Introduction to Computer Theory

2nd Edition



BIBLIOTEX
Digital Library

INTRODUCTION TO COMPUTER THEORY

2nd Edition

BIBLIOTEX
Digital Library
www.bibliotex.com

INTRODUCTION TO COMPUTER THEORY

2ND EDITION



www.bibliotex.com
email: info@bibliotex.com

e-book Edition 2023
ISBN: 978-1-98468-255-0 (e-book)

This book contains information obtained from highly regarded resources. Reprinted material sources are indicated. Copyright for individual articles remains with the authors as indicated and published under Creative Commons License. A Wide variety of references are listed. Reasonable efforts have been made to publish reliable data and views articulated in the chapters are those of the individual contributors, and not necessarily those of the editors or publishers. Editors or publishers are not responsible for the accuracy of the information in the published chapters or consequences of their use. The publisher assumes no responsibility for any damage or grievance to the persons or property arising out of the use of any materials, instructions, methods or thoughts in the book. The editors and the publisher have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission has not been obtained. If any copyright holder has not been acknowledged, please write to us so we may rectify.

Notice: Registered trademark of products or corporate names are used only for explanation and identification without intent of infringement.

© 2023 3G E-learning LLC

In Collaboration with 3G E-Learning LLC. Originally Published in printed book format by 3G E-Learning LLC with ISBN 978-1-98468-005-1

EDITORIAL BOARD



Aleksandar Mratinković was born on May 5, 1988 in Arandjelovac, Serbia. He has graduated on Economic high school (2007), The College of Tourism in Belgrade (2013), and also has a master degree of Psychology (Faculty of Philosophy, University of Novi Sad). He has been engaged in different fields of psychology (Developmental Psychology, Clinical Psychology, Educational Psychology and Industrial Psychology) and has published several scientific works.



Dan Piestun (PhD) is currently a startup entrepreneur in Israel working on the interface of Agriculture and Biomedical Sciences and was formerly president-CEO of the National Institute of Agricultural Research (INIA) in Uruguay. Dan is a widely published scientist who has received many honours during his career including being a two-time recipient of the Amit Golda Meir Prize from the Hebrew University of Jerusalem, his areas of expertise includes stem cell molecular biology, plant and animal genetics and bioinformatics. Dan's passion for applied science and technological solutions did not stop him from pursuing a deep connection to the farmer, his family and nature. Among some of his interest and practices counts enjoying working as a beekeeper and onboard fishing.



Hazem Shawky Fouda has a PhD. in Agriculture Sciences, obtained his PhD. From the Faculty of Agriculture, Alexandria University in 2008. He is working in Cotton Arbitration & Testing General Organization (CATGO).



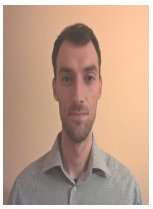
Felecia Killings is the Founder and CEO of LiyahAmore Publishing, a publishing company committed to providing technical and educational services and products to Christian Authors. She operates as the Senior Editor and Writer, the Senior Writing Coach, the Content Marketing Specialist, Editor-in-Chief to the company's quarterly magazine, the Executive and Host of an international virtual network, and the Executive Director of the company's online school for Authors. She is a former high-school English instructor and professional development professor. She possesses a Master of Arts degree in Education and a Bachelor's degree in English and African American studies.



Dr. Sandra El Hajj, Ph.D. in Health Sciences from Nova Southeastern University, Florida, USA is a health professional specialized in Preventive and Global Health. With her 12 years of education obtained from one of the most prominent universities in Beirut, in addition to two leading universities in the State of Florida (USA), Dr. Sandra made sure to incorporate interdisciplinary and multicultural approaches in her work. Her long years of studies helped her create her own miniature world of knowledge linking together the healthcare field with Medical Research, Statistics, Food Technology, Environmental & Occupational Health, Preventive Health and most noteworthy her precious last degree of Global Health. Till today, she is the first and only doctor specialized in Global Health in the Middle East area.



Fozia Parveen has a Dphil in Sustainable Water Engineering from the University of Oxford. Prior to this she has received MS in Environmental Sciences from National University of Science and Technology (NUST), Islamabad Pakistan and BS in Environmental Sciences from Fatima Jinnah Women University (FJWU), Rawalpindi.



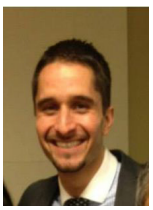
Igor Krunic 2003-2007 in the School of Economics. After graduating in 2007, he went on to study at The College of Tourism, at the University of Belgrade where he got his bachelor degree in 2010. He was active as a third-year student representative in the student parliament.Then he went on the Faculty of science, at the University of Novi Sad where he successfully defended his master's thesis in 2013. The crown of his study was the work titled Opportunities for development of cultural tourism in Cacak". Later on, he became part of a multinational company where he got promoted to a deputy director of logistic. Nowadays he is a consultant and writer of academic subjects in the field of tourism.



Dr. Jovan Pehcevski obtained his PhD in Computer Science from RMIT University in Melbourne, Australia in 2007. His research interests include big data, business intelligence and predictive analytics, data and information science, information retrieval, XML, web services and service-oriented architectures, and relational and NoSQL database systems. He has published over 30 journal and conference papers and he also serves as a journal and conference reviewer. He is currently working as a Dean and Associate Professor at European University in Skopje, Macedonia.



Dr. Tanjina Nur finished her PhD in Civil and Environmental Engineering in 2014 from University of Technology Sydney (UTS). Now she is working as Post-Doctoral Researcher in the Centre for Technology in Water and Wastewater (CTWW) and published about eight International journal papers with 80 citations. Her research interest is wastewater treatment technology using adsorption process.



Stephen obtained his PhD from the University of North Carolina at Charlotte in 2013 where his graduate research focused on cancer immunology and the tumor microenvironment. He received postdoctoral training in regenerative and translational medicine, specifically gastrointestinal tissue engineering, at the Wake Forest Institute of Regenerative Medicine. Currently, Stephen is an instructor for anatomy and physiology and biology at Forsyth Technical Community College.



Michelle holds a Masters of Business Administration from the University of Phoenix, with a concentration in Human Resources Management. She is a professional author and has had numerous articles published in the Henry County Times and has written and revised several employee handbooks for various YMCA organizations throughout the United States.

HOW TO USE THE BOOK

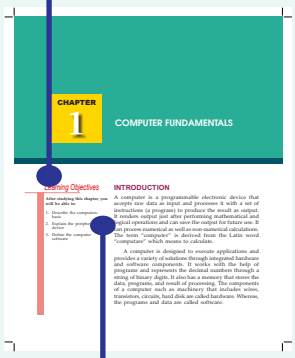
This book has been divided into many chapters. Chapter gives the motivation for this book and the use of templates. The text is presented in the simplest language. Each paragraph has been arranged under a suitable heading for easy retention of concept. Keywords are the words that academics use to reveal the internal structure of an author's reasoning. Review questions at the end of each chapter ask students to review or explain the concepts. References provides the reader an additional source through which he/she can obtain more information regarding the topic.

LEARNING OBJECTIVES

See what you are going to cover and what you should already know at the start of each chapter

ABOUT THIS CHAPTER

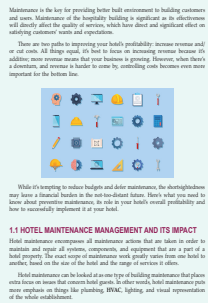
An introduction is a beginning of section which states the purpose and goals of the topics which are discussed in the chapter. It also starts the topics in brief.


CHAPTER 1

COMPUTER FUNDAMENTALS

Learning Objectives

INTRODUCTION


1.1 HOTEL MAINTENANCE MANAGEMENT AND ITS IMPACT

REMEMBER

This revitalizes a must read information of the topic.

KEYWORDS

This section contains some important definitions that are discussed in the chapter. A keyword is an index entry that identifies a specific record or document. It also gives the extra information to the reader and an easy way to remember the word definition.


Keyword

Remember

Better Budgeting and Significant Cost-Cutting with Hotel Preventive Maintenance Software

Global Firewood Production and Consumption

How is Firewood Used for Cooking?

DID YOU KNOW?

This section equip readers the interesting facts and figures of the topic.

EXAMPLE

The book cabinets' examples to illustrate specific ideas in each chapter.



Patent

Trademark and copyright differences can get murky, so let's first review the purpose and use of a patent. When you hear the word "patent," you should think invention – everything from Post-it notes to animal ear protectors.

Copyright

Artists, authors, and other creative seek out copyrights for their work. A copyright covers physical works of art, music recordings of any kind, written works, and performance arts. The length of a copyright lasts for the lifetime of the author plus 70 years.While a copyright technically is created as soon as you create a piece of intellectual property, without some form of proof, it's very difficult enforce that protection in federal court. You also do not need to have the signature (©) with your name and year created on all of your materials, but doing so could potentially help you win your case.

Trademark

Business and product owners use trademarks for names, logos, and symbols that identify commercial goods or services. While they protect the brand name of the goods, they do not protect the actual good or service from being replicated and sold under a different name.

McDonald's might trademark the name "Egg McMuffin," but it will not stop other [illegible] from producing their version of the egg, ham, and cheese breakfast on an English muffin.

This is where branding and marketing comes in. Your competitors might offer similar products, but your customers choose yours because of your recognizable brand and iconography.

How do you trademark a business name and logo?

To register a trademark in the U.S., you have to file an application with the United States Patent Trademark Office (USPTO), the federal agency that governs the enforceability of trademark

too. If your service is based on a certain area, you might extend to other cities. So, select a broad name that can encompass your future growth.

7. ***Make sure you have a related domain:-*** Once you have narrowed down your name options, you have to make sure the name you want to use is available. Check if the name is already registered as a trademark or acquired by another business. If you are planning on having an online presence, which you definitely should, you'll want to make sure the desired domain is available by doing a domain name check.

A domain name can impact your online visibility and influences your marketing efforts as well. That said, you should not necessarily abandon a good business name just because an exact domain is not available.

Here are a couple of things that you can do instead:

- If it is a parked domain, consider buying it. It's probably going to cost you, but a search-friendly, memorable domain is worth it.
- Add modifiers to the name.
- Be creative with domain selection. Nowadays, businesses use different top-level domains to make their URLs more memorable.

Choosing a name can be challenging when you are setting out to start a business, but not something to agonize over! Remember to get opinions and ideas from friends, family or even potential customers. It's always a good idea to get a few different points of view.

A business name is only your first step towards building a brand. Though it'll give your business a good first impression, it's up to you to meet the customer expectations and bring your business to success.

Did You Know?

Finding the right name for your startup can have a significant impact on your success. The wrong name can do worse than fail to connect with customers; it can actually be a monumentable business and legal hurdles. In contrast, a clear, powerful name can be extremely helpful in your marketing and branding efforts.

1.3 BRANDING A BUSINESS NAME

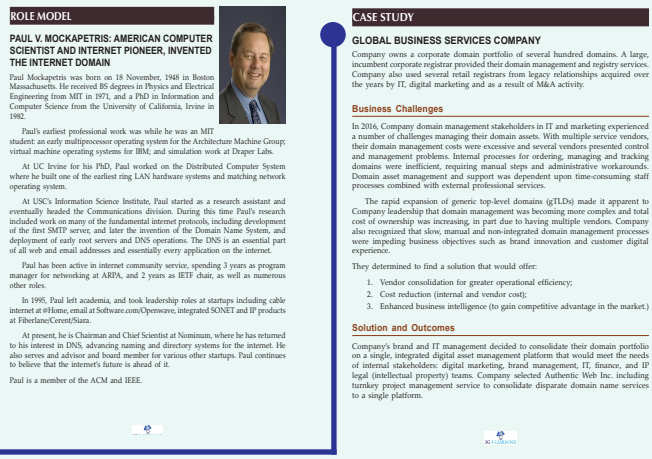
Nowadays, companies around the world (with minor exceptions) are recognized as legal entities, which mean that companies are authorized to make autonomous decisions regarding business activities. As a rule, they are registered and may independently

ROLE MODEL

A biography of someone who has/had acquired remarkable success in their respective field as Role Models are important because they give us the ability to imagine our future selves.

CASE STUDY

This reveals what students need to create and provide an opportunity for the development of key skills such as communication, group working and problem solving.



ROLE MODEL

PAUL V. MOCKAPETRIS: AMERICAN COMPUTER SCIENTIST AND INTERNET PIONEER, INVENTED THE INTERNET DOMAIN

Paul Mockapetris was born on 18 November, 1948 in Boston Massachusetts. He received BS degrees in Physics and Electrical Engineering from MIT in 1971, and a PhD in Information and Computer Science from the University of California, Irvine in 1982.

Paul's earliest professional work was while he was an MIT student: an early multiprocessor operating system for the Architecture Machine Group; virtual machine operating systems for IBM; and simulation work at Draper Labs.

At UC Irvine for his PhD, Paul worked on the Distributed Computer System where he built one of the earliest ring LAN hardware systems and matching network operating system.

At USC's Information Science Institute, Paul started as a research assistant and eventually headed the Communications division. During this time Paul's research included work on many of the fundamental internet protocols, including development of the first SMTP server, and later the invention of the Domain Name System, and deployment of early root servers and DNS operations. The DNS is an essential part of all web and email addresses and essentially every application on the internet.

Paul has been active in internet community service, spending 3 years as program manager for networking at ARPA, and 2 years as IETF chair, as well as numerous other roles.

In 1995, Paul left academia, and took leadership roles at startups including cable internet at @Home, email at Software.com/Openwave, integrated SONET and IP products at Fiberlane/Cerent/Siara.

At present, he is Chairman and Chief Scientist at Nominum, where he has returned to his interest in DNS, advancing naming and directory systems for the internet. He also serves and advisor and board member for various other startups. Paul continues to believe that the internet's future is ahead of it.

Paul is a member of the ACM and IEEE.

CASE STUDY

GLOBAL BUSINESS SERVICES COMPANY

Company owns a corporate domain portfolio of several hundred domains. A large, incumbent corporate registrar provided their domain management and registry services. Company also used several retail registrars from legacy relationships acquired over the years by IT, digital marketing and as a result of M&A activity.

Business Challenges

In 2016, Company domain management stakeholders in IT and marketing experienced a number of challenges managing their domain assets. With multiple service vendors, their domain management costs were excessive and several vendors presented control and management problems. Internal processes for ordering, managing and tracking domains were inefficient, requiring manual steps and administrative workarounds. Domain asset management and support was dependent upon time-consuming staff processes combined with external professional services.

The rapid expansion of generic top-level domains (gTLDs) made it apparent to Company leadership that domain management was becoming more complex and total cost of ownership was increasing, in part due to having multiple vendors. Company also recognized that slow, manual and non-integrated domain management processes were impeding business objectives such as brand innovation and customer digital experience.

They determined to find a solution that would offer:

1. Vendor consolidation for greater operational efficiency;
2. Cost reduction (internal and vendor cost);
3. Enhanced business intelligence (to gain competitive advantage in the market.)

Solution and Outcomes

Company's brand and IT management decided to consolidate their domain portfolio on a single, integrated digital asset management platform that would meet the needs of internal stakeholders: digital marketing, brand management, IT, finance, and IP legal (intellectual property) teams. Company selected Authentic Web Inc. including turnkey project management service to consolidate disparate domain name services to a single platform.

KNOWLEDGE CHECK

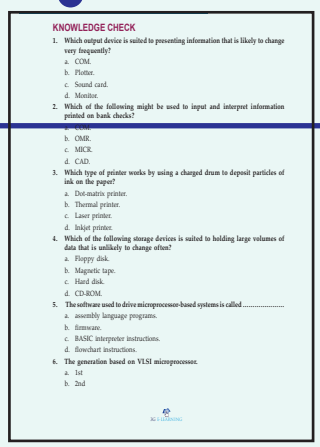
This is given to the students for progress check at the end of each chapter.

REVIEW QUESTIONS

This section is to analyze the knowledge and ability of the reader.

REFERENCES

References refer those books which discuss the topics given in the chapters in almost same manner.



KNOWLEDGE CHECK

1. **Which output device is suited to presenting information that is likely to change very frequently?**
   a. COM.
   b. Plotter.
   c. Sound card.
   d. Monitor.
2. **Which of the following might be used to input and interpret information printed on bank checks?**
   a. OCR.
   b. OMR.
   c. MICR.
   d. CAD.
3. **Which type of printer works by using a charged drum to deposit particles of ink on the paper?**
   a. Dot-matrix printer.
   b. Thermal printer.
   c. Laser printer.
   d. Inkjet printer.
4. **Which of the following storage devices is suited to holding large volumes of data that is unlikely to change often?**
   a. Floppy disk.
   b. Magnetic tape.
   c. Hard disk.
   d. CD-ROM.
5. **The software used to drive microprocessor-based systems is called .....**
   a. assembly language programs.
   b. firmware.
   c. BASIC interpreter instructions.
   d. flowchart instructions.
6. **The generation based on VLSI microprocessor.**
   a. 1st
   b. 2nd

REVIEW QUESTIONS

1. What exactly is a domain name?
2. How to choose a domain name for business? Explain.
3. Discuss how to register a domain name? What will it cost?
4. What's the difference between my domain name and web hosting?
5. What is the best way to secure a domain name?
6. Which domain is best for business?

Answer to Multiple Choice Questions

1. (b) 2. (b) 3. (d) 4. (a) 5. (c)
6. (c) 7. (a) 8. (c) 9. (b) 10. (c)

REFERENCES

1. Department of Revenue Washington State. "Domain Name Registration Services." Accessed Dec. 20, 2019.
2. Domain Name Wire podcast. "Frank Schilling Explains Price Hike – DNW Podcast #127." Accessed Dec. 20, 2019.
3. Domain Name Wire. "ICANN Shuts Down Alpnames Domain Name Registrar." Accessed Dec. 20, 2019.
4. Domain.com. "Domain Registration Agreement." Accessed Dec. 20, 2019.
5. ICANN. "FAQs: How Long Does a Registration Last? Can It be Renewed?" Accessed Dec. 20, 2019.
6. No-IP Knowledge Base. "How Long Until the Domain I just Registered is Active?" Accessed Dec. 20, 2019.
7. PR Newswire. "GoDaddy Acquires Worldwide Media Inc. Domain Name Portfolio." Accessed Dec. 20, 2019.
8. Symbolics.com. "About Symbolics." Accessed Dec. 20, 2019.
9. Verisign. "Verisign Q1 2019 Domain Name Industry Brief." Accessed Dec. 20, 2019.
10. Whois.net. "Terms and Conditions of Whois.net." Accessed Dec. 20, 2019.

TABLE OF CONTENTS

Preface xi

Chapter 1 Computer Fundamentals 1

Introduction 1
1.1 Computers: Basic 2
1.1.1 Generations of Computers 4
1.1.2 Types of Computers 15
1.1.3 Fundamental Uses of Computers 20
1.2 Peripheral Device 21
1.2.1 Input device 21
1.2.2 Output Devices 27
1.2.3 Storage Devices 33
1.3 Computer Software 43
1.3.1 Types of Software 44
1.3.2 Need of Software 48
Summary 53
Knowledge Check 54
Review Questions 55
References 57

Chapter 2 Finite Automata 59

Introduction 59
2.1 Finite Automata: Basic 60
2.1.1 A Precise Specification 64
2.1.2 Designing Automata 66
2.2 Nondeterministic Finite Automata 67
2.2.1 Closure Properties 71
2.2.2 Every regular language is recognized by a DFA 76

2.2.3 Non Regular Languages 77
2.2.4 Regular expressions 81
2.3 Application of Finite Automata 86
Summary 92
Knowledge Check 92
Review Questions 94
References 96

Chapter 3 Pushdown Automata Theory 97

Introduction 97
3.1 Pushdown Automata 98
3.1.1 Basic Structure of PDA 99
3.1.2 Pushdown Automata Acceptance 101
3.1.3 PDA & Context-Free Grammar 103
3.1.4 Parsing 108
3.2 Context-Free Grammar 110
3.2.1 Ambiguity in Context-Free Grammars 114
3.2.2 CFL Closure Property 115
3.2.3 CFG Simplification 116
3.2.4 Chomsky Normal Form 119
3.2.5 Greibach Normal Form 122
Summary 126
Knowledge Check 127
Review Questions 128
References 129

Chapter 4 Turing Machines 131

Introduction 131
4.1 Turing Machines: Basic 132
4.1.1 Formal definition of Turing machine 136
4.1.2 Universal Turing machine 138
4.1.3 Undecidability of the Halting Problem 142
4.2 Computing Functions 145
4.2.1 Shifting 145
4.2.2 Binary Addition 146
4.2.3 Variations 148
Summary 159
Knowledge Check 159
Review Questions 161
References 162

Chapter 5 Algorithms 163

Introduction **163**
5.1 Sorting Algorithms **164**
5.1.1 Bubble Sort 165
5.1.2 Merge Sort 167
5.1.3 Quick Sort 171
5.1.4 Insertion Sort 173
5.1.5 Shell Sort 175
5.1.6 Radix Sort 179
5.2 Numeric Algorithm **182**
5.2.1 Primarily Test 183
5.2.2 Base Conversions 184
5.2.3 Factorial of a Number 186
Summary **189**
Knowledge Check **189**
Review Questions **191**
References **192**

Chapter 6 Distributed Computing 193

Introduction **193**
6.1 Distributed Computing Systems: Basic **195**
6.1.1 Distributed Computing System Models 197
6.1.2 Transparency 202
6.1.3 Reliability 205
6.1.4 Flexibility 209
6.1.5 Performance 210
6.2 Remote Procedure Calls **212**
6.2.1 RPC Operations 212
6.2.2 Transparency of RPC 214
6.2.3 Implementing RPC Mechanism 215
6.2.4 STUB Generation 218
6.2.5 RPC Messages 219
6.2.6 Marshaling Arguments and Results 220
6.2.7 Server Management 221
Summary **225**
Knowledge Check **225**
Review Questions **227**
References **228**

Chapter 7 Computational Complexity 229

Introduction 229

7.1 The Basic of Computational Complexity 230

7.1.1 Big-O notation 230

7.1.2 Polynomial Complexity 232

7.1.3 Brute Force 232

7.1.4 Random and Statistical Methods 233

7.1.5 Space Complexity 234

7.2 The Time Complexity of a Turing Machine, and The Set P 235

7.3 The Set NP And Polynomial Verifiability 240

7.4 Polynomial-Time Reductions and Np-Completeness 246

Summary 249

Knowledge Check 249

Review Questions 250

References 252

Index 255

PREFACE

Theory of Computation or Theory of Automata is the core area of computer science and engineering. It is the branch that aims to try the deep understanding of computational processes by effectively solving the problems with mathematical models, tools and techniques. This understanding is important for its applications, which span various computational models such as algorithms, compilers, and VLSI design to the creation of intelligent technology, cognitive psychology, and philosophy. Automata enables scientists to understand how machines compute functions and solve problems. The main motivation behind the development of automata theory was the development of methods for describing and analyzing the dynamic behavior of discrete systems. To conduct a thorough computational study, computer scientist's work with a mathematical abstraction of computers called a computational model. Several models are used, but the most studied is the Turing machine. Computer scientists study the Turing machine because it is easy to formulate, can be analyzed and used to prove results, and because it represents what many consider to be the most powerful possible reasonable computational model. It may seem that the potentially infinite memory capacity is an unrealizable attribute, but any decidable problem solved by a Turing machine always requires a finite amount of memory. So, in principle, any problem that can be solved by a Turing machine can be solved by a computer that has a finite amount of memory.

Organization of the Book

The content coverage in this edition is organized into seven chapters. This edition is completely revised and new topics are added. The book addresses the fundamental questions about the nature of computation while at the same time helping we better understand the ways in which we interact with the computer.

Chapter one presents the fundamentals of computers. It explores on computer basics, peripheral devices, and computer software.

Chapter Two presents the basics of finite 'automata' that has some typical pronounceable symmetry with the term 'automatic'. In a computer, all processes appear to be done automatically. A given input is processed in the CPU, and an output is generated.

Chapter Three will give you a great understanding of pushdown automata, which is the machine format of the context-free language. It is the same as finite automata with the attachment of an auxiliary amount of storage as stack.

