* of a problem. For example, factorial of a negative number is not possible. Therefore, all valid positive integers $\{0, 1, 2, \ldots\}$ can serve as inputs and every legal input is called an instance. All possible inputs of a problem are often called a *domain* of the input data. The input should be encoded in a suitable form so that computers can process it. The number of binary bits used to represent the given input, say N, is called the *input size*. Input size is important, as a larger input size consumes more computer time and space.

The core question still remains to be answered: how to solve a given problem? To illustrate problem solving, let us consider a simple problem of counting the number of students in a tuition centre who have passed or failed a test, assuming that the pass mark is 50. Details of the students such as their registration number, name, and more importantly, marks obtained, which are necessary for the given problem, are shown in Table 1.1.

Table 1.1 Students' course marks

Registration number	Student name	Course marks
1	Abraham	80
2	Beena	30
3	Chander	83
4	David	23
5	Elizabeth	90
6	Fauzia	78

The class strength of the tuition centre is 6. The pass mark of the course is given as 50. How do we manually solve this problem? First, we will read the student marks. Thus, the inputs for this problem are a set of student marks. The goal of this problem is to print the pass and fail counts of the students, which is also the output of this algorithm. The process of reading a student's mark is done manually. Compare the student mark with the pass mark, that is, 50. If the student mark is greater than or equal to 50, then pass count should be added by one. Otherwise, the fail count should be added by one. This process is repeated for all the students.

The procedure that is done manually can be given as an algorithm. Therefore, informally, the algorithm for this problem can be given as follows:

```
Step 1: Let counter = 1, number of students = 6
Step 2: While (counter ≤ number of students)
        2.1: Read the marks of the students
        2.2: Compare the marks of the student with 50
        2.3: If student mark is greater than or equal to 50
             Then increment the pass count
             Else increment the fail count
        2.4: Increment the counter
Step 3: Print pass count and fail count
Step 4: Exit
```

Thus, algorithmic solving can be observed to be much similar to how we solve problems manually.

Characteristics of Algorithms

Some of the important characteristics are listed as follows:

Input An algorithm can have zero or more inputs.

Output An algorithm should produce at least one or more outputs.

Definiteness An algorithm is characterized by definiteness. Its instructions should be clear and unambiguous without any confusion. All operations should be well defined. For example, operations involving the division of zero or taking a square root of a negative number are unacceptable.

Uniqueness An algorithm should be a well-defined and ordered procedure that consists of a set of instructions in a specific order. The order of the instructions is important as a change in the order of execution leads to a wrong result or uncertainty.

Correctness The algorithm should be correct.

Effectiveness An algorithm should be effective, which implies that it should be traceable manually.

Finiteness An algorithm should have a finite number of steps and should terminate after executing the finite set of instructions. Therefore, finiteness is an important characteristic of an algorithm.

Some of the additional characteristics that an algorithm is supposed to possess are the following:

Simplicity Ease of implementation is another important characteristic of an algorithm.

Generality An algorithm should be generic, independent of any programming language or operating systems, and able to handle all ranges of inputs; it should not be written for a specific instance.

An algorithm that is definite and effective is also called a *computational procedure*. In addition, an algorithm should be correct and efficient, that is, should work correctly for all valid inputs and yield correct outputs. An algorithm that executes fast but gives a wrong result is useless. Thus, efficiency of an algorithm is secondary compared to program correctness. However, if many correct solutions are available for a given problem, then one has to select the best solution (or the optimal solution) based on factors such as speed, memory usage, and ease of implementation.

Algorithms vs Programs

Algorithms can be contrasted with programs. Algorithms, like blueprints that are used for constructing a house, help solve a given problem logically. A program is an expression of that idea, consisting of a set of instructions written in any programming language. The development of algorithms can also be contrasted with software development. At the industrial level, software development is usually undertaken by a team of programmers who develop software, often for third-party customers, on a commercial basis. Software engineering is a domain that deals with large-scale software development. Project management issues such as team management, extensive planning, cost estimation, and project scheduling are the main focus of software engineering. On the other hand, algorithm design and analysis as a field of study take a micro view of program development. Its focus is to develop efficient algorithms for basic tasks that can serve as a building block for various applications. Often project management issues of software development are not relevant to algorithms.

1.4 NEED FOR ALGORITHM EFFICIENCY

Computer resources are limited. Hence, many problems that require a large amount of resources cannot be solved. One good example is the travelling salesperson problem (TSP). Its brief history is given in Box 1.3.

A TSP is illustrated in Fig. 1.2. It can be modelled as a graph. A graph consists of a set of nodes and edges that interconnect the nodes. In this case, cities are the nodes and the paths that connect the cities are the edges. Edges are undirected in this case. Alternatively, it is

Box 1.3 History of travelling salesperson problem

A travelling salesperson problem (TSP) is one of the most important optimization problems studied in history. It is very popular among computer scientists. It was studied and developed in the 19th century by Sir William Rowan Hamilton and Thomas Pennington Kirkman. In 1857, Hamilton created an Icosian game, a pegboard with 20 holes called vertices. The objective is to find a tour starting from a city, visiting all other cities only once, and finally returning to the city where the tour has started. This is called a Hamiltonian cycle. The TSP is to find a Hamiltonian cycle in a graph. The problem was then studied extensively by Karl Menger in Vienna and by Harvard later in 1920.

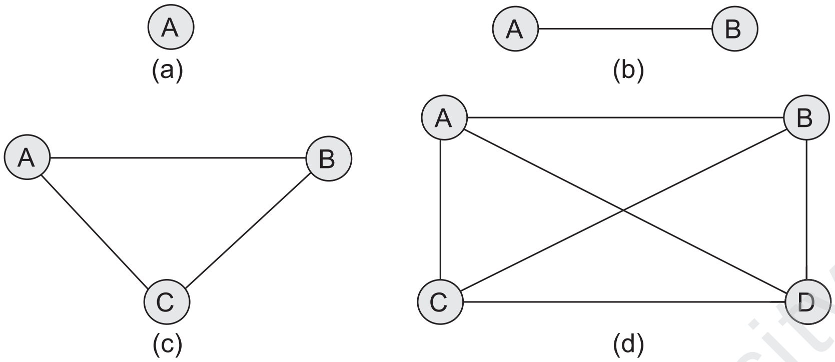


Fig. 1.2 Travelling salesperson problem (a) One city—no path (b) Two cities (c) Three cities (d) Four cities

possible to move from a particular city to any other city. The complete details of graphs are provided in Chapter 5. A TSP involves a travelling person who starts from a city, visits all other cities only once, and returns to the city from where he started.