Chapter Four is about turing machine that manipulates symbols on a strip of tape according to a table of rules.

Chapter Five introduces the concept of algorithm for solving a problem, based on conducting a sequence of specified actions. A computer program can be viewed as an elaborate algorithm.

Chapter Six reveals on distributed computing used to solve complex computational problems that cannot be completed within a reasonable amount of time on a single computer.

Chapter Seven focuses on computational complexity. Computational complexity is a computer science concept that focuses on the amount of computational resources required to perform specific types of tasks. In the theory of computational complexity, researchers assess the types of resources required for a particular type or class of task in order to classify different types of tasks into different levels of complexity.

CHAPTER

1

COMPUTER FUNDAMENTALS

Learning Objectives

After studying this chapter, you will be able to:

1. Describe the computers: basic
2. Explain the peripheral device
3. Define the computer software

INTRODUCTION

A computer is a programmable electronic device that accepts raw data as input and processes it with a set of instructions (a program) to produce the result as output. It renders output just after performing mathematical and logical operations and can save the output for future use. It can process numerical as well as non-numerical calculations. The term "computer" is derived from the Latin word "computare" which means to calculate.

A computer is designed to execute applications and provides a variety of solutions through integrated hardware and software components. It works with the help of programs and represents the decimal numbers through a string of binary digits. It also has a memory that stores the data, programs, and result of processing. The components of a computer such as machinery that includes wires, transistors, circuits, hard disk are called hardware. Whereas, the programs and data are called software.

Below is a picture of a computer with each of the main components. You can see the desktop computer, flat-panel display, speakers, keyboard, and mouse in the picture below. We've also labeled each of the input devices and output devices.



1.1 COMPUTERS: BASIC

A computer is a device that accepts information (in the form of digitalized data) and manipulates it for some result based on a program or sequence of instructions on how the data is to be processed. Complex computers also include the means for storing data (including the program, which is also a form of data) for some necessary duration. A program may be invariable and built into the computer (and called logic circuitry as it is on microprocessors) or different programs may be provided to the computer (loaded into its storage and then started by an administrator or user). Today's computers have both kinds of **programming**.

Keyword

Programming is the process of taking an algorithm and encoding it into a notation, a programming language, so that it can be executed by a computer.



A Computer is an electronic device that can perform activities that involve mathematical, logical and graphical manipulations. Generally, the term is used to describe a collection of devices that function together as a system.

It performs these some operations in sequence.

- It receives data and instructions from the input device
- Processes the data as per instructions
- Provides the result (output) in a desired form

Advantages of Computers

The advantages of computers are given as:

- *High Speed*: Computers have the ability to perform routine tasks at a greater speed than human beings. They can perform millions of calculations in seconds.
- *Accuracy*: Computers are used to perform tasks in a way that ensures accuracy.
- *Storage*: Computers can store large amount of information. Any item of data or any instruction stored in the memory can be retrieved by the computer at lightning speeds.
- *Automation*: Computers can be instructed to perform complex tasks automatically.
- *Diligence*: Computers can perform the same task repeatedly and with the same accuracy without getting tired.
- *Versatility*: Computers are flexible to perform both simple and complex tasks.
- *Cost Effectiveness*: Computers reduce the amount of paper work and human effort, thereby reducing costs.

Disadvantages of Computers

The disadvantages are as:

- Computers need clear and complete instructions to perform a task accurately. If the instructions are not clear and complete, the computer will not produce the required result.
- Computers cannot think.
- Computers cannot learn by experience.

1.1.1 Generations of Computers

Generation in computer talk is a step in technology. It provides a framework for the growth of computer industry. Originally, the term "generation" was used to distinguish between varying hardware technologies but it has now been extended to include both hardware and software that together make up a computer system.

The custom of referring to computer era in terms of generations came into wide use only after 1964. There are totally five computer generations known till today. Although there is a certain amount of overlap between different generations, the approximate period shown against each are normally accepted.

During description of various computer generations, someone will come across several new terminologies and computer jargons that one may not be able to understand properly. However, the idea here is to provide an overview of the major developments and technologies during the five generations of computers.

Each generation along with its identifying characteristics are defied as:

First Generation (19421955)

We have discussed about some of the early computers ENIAC, EDVAC, EDSAC, UNIVAC 1, and IBM 701. These machine and others of their time used thousands of vacuum tubes. A vacuum tube was a fragile glass device, which used filaments as a source of electronics and could control and amplify electronic signals. It was the only high-speed electronic switching device available in those days. These vacuum tubes computer could perform computations in milliseconds and were referred to as first-generation computers.

The memory of these computers used electromagnetic relays, and all data and instructions were fed into the system from punched cards. The instructions were written in machine and assembly languages because high-level programming languages were introduced much later. Since machine and assembly languages are very difficult to work with, only a few specialists understood how to program these early computers.

Characteristic features of first-generation computers are as follows:

- They were the fastest calculating devices of their time.
- They were too bulky in size, requiring large rooms for installation.
- They used thousands of vacuum tubes that emitted large amount of heat and burnt out frequently. Hence, the rooms/areas in which these computers were located had to be properly air-conditioned.
- Each vacuum tube consumed about half a watt of power. Since a computer

typically used more than ten thousand **vacuum tubes**, power consumption of these computers was very high.

- As vacuum tubes used filaments, they had a limited life. Because a computer used thousands of vacuum tubes, these computers were prone to frequent hardware failures.
- Due to low mean time between failures, these computers required constant maintenance.
- In these computers, thousands of individual components were assembled manually by hand in to electronic circuits. Hence, commercial production of these computers was difficult and costly.
- Since these computers were difficult to program and use, they had limited commercial use.

Keyword

Vacuum tube is a device that controls electric current flow in a high vacuum between electrodes to which an electric potential difference has been applied.

Second Generation (19551964)

John Bardeen, Willian Shockley, and Walter Brattain invented a new electronic switching device called transistor at Bell Laboratories in 1947. Transistors soon proved to be a better electronic switching device than vacuum tubes due to their properties:

- They were more rugged and easier to handle than tubes since they were made of germanium semiconductor material rather than glass.
- They were highly reliable as compared to tubes since they had no parts like a filament that could burn out.
- They could switch much faster (almost ten times faster) than tubes. Hence, switching circuits made of transistors could operate much faster than their counterparts made of tubes.
- They consumed almost one-tenth the power consumed by tube.
- They were much smaller than a tube.
- They were less expensive to produce.
- They dissipated much less heat as compared to vacuum tubes.

The second-generation computers were manufactured using transistors instead of vacuum tubes. Due to the properties of

transistors listed above, these computers were more powerful, more reliable, less expensive, smaller, and cooler to operate than the first-generation computers. They used magnetic cores for main memory, and magnetic disk and tape as secondary storage media. Punched cards were still popular and widely used for preparing and feeding programs and data to these computers.

On software front, high-level programming languages (like FORTRAN, COBOL, ALGOL, and SNOBOL) and batch operating system emerged during second generation. High-level programming languages were easier for people to understand and work with than assembly or machine languages making second-generation computers easier to program and use than first-generation computers. Introduction of batch operating system enabled multiple jobs to be batched together and submitted at a time. A batch operating system causes automatic transition from one job to another as soon as the former job completes. This concept helped in reducing human intervention while processing multiple jobs resulting in faster processing, enhanced throughput, and easier operation of second-generation computers.

The first-generation computers were mainly used for scientific computations. However, the second-generation computers were increasingly used in business and industry for commercial data processing applications like payroll, inventory control, marketing, and production planning.

Characteristic features of second-generation computers are as follows:

- They were more than ten times faster than the first-generation computers.
- They were smaller than first-generation computers and required smaller space.
- They consumed less power and dissipated less heat than the first-generation computers. The rooms/areas in which the second-generation computers were located still required to be properly air conditioned.
- They were more reliable and less prone to hardware failures than the first-generation computers.
- They had faster and larger primary and secondary storage as compared to first-generation computers.
- They were easier to program and use than the first-generation computers. Hence, they had wider commercial use.
- In these computers, thousands of individual transistors had to be assembled manually by hand into electronic circuits making commercial production of these computers difficult and costly.

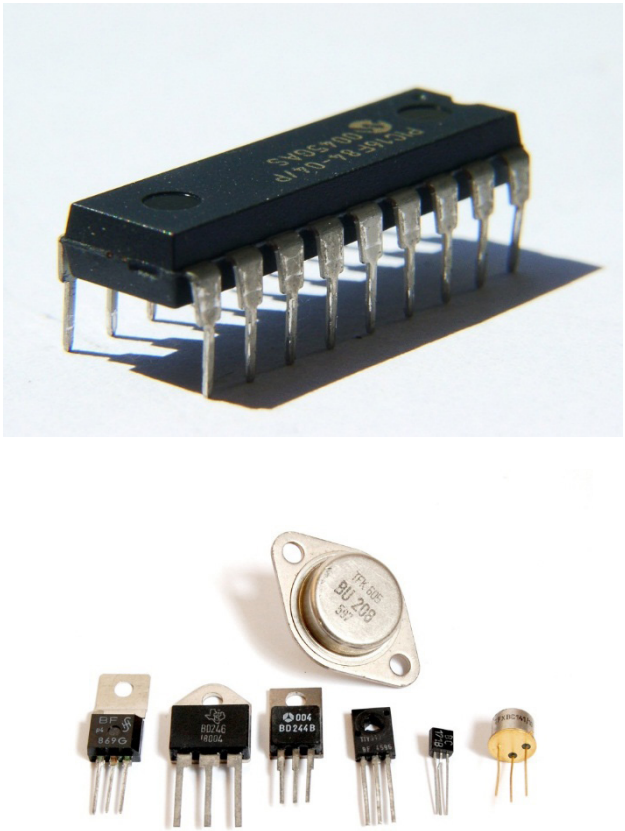


Figure 1.1: Electronic devices used for manufacturing computers of different generations.

Third Generation (1964-1975)

In 1958, Jack Clair Kilby and Robert Noyce invented the first integrated circuit. Integrated circuits (called *ICs)* are circuits consisting of several electronic components like transistors, resistors, and capacitors grown on a single chip of silicon eliminating wired interconnection between components. The IC technology was also known, 0 technologies because it made it possible to integrate larger number of circuit components into very small (less than 5 mm square) surface of silicon, known as "chip" See Figure 1.1. Initially the integrated circuits contained only about ten to twenty components. This technology was named small scale integration (SSI*)*. Later with the advancement in technology for manufacturing ICs, it became possible to integrate up to about components on a single chip. This technology came to be known as medium scale integration (MSI). Computers built using integrated circuits characterized the third generation. Earlier ones used SSI technology and later used MSI technology. The ICs were smaller, less expensive to produce, more rugged and reliable, faster in operations, dissipated less heat, and consumed less power than

circuits built by wiring electronic components manually. Hence third-generation computers were more powerful, more reliable, less expensive, smaller, and cooler to operate than second-generation computers.

Parallel advancements in storage technologies allowed construction of larger magnetic core based random access memory as well as larger capacity magnetic disks and tapes. Hence, third-generation computers typically had few megabytes (less than 5 Megabytes) of main memory and magnetic disks capable of storing few tens of megabytes of data per disk drive.

On software front, standardization of high-level programming languages, timesharing operating systems, unbundling of software from hardware, and creation of an independent software industry happened during third generation. FORTRAN and COBOL were the most popular high-level programming languages in those days.

American National Standards Institute (ANSI) standardized them in 1966 and 1968 respectively, and the standardized versions were called ANSI FORTRAN and ANSI COBOL. The idea was that as long as programmer follows these standards in program writing, a program could be run on any computer with an ANSI, FORTRAN or ANSI COBOL compiler. Some more high-level programming languages were introduced during the third-generation period. Notable among these were PL/1, PASCAL, and BASIC.

We saw that second-generation computers used batch operating system. In these systems, users had to prepare their data and programs and then submit them to a computer center for processing. The operator at the computer centre collected these user jobs and fed them to a computer in batches at scheduled intervals. The respective users then collected their job's output from the computer center. The inevitable delay resulting from this batch processing approach was very frustrating to some users, especially programmers, because often they had to wait for days to locate and correct a few program errors. To rectify this situation, John Kemeny and Thomas Kurtz of Dartmouth College introduced the concept of timesharing operating system. Timesharing operating system enables multiple users to directly access and share computing resources simultaneously in a manner that each user feels that no one else is using the computer. This is accomplished by using a large number of independent relatively low-speed, on-line terminals connected to the main computer simultaneously. Introduction of timesharing concept helped in drastically improving the productivity of programmers and made on-line systems feasible, resulting in new on-line applications like airline reservation systems, interactive query systems, and so on.

Until 1965, computer manufacturers sold their hardware along with all associated software without any separate charges for software.

For example, buyers received language translators for all languages supported on a computer they purchased. From user's standpoint, software was free. However, the situation changed in 1969 when IBM and other computer manufacturers began to price their hardware and software products separately. This unbundling of software from hardware gave users an opportunity to invest only in software of their need and value.

For example, now buyers could purchase only the language translators they needed and not all language translators supported on the purchased computer. This led to the creation of many new software houses and the beginning of an independent software industry.

Development and introduction of minicomputers also took place during the third-generation period. Computers built until the early 1960s were mainframe systems that only very large companies could afford to purchase and use. Clearly, a need existed for low-cost smaller computers to fill the gaps left by the bigger, faster, and costlier mainframe systems. Several innovators recognized this need and formed new firms in the 1960s to produce smaller computers. Digital equipment corporation (DEC) introduced the first commercially available minicomputer, the PDP-8 (programmed data processor), in 1965. It could easily fit in the corner of a room and did not require attention of a full-time computer operator. It used timesharing operating system and a number of users could access it simultaneously from different locations in the same building. Its cost was about one-fourth the cost of a traditional mainframe system making it possible for smaller companies to afford computers. It confirmed the tremendous demand for small computers for business and scientific applications, and by 1971, there were more than 25 computer manufacturers in minicomputer market.

Characteristic features of third-generation computers are as follows:

- They were more powerful than second-generation computers. They were capable of performing about1 million instructions per second.
- They were smaller than second-generation computers requiring smaller space.
- They consumed less power and dissipated less heat than second-generation computers. The rooms/areas in which third-generation computers were located still required to be properly air-conditioned.
- They were more reliable and less prone to hardware failures than second-generation computers requiring lower maintenance cost.
- They had faster and larger primary and secondary storage as compared to second-generation computers.
- They were general-purpose machines suitable for both scientific and commercial applications.
- There manufacturing did not require manual assembly of individual

components into electronic circuits resulting in reduced human labor and cost involved at assembly stage. Hence, commercial production of these systems was easier and cheaper. However, highly sophisticated technology and expensive setup was required for the manufacture of IC chips.

- Standardization of high-level programming languages allowed programs written for one computer to be easily ported to and executed on another computer.
- Timesharing operating system allowed interactive usage and simultaneous use of these systems by multiple users.
- Timesharing operating system helped in drastically improving the productivity of programmers down the time and cost of program development by several fold.
- Timesharing operating system also made on-line systems feasible resulting in usage of these systems oil-line applications.
- Unbundling of software from hardware gave users of these systems an opportunity to invest only in software of their need and value.
- Minicomputers of third-generation made computers affordable even by smaller companies.

Keyword

Large-scale integration (LSI) is the process of integrating or embedding thousands of transistors on a single silicon semiconductor microchip.

Fourth Generation (19751989)

Average of electronic components packed on a silicon chip doubled each year after 1965. This progress soon led to the era of **large scale integration (LSI)** when it was possible to integrate over 30,000 electronic components on a single chip, followed by very large scale integration (VLSI) when it was possible to integrate about one million electronic components on a single chip. This progress led to a dramatic development creation of a microprocessor. A microprocessor contains all circuits needed to perform arithmetic logic and control: functions, the core activities of all computers, on a single chip. Hence, it became possible to build a complete computer with a Microprocessor, a few additional primary storage chips, and other support circuitry. It started a new social revolution - personal computer (PC) revolution. Overnight computers became incredibly compact. They become inexpensive to make, and suddenly it became possible for anyone to own a computer.

During fourth generation, semiconductor memories replaced magnetic core memories resulting in large random access memories with very fast access time. On the other hand, hard disks became cheaper, smaller, and larger in capacity. In addition to magnetic tapes, floppy disks became very popular as a portable medium for porting programs and data from one computer system to another.

Another significant development during fourth-generation period was spread of high-speed computer networking enabling interconnection of multiple computers to enable them to communicate and share data. Local area networks (LANs) became popular for connecting computers within an organization or within a campus. Similarly, wide area networks (WANs) became popular for connecting computers located at larger distances. This gave rise to networks computers and distributed systems.

On software front several new developments emerged to match the new technologies of fourth generation.

For examples, several new operating systems were developed for PCs. Notable among these were MS-DOS, MS-windows, and apple's propriety OS. Since PCs were for individuals who were not computer professionals, companies developed graphical user interfaces for making computers more user friendly (easier to use). A graphical user interface (GUI) provides icons (pictures) and menus (list of choices) that users can select with a mouse. This enables new computer users to learn to use a PC quickly. Several new PC-based applications were also developed to make PCs a powerful tool. Notable among these were powerful word processing packages that allowed easy development of documents, spread sheet package that allowed easy manipulation and analysis of data organized in columns and rows, and graphics packages that allowed easy drawing of pictures and diagrams. Another very useful concept that became popular during fourth-generation period was that of multiple windows on a single terminal screen. This feature allowed users to see the current state of several applications simultaneously in separate windows on same terminal screen. During fourth-generation period, UNIX operating system and C programming language also became very popular.

Characteristic features of fourth-generation computers are as follows:

- The PCs were smaller and cheaper than mainframes or minicomputers of third generation.
- Although the fourth-generation mainframes required proper air-conditioning of the rooms/areas in which they were located, no air-conditioning was required for PCs.
- They consumed less power than third-generation computers.
- They were more reliable and less prone to hardware failures than third-generation computers requiring negligible maintenance cost

- They had faster and larger primary and secondary storage as compared to third-generation computers.
- They were general-purpose machines.
- Their manufacturing did not require manual assembly of individual components into electronic circuits resulting in reduced human labour and cost involved at assembly stage. Hence, commercial production of these systems was easier and cheaper. However, highly sophisticated technology and expensive setup was required for manufacturing LSI and VLSI chips.
- Use of standard high-level programming languages allowed programs written for one computer to be easily ported to and executed on another computer.
- Graphical user interface (GUI) enabled new users to quickly learn how to use computers.
- PC-based applications made PCs a powerful tool for both office and home usage.
- Network of computers enabled sharing of resources like disks, printers, and so on among multiple computers and their users. They also enabled several new types of applications involving interaction among computer users at geographically distant locations. Computer supported cooperative working (CSCW) or groupware is one such application in which multiple members working on a single project and located at distant locations cooperate with each other by using a network of computers.
- In addition to unbundled software, these systems also used add-on hardware feature that allowed users to invest only in the hardware configuration and software of their need and value.
- The PCs of fourth generation made computers affordable even by individuals for their personal use at home.

Fifth Generation (1989Present)

The trend of miniaturization of electronic components, dramatic increase in power of microprocessor -hips, and increases in capacity of main memory and hard disk continued during fifth generation. VLSI technology came from ULSI ((Ultra Large Scale Integration) technology in fifth generation resulting in production of Microprocessor chips having ten million electronic components. In fact, the speed of microprocessors and the size of main memory and hard disk doubled almost every eighteen months. As a result, many features found in the CPUs of large mainframe systems of third- and fourth-generation systems became part of **microprocessor** architecture in fifth generation. This ultimately resulted in availability of very powerful and compact computers becoming available at cheaper rates and death of traditional large mainframe systems. Due to this fast space of advancement in computer technology, we see more

compact and more powerful computers being introduced almost every year at more or less the same price or even cheaper. Notable among these are portable notebook computers that give the power of a PC to their users even while travelling, powerful desktop PCs and workstations, powerful servers, and very powerful supercomputers

Storage technology also advanced very fast making larger and larger main memory and disk storage available in newly introduced systems. During fifth generation, optical disks also emerged as a popular portable mass storage CD-ROM (compact disk read only memory).

During fifth generation, there was tremendous outgrowth of computer networks. Communication technologies become faster day by-day and more and more computers were networked together. This trend resulted in emergence and popularity of the Internet and associated technologies and applications. The Internet made it possible for computer users sitting across the globe to communicate with each other within minutes by use of electronic mail (known as e-mail) facility. A vast ocean of information became readily available to computer users through the World Wide Web (known as WWW). Moreover, several new types of exciting applications like electronic commerce, virtual libraries, virtual classrooms, distance education, and so on, emerged during the period.

Keyword

Microprocessor is a single IC package in which several useful functions are integrated and fabricated on a single silicon semiconductor chip.

Tremendous processing power and massive storage capacity of fifth-generation computers also made them a very popular tool for a wide range of multimedia applications that deal with information containing text, graphics animation, audio, and video data. In general, the data size of multimedia information is much larger than textual information because representation of graphics, animation, audio, or video media in digital form requires larger number of bits than that required for representation of textual information.

Characteristic features of fifth-generation computers are as follows:

- Portable PCs (called notebook computers) are much smaller and handy than PCs of fourth generation allowing users to use computing facility even while travelling.

- Fifth-generation desktop PCs and workstations are several times more powerful than PCs of fourth generation.
- Although fifth-generation mainframes require proper air-conditioning of the rooms/areas in which they are located, no air-conditioning is normally required for notebook computers, desktop PCs, and workstations.
- They consume less power than their predecessors do.
- They are more reliable and less prone to hardware failures than their predecessors were, requiring negligible maintenance cost.
- Many of the large-scale fifth-generation systems have hot-plug feature that enables a failed component to be replaced with a new one without the need to shut down the system. Hence, the uptime of these systems is very high.
- They have faster and larger primary and secondary storage as compared to their predecessors.
- They are general-purpose machines.
- Their manufacturing does not require manual assembly of individual components into electronic circuits resulting in reduced human labour and cost involved at assembly stage. Hence, commercial production of these systems is easier and cheaper. However, highly sophisticated technology and expensive setup (affordable only by a few organizations in the world) is required for manufacturing IJLSI chips.
- Use of standard high-level programming languages allows programs written for one computer to be easily ported to and executed on another computer.
- More user-friendly interfaces with multimedia features make the systems easier to learn and use by anyone, including children.
- Newer and more powerful applications, including multimedia applications, make the systems more useful in every occupation.
- Explosion in the size of the Internet coupled with Internet-based tools and applications have made these systems influence the life of even common people.
- These systems also use the concept of unbundled software and add-on hardware allowing the users to invest only in the hardware configuration and software of their need and value.
- With so many types of computers in all price ranges today, we have a computer for almost any type of user whether the user is a child or a world-fame scientist.

We have looked at the history of computing divided into five generations and we have seen how quickly things have changed in the last few decades. Technological

progress in this area is continuing at the same rate even now. In fact, the fastest-growth period in the history of computing may be still ahead.

1.1.2 Types of Computers

A computer is one of the most brilliant inventions of mankind. With the help of computer technology, we were able to achieve storage and processing of huge amounts of data; we could rest brains by employing computer **memory** capacities for storage of information. Owing to computers, we have been able speed up daily work, carry out critical transactions and achieve accuracy and precision at work. Computers of the earlier times were of the size of a large room and were required to consume huge amounts of electric power. However, with the advancing technology, computers have shrunk to the size of a small watch. Depending on the processing power and size of computers, they have been classified under various types.

Keyword

Memory refers to the computer hardware devices used to store information for immediate use in a computer; it is synonymous with the term "primary storage".

The classifications of computers of computer are given as:

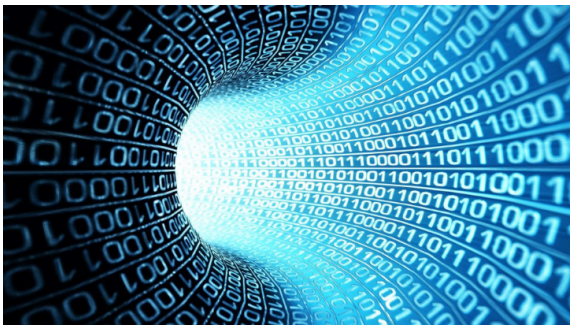
Operational based

Based on the operational principle of computers, they are categorized as analog, digital and hybrid computers.

- *Analog Computers:* These are almost extinct today. These are different from a digital computer because an analog computer can perform several mathematical operations simultaneously. It uses continuous variables for mathematical operations and utilizes mechanical or electrical energy.



- *Digital Computers*: They use digital circuits and are designed to operate on two states, namely bits 0 and 1. They are analogous to states ON and OFF. Data on these computers is represented as a series of 0s and 1s. Digital computers are suitable for complex computation and have higher processing speeds. They are programmable. Digital computers are either general purpose computers or special purpose ones. General purpose computers, as their name suggests, are designed for specific types of **data** processing while general purpose computers are meant for general use.



Keyword

Data is a set of values of qualitative or quantitative variables; restated, pieces of data are individual pieces of information.

- *Hybrid Computers:* These computers are a combination of both digital and analog computers. In this type of computers, the digital segments perform process control by conversion of analog signals to digital ones.



Sizes and Processing Powers

This was the classification of computers based on their style of functioning.

There are classification of the different types of computers based on their sizes and processing powers:

- *Mainframe Computers:* Large organizations use mainframes for highly critical applications such as bulk data processing and ERP. Most of the mainframe computers have capacities to host multiple operating systems and operate as a number of virtual machines. They can substitute for several small servers.
- *Microcomputers*: A computer with a microprocessor and its central processing unit is known as a microcomputer. They do not occupy space as much as mainframes do. When supplemented with a keyboard and a mouse, microcomputers can be called personal computers. A monitor, a keyboard and other similar input-output devices, computer memory in the form of RAM and a **power supply unit** come packaged in a microcomputer. These computers can fit on desks or tables and prove to be the best choice for single-user tasks.



Personal Computers

Personal computers come in different forms such as desktops, laptops and personal digital assistants.

These types of computers are:

- *Desktops*: A desktop is intended to be used on a single location. The spare parts of a desktop computer are readily available at relatively lower costs. Power

DID YOU KNOW?

The differential analyzer, a mechanical analog computer designed to solve differential equations by integration using wheel-and-disc mechanisms, was conceptualized in 1876 by James Thomson, the brother of the more famous Lord Kelvin.

Keyword

Power Supply Unit converts mains AC to low-voltage regulated DC power for the internal components of a computer.

consumption is not as critical as that in laptops. Desktops are widely popular for daily use in the workplace and households.

- *Laptops*: Similar in operation to desktops, laptop computers are miniaturized and optimized for mobile use. Laptops run on a single battery or an external adapter that charges the computer batteries. They are enabled with an inbuilt keyboard, touch pad acting as a mouse and a liquid crystal display. Their portability and capacity to operate on battery power have proven to be of great help to mobile users.
- *Notebooks*: They fall in the category of laptops, but are inexpensive and relatively smaller in size. They had a smaller feature set and lesser capacities in comparison to regular laptops, at the time they came into the market. But with passing time, netbooks too began featuring almost everything that notebooks had. By the end of 2008, netbooks had begun to overtake notebooks in terms of market share and sales.
- *Personal Digital Assistants (PDAs):* It is a handheld computer and popularly known as a palmtop. It has a touch screen and a memory card for storage of data. The PDAs can also be used as portable audio players, web browsers and smartphones. Most of them can access the Internet by means of Bluetooth or Wi-Fi communication.
- *Minicomputers:* In terms of size and processing capacity, minicomputers lie in between mainframes and microcomputers. Minicomputers are also called mid-range systems or workstations. The term began to be popularly used in the 1960s to refer to relatively smaller third **generation** computers. They took up the space that would be needed for a refrigerator or two and used transistor and core memory technologies. The 12-bit PDP-8 minicomputer of the digital equipment corporation was the first successful minicomputer.



- *Servers:* They are computers designed to provide services to client machines in a computer network. They have larger storage capacities and powerful processors. Running on them are programs that serve client requests and allocate resources like memory and time to client machines. Usually they are very large in size, as they have large processors and many hard drives. They are designed to be fail-safe and resistant to crash.
- *Supercomputers:* The highly calculation-intensive tasks can be effectively performed by means of supercomputers. Quantum physics, mechanics, weather forecasting, molecular theory are best studied by means of supercomputers. Their ability of parallel processing and their well-designed memory hierarchy give the supercomputers, large transaction processing powers.
- *Wearable Computers*: A record-setting step in the evolution of computers was the creation of wearable computers. These computers can be worn on the body and are often used in the study of behavior modelling and human health. Military and health professionals have incorporated wearable computers into their daily routine, as a part of such studies. When the users' hands and sensory organs are engaged in other activities, wearable computers are of great help in tracking human actions. Wearable computers do not have to be turned on and off and remain in operation without user intervention.
- *Tablet Computers:* Tablets are mobile computers that are very handy to use. They use the touch screen technology. Tablets come with an onscreen keyboard or use a stylus or a digital pen. Apple's iPod redefined the class of tablet computers.

Keyword

Generation is the term used to distinguish between varying hardware technologies but it has now been extended to include both hardware and software that together make up a computer system.



1.1.3 Fundamental Uses of Computers

With technology continuously evolving and intertwining into everyday life, people are using computers for more and more simple tasks. This is possible, in no small part to the continued efforts of computer manufacturers as well as software designers to simplify computing so that anyone can sit down, power on the machine and jump right in.

Communication

The world has gotten much smaller in recent years by the use of computer and Internet. Many people use their computers to keep in touch with friends and family using instant messenger programs as well as email. A growing communication tool is social networking, with sites like *Facebook* and *Twitter* becoming incredibly popular.

Games

The PCs have long served as recreational devices with hundreds of games available each year. Gaming on a PC can be an expensive hobby, with video cards ranging in price from $ 60 to more than $ 20,000 and fully equipped gaming PCs costing in excess of $ 1500 in many cases. For all the top-tier AAA titles, there are other games that users can find both pre-installed on PCs as well as online.

Entertainment

Almost all computers come with CD or DVD disk drives, which allow one to use the computer as a CD player or DVD player. Some computers are also capable, with the proper hardware, of viewing and recording television onto the machine's hard drives. With an Internet connection, users have a nearly limitless of videos and music available online as well.

Work

Almost every working environment uses computers in one capacity or another. Office buildings use computers to keep track of everything from pay wages to hours logged, retail stores use computers as cash registers and industries such as construction and architecture use computers to help design buildings.

1.2 PERIPHERAL DEVICE

A peripheral is a "device that is used to put information into or get information out of the computer."

There are three different types of peripherals:

- Input, used to interact with, or send data to the computer (mouse, keyboards, etc.)
- Output, which provides output to the user from the computer (monitors, printers, etc.)
- Storage, which stores data processed by the computer (hard drives, flash drives, etc.)

A peripheral device is generally defined as any auxiliary device such as a computer mouse or keyboard that connects to and works with the computer in some way. Other examples of peripherals are expansion cards, graphics cards, image scanners, tape drives, microphones, loudspeakers, webcams, and digital cameras. RAM—random access memory—straddles the line between peripheral and primary component; it is technically a storage peripheral, but is required for every major function of a modern computer and removing the RAM will effectively disable any modern machine. Many new devices such as digital watches, smartphones and tablet computers have interfaces which allow them to be used as a peripheral by a full computer, though they are not host-dependent as other peripheral devices are.

1.2.1 Input device

In computing, an input device is a peripheral (piece of computer hardware equipment) used to provide data and control signals to an information processing system such as a computer or other information appliance.

An input device is any piece of hardware that can be used to send data into the computer system.

Input devices are used to enter data and instructions into computer memory from external world. Examples include:

- Keyboard,
- Mouse,
- Track ball,
- Joy stick,
- Touch screen,
- Light pen,

- Scanner,
- Digital camera,
- Microphone,
- Graphics tablet.

Keyboard

Keyboard is the most familiar input device. It is most widely used input device to enter data and instructions into the computer. Keyboard has a set of keys like a typewriter. It has alphabetic keys for A, B, C…Z or a, b, c, z. It has numeric keys like 0, 1, 2, 3, 9. It has function keys F1, F2,…F12 used to perform specific tasks. It has other keys used for editing like delete, backspace, home, end, etc.



Mouse

Mouse is an input device used to control motion of pointer on screen. A mouse has two or three buttons called left, right and middle button. Buttons are used to perform different functions.



Track Ball

Track ball is an input device like a mouse. It is used in lap top computers to control motion of pointer on screen. It is a pointing device like upside down mouse. It has a ball on its upper side. This ball is moved by fingers or thumb and the pointer moves accordingly on screen.

Joy stick

A joystick is an input device that allows the user to control a character or machine in a computer program, such as a plane in a flight simulator. They look similar to the control device you would find on an arcade game, but nearly always include extra buttons for additional functionality.

There are some computer joystick ports, such as:

Today, most computer joysticks connect to the computer using a USB port. Below is a listing of all of the type of ports that have accepted a joystick.

- Bluetooth
- Game port
- Serial Port
- USB



Touch screen

A touchscreen is a display that also serves as an input device. Some touchscreens require a proprietary pen for input, though most modern touchscreens detect human

touch. Since touchscreen devices accept input directly through the screen, they do not require external input devices, such as mice and keyboards. This makes touchscreens ideal for computer kiosks, as well as portable devices, such as tablets and smartphones.

While a touchscreen may look like an ordinary display, the screen includes several extra layers that detect input. The first layer is a hard protective layer that protects the actual display and the touchscreen components. Beneath the protective layer is an electronic grid that detects input. Most modern touchscreens use capacitive material for this grid, in which the electrical charge changes wherever the screen is touched. Beneath the touchscreen layer is the LCD layer, which is used for the actual display.

While early touchscreens could only detect a single point of input at a time, modern touchscreens support "multi-touch" input. This technology, which was made popular by the original iPhone, enables the screen to detect multiple finger motions at once.

For example, on some touchscreen devices, you can rotate an image by twisting three fingers in a clockwise or counterclockwise motion. Many touchscreen applications also allow you zoom in and out by spreading two fingers apart or pinching them together.

Keyword

Touch screen is an electronic display screen that is also an input device. A user interacts with the computer, tablet, smartphone or touch-controlled appliance by using hand gestures and fingertip movements to tap pictures, moving elements or type words on the screen.

Thanks to multi-touch and other improvements in touchscreen technology, today's touchscreens are easier and more natural to use than they used to be. In fact, improved **touchscreen technology** has greatly contributed to the popularity of the iPad and other tablet PCs.

Light pen

A light pen is a light-sensitive computer input device, basically a stylus that is used to select text, draw pictures and interact with user interface elements on a computer screen or monitor. The light pen works well with CRT monitors because of the way such monitors scan the screen, which is one pixel at a time, giving the computer a way to keep track of the expected scanning time by the electron beam and infer the pen's position based on the latest timestamp of the scanning.

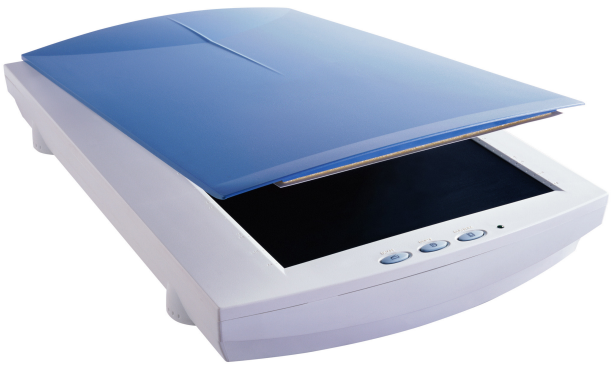
Scanner

A scanner is an input device that scans documents such as photographs and pages of text. When a document is scanned, it is converted into a digital format. This creates an electronic version of the document that can be viewed and edited on a computer.

Most scanners are flatbed devices, which means they have a flat scanning surface. This is ideal for photographs, magazines, and various documents. Most flatbed scanners have a cover that lifts up so that books and other bulky objects can also be scanned. Another type of scanner is a sheet-fed scanner, which can only accept paper documents. While sheet-fed scanners cannot scan books, some models include an automatic document feeder, or ADF, which allows multiple pages to be scanned in sequence.

Scanners work in conjunction with computer software programs, which import data from the scanner. Most scanners include basic scanning software that allows the user to configure, initiate, and import scans. Scanning plug-ins can also be installed, which allow various software programs to import scanned images directly.

If a scanner plug-in is installed for Adobe Photoshop, a user can create new images in Photoshop directly from the connected scanner.



While Photoshop can edit scanned images, some programs like Acrobat and OmniPage can actually recognize scanned text. This technology is called optical character recognition, or OCR. Scanning software that includes OCR can turn a scanned text

document into a digital text file that can be opened and edited by a word processor. Some OCR programs even capture page and text formatting, making it possible to create electronic copies of physical documents.

Digital camera

A digital camera records and stores photographic images in digital form. Many current models are also able to capture sound or video, in addition to still images. Capture is usually accomplished by use of a photo sensor, using a charged coupled device (CCD).These stored images can be uploaded to a computer immediately or stored in the camera for to be uploaded into a computer or printer later. Images may also be archived on a photographic compact disc or external hard disk.

Microphone

A microphone is a device that captures audio by converting sound waves into an electrical signal. This signal can be amplified as an analog signal or may be converted to a digital signal, which can be processed by a computer or other digital audio device. Sometimes abbreviated as mic, a microphone is a hardware peripheral originally invented by Emile Berliner in 1877 allows computer users to input audio into their computers.

Below is a short list of all the different uses a microphone could be used for on a computer.

- Audio for a video
- Voice recorder
- VoIP
- Voice recognition
- Computer gaming
- Online chatting
- Recording voice for dictation, singing, and podcasts
- Recording musical instruments

Graphics tablet

A graphics tablet is a hardware input device that enables an artist to draw or sketch digitally using a pen or stylus. They are helpful because they provide a more natural and precise feel than a standard computer mouse. The picture to the right shows an example of a Pen Graphics Tablet from Wacom.

A graphics tablet consists of a special pen called stylus and a flat pad. The image is created on the monitor screen as the user draws it on the pad with the help of stylus (special pen).

Below is a list of professions and people who are more likely to use a graphics tablet.

- Architects and Engineers
- Artists
- Cartoonist
- Fashion designers
- Graphic designers
- Illustrators
- Photographers
- Teachers



1.2.2 Output Devices

An output device is a piece of hardware that is used receiving information from a computer. Some devices are both an input and output device and can transfer information in one of two directions depending on the current situation. A disk drive is an example of an input/output device. Some devices can be used only for output *for example*, a printer and a monitor. This is the most commonly used output device. It displays what someone have typed or otherwise entered in to the computer on the screen in front of one, monitors can be monochrome (black and white, black and green or black and amber), or color. Color monitors come in various types. Each type has a different number of colors to use and a different quality of picture (resolution). Higher resolution and better quality of pictures on screen, resolution is measured in pixels. The screen is divided into a grid. Each square on the grid is a pixel.

Any peripheral that receives or displays output from a computer. In the picture to the right, are an inkjet printer and a good example of an output device that can make a hard copy of anything on a computer. An output device is an electromechanical device that accepts data from a computer and translates them into a form suitable for use by outside world (users). Several output devices are available today.

There are some output devices, such as:

- Monitor
- Plotter
- Projector
- Sound Card
- Speaker
- Speech-Generating Device
- Printer
- Computer Output Microfilm

Monitor

A monitor is only used to display (output) information from a computer and provides no source of input.

Alternatively referred to as a video display terminal (VDT) and video display unit (VDU), a monitor is a device that displays video images to the user. A monitor is made up of circuitry, a screen, a power supply, and buttons to adjust screen settings, and casing that holds all of these components.



Like most early TVs, the first computer monitors were comprised of a CRT (Cathode Ray Tube) and a fluorescent screen. Today, all monitors are created using flat panel display technology, usually backlit with LEDs. The image to the right shows an ASUS LCD monitor.

Plotter

A plotter is a computer hardware device much like a printer that is used for printing vector graphics. Instead of toner, plotters use a pen, pencil, marker, or another writing tool to draw multiple, continuous lines onto paper rather than a series of dots like a traditional printer. Though once widely used for computer-aided design,

these devices have more or less been phased out by wide-format printers. Plotters are used to produce a hard copy of schematics and other similar applications.



Advantages of plotters

There are some advantages of plotters, such as:

- Plotters can work on very large sheets of paper while maintaining high resolution.
- They can print on a wide variety of flat materials including plywood, aluminum, sheet steel, cardboard, and plastic.
- Plotters allow the same pattern to be drawn thousands of times without any image degradation.

Disadvantages of plotters

There some disadvantages of plotters, such as:

- Plotters are quite large when compared to a traditional printer.
- Plotters are also much more expensive than a traditional printer.

Projector

A projector is an output device that can take images generated by a computer or Blu-ray player and reproduce them onto some surface, usually a large, flat, lightly colored one. *For example,* you could use a projector to show a presentation on a large screen so that everyone in the room can see it. Projectors can produce either still (slides) or moving images (videos). A projector is often about the size of a toaster and weighs only a few pounds. The image to the right is that of a ViewSonic projector.

The different ways a projector can be used today:

- Project a PowerPoint presentation at a business meeting.
- Project a computer screen to teach a class in school.
- Project a TV or computer with a movie playing to a large screen.
- Demo a product or service at a convention center.
- Transform a wall, house, or other object to give it a different appearance.



Sound Card

A sound card is an expansion card or IC for producing sound on a computer that can be heard through speakers or headphones. Although the computer does not need a sound device to function, they are included on every machine in one form or another, either in an expansion slot (sound card) or on the motherboard (onboard).



The picture is an example of a sound card audio ports or audio jacks on the back of your computer, associated colors, and the connector symbols.

- Digital Out (White or Yellow; words: "Digital" or "Digital Out") - Used with surround sound or loudspeakers.
- Sound in or line in (Blue; Arrow pointing into waves) - Connection for external audio sources, e.g. tape recorder, record player, or CD player.
- Microphone or Mic (Pink; Microphone) - The connection for a microphone or headphones.
- Sound out or line out (Green; Arrow pointing out of waves) - The primary sound connection for your speakers or headphones. This sound card also has a second (black) and third (orange) sound out connector.
- Firewire (Not pictured) - Used with some high-quality sound cards for digital video cameras and other devices.
- MIDI or joystick (15 pin yellow connector) - Used with earlier sound cards to connect MIDI keyboard or joystick.

Speaker

A speaker is a term used to describe the user who is giving vocal commands to a software program. A computer speaker is a hardware device that connects to a computer to generate sound. The signal used to produce the sound that comes from a computer speaker is created by the computer's sound card. The picture shows the Altec Lansing VS2221 speakers with subwoofer, an example of external computer speakers.

Speakers are made up of a cone, an iron coil, a magnet, and housing (case). When the speaker receives electrical input from a device, it sends the current through the causing it to move back and forth. This motion then vibrates the outer cone, generating sound waves picked up by our ears.

Speech-Generating Device

Speech-Generating Device (SGD), is an electronic output device that is used to help individuals with severe speech impairments or other issues that result in difficulty in communicating.

For example, SGDs can be utilized by those who have Lou Gehrig's disease (such as astrophysicist Steven Hawking, shown right) to supplement or replace speech or writing for them. Speech-generating devices also find use with children who are suspected of having speech deficiencies.

How do they work?

Although SGDs vary in their design, most consist of several pages of words or symbols

on a touch screen that the user may choose. As the individual makes their choices, suggestions are made for the next symbol or word based on what they might want to say. Additionally, the communicator can navigate the various pages manually if the predictive system fails. Most speech-generating can produce electronic voice output by utilizing either speech synthesis or via digital recordings of someone speaking.

Printer

A printer is an external hardware output device that takes the electronic data stored on a computer or other device and generates a hard copy of it.



For example, if you created a report on your computer you could print several copies to hand out at a staff meeting. Printers are one of the most popular computer peripherals and are commonly used to print text and photos. The picture to the right is an example of an inkjet computer printer, the Lexmark Z605.

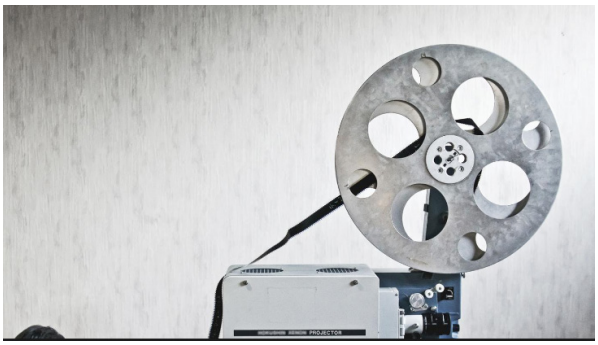
Types of printers

Today, the most common printers used with a computer are Inkjet and Laser printers.

- 3D printer
- All-in-one (AIO) printer
- Dot matrix printer
- Inkjet printer
- Laser printer
- LED printer
- Multifunction printer (MFP)
- Plotter
- Thermal printer

Computer Output Microfilm

Computer output microfilm is the product of copying information from electronic media onto microfilm. COM technology, with a history that dates back to the first patent for microphotography in 1839, has been used for document and newspaper archival since the 1920s. And, with the advent of the personal computer, computer-produced microforms are becoming more popular for non-library use, such as storing catalogs and patient records.



An important advantage to using computer output microfilm for document archival is its storage capacity. The Indiana Commission on Public Records states that a single microfiche card can hold 230 images, and a 1-cubic-foot storage box, containing 6,000 cards, can hold 1,380,000 images. The commission concluded that if each of these images were to be printed on a single sheet of paper, their storage would require 460 boxes, as each would only hold 3,000 pages.

In addition to increased storage efficiency, the commission found that newer COM technology offers superior image quality at a cost that is comparable to that of paper printing. If their per-sheet cost of printing a black-and-white document on standard-sized paper from a centralized printer is 3 cents, the same document could be produced on COM microfiche for 0.003 cents per sheet. These figures do not include the additional cost reduction from savings on storage space.

Remember

A read/write head only covers a part of the surface so that the head or medium or both must be moved relative to another in order to access data.

1.2.3 Storage Devices

A storage device is any computing hardware that is used for storing, porting and extracting data files and objects. It can

hold and store information both temporarily and permanently, and can be internal or external to a computer, server or any similar computing device.

Storage Device is a device for recording (storing) information (data). Recording can be done using virtually any form of energy. Storage device is a hardware device capable of storing information. There are two storage devices used in computers; a primary storage device such as computer RAM and a secondary storage device such as a computer hard disk drive.

Magnetic storage devices

Magnetic storage is one of the most common types of storage used with computers and is the technology that many computer hard drives use.

There are some magnetic storage devices, such as:

- Floppy diskette
- Hard drive
- Magnetic strip
- SuperDisk
- Tape cassette
- Zip diskette

Floppy Disk Drive

A Floppy Disk Drive, also called FDD or FD for short, is a computer disk drive that enables a user to save data to removable diskettes. Although 8" disk drives were first made available in 1971, the first real disk drives used were the 5 1/4" floppy disk drives, which were later replaced with the 3 1/2" floppy disk drives.



A 5 1/4"floppy disk was capable of storing between 360KB and 1.2MB of data, and the 3 1/2" floppy disk was capable of storing between 360KB and 1.44MB of data. For both sizes of floppy disk, the amount of data that could be stored was

dependent on whether the disk was single or double sided and whether the disk was regular or high density.

Hard drive

A hard disk drive (sometimes abbreviated as hard drive, HD, or HDD) is a non-volatile memory hardware device that permanently stores and retrieves data on a computer. A hard drive consists of one or more platters to which data is written using a magnetic head, all inside of an air-sealed casing. Internal hard disks reside in a drive bay, connect to the motherboard using an ATA, SCSI, or SATA cable, and are powered by a connection to the PSU (power supply unit).



Magnetic card

A magnetic card is a rectangular object that contains either a magnetic strip on the outside or a magnetic object within the card that contains data. A magnetic card may contain information about an individual such as available credit on a credit card or pass codes for entering secure buildings. The picture is an example of a Magnetic card being swiped through a magnetic card reader.

SuperDisk

Alternatively referred to as an LS-120 and LS-240, the SuperDisk is a disk drive and diskette introduced by 3M, which later became Imation. The drive was most popular with OEM computers, such as Compaq and Packard Bell computers.