A brute force technique can be used for solving this problem. Let us enumerate all the possible routes. For a TSP involving only one city, there is no path. For two cities, there is only one path (A–B). For three cities, there are two paths. In Fig. 1.2, these two paths are A–B–C and A–C–B, assuming that A is the origin from where the travelling salesperson started. For four cities, the paths are {A–D–B–C–A, A–D–C–B–A, A–B–C–D–A, A–B–D–C–A, A–C–D–B–A, and A–C–D–B–A}.

Thus, every addition of a city can be noted to increase the path exponentially. Table 1.2 shows the number of possible routes.

Therefore, it can be observed that, for N cities, the number of routes would be $(N - 1)!$ for $N \geq 2$. The availability of an algorithm for a problem does not mean that the problem is solvable by a computer. There are many problems that cannot be solved by a computer and for many problems algorithms require a huge amount of resources that cannot be provided practically. For example, a TSP cannot be solved in reality. Why? Let us assume that there are 100 cities. As $N = 100$, the possible routes are then $(100 - 1)! = 99!$.

The value of the number 50! is 30414093201713378043612608166064768844377641568960512000000000000. Therefore, 99! is a very large number, and even if a computer takes one second for exploring a route, the algorithm will run for years, which is plainly not acceptable. Just imagine how much time this algorithm will require, if the TSP is tried out for all cities of India or USA. This shows that the development of efficient algorithms is very important as computer resources are often limited.

Table 1.2 Complexity of TSP

Number of cities	Number of routes
1	0 (as there is no route)
2	1
3	2
4	6
5	24
6	120

1.5 FUNDAMENTAL STAGES OF PROBLEM SOLVING

Problem solving is both an art and a science. The problem-solving process starts with the understanding of a given problem and ends with the programming code of the given problem. The stages of problem solving are shown in Fig. 1.3.

The following are the stages of problem solving:

1. Understanding the problem
2. Planning an algorithm
3. Designing an algorithm
4. Validating and verifying an algorithm
5. Analysing an algorithm
6. Implementing an algorithm
7. Performing empirical analysis (if necessary)

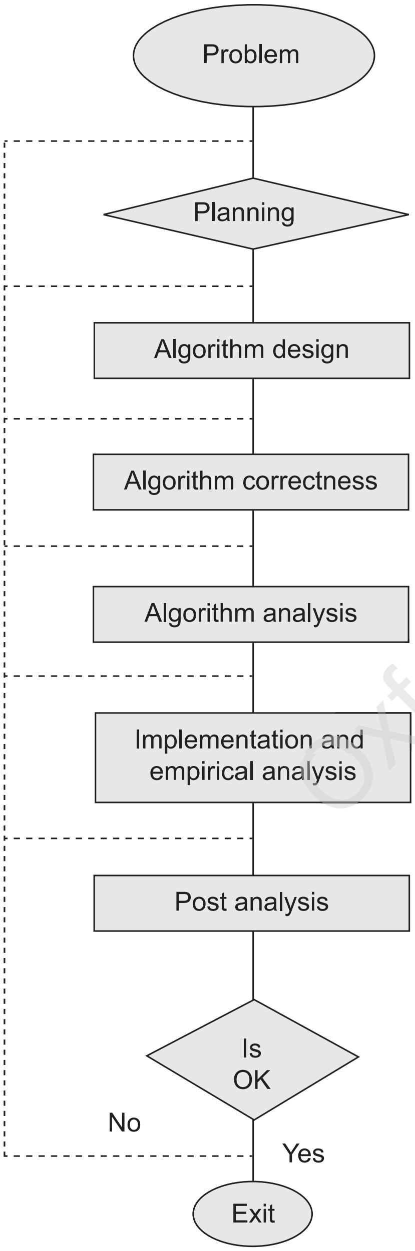


Fig. 1.3 Stages of problem solving

1.5.1 Understanding the Problem

The study of an algorithm starts with the *computability theory*. Theoretically, the primary question of computability theory is the following: Is the given problem solvable? Therefore, a guideline for problem solving is necessary to understand the problem fully. Hence, a proper problem statement is required. Any confusion regarding or misunderstanding of a problem statement will ultimately lead to a wrong result. Often, solving the numerical instances of a given problem can give an insight to the problem. Similarly, algorithmic solutions of a related problem will provide more knowledge for solving a given algorithm.

Generally, computer systems cannot solve a problem if it is not properly defined. Puzzles often fall under this category, which needs supreme level of intelligence. These problems illustrate the limitations of computing power. Humans are better than computers in solving these problems.

1.5.2 Planning an Algorithm

The second important stage of problem solving is planning. Some of the important decisions are detailed in the following subsections.

Model of Computation

A *computation model* or *computational model* is an abstraction of a real-world computer. A model of computation is a theoretical mathematical model and does not exist in physical sense. It is thus a virtual machine, and all programs can be executed on this virtual machine theoretically. What is the need for a computing model? It is meaningless to talk about the speed of an algorithm as it varies from machine to machine. An algorithm may run faster in machine A compared to in machine B. This kind of an analysis is not correct as algorithm analysis should be independent of machines. A computing model thus facilitates such machine-independent analysis.

First, all the valid operations of the model of computation should be specified. These valid operations help specify the input, process, and

Box 1.4 Alan Turing

Alan Turing (1912–1954) is well known for his contribution towards computing. He is considered by many as the 'father of modern computing'. His contribution towards computability theory and artificial intelligence is monumental. He designed a theoretical machine, called the Turing machine that is used widely in computability theory and complexity analysis. Alan Turing is also credited with the designing of the 'Turing Test' as a measure of testing the intelligence of a machine.

output. In addition, the computing models provide a notion of the necessary steps to compute time and space complexity of a given algorithm.