The original SuperDisk (shown in the picture to the right) is capable of holding 120 MB on a single disk the same size of a traditional 1.44MB floppy diskette. Later SuperDisks were capable of holding 240MB on a disk. The SuperDisk drive was also backwards compatible with 1.44MB disks. Its available interfaces were IDE(ATAPI), parallel port, SCSI, and USB.

The SuperDisk did not become very popular primarily because of the Iomega Zip drive, which had already gained success before the SuperDisk was released. Today,

neither the SuperDisk nor the Zip drive are used because of technology like the CD-R disc and USB thumb drives.

Tape

A magnetically thin coated piece of plastic wrapped around wheels capable of storing data. Tape is much less expensive than other storage mediums, but it is a much slower solution that was typically only used for backup.

Today, tape has mostly been abandoned for faster and more reliable solutions like disc drives, hard drives, and flash drives. The image to the right is a picture and example of magnetic tape taken by KENPEI and shared under the creative commons.

Remember

Magnetic core memory would become the dominant form of memory until the development of transistor-based memory in the late 1960s.

Zip drive

A Zip drive and Zip disk is a hardware data storage device developed by Iomega that functions like a standard 1.44" floppy drive and diskette. What makes the Iomega Zip drive unique is its capability to hold up to 100 MB of data or 250 MB of data on the later models. Iomega Zip drives became very popular in late 1990s but quickly became less popular as users needed larger storage capabilities. The drive was eventually replaced by CD-R and CD-RW drives and discs as they became cheaper since they offered much more storage and compatibility.



Optical storage devices

Another common storage is optical storage, which uses lasers and lights as its method of reading and writing data.

There are some optical storage devices, such as:

- Blu-ray disc
- CD-ROM disc
- CD-R and CD-RW disc
- DVD-R, DVD+R, DVD-RW, and DVD+RW disc

Blu-ray Disc

The Blu-ray discs and drives are an upgrade and replacement to the CD and DVD discs and drives.

Blu-ray Disc, BD or BD-ROM, is an optical disc format developed by thirteen consumer electronics and PC companies.

For example, these companies include Dell, Hitachi, Hewlett Packard, LG, Mitsubishi, Panasonic, Sony, and TDK. Blu-ray was first introduced at the CES on January 4, 2006 and can store up to 25 GB single-layer disc and 50 GB on a dual-layer disc, each disc being the same size as a standard CD. The picture shows a blue laser in a Blu-ray disc player.

CD-ROM

A CD-ROM is an optical disc which contains audio or software data whose memory is read only. A CD-ROM Drive or optical drive is the device used to read them. CD-ROM drives have speeds ranging from 1x all the way up to 72x, meaning it reads the CD roughly 72 times faster than the 1x version. As you would imagine, these drives are capable playing audio CDs and reading data CDs. Below is a picture of the front and back of a standard CD-ROM drive.



CD-R

Alternatively referred to as a cd writer, CD-WO (Write once), WORM (Write Once Read Many) drive. CD-R is short for CD-Recordable and is a writable disc and drive that is capable of having information written to the disc once and then having that disc read many times after that. Data cannot be deleted from a CD-R disc and a CD-R disc cannot be formatted. If the data is not written to the disc properly, it cannot be corrected and is often jokingly referred to as a coaster.

How does the CD-R work?

A CD-R disc is coated with a photosensitive organic dye that allows a user to record information. Once the CD-R disc is placed within the computer, the recording process begins. The laser inside the drive heats the dye to reveal areas that diffuse the light like a traditional CD pit. The CD-R drive does not create pits on the CD; instead the burner creates reflective sections on the CD causing the computer's CD-ROM laser to interpret it as a pit. Once a CD-R disc is finished recording, the CD can be used in any standard CD-ROM, CD-R, CD-RW, or DVD drive. Although, there were issues with the first revision DVD drives reading CD-R disc, today's drives no longer suffer from any reading problems.

Recordable DVD drives

Recordable DVD drives are disc drives capable of creating DVD discs. Unfortunately, unlike recordable CD drives, there are many different competing standards for creating DVD discs. *For example*, DVD-R, DVD-RW, DVD+R, DVD+RW, DVD+R DL (DVD+R9), and DVD-RAM are all different competing standards. Below is a brief explanation of each of these standards and related links to each of these standards.

- *DVD-R:* DVD-R is a drive capable of recording once to a disc and reading many times after it has been created. DVD-R is an approved standard by DVD Forum and the drives are capable of recording to DVD-R discs, also known as DVD-5 and DVD-10 discs.
- *DVD-RW (DVD-R/W):* Short for Digital Versatile Disc-Read/Write, DVD-RW is an approved standard by DVD Forum. Similar to CD-RW, it is a technology that enables a user to read and write to a DVD-RW or DVD-R disc multiple times. DVD-RW drives are capable of recording to DVD-R and DVD-RW discs and are also known as DVD-5 and DVD-10 discs.
- *DVD+R:* Short for Digital Versatile Disc-Recordable, DVD+R is DVD+RW Alliance standard of disc. DVD+R is a technology that enables a user to read and write to a DVD+RW or DVD+R disc several times. DVD+RW drives are capable of recording DVD+R discs, also known as DVD-5and DVD-10 discs.

- *DVD+RW:* Short for Digital Versatile Disc-Read/Write, DVD+R is a DVD+RW Alliance standard much like CD-RW and DVD-R. With DVD+RW you can read and write to a DVD+RW or DVD+R disc several times. DVD+RW drives are capable of recording DVD+R and DVD+RW discs, also known as DVD-5 and DVD-10 discs.
- *DVD+R DL (DVD+R9):* Short for Digital Versatile Disc-Recordable Dual Layer, DVD+R DL is a DVD+RW Alliance standard that almost doubles the capacity of a DVD when compared with the DVD+R or DVD+RW discs. DVD+R DL are also known as DVD-9 and DVD-18 discs.
- *DVD-RAM:* Short for Digital Versatile Disc-Random Access Memory, DVD-RAM drives are a DVD+RW Alliance standard similar to CD-RW in that they enable users to read and write to a disc several times.

Flash memory devices

Flash memory has started to replace magnetic media as it becomes cheaper as it is the more efficient and reliable solution.

There are some flash memory devices, such as:

- Jump drive or flash drive
- Memory card
- Memory stick
- SSD

Remember

Nearly everything a computer programmer does requires him or her to consider how to manage memory. Even storing a number in memory requires the programmer to specify how the memory should store it.

Jump drive

Alternatively referred to as a USB flash drive, data stick, pen drive, memory unit, key chain drive and thumb drive, a jump drive is a portable storage device. It is often the size of a human thumb (hence the name), and it connects to a computer via a USB port. Flash drives are an easy way to store and transfer information. They are available in sizes ranging from 2 GB to 1 TB.

Unlike a standard hard drive, the flash drive has no movable parts; it contains only an integrated circuit memory chip that is used to store data. Flash drives usually have plastic or aluminum

casings surrounding the memory chip and a USB connector for use with most modern computers.

Memory card

A memory card is a type of storage media that is often used to store photos, videos, or other data in electronic devices. Devices that commonly use a memory card include digital cameras, digital camcorders, handheld computers, MP3 players, PDAs, cell phones, game consoles, and printers. The picture to the right is a MicroSD flash memory card, which is just one of the types of memory cards available.

There are a handful of different types of memory cards on the market, each varying in size, compatibility, and storage capacity. Links to more information on the most commonly used types of memory cards are listed below.

- CompactFlash (CF)
- MicroSD
- MMC
- SD Card
- SDHC Card
- SmartMedia Card
- Sony Memory Stick
- xD-Picture Card



Memory Stick

The memory stick including the memory stick PRO, memory stick Duo, memory stick PRO Duo, memory stick Micro (M2), and memory stick PRO-HG. The capacity of

the Memory Sticks range in sizes of 4MB to 256GB, with a theoretically maximum capacity of 2TB, but they have not been produced in that capacity to date.

Solid-State Drive (SSD)

Solid-State Drive or Solid-State Disk, SSD is a drive that uses non-volatile memory as a means of storing and accessing data, much like computer RAM. Unlike a hard drive, an SSD has no moving parts, which gives it advantages such as accessing stored information faster, no noise, often more reliable, and consuming less power. The picture shows a Crucial SSD and is an example of an SSD.

Online and cloud

Storing data online and in cloud storage is becoming popular as people need to access their data from more than one device.

There are some online and cloud, such as:

- Cloud storage
- Network media

Cloud computing

Cloud computing is a term used to describe services provided over a network by a collection of remote servers. This abstract "cloud" of computers provides massive, distributed storage and processing power, which can be accessed by any Internet-connected device running a web browser.

Because the term cloud computing is a broad term, it is likely if you have spent any time on the Internet or use devices connected to the Internet that you have used some cloud computing. Below are some common examples of cloud computing you have likely heard of or used.

Network Media

Network media is any audio, video, images or text, used on a computer network, like the Internet. It almost always requires a computer to send and receive, as well as a community of people to create and consume the content. Network media are decentralized and contributed by many people, who not only provide it to others, but are also recipients and users of the media.

Paper storage

Early computers had no method of using any of the above technologies for storing information and had to rely on paper. Today, these forms of storage are rarely used or found. The picture to the right is an example of a woman entering data to a punch card using a punch card machine.

There are some paper storage, such as:

- OMR
- Punch card

Optical Mark Reading (OMR)

Optical mark reading or optical mark recognition, OMR is the process of gathering information from human beings by recognizing marks on a document. OMR is accomplished by using a hardware device (scanner) that detects a reflection or limited light transmittance on or through a piece of paper.



OMR allows for the processing of hundreds or thousands of documents per hour.

For example, students may recall taking tests or surveys where they filled in bubbles on paper (shown right) with a pencil. Once the form had been completed, a teacher or teacher's assistant would feed the cards into a system that grades or gathers information from them.

Punch card

Punch cards also known as Hollerith cards are paper cards containing several punched holes that were punched by hand or machine to represent data. These cards allowed companies to store and access information by entering the card into the computer. The picture is an example of a woman using a punch card machine to create a punch card. Using a punch card machine like that shown in the picture above, data can be entered into the card by punching holes on each column to represent one character. Below is an example of a punch card.

Once a card has been completed or the return key has been pressed the card technically "stores" that information. Because each card could only hold so much data if you wrote a program using punch cards (one card for each line of code) you would have a stack of punch cards that needed to remain in order.

To load the program or read a punch card data, each card is inserted into a punch card reader that input the data from the card into a computer. As the card is inserted, the punch card reader starts on the left-top-side of the card and reads vertically starting at the top and moving down. After the card reader has read a column it moves to the next column.

1.3 COMPUTER SOFTWARE

Software is a general term used to describe a collection of computer programs, procedures and documentation that perform some tasks on an operating system. Software is the way to perform different tasks electronically. Software is a set of rules to perform a specific task. The software is the information that the computer uses to get the job done. Software needs to be accessed before it can be used. There are many terms used for the process of accessing software including running, executing, starting up, opening, and others. Computer programs allow users to complete tasks.

Remember

An instruction may also (indirectly) cause something to appear on a display of the computer system—a state change which should be visible to the user.



A program can also be referred to as an application and the two words are used interchangeably. Software is the general term for information that is recorded onto some kind of medium.

For example, when a person goes to the video store and rent or buys a tape or digital versatile disc (DVD, what he/she really getting is the software that is stored on that tape or disk. Videocassette recorder (VCR or DVD player are hardware devices that are capable of reading the software from a tape or disk and projecting it onto TV screen, in the form of a movie.

1.3.1 Types of Software

Software is a set of programs, which is designed to perform a well-defined function. A program is a sequence of instructions written to solve a particular problem.

Although the range of software available today is vast and varied, most software can be divided into three major categories:

- System software
- Application software
- Programming software

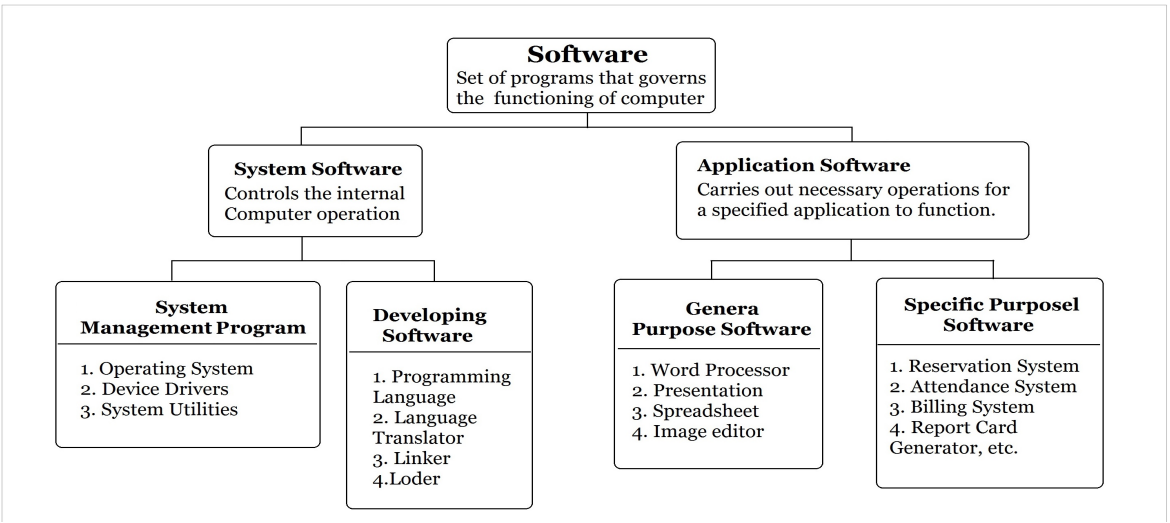


Figure 1.2: Type of software.

System Software

The system software is collection of programs designed to operate, control, and extend the processing capabilities of the computer itself. System software are generally prepared by computer manufactures. These software products comprise of programs written in low-level languages which interact with the hardware at a very basic level. System software serves as the interface between hardware and the end users.

It is a set of one or more programs designed to control the operation and extend the processing capability of a computer system. In general, a computer's system software performs one or more of the following functions:

- Supports development of other application software.
- Supports execution of other application software.
- Monitors effective use of various hardware resources such as CPU, memory, peripherals, etc.
- Communicates with and controls operation of peripheral devices such as printer, disk, tape, etc.

Hence, system software makes the operation of a computer system more effective and efficient. It helps the hardware components work together, and provides support for the development and execution of application software (programs). The programs included in a system software package are called system programs. The programmers who prepare system software are referred to as system programmers.

Types of system software

Some commonly known types of system software are:

- *Operating Systems*: Operating system software takes care of effective and efficient utilization of all hardware and software components of a computer system.
- *Programming Language Translators*: Programming language translators transform the instructions prepared by programmers in a programming language into a form that can be interpreted and executed by a computer system.
- *Communications Software*: In a network environment (where multiple computers are interconnected together by communications network) communications software enables transfer of data and programs from one computer system to another.
- *Utility Programs*: Utility programs (also known as utilities) are a set of programs that help users in system maintenance tasks, and in performing tasks of routine nature. Some tasks commonly performed by utility programs include formatting of hard disks or floppy disks, taking backup of files stored on hard disk on to a tape or floppy disk, sorting of the records stored in a file based on some key field(s), etc.

Application Software

Application software products are designed to satisfy a particular need of a particular environment. All software applications prepared in the computer lab can come under

the category of Application software. Application software may consist of a single program, such as a Microsoft's notepad for writing and editing simple text. It may also consist of a collection of programs, often called a software package, which work together to accomplish a task, such as a spreadsheet package. It is a set of one or more programs designed to solve a specific problem, or do a specific task.

For example, payroll processing software, examination results processing software, railway/airline reservation software, computer games software are all application software. Similarly, a program written by a scientist to solve a research problem is also application software. The programs included in an application software package are called application programs. The programmers who prepare application software are referred to as application programmers.

There is literally millions of application software available for a wide range of applications. They range from simple applications such as word processing, inventory management, preparation of tax returns, banking, hospital administration, insurance, publishing, to complex scientific and engineering applications such as weather forecasting, space shuttle launching, oil and natural gas exploration, design of complex structures like aircrafts, ships, bridges, sky-rise buildings, etc.

With so many applications available, it is not possible to categorize them all and to cover them here. Some commonly known application software is:

- *Word-processing Software*: Word-processing software enables us to make use of a computer for creating, editing, viewing, formatting, storing, retrieving, and bringing documents (written material, such as letters, reports, books, etc.).
- *Spreadsheet Software*: Spreadsheet software is a numeric-data-analysis tool that allows us to create a kind of computerized ledger. A manual ledger is a book having rows and columns that accountants use for keeping a record of financial transactions and for preparing financial statements.
- *Database Software*: A database is a collection of related data stored and treated as a unit for information retrieval purposes. *A* database software is a set of programs that enable us to create a database, maintain it (add, delete, and update its records), organize its data in desired fashion (*for example*, sort its records alphabetically name-wise), and selectively retrieve useful information from it.
- *For example*, queries such as get the telephone number of the person named Adam from the address database, or get the names of all currently enrolled students whose birthdays fall today from the student database can be handled easily.

- *Graphics Software:* Graphics software enables us to use a computer system for creating, editing, viewing, storing, retrieving, and printing of designs, drawings, pictures, graphs, etc.
- *Personal Assistance Software*: Personal assistance software allows us to use personal computers for storage and retrieval of our personal information, as well as planning and management of schedules, contacts, finances, and inventory of important items.
- *Education Software*: Education software allows a computer to be used as a teaching and learning tool. A few examples of such software include those used for teaching mathematics, grammar, language, or any other subject.
- *Entertainment Software*: Entertainment software allows a computer to be used as an entertainment tool. Computer video games belong to this category of software.

Programming Software

Programming software usually provides tools to assist a programmer in writing computer programs, and software using different programming languages in a more convenient way.

The tools include:

Compilers

A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language, often having a binary form known as object code). The most common reason for wanting to transform source code is to create an executable program.

Debuggers

A debugger or debugging tool is a computer program that is used to test and debug other programs (the "target" program). The code to be examined might alternatively be running on an instruction set simulator (ISS), a technique that allows great power in its ability to halt when specific conditions are encountered but which will typically be somewhat slower than executing the code directly on the appropriate (or the same) processor. Some debuggers offer two modes of operation full or partial simulation, to limit this impact.

Interpreters

An interpreter normally means a computer program that executes, i.e. performs, instructions written in a programming language.

An interpreter may be a program that either:

- Executes the source code directly
- Translates source code into some efficient intermediate representation (code) and immediately executes this
- Explicitly executes stored precompiled code made by a compiler which is part of the interpreter system

Linkers

A linker or link editor is a program that takes one or more objects generated by a compiler and combines them into a single executable program.

Text Editors

A text editor is a type of program used for editing plain text files. Text editors are often provided with operating systems or software development packages, and can be used to change configuration files and programming language source code.

1.3.2 Need of Software

The problem with computers is that humans do not speak their language. We need a translator, something that can understand our needs and put the computer to work for us. The translator is called software and it makes computers useful. *For example*, like a typewriter, a computer without software is just a lifeless machine. By adding software, the computer becomes more alive, easy to use, and built for us. Most computers have two basic kinds: The operating system and software programs. If we want to use a computer, we have to use an operating system. From saving files, to using a mouse or fixing problems, the operating system covers the basics. Operating systems come with all new computers and do a lot of same things. We have seen those called Windows, Mac and Linux.

But the operating system is only part of what we need. To make them personalized and more useful, we can add software programs.

For example, if we need to edit a photo we can add a software program that is built for that purpose. If we need to design a house, we can add a software program that lets we see the house from all sides. By adding and removing software programs,

we can make the computer fit with exactly what we want to do. And adding most software programs is easy. We can get them from a computer store or download them from the Internet. No computer nerds required.

What is a software program? What is really happening when one opens one? Think about it this way. Computers are really good at following instructions. And a software program is essentially a set of instructions that tells the computer exactly what to do?

When one open a program, the computer goes to work, completing the instructions until the program is ready for use. The ability to add and remove software programs means that everyone's computer can be different and unique to them.

Now we have talked about operating systems that take care of the basics, and software programs that make computers personalized. It is combination that makes computers so useful. But it is not limited to computers on desk.

Consider cell phone, just like a computer without software, it is a lifeless machine that does not speak our language. Cell phones have software that brings them to life.

The same thing is true for many cameras, music players, and even our cars. Every day we rely on software to bring machines to life and make them personalized and useful.

ROLE MODEL



JOHN VON NEUMANN: THE SCIENTIFIC GENIUS WHO PIONEERED THE MODERN COMPUTER, GAME THEORY, AND NUCLEAR DETERRENCE

John von Neumann was a famous Hungarian-American mathematician, who is still revered for his unparalleled contributions to disciplines like mathematics, science, economics and many more. Born in a wealthy family to Jewish parents, he was a child prodigy, exhibiting great analytic and computing skills. He was able to divide eight - digit numbers in mind in very early ages, which proves his great analytical and computing skills. He was a passionate learner and was interested in many subjects like physics, economics, statistics, history etc., apart from mathematics. He was a prominent member of the Manhattan Project and the Institute for Advanced Study in Princeton. His unusual abilities astonished many great personalities like mathematician Jean Dieudonne Peter Lax, Hans Bethe etc. Hans Bethe once said, "I have sometimes wondered whether a brain like von Neumann's does not indicate a species superior to that of man". Read through this biography to know more about the career and life of this genius.

Early Life

John von Neumann was born in Budapest, the capital city of Hungary, on 28 December 1903. He was the elder son of Neumann Miksa and Kann Margit. The nature of the numbers and the mathematical logic induced interest in him at a very early age. However, mathematics was not the only subject which fascinated him; Neumann was greatly interested in history as well that he went through forty four volumes of universal history at an early age of eight. This indicated that Neumann found comfort dealing with both logical and social areas. He was fortunate enough to have very supportive parents, who stood by his interests. In 1914, at the age of ten, he joined the Lutheran Gymnasium, one among the three best organizations in Budapest. He published his first paper in the Journal of the German Mathematical Society in 1922, which dealt with the zeros of certain minimal polynomials.

Berlin, Zurich and Budapest

Neumann had little interest towards chemistry and engineering but, his father

encouraged him to pursue engineering as it was considered as an elite career during that time. He attended Pázmány Péter University in Budapest to pursue advanced doctorate in mathematics along with completing his undergraduate course, diploma in chemical engineering from ETH Zurich. In his thesis for PhD, he wanted to give an attempt to the axiomatization of the set theory which was developed by George Cantor. It was an achievement indeed that a seventeen year old boy pursued both undergraduate and PhD simultaneously. He received good scores for his exams for both the courses and thus, completed under graduation degree in chemical engineering and PhD in mathematics at the age of twenty-two.

Quantum Mechanics

After receiving his degrees in 1926, Neumann attended the university in Gottingen, Germany, where he dealt with quantum mechanics. He was creative and an original thinker, and came up with complete and logical concepts. He worked on the theories of quantum mechanics in 1926 to reconcile and improve them. Neumann worked to find out the similarities between wave mechanics and matrix mechanics. Also, von Neumann worked on the rules of 'abstract Hilbert space' and devised a mathematical structure in terms of quantum theory.

Personal Life

During the time from 1927-1929, after Neumann formulated the quantum mechanics, he attended extensive conferences and colloquia. By 1929, he wrote about 32 papers in German. These papers were was well-organized so that the other mathematicians could incorporate Neumann's works into their theories. By this time, he became famous in the academic world, owing to his creative yet new ideas. By the end of 1929, he was offered a post of lecturer at Princeton in America. It was during the same time, when he married Mariette Kovesi, his childhood friend. In 1935, Mariette gave birth to a girl child, Marina. The marital relationship did not last long; they separated in 1936. Mariette went back to Budapest and Neumann drifted around Europe for a brief period and returned to United States. He met Klari Dan on his way to Budapest and married her in 1938.

Death

John von Neumann was diagnosed with cancer. Despite his ailment and weakness, he took part in the ceremonies organized to honor him, in a wheel chair, and kept contact with family and friends during this time. He passed away on 8 February 1957.

What Is The Moral Here?

Neumann contributed to the Los Alamos project devised by the American government in the development of the nuclear weapons and also in developing the concept and design of explosive lenses. The mathematical modelling which he used at Los Alamos helped him in the development of the modern computer. Along with gathering resources, he also funded the project for the development of the modern computer. He worked on the architecture of the computer as well. His efforts made other scientists realize that computer is not just a 'bigger calculator'. Quantum logic, game theory, linear programming and mathematical statistics are some of the many contributions he had made to the field of science.

SUMMARY

- A computer is a device that accepts information (in the form of digitalized data) and manipulates it for some result based on a program or sequence of instructions on how the data is to be processed.
- Generation in computer talk is a step in technology. It provides a framework for the growth of computer industry. Originally, the term "generation" was used to distinguish between varying hardware technologies but it has now been extended to include both hardware and software that together make up a computer system.
- Digital computers are either general purpose computers or special purpose ones. General purpose computers, as their name suggests, are designed for specific types of data processing while general purpose computers are meant for general use.
- In computing, an input device is a peripheral (piece of computer hardware equipment) used to provide data and control signals to an information processing system such as a computer or other information appliance.
- A joystick is an input device that allows the user to control a character or machine in a computer program, such as a plane in a flight simulator. They look similar to the control device you would find on an arcade game, but nearly always include extra buttons for additional functionality.
- A light pen is a light-sensitive computer input device, basically a stylus that is used to select text, draw pictures and interact with user interface elements on a computer screen or monitor.
- A scanner is an input device that scans documents such as photographs and pages of text. When a document is scanned, it is converted into a digital format. This creates an electronic version of the document that can be viewed and edited on a computer.
- A storage device is any computing hardware that is used for storing, porting and extracting data files and objects. It can hold and store information both temporarily and permanently, and can be internal or external to a computer, server or any similar computing device.
- Software is a general term used to describe a collection of computer programs, procedures and documentation that perform some tasks on an operating system. Software is the way to perform different tasks electronically.

KNOWLEDGE CHECK

1. **Which output device is suited to presenting information that is likely to change very frequently?**
   a. COM.
   b. Plotter.
   c. Sound card.
   d. Monitor.
2. **Which of the following might be used to input and interpret information printed on bank checks?**
   a. COM.
   b. OMR.
   c. MICR.
   d. CAD.
3. **Which type of printer works by using a charged drum to deposit particles of ink on the paper?**
   a. Dot-matrix printer.
   b. Thermal printer.
   c. Laser printer.
   d. Inkjet printer.
4. **Which of the following storage devices is suited to holding large volumes of data that is unlikely to change often?**
   a. Floppy disk.
   b. Magnetic tape.
   c. Hard disk.
   d. CD-ROM.
5. **The software used to drive microprocessor-based systems is called .....**
   a. assembly language programs.
   b. firmware.
   c. BASIC interpreter instructions.
   d. flowchart instructions.
6. **The generation based on VLSI microprocessor.**
   a. 1st
   b. 2nd

c. 3rd
d. 4th

7. **A term in computer terminology is a change in technology a computer is/was being used.**
   a. development
   b. generation
   c. advancement
   d. growth

8. **…… generation of computer started with using vacuum tubes as the basic components.**
   a. 1st
   b. 2nd
   c. 3rd
   d. 4th

9. **Batch processing was mainly used in this generation.**
   a. 1st
   b. 2nd
   c. 3rd
   d. 4th

10. **The processed form of data is called;**
    a. Field
    b. Information
    c. Words
    d. File

REVIEW QUESTIONS

1. What do you understand by computers? Explain its advantages.
2. What are the generations of computer explain?
3. Explain the types of computers.
4. Describe the peripheral device.
5. Explain the input and output device.

6. What are the storage devices?
7. What is computer software? What are the needs of software?
8. Explain the different types of software.

Check Your Result

1. (d)	2. (c)	3. (c)	4. (d)	5. (a)
6. (d)	7. (b)	8. (a)	9. (a)	10. (b)

REFERENCES

1. Campbell-Kelly, Martin, and William Aspray. 1996/2004/2013. Computer: A History of the Information Machine. New York: Basic Books; 2nd and 3rd editions Westview.
2. Care, Charles. Technology for Modelling: Electrical Analogies, Engineering Practice, and the Development of Analogue Computing. Springer, 2010.
3. Ceruzzi, Paul E. A History of Modern Computing. Cambridge: MIT Press, 1998; second edition 2003.
4. Chandler, Alfred D., Jr. Inventing the Electronic Century: The Epic Story of the Consumer Electronics and Computer Science Industries. New York: Free Press, 2001.
5. Chandler, Alfred D., Jr., and James W. Cortada, eds. A Nation Transformed by Information: How Information Has Shaped the United States from Colonial Times to the Present. Oxford University Press, 2000.
6. Mahoney, Michael S. "The Histories of Computing(s)." Interdisciplinary Science Reviews 30 #2 (June 2005).
7. Mahoney, Michael S. Histories of Computing. Harvard University Press 2011.
8. Swedin, Eric G., and David L. Ferro. Computers: The Life Story of a Technology. Johns Hopkins University Press, 2007

CHAPTER

2

FINITE AUTOMATA

Learning Objectives

After studying this chapter, you will be able to:

1. Discuss about the finite automata: basic
2. Explain the nondeterministic finite automata
3. Describe the application of finite automata

INTRODUCTION

Finite automata or finite state machine is the simplest machine that is used for recognizing patterns. It is an abstract machine that has five elements or tuples. It possesses a set of states and rules that are used for moving from one state to another, but it is dependent on the applied input symbol.

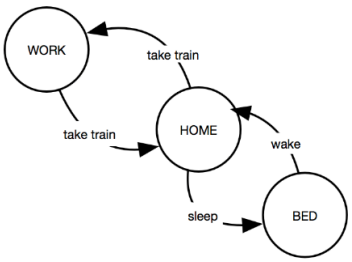
Essentially, it is pretty much an abstract model of a digital computer. It is a mathematical model of computation.

This abstract machine can be in exactly one of a finite number of states at any given time. The Finite-State Machine (FSM) can be in exactly one of a finite number of states at any specific time.

A finite state machine can transition or change from one state to another in response to some inputs. A finite automaton is defined by a list of its states, its initial state, and the inputs that trigger each transition.

You can observe the behavior of state machines in various devices in modern society that perform a predetermined sequence of actions depending on the sequence of events with which they are presented.

The finite state machine possesses less computational power than certain other models of computation like the Turing machine. The computational power distinction signifies that there are tasks there are computational tasks that a Turing machine could perform but finite automata can't.



2.1 FINITE AUTOMATA: BASIC

Finite automata model very simple computational devices or processes. These devices have a constant amount of memory and process their input in an online manner. By that we mean that the input is read a character at a time, and for each character, as it is read, a constant amount of processing is performed.

One might wonder whether such simple devices could be useful. In fact, they are widely used as controllers in mechanical devices such as elevators and automatic doors.

Example. A controller for the doors on the exit hatchway of a spaceship. The rather simple spaceship in this example has an exit area with two doors: one leads out into space, the other leads into the spaceship interior, as shown in figure 2.1.

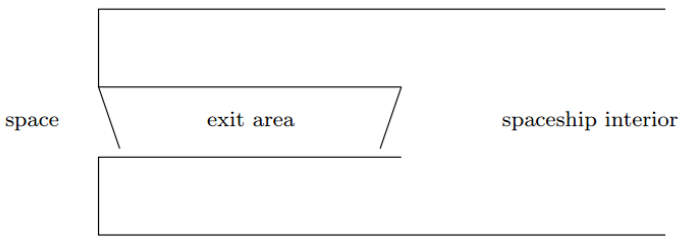


Figure 2.1: Spaceship Exit Doors

The critical rule is that at most one door may be open at any given time (for otherwise all the air in the spaceship could vent out into the vacuum of space). The state of the doors in managed by a controller. The controller can receive signals or requests to open or close a door. For simplicity, we suppose that only one request can be received at one time. The diagram in figure 2.2 specifies the operation of the controller. It is simply a directed graph with edge labels. The vertex names in this graph specify the door state to which the vertex corresponds. The edges exiting a vertex indicate the actions taken by the controller on receiving the request labeling the edge.

For example, if both doors are closed and a request to open the exit door is received, then the controller opens this door and the state changes accordingly. On the other hand, if currently the interior door is open and a request to open the exit door is made, then nothing changes; i.e. the interior door remains open and the exit door remains closed. Note that there are many requests that result in no change.

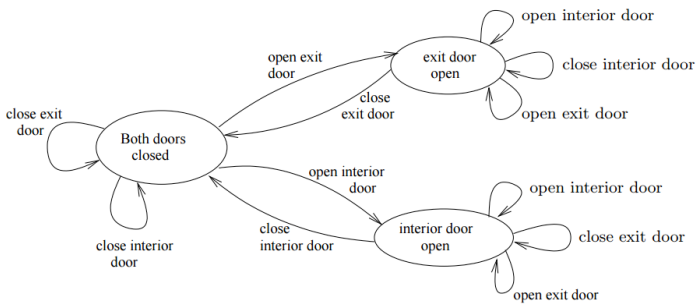


Figure 2.2: Spaceship Door Controller

We can simplify the diagram so as to show only those requests that cause a change of state, as shown in figure 2.3.

Remember

One should note that finite automata are not often used to model processes such as this and the next one, for the reason that other models usually do it better. Nevertheless, we can view each of these two as true finite automata, although with fewer transitions than is usual.

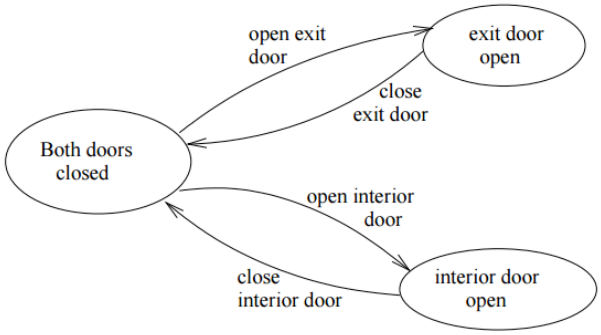


Figure 2.3: Spaceship Door Controller, Sparse Diagram

As requests are received the controller cycles through the appropriate states. To implement the device it suffices to have a 2-bit memory to record the current state (2 bits can distinguish 4 states) plus a little logic to decide what state change to make as requests are received.

Similar devices are also very useful as a front end in many programs, where they are used to partition the input into meaningful tokens.



A compiler, which is a program that processes input programs so they can then be executed, needs to partition the input into keywords, variables, numbers, operators, comments, etc.

For example, we could imagine recognizing the keywords if, then, else, end with the help of the following device or procedure, represented as a graph.

The processing begins at the vertex ***start*** and follows the path labeled by the input characters as they are read. If there is no edge to follow, then the computation fails; it recognizes a keyword if it finishes at a vertex represented by a double circle.

For example, on reading the input string ***if***, the finite automata goes from the start vertex to the vertex labeled ***if***. Note that it is conventional for the start vertex, which need not be named ***start***, to be indicated in the graph by means of a double arrow.

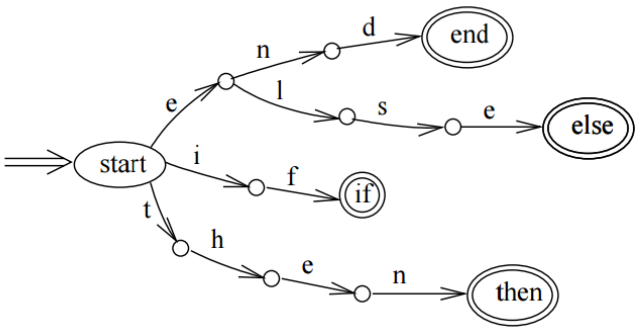


Figure 2.4: Keyword Recognizer

In particular, the compiler has to be able to decide when it has reached the end of a keyword rather than being in the middle of a variable (e.g. *ended*) whose name begins with the keyword. But these essentially straightforward and tedious, albeit important, details are not the focus here.

Variables are another collection of strings that compilers need to recognize. Let us suppose variables consist of all strings of (lower-case) letters other than keywords, and to simplify the illustration below let us suppose there is just one keyword: ***if***. Then the following procedure, shown in figure 2.5, will recognize variables.

Remember

The set of all strings accepted by an NFA is the language the NFA accepts. This language is a regular language.

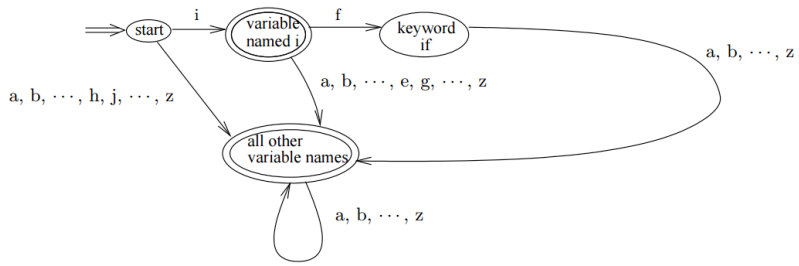


Figure 2.5: Variable Recognizer

The graph has a self-loop; also, some of its edges have more than one label (alternatively, we could think of this as a collection of parallel edges, each with a distinct label). Again, to use the variable recognizer, the compiler follows the path spelled out by the input. Any input other than the word ***if*** leads to a vertex represented as a double circle and hence represents a variable.

2.1.1 A Precise Specification

We need to introduce a little notation and terminology. Recall that an alphabet is a set of characters. *For example*:

- Binary = {0,1}
- Boolean = {T,F}
- English = {a, b, · · ·, z}

The usual way of writing an unspecified *k*-character alphabet is as $\Sigma = \{a_1, a_2, \cdots, a_k\}$.

By convention, a finite automaton is named *M*, or M_1, M_2, · · · if there are several. As we shall see subsequently, other devices will also be named by *M*. A finite automaton is a directed graph plus a finite alphabet Σ in which:

- One vertex is designated as the start vertex.
- A subset of the vertices, conventionally called *F*, form the collection of Recognizing or *Final* vertices.
- Each edge is labeled by a character of Σ.
- Each vertex has |Σ| outgoing edges, one for each character in Σ.

In the corresponding drawing of the automaton, the start vertex is indicated by a double arrow, and the vertices in *F* are indicated by drawing them with double circles.

M processes an input string ***s*** as follows: it determines the end of the path, which begins at the start vertex and follows the labels spelled out by reading ***s***.

For example, in the keyword recognizer (see Figure 2.4), the input ***if*** specifies the path from the vertex ***start*** to the vertex named ***if***. If the end of the path is a vertex in *F* then M recognizes or *accepts* ***s***; otherwise *M* rejects ***s***. Continuing the example, input ***if*** is recognized by the automaton as the vertex ***if*** is in *F*.

Thus we can view *M* as partitioning the set of all possible strings into those it accepts and those it rejects. The set of strings accepted by *M* is often called the language accepted by *M,* and is sometimes written as *L(M)* or *L* for short.

Conventionally, the vertices of a finite automata are called its states and are written as $Q = \{q_1, q_2, \cdots, q_r\}$ rather than $V = \{v_1, v_2, \cdots, v_r\}$. The index n tends to be reserved for the length of the input string. However, we will tend to use more descriptive names for the vertices in our examples.

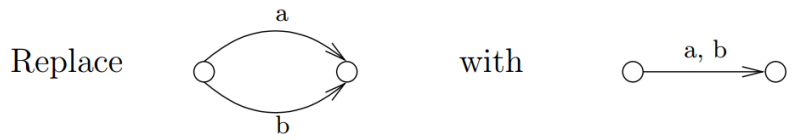


Figure 2.6: Simplifying Figures of Finite Automata

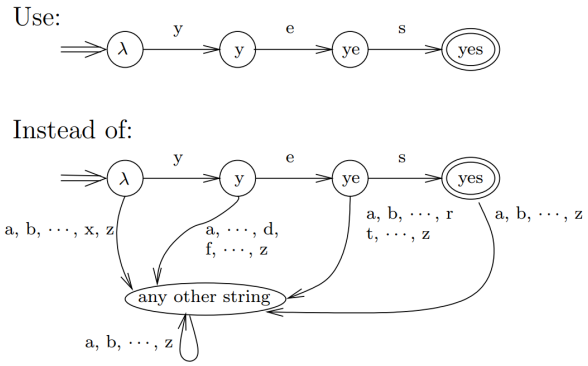


Figure 2.7: Omitting the Sink State

For simplicity in drawing we replace multiple parallel edges by a single edge with multiple labels as shown in figure 2.6.

Also, it is often convenient to omit a "sink" state (a non-final state which cannot be left); see figure 2.7.

The descriptive names being used for the vertices. These names help explain for which strings the automata reaches which vertices.

Finally, it is helpful to have a notation for representing labeled edges. We write $\delta(p, a) = q$ to mean that there is an edge labeled a from vertex p to vertex q; a more active interpretation is that starting at vertex (state) p, on reading a the automata goes to vertex (state) q. In other words the destination on reading a when at vertex p is vertex q. conventionally δ is called the transition function.

It is also useful to extend the destination (or transition) **function** to the reading of strings; this is denoted by δ^*. $\delta^*(p, s) = r$ means that vertex r is the destination on reading string s starting at vertex p.

Keyword

Function is a relation between a set of inputs and a set of permissible outputs with the property that each input is related to exactly one output.

For example, in the variable recognizer (see figure 2.5), δ^* (start, it) = "all other variables". Observe that $\delta^*(p, \lambda) = p$: reading the empty string, i.e. reading no characters, leaves the state unchanged

Some Particular Languages

- L(M) = ϕ — This is the empty language, the language with no strings in it. So the finite automata M accepts nothing.
- L(M) = $\{\lambda\}$ — This is the language comprising the single string λ. Note that $\phi \neq \{\lambda\}$; these two languages are distinct.

2.1.2 Designing Automata

The first issue is to decide what has to be remembered as the input is read. The easiest thing, it might seem, is to remember the whole string. But one cannot in general, as the automata has a finite memory.

For example, with $\Sigma = \{a, b\}$, consider recognizing those strings with an odd number of *b*'s. It suffices to keep track of whether an even or an odd number of *b*'s have been read so far. This leads to the transition diagram shown in figure 2.10(a). To obtain the full automata, it is enough to note that at the start zero *b*'s have been read (an even number), and the Final state corresponds to an odd number of *b*'s having been read; see figure 2.10(b). Next, consider the language $L(M) = \{w \mid w$ contains at least one $a\}$, with $\Sigma = \{a, b\}$. The natural states correspond to "no *a* read" and "at least one *a* read". The transitions are now evident and are shown in figure 2.11. Finally, consider the language $L(M) = \{w \mid w$ contains at least two *b*'s$\}$, with $\Sigma = \{a, b\}$. The natural three states correspond to: "no *b* read", "one *b* read", and "at least two *b*'s read". Again, the transitions are evident and are shown in figure 2.12.

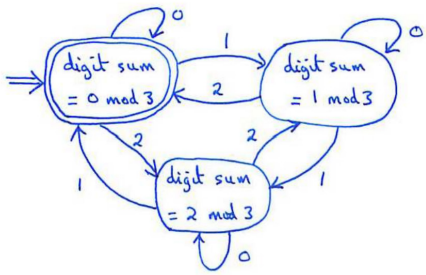


Figure 2.8: Example Automata M_4: $L(M_4) = \{x \mid \sum_{i=1}^{|x|} x_i = 0 \mod 3$, where $x = x_1 x_2 \cdots x_{|x|}\}$. Note that $\lambda \in L(M_4)$.

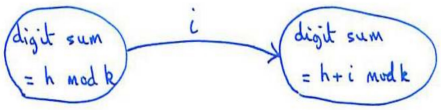


Figure 2.9: Example Automata M_5: The Transition Function

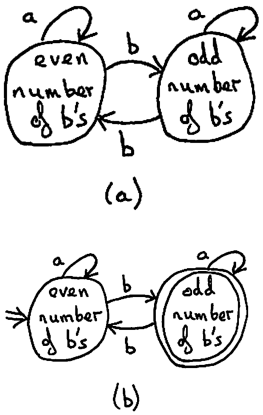


Figure 2.10: Automata recognizing strings with an odd number of *b*'s

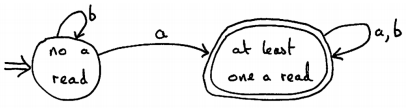


Figure 2.11: Automata recognizing strings with at least one a

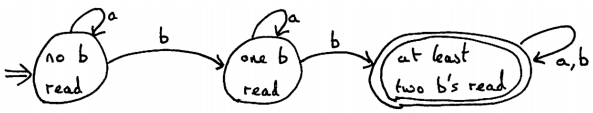


Figure 2.12: Automata recognizing strings with at least two *b*'s

2.2 NONDETERMINISTIC FINITE AUTOMATA

In a nondeterministic finite automaton (NFA), for each state there can be zero, one, two, or more transitions corresponding to a particular symbol. If NFA gets to state with more than one possible transition corresponding to the input symbol, we say it branches. If NFA gets to a state where there is no valid transition, then that branch dies.

NFA Acceptance

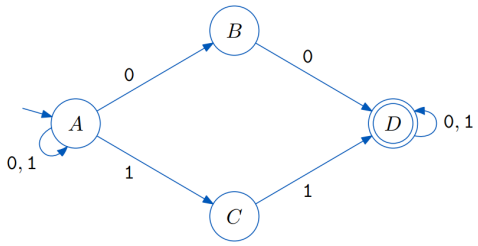
An NFA accepts the input string if there exists some choice of transitions that leads to ending in an accept state.

Thus, one accepting branch is enough for the overall NFA to accept, but every branch must reject for the overall NFA to reject.

This is a model of computation. We write DFA to specify a deterministic finite automaton (the one defined earlier). If type does not matter, we now just write FA.

Example: Doubles

It accepts any binary string that contains 00 or 11 as a substring.

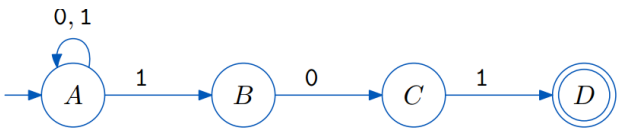


Remember

Explicitly propagate tokens through the transition structure of the NFA and match whenever a token reaches the final state. This is sometimes useful when the NFA should encode additional context about the events that triggered the transition.

Example: ***Ending of Strings***

An NFA that accepts all **binary strings** that end with 101.

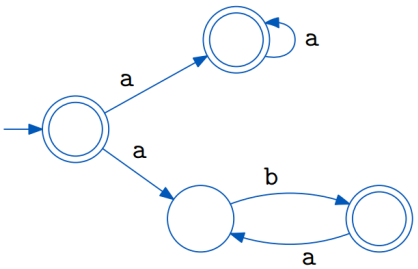


Keyword

Binary string is a sequence of 0's and 1's. Let Σ be the set {0,1}. Then the set of finite binary strings is written as Σ*, and the set of finite and infinite binary stringsis written as Σ**.

Example: Simultaneous Patterns

An NFA for a * + (ab) *



Non-deterministic finite automata, NFAs for short, are a generalization of the machines we have already defined, which

are often called deterministic finite automata by contrast, or DFAs for short. The reason for the name will become clear later.

As with a DFA, an NFA is simply a graph with edges labeled by single letters from the input alphabet Σ. There is one structural change.

For each vertex v and each character $a \in \Sigma$, the number of edges exiting v labeled with a is unconstrained; it could be *0, 1, 2* or more edges.

This obliges us to redefine what an automaton M is doing, given an input x. Quite simply, M on input x determines all the vertices that can be reached by paths starting at the start vertex, ***start***, and labeled x. If any of these reachable vertices is in the set of Accepting or Final vertices then M is defined to accept x. Another way of looking at this is that M recognizes x exactly if there is some path (and possibly more than one) labeled x, going from ***start*** to a vertex (state) $q \in F$; such a path is called a recognizing or accepting path.

Example. Let $A = \{w \mid$ the third to last character in w is a "b"$\}$. The machine M with $L(M) = A$ is shown in figure 2.13.

Note that on input abbb all four vertices can be reached, whereas on input abab the second from rightmost vertex cannot be reached.