For algorithm study, a computation model such as a random access machine (RAM) or a Turing machine is used to perform complexity analysis of algorithms. RAM is discussed in Chapter 3 and Turing machines are discussed in Chapter 17. The contribution of Alan Turing is vital for the study of computability and computing models (refer to Box 1.4).

Data Organization

Data structure concerns the way data and its relationships are stored. Algorithms require data for solving a given problem. The nature of data and their organization can have impacts on the efficiency of algorithms. Therefore, algorithms and data structures together often constitute an important aspect of problem solving.

Data organization is something we are familiar with in our daily life. Figure 1.4 shows an example of data organization called a *queue*. A gas station with one servicing point should have a queue, as shown in the figure, to avoid chaos. A queue (or first come first serve—FCFS) is an organization where the processing (filling of gas) is done in one end and a vehicle is added at the other end. All vehicles have same priority in this case. Often a problem dictates the choice of structures. However, this structure may not be valid in cases of, for example, handling medical emergencies, where highest priority should be given to urgent cases.

Thus, data organization is a very important issue that influences the effectiveness of algorithms. At some point of problem solving, careful consideration should be given to storing the data effectively. A popular statement in computer science is 'algorithm + data structure = program'. A wrong selection of a data structure often proves fatal in the problem-solving process.

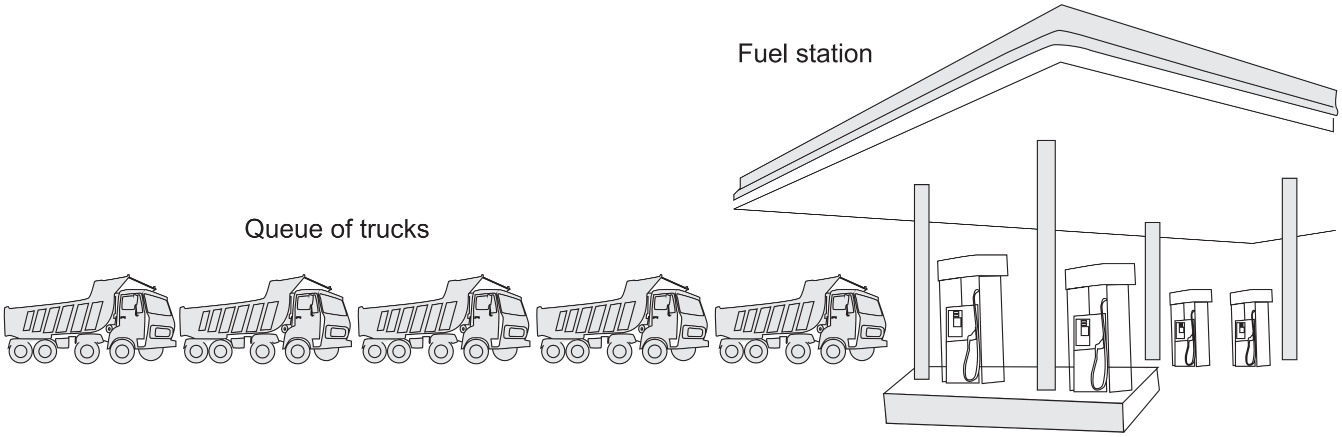


Fig. 1.4 Example of a queue

1.5.3 Designing an Algorithm

Algorithm design is the next stage of problem solving. The following are the two primary issues of this stage:

1. How to design an algorithm?
2. How to express an algorithm?

Algorithm design is a way of developing algorithmic solutions using an appropriate design strategy. Design strategies differ from problem to problem. Just imagine searching a name in a telephone directory. If anyone starts searching from page 1, it is termed as a *brute force strategy*. Since a telephone directory is indexed, it makes sense to open the book and use the index at the top to locate the name. This is a better strategy. One may come across many design paradigms in the algorithm study. Some of the important design paradigms are divide and conquer, and dynamic programming. Divide and conquer, for example, divides the problem into sub-problems and combines the results of the sub-problems to get the final solution of the given problem. Dynamic programming visualizes the problem as a sequence of decisions. Then it combines the optimal solutions of the sub-problems to get an optimal global solution. Many design variants such as greedy approach, backtracking, and branch and bound techniques have been dealt with in this book. The role of algorithm design is important in developing efficient algorithms. Thus, one important skill required for problem solving is the selection and application of suitable design paradigms. A skilled algorithm designer is called an 'algorist'.

Algorithm Specification

After the algorithm is designed, it should be communicated to a programmer so that the algorithm can be coded as a program. This stage is called *algorithm specification*. Only three possibilities exist for communicating the idea of algorithms. One is to use natural languages such as English to communicate the algorithm. This is preferable. However, the natural language has some disadvantages such as ambiguity and lack of precision. Hence, an algorithm is often written and communicated through pseudocode. Pseudocode is a mix of the natural language and mathematics. Another way is to use a programming language. The problem with programming language notation is that readers often get bogged down by the programming code details. Therefore, the pseudocode approach is preferable. Even though there are no specific rules for algorithm writing, certain guidelines can be followed to make algorithms more understandable. Some of these guidelines are presented in Chapter 2.

1.5.4 Validating and Verifying an Algorithm

Algorithm validation and verification are the methods of checking algorithm correctness. An algorithm is expected to give correct outputs for all valid inputs. Sometimes, algorithms may not give correct outputs due to logical errors. Hence, validation of algorithms becomes a necessity. *Algorithm validation* is a process of checking whether the given algorithm gives correct outputs for valid inputs or not. This is done by comparing the output of the algorithm with the expected output.