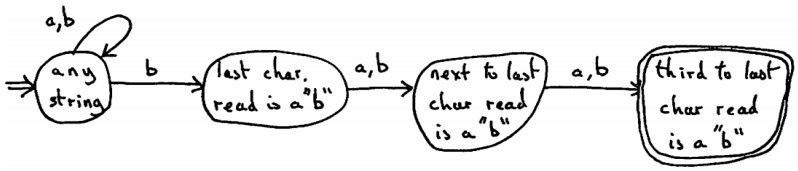


Figure 2.13: NFA recognizing *{w | the third to last character in w is a b}.*

The collection of languages recognized by NFAs is exactly the collection of languages recognized by DFAs. That is, for each NFA N there is a DFA M with $L(M) = L(N)$. Thus NFAs do not provide additional computational power. However, they can be more compact and easier to understand, and as a result they are quite useful.

It is helpful to consider how one might implement an NFA. One way is to maintain *a* bit vector over the vertices, so as to record which vertices are reachable currently. That is, suppose that reading input w can lead to any of the vertices in set R, say. The set S of vertices reachable on reading a further character a is obtained by following all the edges labeled a that exit any of the vertices in R. So in the automata of figure 2.13, the set of vertices reachable on reading input b is {"any string", "last char read is *a b*"}, and the set of vertices reachable on reading ba is {"any string", "next to last char read is *a b*"}. We define recognition for NFAs as follows: An NFA M recognizes input string w if there is a path from M's start vertex to a final vertex

(state) such that the labels along the edges of the path, when concatenated, form the string w. That is, were we to follow this path, reading the edge labels as we go, the string read would be exactly w; we say this path has label w. The implementation described above checks every path with label w; for acceptance, one needs to have at least one label w path leading to a final vertex. We call such a path a recognizing or accepting path for w.

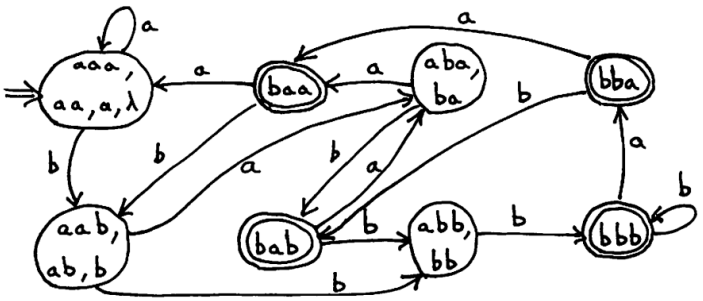


Figure 2.14: DFA recognizing $\{w \mid$ the third to last character in w is a $b\}$. By naming the last three characters or all the characters in the string, each vertex specifies the strings for which the DFA reaches it.

Keyword

Empty String is the special case where the sequence has length zero, so there are no symbols in the string.

To define NFAs formally, we need to redefine the destination (transition) functions δ. Now δ takes as its arguments a set of vertices (states) $R \subseteq V$ and a character $a \in \Sigma$ and produces as output another set of vertices (states) $S \subseteq V$: $\delta(R, a) = S$. The meaning is that the set of vertices (states) reachable from R on reading a is exactly the set of vertices (states) S.

δ^* is also redefined. $\delta^*(R, w) = S$ means that the set of vertices (states) S are the possible destinations on reading w on starting from a vertex (state) in R. So, in particular, w is recognized by M, $w \in L(M)$, exactly if $\delta^*(\{start\}, w) \cap F \neq \phi$; i.e., when starting at vertex start, on reading w, at least one destination vertex (state) is an accepting or final vertex (state).

δ is often defined formally as follows: $\delta : 2^Q \times \Sigma \rightarrow 2^Q$. In this context, 2^Q means the collection of all possible subsets of Q, the set of states of the NFA, and so δ does exactly what it should. Its inputs are (*i*) a set of vertices (states), that is one of the elements of the collection 2^Q, and (ii) a character in Σ; its output is also a set of vertices (states).

Next, we add one more option to NFAs: edges labeled with the **empty string**, as shown in figure 2.15. We call these edges λ-edges for short.

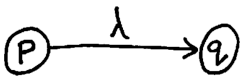


Figure 2.15: λ label on an edge.

The meaning is that if vertex (state) p is reached, then vertex (state) q can also be reached without reading any more input. Again, the same languages as before are recognized by NFAs with λ-edges and by DFAs; however, the λ-edges are very convenient in creating understandable machines. The meanings of the destination functions δ and δ^* are unchanged.

Example. Let A and B be languages over the alphabet Σ that are recognized by NFAs M_A and M_B, respectively. Then the NFA shown in figure 2.16 recognizes the language $A \cup B$.

Its first step, prior to reading any input, is to go to the start vertices for machines M_A and M_B. Then the computation in these two machines is performed simultaneously. The set of final vertices for M is the union of the final vertices for M_A and M_B; thus M reaches a final vertex on input w exactly if at least one of M_A or M_B reaches a final vertex on input *w*. In other words $L(M) = L(M_A) \cup L(M_B)$.

Deterministic vs. Nondeterministic

Another way of viewing the computation of an NFA M on input x is that the task is to find a path labeled x from the start vertex to a final vertex if there is one, and this is done correctly by (inspired) guessing. This process of correct guessing is called Non-deterministic computation. Correspondingly, if there is no choice or uncertainty in the computation, it is said to be Deterministic. This is not an implementable view; it is just a convenient way to think about what non determinism achieves.

2.2.1 Closure Properties

A language is said to be regular if it can be recognized by an NFA. The regular languages obey three closure properties: if A and B are regular, then so are $A \cup B$, $A \circ B$ and A^*. We have already seen a demonstration of the first of these. We now give a simpler demonstration of the first property and then show the other two.

First, the following technical lemma is helpful.

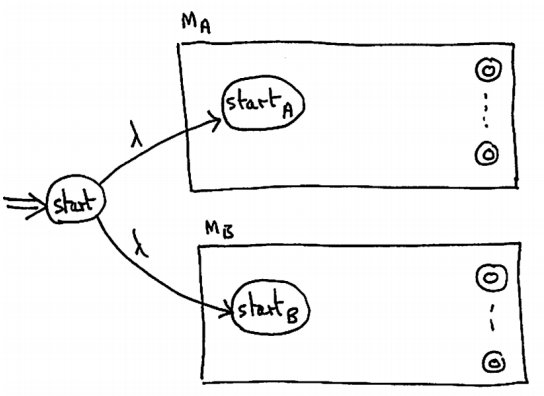


Figure 2.16: An NFA recognizing language A ∪ B

Remember

When the last input symbol is consumed, the NFA accepts if and only if there is some set of transitions that will take it to an accepting state. Equivalently, it rejects, if, no matter what transitions are applied, it would not end in an accepting state.

Lemma. Let M be an NFA. Then there is another NFA, N, which has just one final vertex, and which recognizes the same language as M: L(N) = L(M).

Proof. The idea is very simple: N is a copy of M with one new vertex, *new_final*, added; *new_final* is N's only final vertex. λ-edges are added from the vertices that were final vertices in M to *new_final*, as illustrated in figure 2.17.

Recall that a recognizing path in an NFA goes from its start vertex to a final vertex. It is now easy to see that M and N recognize the same language. The argument has two parts.

First, any recognizing path in M, for string w say, can be extended by a single λ-edge to reach N's final vertex, and thereby becomes a recognizing path in N; in addition, the edge addition leaves the path label unchanged (as $w\lambda = w$). It follows that if M recognizes string w then so does N. In other words, L(M) ⊆ L(N).

Second, removing the last edge from a recognizing path in N yields a recognizing path in M having the same path label (for the removed edge has label λ). It follows that if N recognizes string w then so does M. In other words, L(N) ⊆ L(M).

Together, these two parts yield that L(M) = L(N), as claimed.

Closure properties

There are some closure properties, such as:

- If A and B are regular then so is A ∪ B We show this using the NFA $M_{A \cup B}$ in figure 2.18. $M_{A \cup B}$ is built using M_A and M_B, NFAs with single final vertices recognizing A and B, respectively. It has an additional start vertex start, connected to M_A and M_B's start vertices by λ-edges, and an additional final vertex final, to which M_A and M_B's final vertices are connected by λ-edges.

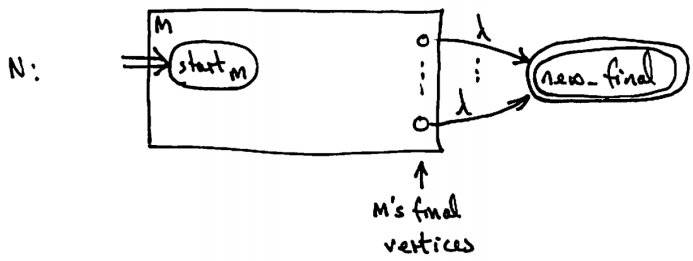


Figure 2.17: NFA N with a single final vertex

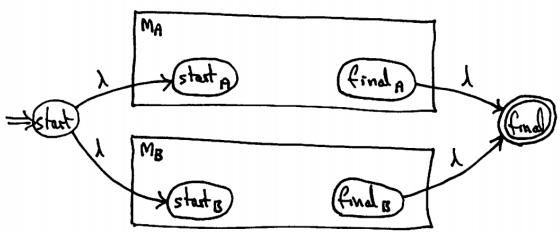


Figure 2.18: NFA $M_{A \cup B}$ recognizing A ∪ B

As in the earlier construction, each recognizing path in $M_{A \cup B}$ corresponds to a recognizing path in one of M_A and M_B, and vice-versa, and thus $w \in L(M_{A \cup B})$ if and only if $w \in A \cup B$. We now show this more formally.

Lemma. $M_{A \cup B}$ recognizes A ∪ B.

Proof. We show that $A \cup B = L(M_{A \cup B})$ by showing that $L(M_{A \cup B}) \subseteq A \cup B$ and that $A \cup B \subseteq L(M_{A \cup B})$.

To show that $L(M_{A \cup B}) \subseteq A \cup B$, it is enough to show that if $w \in L(M_{A \cup B})$ then $w \in A \cup B$ also. So let $w \in L(M_{A \cup B})$ and let P be a recognizing path for w in $M_{A \cup B}$. Removing the first and last edges of P, which both have label λ, yields a recognizing path

DID YOU KNOW?

Nondeterministic finite automaton were introduced in 1959 by Michael O. Rabin and Dana Scott, who also showed their equivalence to DFAs. NFAs are used in the implementation of regular expressions: Thompson's construction is an algorithm for compiling a regular expression to an NFA that can efficiently perform pattern matching on strings.

in one of M_A or M_B, also having label w. This shows that if $w \in L(M_{A \cup B})$, then either $w \in L(M_A) = A$ or $w \in L(M_B) = B$ (of course, it might be in both, but this argument does not reveal this); in other words, if $w \in L(M_{A \cup B})$ then $w \in A \cup B$.

To show that $A \cup B \subseteq L(M_{A \cup B})$, it is enough to show that if $w \in A \cup B$ then $w \in L(M_{A \cup B})$ also. Now, if $w \in A \cup B$, then either $w \in A$ or $w \in B$ (or possibly both). Consider the case that $w \in A$. Then there is a recognizing path P_A for w in M_A. Preceding P_A with the appropriate λ-edge from start and following it with the λ-edge to final, yields a recognizing path in $M_{A \cup B}$, also having label w. This shows that if $w \in L(M_A) = A$ then $w \in L(M_{A \cup B})$ also. Similarly, if $w \in B$ then $w \in L(M_{A \cup B})$ too. Together, this gives that if $w \in A \cup B$ then $w \in L(M_{A \cup B})$.

- If A and B are regular then so is $A \circ B$ We show this using the NFA M_{AB} displayed in figure 2.19. M_{AB} is built using M_A and M_B, NFAs with single final vertices, recognizing A and B, respectively. M_{AB} comprises a copy of M_A plus a copy M_B, plus one additional edge. M_A's start vertex is also M_{AB}'s start vertex, and M_B's final vertex is M_{AB}'s only final vertex. Finally, the new edge, e, joins M_A's final vertex to M_B's start vertex.

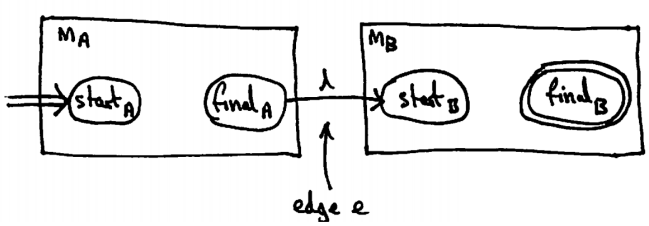


Figure 2.19: NFA M_{AB} recognizing $A \circ B$

The idea of the construction is that a recognizing path in M_{AB} corresponds to recognizing paths in M_A and M_B joined by edge *e*. It then follows that $w \in L(M_{AB})$ if and only if w is the concatenation of strings u and v, $w = uv$, with $u \in A$ and $v \in B$. We now show this more formally.

Lemma. M_{AB} recognizes $A \circ B$.

Proof. We show that $L(M_{AB}) = A \circ B$ by showing that $L(M_{AB}) \subseteq A \circ B$ and that $A \circ B \subseteq L(M_{AB})$.

To show that $L(M_{AB}) \subseteq A \circ B$, it is enough to show that if $w \in L(M_{AB})$ then $w \in A \circ B$ also. So let $w \in L(M_{AB})$ and let P be a recognizing path for w in M_{AB}. Removing edge e from P creates two paths P_A and P_B, with P_A being a recognizing path in M_A and P_B a recognizing path in M_B. Let u and v be the path labels for P_A and P_B, respectively. So $u \in A$ and $v \in B$. As e is a λ-edge, $w = u\lambda v = uv$.

This shows that $w \in A \circ B$.

To show that $A \circ B \subseteq L(M_{AB})$, it is enough to show that if u ∈ A and v ∈ B then uv ∈ $L(M_{AB})$. So let $u \in A$, $v \in B$, let P_A be a recognizing path for u in M_A, and let P_B be a recognizing path for v in M_B. Then form the path $P = P_A, e, P_B$ in M_{AB}; clearly, this is a recognizing path. Further, it has label $u\lambda v = uv$. So $uv \in L(M_{AB})$.

- If A is regular then so is A^* We show this using the NFA M_{A^*} in figure 2.20. M_{A^*} is built using M_A, an NFA with a single final vertex recognizing A. M_{A^*} comprises a copy of M_A plus a new start vertex, plus two additional λ-edges, e and *f*. *e* joins M_{A^*} 's start vertex to M_A's start vertex, and *f* joints M_A's final vertex to M_{A^*} 's start vertex. M_{A^*} 's start vertex is also its final vertex.

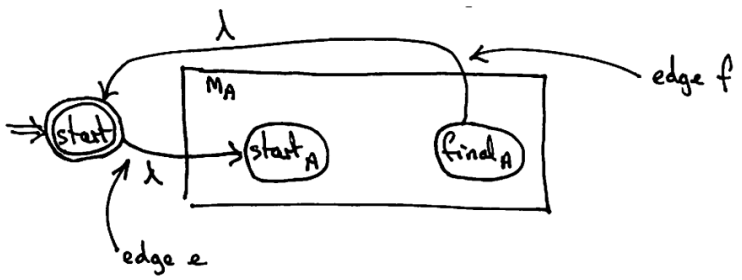


Figure 2.20: NFA M_{A^*} recognizing A^*

The construction is based on the observation that removing all copies of e and f from a recognizing path in M_{A^*} yields subpaths, *k* of them say, each of which is a recognizing path in M_A. It then follows that a string w is recognized by M_{A^*} if and only if it is the concatenation of *k* strings recognized by M_A, for some $k \geq 0$. Now we show this more formally.

Lemma. M_{A^*} *recognizes* A^* .

Proof. We show that $L(M_{A^*}) = A^*$ by showing that $L(M_{A^*}) \subseteq A^*$ and that $A^* \subseteq L(M_{A^*})$

To show that $L(M_{A^*}) \subseteq A^*$, it is enough to show that if $w \in L(M_{A^*})$ then $w \in A^*$ also. So let $w \in L(M_{A^*})$ and let P be a recognizing path for *w* in M_{A^*}. Removing all instances of edges e and f from P creates k subpaths, for some $k \geq 0$, where each subpath is a recognizing path in M_A. Let the path labels on these k subpaths be $u_1, \cdots, u_k$, respectively. So $u_i \in A$, for $1 \leq i \leq k$. As e and f are λ-edges, $w = \lambda u_1 \lambda\lambda u_2 \lambda\lambda \cdots \lambda\lambda u_k \lambda = u_1 u_2 \cdots u_k$. Thus $w \in A^*$.

To show that $A^* \subseteq L(M_{A^*})$, it is enough to show that if $w \in A^*$ then $w \in L(M_{A^*})$. If $w \in A^*$ then $w = u_1 u_2 \cdots u_k$, with $u_1, u_2, \cdots, u_k \in A$, for some k ≥ 0. As $u_i \in A$, there is an accepting path P_i in M_A for u_i. Let P be the following path in M_{A^*}: $e, P_1, f, e, P_2, f, \cdots, e, P_k, f$ (for k = 0 we intend the path of zero edges). P is a recognizing path in M_{A^*}, and it has label $u_1 u_2 \cdots u_k = w$, as e and f are λ-edges. Thus $w \in L(M_{A^*})$.

2.2.2 Every regular language is recognized by a DFA

Let N be an NFA. We show how to construct a DFA M with L(M) = L(N).

Recall that to implement an NFA we keep track of the set of currently reachable vertices. This is what M will do with its vertices or states, which we call superstates henceforth. Each of M's superstates is a subset of the set Q of vertices or states in N. So M's collection (set) of superstates is the power set of Q, 2^Q.

The relation between superstates of M and sets of vertices in N is specified by the following assertion.

Assertion. On input w, M reaches superstate R if and only if N can reach the set of states R.

To achieve assertion, we define the transition function δ for M to be identical to the transition function δ for N. Note that this specifies where M's edges go.

To make sure the assertion is correct initially, that is for w = λ, we set M's start superstate to be the set of N's vertices that N can reach on input λ. Finally, we set the final superstates of M to be those superstates that include one or more of the final vertices of N: so if F is the set of N's final vertices, then R is a final superstate for M exactly if R ∩ F ≠ ϕ.

Lemma. L(M) = L(N).

Proof. We begin by arguing by induction on the length of the input string read so far that above assertion is true. The base case is for the empty string: the claim is true by construction. For the inductive step, if the claim is true for strings of length k, then we argue that it is also true for strings of length k + 1. By the inductive hypothesis, on an input w of length k, R is the superstate reached by M if and only if R is the set of vertices reachable by N. But then, on input wa, M reaches superstate δ(R, a), and N can reach the set of vertices δ(R, a), so the claim is true for strings of length k + 1 also.

It remains to consider which strings the two machines recognize. M recognizes input w if and only if on input w it reaches superstate R where R∩F ≠ϕ; and N recognizes input w if and only if it can reach a set of vertices S where S ∩F ≠ ϕ. But we have shown that S = R. So the two machines recognize the same collection of strings, that is L(M) = L(N).

Implementation

The advantage of an NFA is that it may have far fewer vertices (states) than a DFA recognizing the same language, and thus use much less **memory** to store them. On the other hand, when running the machines, the NFA may be less efficient, as the set of reachable vertices had to be computed as the input is read, whereas in the DFA one just needs to follow a single edge. Which choice is better depends on the particulars of the language and the implementation environment.

Keyword

Memory refers to the computer hardware devices used to store information for immediate use in a computer; it is synonymous with the term "primary storage".

2.2.3 Non Regular Languages

The proof of non-regularity of a language using the pumping lemma is a proof by contradiction. The goal is to assume that the language is regular and then derive strings which are not in the language, thereby contradicting the regularity assumption.

We turn now to a method for demonstrating that some languages are not regular. For example, as we shall see, L = {a^n b^n |n ≥ 1} is not a regular language. Intuition suggests that to recognize L we would need to count the number of a's in the input string; in turn, this suggests that any automata recognizing L would need an unbounded number of vertices.

We use a proof by contradiction. So suppose for a contradiction that L were regular. Then there must be a DFA M that accepts L. M has some number k of vertices. Let's consider feeding M the input a^k b^k . Look at the sequence of k + 1 vertices M goes through on reading a^k: $r_0 = start, r_1, r_2, \cdots, r_k$, where some of the r_i 's may be a repeated instance of the same vertex. This is illustrated in figure 2.21.

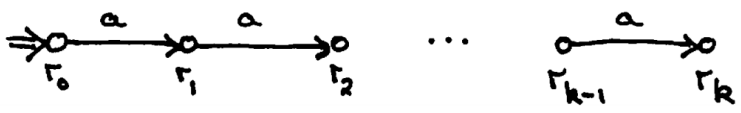


Figure 2.21: The vertices traversed on input a^k.

As there are only k distinct vertices, there is at least one vertex that is visited twice in this sequence, $r_i = r_j$, say, for some pair i,j, $0 \le i < j \le k$. This is shown in figure 2.21. In fact we see that $r_{j+1} = r_{h+1}$ if $j + 1 \le k$, $r_{j+2} = r_{h+2}$ if $j + 2 \le k$, and so forth. But all we will need for our result is the presence of one loop in the path, so we will stick with the representation in the figure above.

It is helpful to partition the input into four pieces: a^i, a^{j-i}, a^{k-j}, b^k. The first a^i takes M from vertex r_0 to r_i (to the start of the loop), the next a^{j-i} takes M from r_i to r_j (once around the loop), the final a^{k-j} takes M from r_j to r_k, and b^k takes M from r_ν to a final vertex, as shown in figure 2.22. What happens on input $a^i a^{j-i} a^{j-i} a^{k-j} b^k = a^{k+j-i} b^k$?

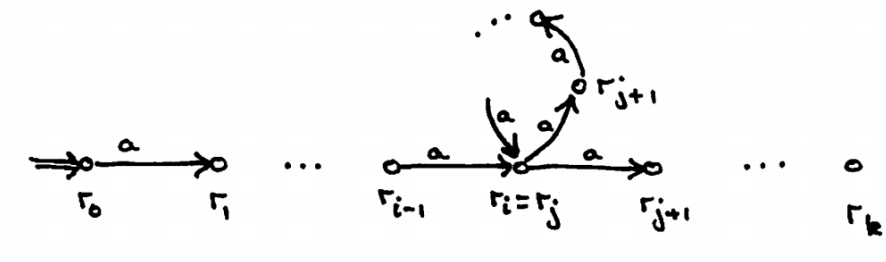


Figure 2.21: The path traversed on input a^k .

The initial a^i takes M from r_0 to r_i, the first a^{j-i} takes M from r_i to $r_j = r_i$ (once around the loop), the second a^{j-i} takes M from r_i to r_j (around the loop again), the a^{k-j} takes M from r_j to r_k, and the b^k takes M from r_k to a final vertex. So M accepts $a^{k+j-i} b^k$, which is not in L as $j - i > 0$. This is a contradiction, and thus the initial assumption, that L was regular, must be incorrect.

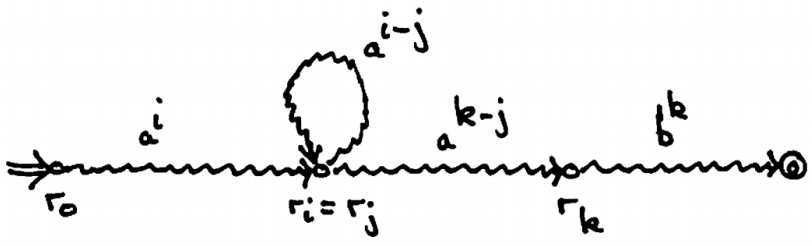


Figure 2.22: The path traversed on input a k

Lemma. (Pumping Lemma) Let L be a **regular language**. Then there is a number p = pL, the pumping length for L, with the property that for each string s in L of length at least p, s can be written as the concatenation of three substrings x, y, and z, that is s = xyz, and these substrings satisfy the following conditions.

- $|y| > 0$,
- $|xy| \leq p$,
- For each integer $i \geq 0$, $xy^i z \in L$.