Once validation is over, *algorithm verification* or *algorithm proving* begins. Algorithm verification is a process of providing a mathematical proof that the given algorithm works correctly for all instances of data. One way of doing it is by creating a set of assertions that are expressed using mathematical logic. Assertions are statements that indicate the conditions of algorithm variables at various points of the algorithm. Preconditions indicate the conditions or variables before the execution of an algorithm, and postconditions indicate the status of the variables at the end of the execution of an algorithm. Remember, still the algorithm has not been translated or converted into a code of any programming language. Hence, all these executions are first carried out theoretically on a paper and proof for the algorithm is determined. A *proof* of an algorithm is said to exist if the preconditions can be shown to imply the postconditions logically. A complete proof is to write each statement clearly for proving that the algorithm is right. In addition to assertions, mathematical proof techniques such as mathematical induction can be used for proving that algorithms do work correctly. *Mathematical induction* is one such useful proof technique that is often used to prove that the algorithm works for all possible instances. Mathematical proofs are rigorous and always better than algorithm validation. Program correctness itself is a major study and extensive research is done in this field.

1.5.5 Analysing an Algorithm

In algorithm study, complexity analysis is important as we are mainly interested in finding optimal algorithms that use fewer computer resources. *Complexity theory* is a field of algorithms that deals with the analysis of a solution in terms of computational resources and their optimization. Humans are more complex than, say, amoeba. So what does the word 'complexity' refer to here? Complexity is the degree of difficulty associated with a problem and the algorithm. The complexity of an algorithm can be observed to be related to its input size. For example, an algorithm for sorting an array of 10 numbers is easy, but the same algorithm becomes difficult for 1 million inputs. This shows the connection between complexity and the size of the input.

Thus, complexity analysis is useful for the following two purposes:

1. To decide the efficiency of the given algorithms
2. To compare algorithms for deciding the effective solutions for a given problem

Consider the following scenario: for a problem Z, two algorithms A and B exist. Find the optimal algorithm. To solve this, there must be some measures based on which comparisons can be made. In general, two types of measures are available for comparing algorithms—subjective and objective. *Subjective measures* are factors such as ease of implementation, algorithm style, and readability of the algorithm. However, the problem with these subjective measures is that these factors cannot be quantified. In addition, a measure such as the ease of implementation, style of the algorithm, or understandability of algorithms is a subjective measure that varies from person to person. Therefore, in algorithm study, comparisons are limited to some objective measures. *Objective measures*, unlike subjective measures, can be quantified. The advantages of objective measures are that they are measurable and quantifiable, and can be used for predictions. Often time and space are used as objective measures for analysing algorithms.

Time complexity means the time taken by an algorithm to execute for different increasing inputs (i.e., differently-scaled inputs). In algorithm study, two time factors are considered—execution time and run time. Execution time (or compile time) does not depend on problem instances. Additionally, a program may be compiled many times. Therefore, time in an algorithm context always refers to the run time as only this is characterized by instances. Another complexity is *space complexity*, which is the measurement of memory space requirements of an algorithm. Technically, an algorithm is efficient if lesser resources are used. Chapter 3 focuses on the analysis of algorithms.

In algorithm study, time complexity is not measured in absolute terms. For example, one cannot say the algorithm takes 3.67 seconds. It is wrong as time complexity is always denoted as a complexity function $t(n)$ or $T(N)$, where $t(n)$ is a function of input size 'n' and is not an absolute value. The variables n and N are used interchangeably and always reserved to represent the input size of the algorithm. Recollect that input size is the number of binary bits used to represent the input instance. The logic here is that for larger inputs the algorithm would take more time. For example, sorting a list of 10 elements is easy, but sorting a list of 1 billion elements is difficult. Generally, algorithms whose time complexity function is a polynomial, for example, say N or $\log N$, can be solved easily, and but problems whose algorithms have exponential functions, say 2^N, are difficult to solve.

Example 1.1 Assume that there are two algorithms A and B for a given problem P. The time complexity functions of algorithms A and B are, respectively, $3n$ and 2^n. Which algorithm should be selected assuming that all other conditions remain the same for both algorithms?

Solution Assuming that all conditions remain the same for both algorithms, the best algorithm takes less time when the input is changed to a larger value. Results obtained employing different values of n are shown in Table 1.3.

Table 1.3 Time complexities of algorithms A and B

Input size (n)	Algorithm A $T(n) = 3n$	Algorithm B $T(n) = 2^n$
1	3	2
5	15	32
10	30	1024
100	300	2^{100}

It can be observed that algorithm A performs better as n is increased; time complexity increases linearly and gives lesser values for larger values of n. The second algorithm instead grows exponentially, and 2^{100} is a very large number.

Therefore, algorithm A is better than algorithm B.

Example 1.2 Let us assume that, for a problem P, two algorithms are possible—algorithm A and algorithm B. Let us also assume that the time complexities of algorithms A and B are, respectively, $3n$ and $10n^2$ instructions. In other words, the instructions or steps of algorithms A and B are $3n$ and $10n^2$ respectively. Here, n is the input size of the problem. Let the input size n be 10^5 instructions. If the computer system executes 10^9 instructions per second, how much time do algorithms A and B take?

Solution Here, $n = 10^5$, and the computer can execute 10^9 instructions per second.

Therefore, algorithm A (time complexity $3n$) would take $\frac{3 \times 10^5}{10^9} = \frac{3}{10^4} = 0.0003$ seconds.

Algorithm B (time complexity $10n^2$) would take $\frac{10 \times (10^5)^2}{10^9} = \frac{10 \times 10^{10}}{10^9} = 100$ seconds.

It can be observed that algorithm A would take very less time compared to algorithm B. Hence, it is natural that algorithm A is the preferred one.

One may wonder whether this has got anything to do efficiency. Imagine that we are now executing algorithm A on a slower machine—a machine that executes only 10^5instructions. What would be the scenario in this case?

Algorithm A would take $\frac{3 \times 10^5}{10^5} = 3$ seconds.

We can see that algorithm A is still better compared to algorithm B that takes $\frac{10 \times (10^5)^2}{10^5} = 10^6$ seconds. Therefore, the important point to be noted here is that speed of the machine does not affect selection of algorithm A as a better algorithm. Hence, while computer speed is crucial, the role of a better designed algorithm is still significant.

1.5.6 Implementing an Algorithm and Performing Empirical Analysis

After the algorithm is designed, it is expressed as a program and a suitable programming language is chosen to implement the algorithm as a code. After the program is written, it must be tested. Even if the algorithm is correct, sometimes the program may not give expected results. This may be due to syntax errors in coding the algorithm in a programming language or hardware fault. The error due to syntax problems of a language is called a *logical error*. The process of removing logical errors is called *debugging*. If the program leads to an error even in the absence of syntax errors, then one has to go back to the design stage to correct the algorithmic errors.