Definition. We say s can be pumped if s ∈ L can be written as the concatenation of substrings x, y and z, and conditions one to three of the pumping lemma hold.

Proof. Again, we use a proof by contradiction. So suppose that L were regular and let M be a DFA accepting L. Now suppose that M has p states p, the number of states in M, will be the pumping length for L.

Let s be a string in L of length n ≥ p. Write s as $s = s_1 s_2 \cdots s_n$, where each s_i is a character in Σ, the alphabet for L. Consider the substring $s' = s_1 s_2 \cdots s_p$. We look at the path M follows on input s'. It must go through p + 1 vertices, and as M has only p vertices, at least one vertex must be repeated. Let start = $r_0, r_1, \cdots, r_p$ be this sequence of vertices and suppose that $r_i = r_j$, where $0 \leq i < j \leq p$, is a repeated vertex, as shown in figure 2.23. Let x denote $s_1 \cdots s_i$, y denote $s_{i+1} \cdots s_j$, and z denote $s_{j+1} \cdots s_p s_{p+1} \cdots s_n$. The path traversed on input s = xyz is illustrated in figure 2.24. As j > i, $|y| = j - i > 0$. Also $|xy| = |s_1 \cdots s_j| = j \leq p$. So (1) and (2) conditions are true.

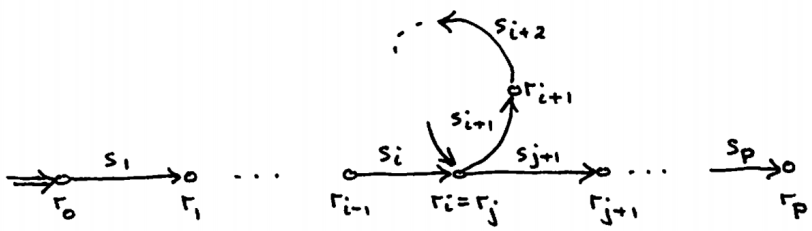


Figure 2.23: The path traversed on input $s' = s_1 s_2 \cdots s_p$.

Keyword

Regular language satisfies the following equivalent properties: it is the language of a regular expression (by the above definition) it is the language accepted by a nondeterministic finite automaton (NFA) it is the language accepted by a deterministic finite automaton (DFA) it can be generated by a regular grammar.

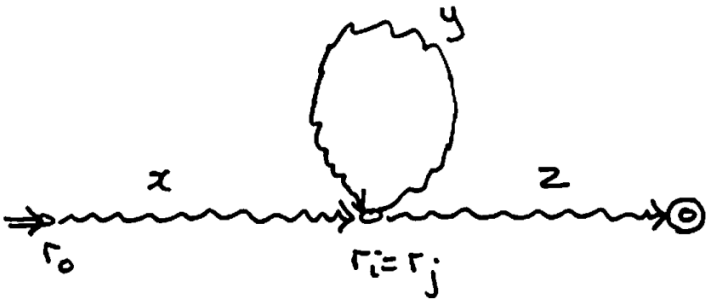


Figure 2.24: The path traversed on input s = xyz.

Clearly, xz is recognized by M, for x takes M from r_0 to $r_i = r_j$ and z takes M from r_j to a final vertex.

Similarly $xy^i z$ is recognized by M for any $i \geq 1$, for x takes M from r_0 to r_i, each repetition of y takes M from r_i (back) to $r_j = r_i$, and then the z takes M from r_j to a final vertex. Thus (3) condition is also true, proving the result.

Now, to show that a language L is non-regular we use the pumping Lemma in the following way. We begin by assuming L is regular so as to obtain a contradiction. Next, we assert that there is a pumping length p such that for each string s in L of length at least p the three conditions of the Pumping Lemma hold. The next (and more substantial) task is to choose a particular string s to which we will apply the conditions of the Pumping Lemma, and condition (3) in particular, so as to obtain a contradiction.

Example. Let us look at the language $L = \{a^n b^n \mid n \geq 1\}$ again. The argument showing that L is not regular proceeds as follows.

Step 1. Suppose, so as to obtain a contradiction, that L were regular. Then L must have a pumping length $p \geq 1$ such that for any string $s \in L$ with $|s| \geq p$, s can be pumped.

Step 2. Choose s. Recall that we chose the string $s = a^p b^p$.

Step 3. By pumping s, obtain a contradiction.

As s can be pumped (for $s \in L$ and $|s| = 2p \geq p$), we can write s = xyz, with $|y| > 0$, $|xy| \leq p$, and $xy^i z \in L$ for all integer $i \geq 0$.

As the first p characters of s are all a's, the substring xy must also be a string of all a's. By condition (3), with i = 0, we have that $xz \in L$; but the string xz has removed |y| a's from s, that is $xz = a^{p-|y|} b^p$, and this string is not in L since $p - |y| \neq p$. This is a contradiction for we have shown that $xz \in L$ and $xz \notin L$.

Step 4. Consequently the initial assumption is incorrect; that is, L cannot be recognized by a DFA, so L is not regular.

Let me stress the sequence in which the argument goes. First the existence of pumping length p for L is asserted (different regular languages L may have different pumping lengths, but each regular language with arbitrarily long strings will have a pumping length). Then a suitable string s is chosen. The length of s will be a function of p. s is chosen so that when it is pumped a string outside of L is obtained. An important point about the pumped substring y is that while we know y occurs among the first p characters of s, we do not know exactly which ones form the substring y. Consequently, a contradiction must arise for every possible substring y of the first p characters in order to show that it is not regular.

Keyword

Argument is a series of statements typically used to persuade someone of something or to present reasons for accepting a conclusion.

There is a distinction here: for a given value of p, s is fully determined; for instance, in above example for p = 3, s = aaabbb. By contrast, all that we know about x and y is that together they contain between 1 and 3 characters, and that y has at least one character. There are six possibilities in all for the pair (x, y), namely: (λ, a), (λ, aa), (λ, aaa), (a, a), (a, aa), (aa, a). In general, for $|xy| \leq p$, there are $\frac{1}{2}p(p+1)$ choices of x and y. The **argument** leading to a contradiction must work for every possible choice of p and every possible partition of s into x, y, and z. Of course, you are not going to give a separate argument for each case given that there are infinitely many cases. Rather, the argument must work regardless of the value of p and regardless of which partition of s is being considered.

2.2.4 Regular expressions

Regular expressions provide another, elegant and simple way of describing regular languages. We denote regular expressions by small letters such as r, s, r_1, etc. As with regular languages, they are defined with respect to a specified alphabet Σ. They are defined by the following six rules, comprising three base cases and three combining rules.

We begin with the base cases.

- ϕ is a regular expression; it represents ϕ, the empty set.

- λ is a regular expression; it represents the language $\{\lambda\}$, the language containing the empty string alone.
- a is a regular expression; it represents the language {a}.
- For the combining rules, let regular expressions r and s represent languages R and S. The rules follow.
- $r \cup s$ is a regular expression; it represents language $R \cup S$.
- $r \circ s$ is a regular expression; it represents language $R \circ S$. We write rs for short.
- r^* is a regular expression; it represents language R^* .

Parentheses are used to indicate the scope of an operator. It is also convenient to introduce the notation r^+; while not a standard regular expression, it is shorthand for rr^* , which represents the language RR^* . Also Σ is used as a shorthand for $a_1 \cup a_2 \cup \cdots \cup a_k$, where $\Sigma = \{a_1, a_2, \cdots, a_k\}$. Finally, L(r) denotes the language represented by regular expression r.

Examples.

1. $a^* ba^* = \{w \mid w \text{ contains exactly one b}\}$.

2. $\Sigma^* bab\Sigma^* = \{w \mid w \text{ contains bab as a substring}\}$.

3. $(\Sigma\Sigma)^* = \{w \mid w \text{ has even length}\}$.

4. $a\Sigma^* a \cup b\Sigma^* b \cup a \cup b = \{w \mid w \text{ has the same first and last character}\}$.

5. $(a \cup \lambda)b^* = ab^* \cup b^*$.

6. $a^*\phi = \phi$.

Lemma. Let r be a regular expression. There is an NFA N_r that recognizes the language represented by r: $L(N_r) = L(r)$.

Proof. The proof is by induction on the number of operators (union, concatenation, and star) in regular expression r.

The base case is for zero operators, which are also the base cases for specifying regular expressions. It is easy to give DFAs that recognize the languages specified in each of these base cases and this is left as an exercise for the reader.

For the inductive step, suppose that r is given by one of the recursive definitions, $r_1 \cup r_2$, $r_1 \circ r_2$, or r_1^*. Since r_1, and r_2 if it occurs, contain fewer operators than r, we can assume by the inductive hypothesis that there are NFAs recognizing the languages represented by regular expressions r_1 and r_2.

We can conclude that there is an NFA recognizing L(r).

To prove the converse, that every regular language can be represented by a regular expression takes more effort. To this end, we introduce yet another variant of NFAs, called GNFAs (for Generalized NFAs).

In a GNFA each edge is labeled by a regular expression r rather than by one of λ or a character $a \in \Sigma$. We can think of an edge labeled by regular expression r being traversable on reading string x exactly if x is in the language represented by r. String w is recognized by a GNFA M if there is a path P in M from its start vertex to a final vertex such that P's label, the concatenation of the labels on P's edges, forms a regular expression that includes w among the set of strings it represents.

Suppose P consists of edges $e_1, e_2, \cdots, e_k$, with labels $r_1, r_2, \cdots, r_k$, respectively; then P is a w-recognizing path if w can be written as $w = w_1 w_2 \cdots w_k$ and each w_i is in the language represented by r_i, for $1 \le i \le k$.

Clearly, every NFA is a GNFA, so it will be enough to show that any language recognized by a GNFA can also be represented by a regular expression.

We begin with two simple technical lemmas.

Lemma. Suppose that GNFA M has two vertices p and q with h edges from p to q, edges $e_1, e_2, \cdots, e_h$. Suppose further that these edges are labeled by regular expressions $r_1, r_2, \cdots, r_h$, respectively. Then replacing these h edges by a new edge e labeled $r_1 \cup r_2 \cup \cdots \cup r_h$, or r for short, yields a GNFA N recognizing the same language as M.

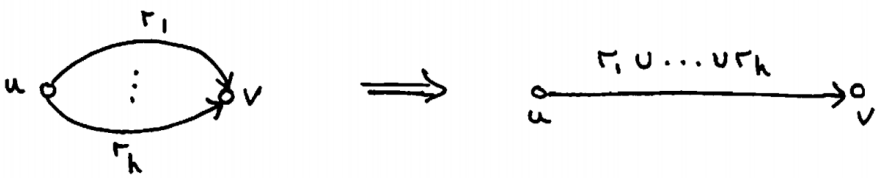


Figure 2.25: Edge replacement in lemma.

Proof. The edge replacement is illustrated in figure 2.25.

First we show that $L(M) \subseteq L(N)$. Suppose that M recognizes string w. Let P be a w-recognizing path in M. Replacing each instance of edge e_i, for $1 \le i \le h$, by edge e yields a new path P' in N, with P' being w-recognizing (for a substring w_i read on traversing edge e_i, and hence represented by r_i, is also represented by r and so can be read on traversing edge e also. This shows $w \in L(N)$.

Next we show that L(N) ⊆ L(M). Suppose that N recognizes string w. let P be a wrecognizing path in N. Suppose that edge e is traversed j times in P, and let w_1, w_2, · · · , w_j be the substrings of w read on these j traversals. Each substring w_g, 1 ≤ g ≤ j, is represented by regular expression r, and as $r = r_1 \cup \cdots \cup r_h$, each w_g is represented by one of r_1, r_2, · · · , r_h; so let w_g be represented by r_{i_g}, where $1 \le i_g \le h$. Replacing the corresponding instance of e in P by e_{i_g} yields a path P' in M which is also w-recognizing. This shows that w ∈ L(M).

Lemma. suppose GNFA M has three vertices u, v, q, connected by edges labeled as shown in figure 2.26 and further suppose that they are the only edges incident on q. Then removing vertex q and creating edge (u, v) with label $r_1 r_2^* r_3$ yields a GNFA N recognizing the same language as M.

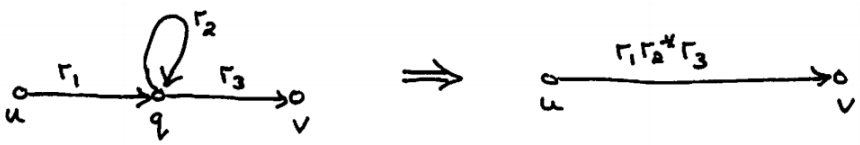


Figure 2.26: Edge replacement in lemma.

Proof. First we show that L(M) ⊆ L(N). Suppose that w ∈ L(M), and let P be a wrecognizing path in M. We build a w-recognizing path P' in N. Consider a segment of the path from u to v going through q in P. It consists of edge (u, q), followed by some k ≥ 0 repetitions of edge (q, q), and followed by edge (q, v). This subpath has label $r_1 r_2^k r_3$. But all strings represented by $r_1 r_2^k r_3$ are represented by $r_1 r_2^* r_3$, and so we can replace this subpath by the new edge (u, v) in N. Thus w is recognized by N.

Next, we show that L(N) ⊆ L(M). Suppose that w ∈ L(N), and let P be a w-recognizing path in N. We build a w-recognizing path P' in M. Consider an instance of edge e with e = (u, v) on P, if any. Suppose string w_i is read on traversing e. Then w_i is in the language represented by $r_1 r_2^k r_3$ for some k ≥ 0 (for this is what $r_1 r_2^* r_3$ means: $r_1 r_2^* r_3 = r_1 r_3 \cup r_1 r_2 r_3 \cup r_1 r_2^2 r_3 \cup r_1 r_2^3 r_3 \cup \cdots$.) So we can replace this instance of edge e with the edge sequence (u, q) followed by k instances of edge (q, q), followed by edge (q, v). When all instances of e in P are replaced in this manner, the resulting path P' in M is still w-recognizing, so w ∈ M.

Lemma. Let M be a GNFA. Then there is a regular expression r representing the language recognized by M: L(r) = L(M).

Proof. We begin by modifying M so that its start vertex has no in-edge (if need be by introducing a new start vertex) and so that it has a single final vertex with

no out-edges (again, by adding a new vertex, if needed). Also, for each pair of vertices, if there are several edges between them, they are replaced by a single edge by applying lemma. Let N_0 be the resulting machine and suppose that it has n + 2 vertices {start, q_1, q_2, · · · , q_n, final}.

We construct a sequence N_1, N_2, · · · , N_n of GNFAs, where N_i is N_{i-1} with qi removed and otherwise modified so that N_{i-1} and N_i recognize the same language, for $1 \leq i \leq n$.

Thus $L(N_n) = L(M)$. But Ln has two vertices, start and final, joined by a single edge labeled by a regular expression r, say. Clearly, the language recognized by N_n is L(r). So it remains to show how to construct N_i given N_{i-1}

Step 1. Let $q = q_i$ be the vertex being removed. For each pair of vertices u, $v \neq q$ such that there is a path u, q, v we make a new copy $q_{u,v}$ of q as shown in figure 2.27, and then remove the vertex q and the edges incident on it.

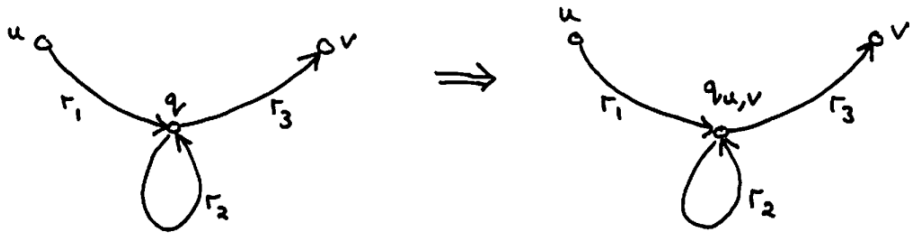


Figure 2.27: Vertex duplication in step 1.

Clearly, a path segment (u, q repeated $k \geq 1$ times, v) in N_{i-i} has the same label as the path segment (u, $q_{u,v}$ repeated k times, v), and consequently the labels on the recognizing paths in N_{i-1} and the machine following the Step 1 modification are the same.

Step 2. In turn, for each vertex $q_{u,v}$, apply lemmas to the sub graph formed by u, $q_{u,v}$, and v, for each pair of vertices u, v. The resulting GNFA is N_i. Clearly, $L(N_i) = L(N_{i-1})$. The effect of the application of these lemmas is illustrated in figure 2.28.

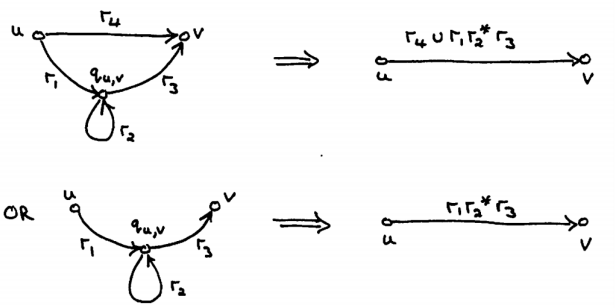


Figure 2.28: Change due to step 2.

2.3 APPLICATION OF FINITE AUTOMATA

We have seen an example of use of finite automata in describing the operation of a simplified version of vending machine. Many other systems operating in practice can also be modeled by finite automata such as control circuits of computers, computer network communication protocols, and lexical analyzers for compilers etc. Many of those systems fall into the class of systems called reactive system. A reactive system is a system that changes its actions, outputs and conditions/status in response to stimuli from within or outside it. It is an event driven or control driven system continuously having to react to external and/or internal stimuli. The inputs for a reactive system are never ready unlike for example when two numbers are added together by an adder (Here we are considering an adder at a higher level of abstraction than physical devices level ignoring for example the transient states of the electronic circuit that realizes an adder). An adder does not respond unless the input i.e. two numbers to be added are ready. A system such as an adder is called a transformational system. Vending machine or **communication protocol,** on the other hand, a system must respond to each stimulus, even to a fragment of input such as each coin tossed in for a can of soda or every message received.

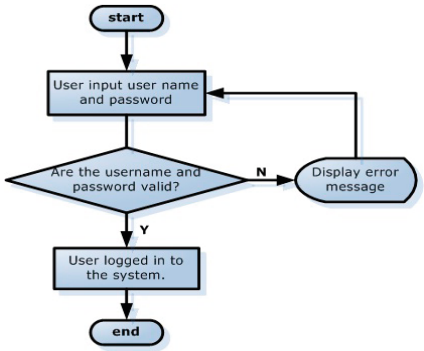
Keyword

Communications protocol is a set of formal rules describing how to transmit or exchange data, especially across a network. A standardized communications protocol is one that has been codified as a standard.

It is generally agreed that finite automata are a natural medium to describe dynamic behaviors of reactive systems. Finite automata are formal and rigorous and computer programs can be easily written to simulate their behaviors. To model a reactive system with finite automaton, first the states the system goes in or the modes of its operation are identified. These become the states of the finite automaton that models it. Then the transitions

between the states triggered by events and conditions, external or internal to the system, are identified and they become arcs in the transition diagram of the finite automaton. In addition actions that may take place in those states can also be added to the model.

For example consider the following very simplified version of login process to a computer from the computer point of view. Let us assume for simplicity that this computer accepts a single user at a time.

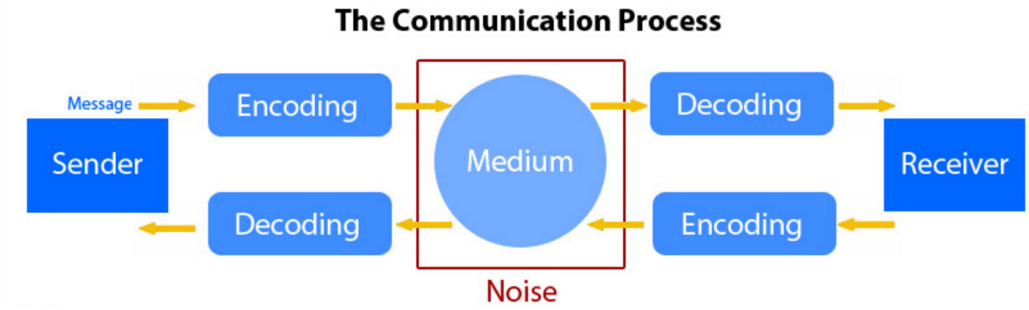


Initially the computer waits for a user name to be typed in. This is one state of the system. When a name is typed in, it checks whether or not the name is valid. If it is valid, then it asks for and then waits for the password, which is another state. If the user name typed in is not valid, it goes back to the initial state. We could make it go to a different state and count the number of login attempts for security purpose. But let us make it simple. When a password is typed in and it is correct, then it accepts the user and starts a session. That is another state though it could further be broken down into a number of more states. When the session terminates, it gets a signal, goes back to the initial state and waits for another login. If the password typed in is incorrect, then it informs the user of that and waits for the next try. That is a fourth state. If the second password fails, it goes to the initial state and starts all over again. Again what we have seen is a model for one level of abstraction.

The next example is a protocol for a computer to follow in communicating with another computer. Again it is a very simplified version.

Remember

If a process is running and must wait on an event, it is placed in the blocked state. Upon completion of the external event, the process becomes ready to run and is moved to the ready state, from which it will eventually be dispatched.



Initially the computer is in wait state waiting for "Request for Next Message" (RFNM) to come from another computer. When a RFNM starts coming, it goes into the state of receiving it (Our interpretation is that the computer is in a state of receiving an RFNM and it is taking the action of receiving the RFNM). Upon completion of the RFNM, it sends "Acknowledgement" (ACK) to the other computer, which is another state. After sending the ACK, it starts sending the requested message to the other party. When it is complete, it goes into another wait state waiting for an ACK to come from the other computer. If a negative ACK is received, it resends the message. If a positive ACK is received, it goes back to the initial state and waits for another RFNM to come. Thus a finite automaton that models this protocol has the following five states: initial state (wait for RFNM), receiving RFNM, sending ACK, sending message and waiting for ACK.

Again depending on the level of abstraction, different states and transitions would have to be chosen.

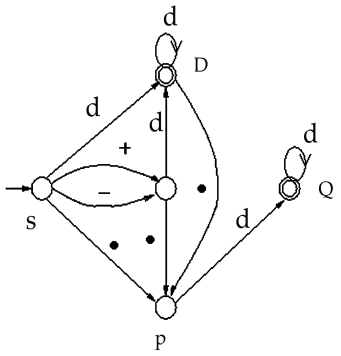
- One such system initially waits for the first symbol to come in. If the first symbol is a sign, then it goes into a state, denote it by G that indicates that a sign has been received.
- If the first digit is received before a decimal point, regardless of whether a sign has been read or not, it goes into a state, denote it by D that indicates a digit has been read before a decimal point.
- If a decimal point is received before a digit, then it goes into a state, denote it by P that indicates that a decimal point has been read.
- If a decimal point has been read (i.e. in state P), then it must receive at least one digit after that. After one digit it can continue receiving digits. Therefore from state P it goes to another state, denote it by Q, after reading a digit and stays there as long as digits are read. This Q is an accepting state.
- On the other hand if a digit has been read before a decimal point, i.e. it is in state D, then it can continue receiving digits and stay in D. D is another accepting state. If a decimal point is read while in D, then it goes to state P indicating that a decimal point has been read.

This system can also be described by a regular expression.

Since these numbers are represented by strings consisting of a possible sign, followed by zero or more digits, followed by a possible decimal point, followed by one or more digits, they can be represented by the following regular expression:

$(s_+ + s_- + \Lambda) (d^+.d^+ + d^+ + .d^+)$,

where s_+ and s_- represent the positive and negative signs, respectively and $d \in \{ 0 , 1 , 2 , . . . , 9 \}$. This system can be modeled by the following finite automaton:



Number Recognozer

ROLE MODEL

MICHAEL OSER RABIN: ISRAELI COMPUTER SCIENTIST



Michael Oser Rabin, (born September 1, 1931, Breslau, Germany [now Wrocław, Poland]), German-born Israeli American mathematician and computer scientist and cowinner of the 1976 A.M. Turing Award, the highest honour in computer science. Rabin and the American mathematician and computer scientist Dana S. Scott were cited for their early joint paper "Finite Automata and Their Decision Problem," which has had a lasting impact on the field of automata theory, and for their subsequent independent work.