Now, complexity analysis can be performed for the developed program. The analysis that is carried out after the program is developed is called *empirical analysis* (or *a priori* analysis or theoretical analysis). This analysis is popular for larger programs; a new area, called *experimental algorithmics*, where analysis is performed for larger programs using a dataset, is emerging. A *dataset* is a huge collection of valid input data of an algorithm; the standard dataset that is often used for testing of algorithms is called a *benchmark dataset*. Interpretation of the results of a program on a benchmark dataset provides vital statistical information about the behaviour of the algorithm. This kind of analysis is called empirical analysis (also called *a posteriori* analysis). In addition, the term *profiling* is often used to denote the process of running a program on a dataset and measuring the time/space requirement of the program empirically.

1.5.7 Post (or Postmortem) Analysis

A problem-solving process ends with postmortem analysis. Any analysis should end with a valuable insight. The following are the questions that are often raised in this problem-solving process:

1. Is the problem solvable?
2. Are there any limits for this algorithm? Is there any theoretical limit for the efficiency of this problem?
3. Is the algorithm efficient, and are there any other algorithms that are faster than the current algorithm?

Answers to these questions give a lot of insight into the algorithms that are necessary for refining the design of algorithms to make them more efficient. The best possible or optimal solution that is theoretically possible for a given problem specifies a *lower bound* of a given problem. The worst-case estimate of resources required by the algorithm is called an *upper* *bound*. Theoretically, the best solution of a problem should be closer to its lower bound. The difference between the lower and upper bounds should be minimal for a better solution. The difference between the upper and the lower bounds is called an *algorithmic gap*. Technically, this should be zero. However, in practice, there may be vast difference between an upper and a lower bound. A problem-solving process tries to reduce this difference by focusing on better planning, design, and implementation on a continuous basis.

1.6 CLASSIFICATION OF ALGORITHMS

As algorithms are important for us, let us classify them for further elaboration. No single criterion exists for classifying algorithms. Figure 1.5 shows some of the criteria used for classification of algorithms.

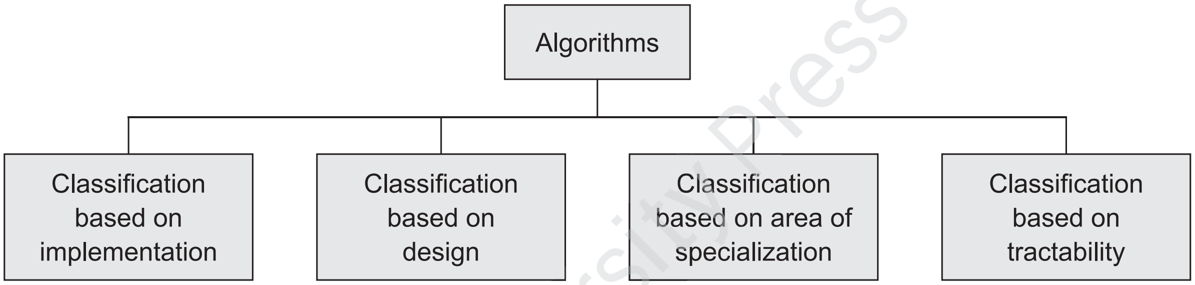

Algorithms
Classification based on implementation
Classification based on design
Classification based on area of specialization
Classification based on tractability

Fig. 1.5 Classification of algorithms

1.6.1 Based on Implementation

Based on implementation, one can classify the algorithms as follows

Recursive Algorithms vs Non-recursive (Iterative) Algorithms

Problem reduction is a scientific principle for problem solving. Take a problem, reduce it, and repeat the reduction process till the problem is reduced to a level where it can be solved directly. Using recursion, the problem is reduced to another problem with a decrease in input instance. The transformed problem is the same as the original one but with a different input, which is less than that of the original problem. The problem reduction process is continued till the given problem is reduced to a smaller problem that can be solved directly. Then the results of the sub-problems are combined to get the result of the given problem. This strategy of problem solving is called recursion.

Recursive algorithms use recursive functions for creating repetitions required for solving a given problem.

A good example of recursion is computing the factorial of a number *n*. Factorial of a number can be computed using the recursive function as follows:

$$n! = \begin{cases} 0 & \text{if } \; n = 0 \\ n \times (n-1)! & \text{for } n \geq 1 \end{cases}$$

Thus, a recursive function has a base case and an inductive case. A base case is the simplest problem that can be solved directly. In this factorial example, the base case is finding 0!, as 0! can be solved directly. An inductive case is a recursive definition of the problem that captures the essence of a problem reduction process. It can be observed from Fig. 1.6, which shows the computation of 4!, that recursion works by the *principle of work postponement* or *delaying the*

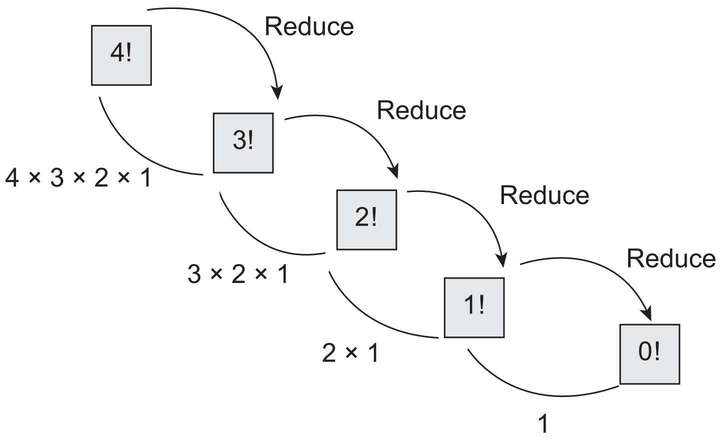


Fig. 1.6 Computation of 4! using recursion

work. For a given factorial program, the factorial function calls itself by reducing its input by 1 till the program reaches 0! This is the base case and the algorithm stops here; results of the sub-problems are collected for computing the final answer of the given problem.

The correctness of recursive algorithms is due to its close relationship with the concept of mathematical induction. In other words, recursion is a mirror of mathematical induction.

Non-recursive (or *iterative*) *algorithms*, on the other hand, are deductive in nature. Non-recursive algorithms do not use the recursion concept but, instead, rely on looping constructs, such as for or while statement, to create repetition of tasks. Non-recursive (or iterative) algorithms and their analyses are discussed in Chapter 2, and recursive algorithms and their analyses in Chapter 3.

Nature and Number of Processors

An algorithm that is designed for a single processor is called a *sequential algorithm*.

A *parallel algorithm* is designed for systems that use a set of processors. The concept of parallel processing and distributed processing are interrelated. Parallel systems have multiple processors that are located closely. Distributed systems, on the contrary, have multiple processors that are located at different places separated by a vast distance geographically. Hence, distributed systems are called loosely coupled systems, while parallel systems are called tightly coupled systems. *Distributed algorithms* are implemented in a distributed system environment. Parallel algorithms are discussed in Chapter 20.

Exact vs Approximation Algorithms

An *exact algorithm* finds the exact solutions for a given problem. Some problems are so complex that finding their exact solutions is difficult. *Approximation algorithms* (discussed in Chapter 19 of this book) find equivalent or approximate solutions for a given problem.

Deterministic vs Non-deterministic Algorithms

Deterministic algorithms always provide fixed predictable results for a given input. In contrast, non-deterministic algorithms or randomized algorithms take a different approach. For deterministic algorithms, the output should always be true. On the other hand, randomized algorithms relax this condition. It is argued that outputs based on random decisions may not often result in correct answers or the algorithm may not terminate at all. Thus, the accuracy of an output is associated with a probability. In daily life, we often use random decisions, for example, in games such as dice. Industries conduct random quality checks on products. Randomized samples are used to predict poll results and so on. One can be very sceptical about randomized algorithms. However, randomized algorithms have been proved to be very effective. Randomized algorithms are discussed in Chapter 19.

1.6.2 Based on Design

Based on design techniques, algorithms can be classified into various categories. As discussed earlier, every design technique uses a strategy for solving a problem. Algorithms can be classified based on design as brute force, divide and conquer, dynamic programming, greedy approach, backtracking, and branch and bound algorithms. This textbook is organized based on this classification of algorithms only.

1.6.3 Based on Area of Specialization

Algorithms can be classified based on the area of specialization. General algorithms such as searching and sorting, and order statistics such as finding mean, median, rank, and so on are useful for all fields of specialization. These algorithms can be considered as building blocks of any area of specialization. Other than these basic algorithms, some algorithms are specialized for a particular domain. String algorithms, graph algorithms, combinatorial algorithms, and geometric problems are some examples of specialized algorithms that are discussed in this book. Let us discuss some of these algorithms now.

General Algorithms

Sorting algorithm is a general algorithm that is useful in all areas of specialization. Sorting problem is a structural problem. It involves structuring or rearranging the sequence in a specific order. The importance of this problem arises from the fact that all modern applications require sorting. For example, an organization may require sorting of employee records based on employee identification number (called primary key). A library system may want books to be ordered based on titles or ISBN numbers. Sorting problems for small instances are relatively easy. When sorting is required for a larger number, say a billion elements, designing sorting algorithms becomes more challenging.

Domain-specific Specialized Algorithms

Let us discuss some of these domain-specific algorithms now:

String algorithms String algorithms are used frequently in document editing, web searches, and pattern matching.

A sequence of characters is called a string. The sequence can be text, bits, numbers, or gene sequences (A, C, G, or T). One problem that is of considerable interest is string matching, which takes a pattern, searches the text string, and reports whether a pattern is present or not. String algorithms, which are discussed in Chapter 16, constitute a very important field of study.

Graph algorithms A graph is used for modelling complex systems by representing the relationships among the subsystems. A graph represents a set of nodes and edge. The nodes are also called vertices. The nodes are connected by edges, which are also known as arcs. Consider the graph shown in Fig. 1.7. Here, the nodes are given as {A, B, C, D, E} and the edges as {(A,B), (A,C), (A,D), (B,A), (B,E), (B,C), (C,A), (C,B), (C,D), (C,E), (D,A), (D,C), (D,E), (E,B), (E,C), and (E,D)}.

Many interesting problems can be formulated using this graph structure. The following are some of the interesting graph algorithms:

Travelling salesperson problem As stated earlier, a salesperson starts from a node, visits all other nodes only once, and gets back to the starting node. This problem is reduced to a Hamiltonian cycle if the graph is undirected. In the case of a weighted graph (where edges carry some weights), the TSP is about finding a tour of minimum cost.

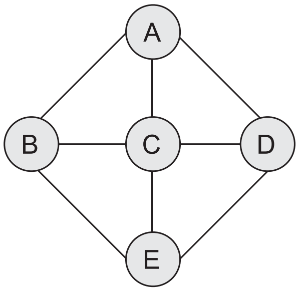


Fig. 1.7 Sample graph

Chinese postman problem This problem is the same as a TSP, but instead of vertices, an edge should be visited only once. There is no restriction on vertices in this problem.

Graph colouring problem This problem is about how to colour all the vertices distinctly using only a small number of colours such that no two neighbouring vertices share the same colour.

Combinatorial algorithms The focus of combinatorial algorithms is to find a combinatorial object inherent in the problem such as permutations, combinations, or a subset that satisfies some constraints and objective functions. These problems are difficult to solve, and many problems do not have algorithmic solutions. TSPs and graph colouring problems are examples of combinatorial problems.

Geometric algorithms Geometric algorithms deal with geometrical objects such as points, lines, and polygons. The following are some of the algorithms discussed in this textbook:

Closest pair problem This problem deals with finding the distance between points in a 2D space and finding a pair of points that are closest to each other.

Convex hull problem This problem deals with finding the smallest convex polygon that includes all points in a 2D space.

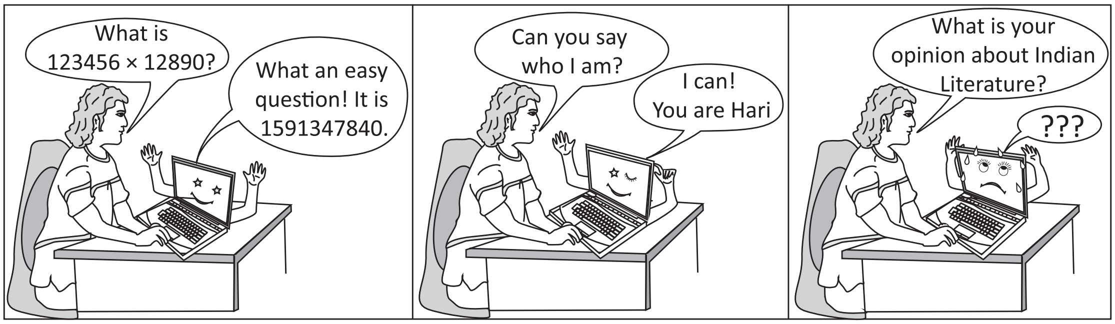
1.6.4 Based on Tractability

Tractability means solvability of a given problem within a reasonable amount of time and space. An intractable problem is difficult to solve within the reasonable amount of computer resources. Based on tractability, the following categories are possible:

Easily solvable problems These problems have polynomial time complexity or their upper bounds are characterized by a polynomial. These problems are solvable. For example, sorting is an example of solvable or polynomial problems.

Unsolvable problems Problems such as halting problems cannot be solved at all. These are called unsolvable or non-computable problems. Knowledge about the non-computability of these problems helps in project management.

Intractable problems These problems are of two categories. One category comprises a set of problems that have algorithmic solutions but require more computer resources. Hence, these solutions are practically non-implementable. In addition, these have been proved to be computationally hard. The other category consists of a set of problems that have been proved to be computationally hard. For example, a TSP is a proven computationally hard problem; it had already been discussed that time complexity of the algorithm increases exponentially with an increase in the number of cities.


What is 123456 × 12890?
What an easy question! It is 1591347840.
Can you say who I am?
I can! You are Hari
What is your opinion about Indian Literature?
???

Thus, algorithm study can be concluded as the central theme of computer science. An aspiring computer professional should be a better problem solver. The problem-solving process starts with the understanding of a given problem and ends with finding an efficient programming code for the given problem. Problem solving poses many challenges as it is a creative process. Chapter 2 introduces the basics of a problem-solving process and also introduces basic guidelines for writing algorithms.

SUMMARY

- An algorithm is a step-by-step procedure for solving a given problem.
- A computational problem is characterized by two factors—specification of valid input and output parameters of the algorithm, and specification of the relationship between inputs and outputs.
- An algorithm that yields the correct output for a legal input is called an algorithmic solution.
- The art of designing, analysing, and implementing algorithms is called algorithmics.
- Algorithms can be contrasted with programs. A program is an expression of algorithm in a programming language.
- A valid input is called an instance. The number of binary bits necessary to encode inputs of an algorithm is called the input size.
- An algorithm should have characteristics such as a well-defined order, inputs, outputs, finiteness, definiteness, efficiency, and generality.
- Problem solving starts with the understanding of a problem statement without any confusion.
- A computation or computational model is an abstraction of a real-world computer.
- Algorithm design is a way of developing algorithmic solutions using a suitable course of action called a design strategy. Algorithm specification is about communicating the design strategy to a programmer often in the form of a pseudocode.
- Algorithmic validation means checking whether the algorithm gives a correct result or not.
- Algorithm verification is a process of providing a mathematical proof that the given algorithm works correctly for all instances of data.
- A proof of an algorithm is said to exist if the preconditions can be shown to imply postconditions logically.
- An estimation of the time and space complexities of an algorithm for varying input sizes is called algorithm analysis.
- Time complexity refers to the measurement of run time of an algorithm in terms of its input size, and space complexity is the measurement of space required for a given algorithm.
- A dataset is a huge collection of valid input data of an algorithm. The standard dataset that is often used for testing of algorithms is called a benchmark dataset.
- The theoretically best possible or optimal solution for a given problem specifies a lower bound of a given problem. The worst-case estimate of resources that can be required by an algorithm is called an upper bound. The difference between the upper and the lower bound is called an algorithmic gap.
- Problem reduction is a scientific principle for problem solving. Take a problem, reduce it, and repeat the reduction process till the problem is reduced to a level where it can be solved directly.
- Algorithms can be categorized based on their implementation methods, design techniques, field of study, and tractability.

GLOSSARY

Agent A performer of an algorithm

Algorist A person who is skilled in algorithm development

Algorithm A step-by-step procedure for solving a given problem

Algorithm gap The difference between lower and upper bounds

Algorithm specification Formalization of an algorithm in a suitable form that can be conveyed to a programmer

Algorithm verification The process of providing a mathematical proof that the algorithm works correctly for all valid inputs

Algorithm validation The process of checking the correctness of an algorithm, this is done by giving valid inputs to it and checking its results with expected values

Approximation algorithms Algorithms that provide approximate solutions for problems whose exact solutions are difficult to obtain

Complexity theory A field of algorithm study that deals with the analysis of algorithms

Computability theory A field of study that deals with the theory of algorithms and solvability of the algorithms

Computational model An abstraction of a real-world computer system

Computational problem A problem that can be solved by computer systems

Deterministic algorithm Algorithms that always give the same result for a fixed input

Exact algorithm An algorithm that is designed to give an exact solution for a given problem

Experimental algorithmics A field of study that deals with the analysis of algorithms experimentally using a large dataset

Intractable problems Problems that cannot be solved within a reasonable amount of computer resources

Lower bound A theoretical best solution for a given problem

Non-deterministic algorithm An algorithms that uses randomized choices that determine the course of execution of instructions; hence, the output of an algorithm varies even for a fixed input

Non-recursive algorithm (iterative algorithm) An algorithm that is developed using iterative constructs for creating repetition of tasks instead of using recursive functions

Parallel algorithm An algorithm designed for systems that use multiple processors

Profiling The process of measuring time and space complexity by running a program on a larger dataset

Proof The logical derivation showing that the preconditions of algorithms imply their postconditions

Recursive algorithm An algorithm that uses a recursive function

Sequential algorithm An algorithm that is designed for a single-processor system

Space complexity Measurement of space required for an algorithm

Strategy A way of designing algorithms

Time complexity Run time measurement of the algorithm when the input is scaled to a larger value

Tractable problems Problems for which algorithms that find solutions within a reasonable amount of computer resources exist

Upper bound The worst-case complexity of an algorithm

■ REVIEW QUESTIONS ■

1.1 Define an algorithm.

1.2 What are the characteristics of an algorithm?

1.3 Survey the Internet and list out at least five algorithms that have huge impact on our daily lives.

1.4 What are the stages of problem solving?

1.5 What is meant by time and space complexities?

1.6 Define the term—design paradigm.

1.7 What is an algorithm specification? What is a pseudocode and how does it help in algorithm specification?

1.8 What is the difference between algorithm verification and algorithm validation?

1.9 How is an algorithm validated and verified? Explain with an example.

1.10 What are the criteria used for classification of algorithms?

■ EXERCISES ■

1.1 Assume that there are two algorithms A and B for a given problem P. The time complexity functions of algorithms A and B are, respectively, $5n$ and $\log_2 n$. Which algorithm should be selected assuming that all other conditions remain the same for both the problems?

1.2 Let us assume that for a telephone directory search problem P, three algorithms exist: A, B, and C. Time complexities of A, B, and C are $3n$, $5n$, and $\log n$, respectively. Assume that the input instance n is 10^3. Assume that the machine executes 10^9 instructions per second. How much time will algorithms A, B, and C take? Which algorithm will be the best?

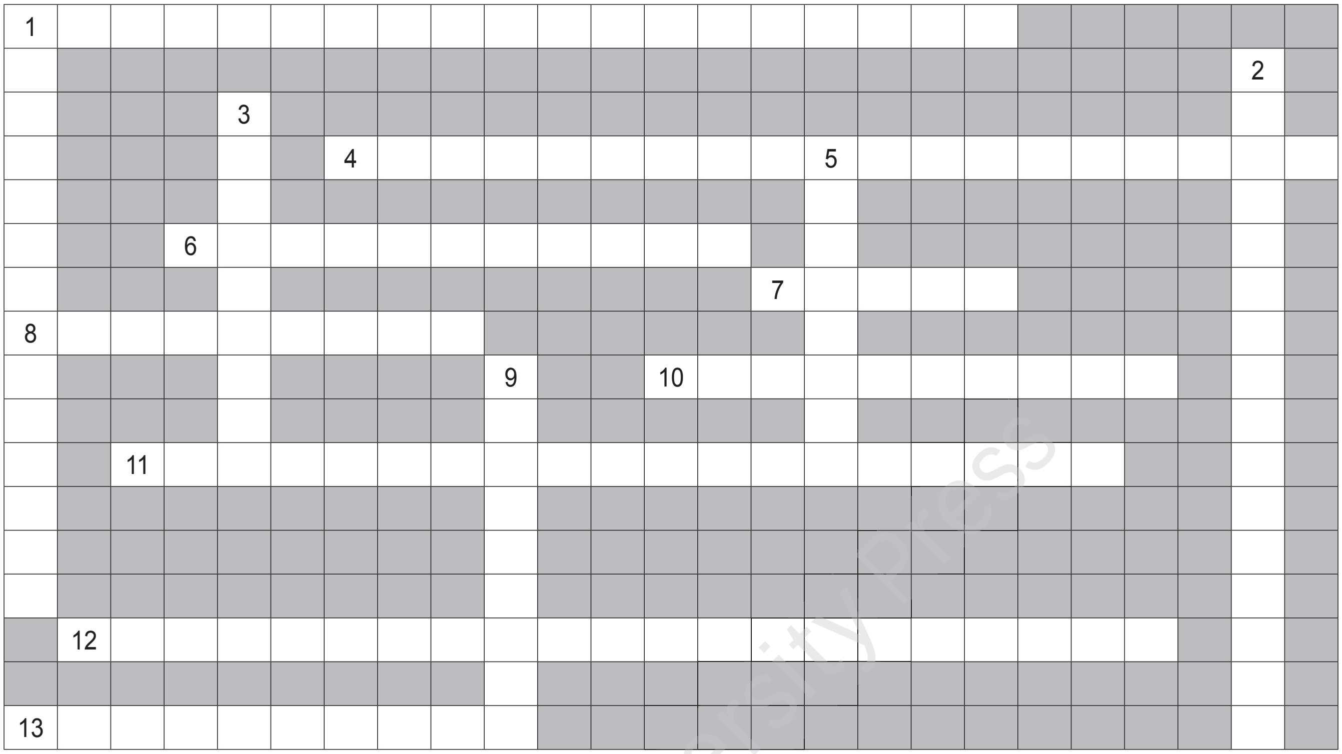
■ ADDITIONAL PROBLEM ■

1.1 John MacCormick had written a book titled *Nine Algorithms That Changed the Future: The Ingenious Ideas that drive Today's Computers,* Princeton University Press, Princeton, that had listed the nine wonderful algorithms, namely, search engine indexing, page rank, public key cryptography, error correcting codes, pattern recognition, data compression, databases, and digital signatures, that changed the world.

(a) What are these algorithms? Search the Internet and find what these algorithms are for.

(b) Identify one more algorithm that you feel changes the world we live in.

CROSSWORD



Across

1. Algorithms that are solved within reasonable amount of computable resources
4. Algorithms that use recursive functions
6. Who coined the word algorithm analysis
7. An abstraction of a real world problem
8. Statistical process of measuring resources on a larger dataset
10. Process of checking algorithms
11. Theory of algorithms
12. Analytical skills
13. Attributed as Father of Modern Computing by many

Down

1. Measurement of time
2. Study of analysis of algorithms
3. A step-by-step procedure
5. A person skilled in algorithms
9. Process of removing syntax errors

Answers to the Crossword are available as online resources