Rabin's family immigrated to Palestine in 1935. In 1953 Rabin earned a master's degree in mathematics from Hebrew University of Jerusalem, and then he went to the United States to earn a doctorate (1957) in mathematics from Princeton University. Rabin taught at Princeton (1956–58) before returning to Israel to accept a professorship in mathematics at Hebrew University in 1958. In 1980 he became the school's Albert Einstein Professor of Mathematics, a position he held until he retired as emeritus professor in 1999. During that time he also held a joint appointment at Harvard University, first as the Gordon McKay Professor of Computer Science (1981–83) and then as the Thomas J. Watson, Sr., Professor of Computer Science (1983–12); in 2013 he became Thomas J. Watson, Sr., Research Professor of Computer Science.

In addition to his work on the decision problem in mathematics, Rabin codeveloped the Miller-Rabin test, an algorithm for determining if a given number is a prime number. This was just one aspect of Rabin's numerous contributions to the fields of cryptography and data encryption. Perhaps his most far-reaching work was his invention, with the Israeli American computer scientist Yonatan Aumann and the Chinese computer scientist Yan Zong Ding, of Hyper-Encryption, the first provably unbreakable encryption scheme. In addition to the Turing Award, Rabin was honoured with the 1974 Rothschild Prize in Mathematics, the 1980 Harvey Prize in Science and Technology, the 1995 Israel Prize in Exact Sciences/Computer Science, the 2000 IEEE Charles Babbage Award in Computer Science, and the 2004 EMET Prize in the Exact Sciences: Computer Sciences. Rabin served on the editorial boards of the Journal of Computer and Systems Science, the Journal of Combinatorial Theory, and the Journal of Algorithms. He was elected to the American Academy of Arts and Sciences (1975), the Israel Academy of Sciences and Humanity (1982), the U.S. National Academy

of Sciences (1984), the American Philosophical Society (1988), the French Academy of Sciences (1995), the Royal Society of London (2007), and the European Academy of Science.

SUMMARY

- Finite automata or finite state machine is the simplest machine that is used for recognizing patterns. It is an abstract machine that has five elements or tuples. It possesses a set of states and rules that are used for moving from one state to another, but it is dependent on the applied input symbol.
- The compiler has to be able to decide when it has reached the end of a keyword rather than being in the middle of a variable (e.g. ended) whose name begins with the keyword.
- The vertices of a finite automata are called its states and are written as Q = {q1, q2, · · ·, qr} rather than V = {v1, v2, · · ·, vr}. The index n tends to be reserved for the length of the input string.
- In a nondeterministic finite automaton (NFA), for each state there can be zero, one, two, or more transitions corresponding to a particular symbol.
- In a nondeterministic finite automaton (NFA), for each state there can be zero, one, two, or more transitions corresponding to a particular symbol.
- That every regular language can be represented by a regular expression takes more effort. To this end, we introduce yet another variant of NFAs, called GNFAs (for Generalized NFAs).
- A reactive system is a system that changes its actions, outputs and conditions/ status in response to stimuli from within or outside it. It is an event driven or control driven system continuously having to react to external and/or internal stimuli.
- Initially the computer is in wait state waiting for "Request for Next Message" (RFNM) to come from another computer. When a RFNM starts coming, it goes into the state of receiving it (Our interpretation is that the computer is in a state of receiving an RFNM and it is taking the action of receiving the RFNM).

KNOWLEDGE CHECK

1. Which of the following pairs of regular expressions are equivalent?

a. 1 (01)* and (10)* 1

b. x (xx) * and (xx) * x

c. x+ and x+x*+

d. All of these

2. Number of states of the FSM required to simulate behavior of a computer with a memory capable of storing "m" words, each of length 'n'

a. m x 2^n

b. 2^{mn}

c. 2^{m+n}

d. All of these

3. To describe the complement of a language, it is very important to describe the ---------- of that language over which the language is defined.

a. Alphabet

b. Regular Expression

c. String

d. Word

4. Finite automata requires minimum _____ number of stacks.

a. 1

b. 0

c. 2

d. None of the mentioned

5. Any language that cannot be expressed by a RE is said to be regular language.

a. True

b. False

6. There are ..... tuples in finite state machine.

a. 4

b. 5

c. 6

d. unlimited

7. Final state is represented by?

a. Square

b. Circle

c. Double Circlue

d. Rhombus

8. In finite automata, Q stands for?

a. Finite set of input symbols

b. Transition function

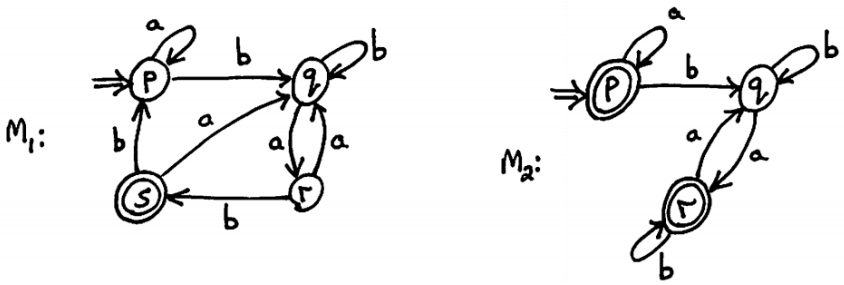
c. Finite set of states

d. One Start state

9. **In Final state, the number of odd arrows are one greater than even, i.e. odd = even+1?**
   a. TRUE
   b. FALSE
   c. Can be true or false
   d. Cannot say
10. **Languages of a automata is**
   a. If it halts
   b. If it is accepted by automata
   c. If automata touch final state in its life time
   d. All language are language of automata

REVIEW QUESTIONS

1. What do you understand by finite automata? Explain its application
2. Write the short note on "nondeterministic finite automata".
3. Consider the Finite Automata shown in below figure.



Find the answer the following questions regarding each of the automata.

   a. Name its start vertex.
   b. List its final vertices.
   c. List the series of vertices the finite automata goes through as the following input is read: ababb.
   d. Is ababb recognized by the automata?

4. For each of the following regular expressions answer the following questions:
   a. $(a \cup b)^*a(a \cup b)^*$.
   b. $(\lambda \cup a)b^*$

5. If for language L the pumping length is three and $a^3\ b^3 \in L$, then if you choose $s = a^3\ b^3$ and apply the Pumping Lemma to s, how many pairs x, y are there, and what are these pairs?

Check Your Result

1. (d) 2. (b) 3. (a) 4. (b) 5. (b)

6. (b) 7. (c) 8. (c) 9. (a) 10. (b)

REFERENCES

1. Aho, Sethi, Ullman, Compilers Principles, Techniques and Tools, Pearson Education, 2003.
2. Gandy, R.O. and C.E.M. Yates (editors). Collected Works of A.M. Turing: Mathematical Logic. Amsterdam: Elsevier Science, 2001.
3. Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D. (2001). Introduction to Automata Theory, Languages, and Computation (2nd ed.). Reading Mass: Addison-Wesley.
4. Kohavi, ZVI, Switching And Finite Automata Theory, Tata McGraw-Hill, 2006.
5. Mishra, KLP, Chandrasekaran, N. Theory of Computer Science, (Automata, Languages and Computation) PHI, 2002.
6. Mordeson, John N. and Davendar S. Malik. Fuzzy Automata and Languages. Boca Raton: CRC Press, 2002.
7. Pandey, An Introduction to Automata Theory and Formal Languages, S. K. Kataria & Sons, 2010.
8. Rodger, Susan; Finley, Thomas (2006). JFLAP: An Interactive Formal Languages and Automata Package (1st ed.). Sudbury, MA: Jones and Bartlett.
9. Sipser, Michael (2006). Introduction to the Theory of Computation (2nd ed.). Boston Mass: Thomson Course Technology.
10. V. Diekert and P. Gastin. First-order definable languages. In Logic and Automata: History and Perspectives, Texts in Logic and Games, pp. 261–306. Amsterdam University Press, (2008).
11. Wolfram, Stephen. A New Kind of Science. Champaign: Wolfram Media, 2002.

CHAPTER

3

PUSHDOWN AUTOMATA THEORY

Learning Objectives

After studying this chapter, you will be able to:

1. Describe the pushdown automata
2. Explain the context-free grammar

INTRODUCTION

A pushdown automaton (PDA) is a type of automaton that employs a stack. Pushdown automata are used in theories about what can be computed by machines. They are more capable than finite-state machines but less capable than Turing machines. Deterministic pushdown automata can recognize all deterministic context-free languages while nondeterministic ones can recognize all context-free languages, with the former often used in parser design. The term "pushdown" refers to the fact that the stack can be regarded as being "pushed down" like a tray dispenser at a cafeteria, since the operations never work on elements other than the top element. A stack automaton, by contrast, does allow access to and operations on deeper elements. Stack automata can recognize a strictly larger set of languages than pushdown automata. A nested stack automaton allows full access, and also allows stacked values to be entire sub-stacks rather than just single finite symbols.

3.1 PUSHDOWN AUTOMATA

A pushdown automaton (PDA) is a finite automaton equipped with a stack-based memory.

A rough sketch of the architecture of a DFA or NFA is given in figure 3.1: the control unit labeled 'Automaton' processes the input that is provided, and produces a yes/no answer for the input string processed so far. Yes, if the current state of the control unit is a final state; no, if the current state of the **control unit** is a non-final state. The only 'memory' that is available are the states of the control unit.

Keyword

Control unit is a component of a computer's central processing unit (CPU) that directs the operation of the processor.



Figure 3.1: Architecture of finite automaton

We next consider an abstract machine model that does have memory, viz. memory in the form of a stack. The stack can be accessed only at the top: something can be added on top of the stack via a push operation, or something can be removed from the top of the stack via a pop operation. Items under the top of the stack cannot be inspected directly.

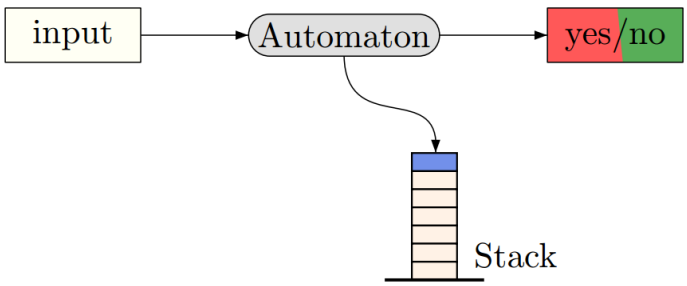


Figure 3.2: Architecture of a push-down automaton

For this reason, the machine model is generally referred to as a push-down automaton, PDA for short. We assume that we have means to check if the stack is empty, i.e. to observe that there is no item on (top of) the stack. See figure 3.2 for the augmented architecture of a push-down automaton.

3.1.1 Basic Structure of PDA

A pushdown automaton is a way to implement a context-free grammar in a similar way we design DFA for a regular grammar. A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.

Basically a pushdown automaton is –

"Finite state machine" + "a stack"

A pushdown automaton has three components –

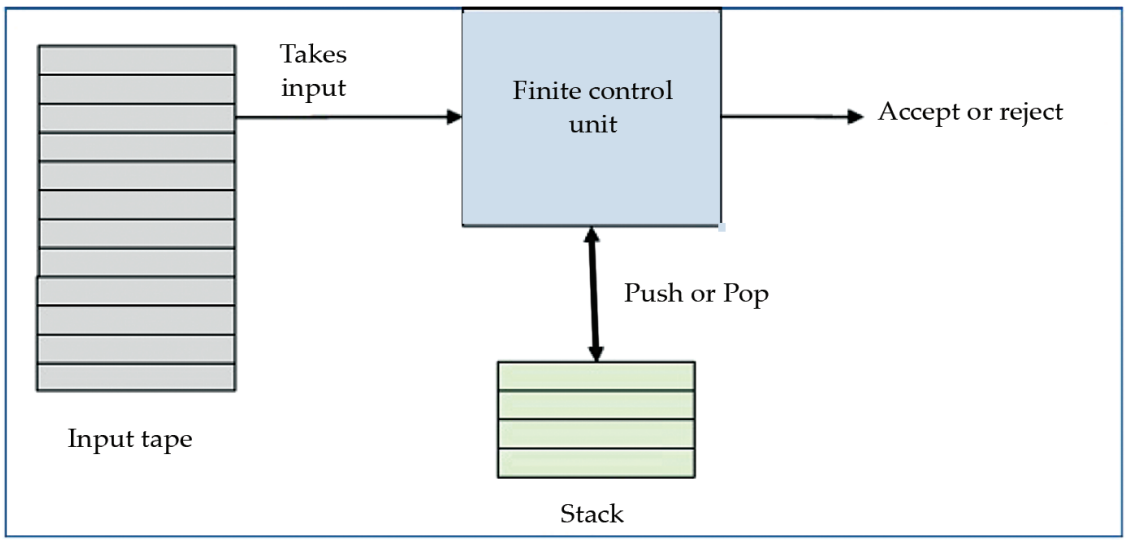
- an input tape,
- a control unit, and
- a stack with infinite size.

The stack head scans the top symbol of the stack.

A stack does two operations –

- Push – a new symbol is added at the top.
- Pop – the top symbol is read and removed.

A PDA may or may not read an input symbol, but it has to read the top of the stack in every transition.



If we have seen 5 b's and 3 a's in any order, then the stack should be "bbc". The transition to the final state represents the only non-determinism in the PDA in that it must guess when the input is empty in order to pop off the stack bottom.

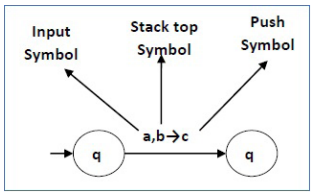
A PDA can be formally described as a seven-tuple (Q, ∑, S, δ, q_0, I, F) –

- **Q** is the finite number of states
- **∑** is input alphabet
- **S** is stack symbols
- **δ** is the transition function: Q × (∑ ∪ {ε}) × S × Q × S*
- $\mathbf{q_0}$ is the initial state ($q_0 \in Q$)
- **I** is the initial stack top symbol (I ∈ S)
- **F** is a set of accepting states (F ∈ Q)

Remember

Additionally, the CU's orderly hardware coordination properly sequences these control signals then configures the many hardware units comprising the CPU, directing how data should also be moved, changed, and stored outside the CPU (i.e. memory) according to the instruction's objective.

The following diagram shows a transition in a PDA from a state q_1 to state q_2, labeled as a,b → c –



This means at state q_1, if we encounter an input string 'a' and top symbol of the **stack** is 'b', then we pop 'b', push 'c' on top of the stack and move to state q_2.

Terminologies Related to PDA

There are some terminologies related to PDA, such as:

Instantaneous Description

The instantaneous description (ID) of a PDA is represented by a triplet (q, w, s) where

- **q** is the state
- **w** is unconsumed input
- **s** is the stack contents

Turnstile Notation

The "turnstile" notation is used for connecting pairs of ID's that represent one or many moves of a PDA. The process of transition is denoted by the turnstile symbol "⊢".

Consider a PDA (Q, ∑, S, δ, q_0, I, F). A transition can be mathematically represented by the following turnstile notation −

(p, aw, Tβ) ⊢ (q, w, αb)

This implies that while taking a transition from state p to state q, the input symbol 'a' is consumed, and the top of the stack 'T' is replaced by a new string 'α'.

Note − If we want zero or more moves of a PDA, we have to use the symbol (⊢*) for it.

Keyword

Stack is an abstract data type that serves as a collection of elements, with two principal operations: push, which adds an element to the collection, and pop, which removes the most recently added element that was not yet removed.

3.1.2 Pushdown Automata Acceptance

There are two different ways to define PDA acceptability.

Final State Acceptability

In final state acceptability, a PDA accepts a string when, after reading the entire string, the PDA is in a final state. From the starting state, we can make moves that end up in a final state with any stack values. The stack values are irrelevant as long as we end up in a final state.

For a PDA (Q, ∑, S, δ, q_0, I, F), the language accepted by the set of final states F is −

L(PDA) = {w | (q_0, w, I) ⊢* (q, ε, x), q ∈ F}

for any input stack string **x**.

Empty Stack Acceptability

Here a PDA accepts a string when, after reading the entire string, the PDA has emptied its stack.

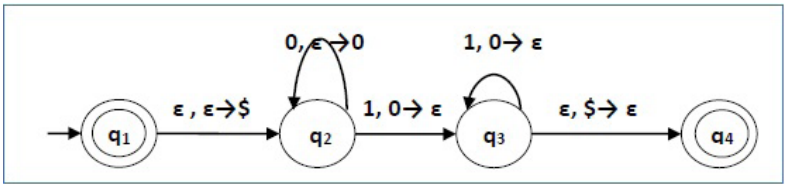
For a PDA (Q, ∑, S, δ, q_0, I, F), the language accepted by the empty stack is −

L(PDA) = {w | $(q_0, w, I) \vdash^* (q, \varepsilon, \varepsilon), q \in Q$}

Example:

Construct a PDA that accepts L = $\{0^n 1^n \mid n \geq 0\}$

Solution



PDA for L= {0^n 1^n | n≥0}

This language accepts L = {ε, 01, 0011, 000111,.....}

Here, in this example, the number of 'a' and 'b' have to be same.

- Initially we put a special symbol '$' into the empty stack.
- Then at state q_2, if we encounter input 0 and top is Null, we push 0 into stack. This may iterate. And if we encounter input 1 and top is 0, we pop this 0.
- Then at state q_3, if we encounter input 1 and top is 0, we pop this 0. This may also iterate. And if we encounter input 1 and top is 0, we pop the top element.
- If the special symbol '$' is encountered at top of the stack, it is popped out and it finally goes to the accepting state q_4.

Example

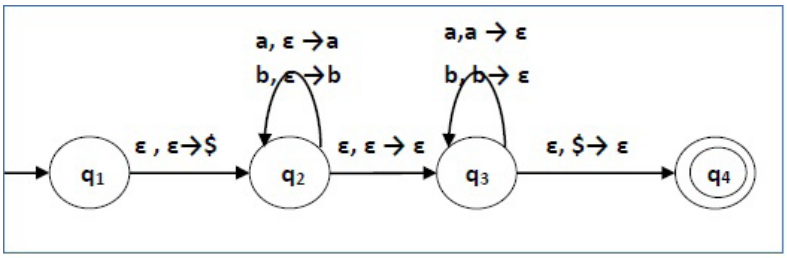
Construct a PDA that accepts L = $\{ww^R \mid w = (a+b)^*\}$

Solution

Initially we put a special symbol '$' into the empty stack. At state q_2, the w is being read. In state q_3, each 0 or 1 is popped when it matches the input. If any other input is given, the PDA will go to a dead state. When we reach that special symbol '$', we go to the accepting state q_4.

Remember

Using the same stack for both data and procedure calls has important security implications of which a programmer must be aware in order to avoid introducing serious security bugs into a program.



PDA for L= {ww^R | w = (a+b)*}

3.1.3 PDA & Context-Free Grammar

If a grammar **G** is context-free, we can build an equivalent nondeterministic PDA which accepts the language that is produced by the context-free grammar **G**. A parser can be built for the grammar G.

Also, if P is a pushdown automaton, an equivalent context-free grammar G can be constructed where

$$L(G) = L(P)$$

Convert CFG to PDA

The idea behind the conversion from a CFG to an NPDA using the SLR(1) parsing method, is to push terminals from the strings on the stack, pop right-hand sides of productions off the stack, and push their left-hand sides on the stack, until the start variable is on the stack.

Algorithm to find CFG corresponding to a given PDA

Input − A CFG, G = (V, T, P, S)

Output − Equivalent PDA, P = (Q, ∑, S, δ, q_0, I, F) such that the non- terminals of the grammar G will be $\{X_{wx} \mid w,x \in Q\}$ and the start state will be $A_{q0,F}$.

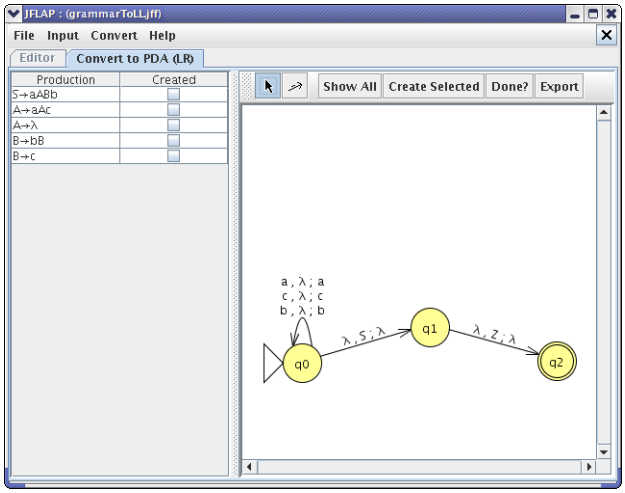
- Step 1 − For every w, x, y, z ∈ Q, m ∈ S and a, b ∈ ∑, if δ (w, a, ε) contains (y, m) and (z, b, m) contains (x, ε), add the production rule $X_{wx} \rightarrow a\ X_{yz} b$ in grammar G.
- Step 2 − For every w, x, y, z ∈ Q, add the production rule $X_{wx} \rightarrow X_{wy} X_{yx}$ in grammar G.
- Step 3 − For w ∈ Q, add the production rule $X_{ww} \rightarrow \varepsilon$ in grammar G.

How to Convert CFG to PDA (LR)

We will begin by loading the same grammar that we used for building the LL(1) parse table. After loading the grammar, click on Convert and click Convert CFG to PDA (LR).

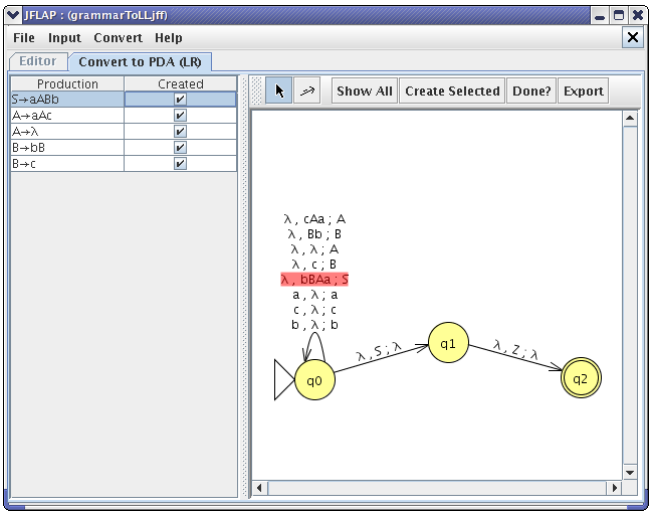
DID YOU KNOW?

The efficiency of context-free grammar parsing is determined by the automaton that accepts it. Deterministic context-free grammars are accepted by deterministic pushdown automata and can be parsed in linear time, for example by the LR parser.[

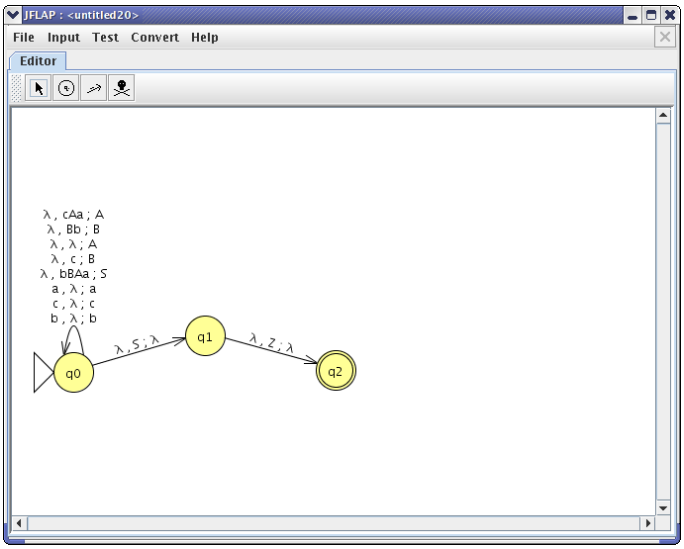


Just like the conversion to PDA using LL parsing, we need to add transitions between the states. However this time transitions are going to be quite different, because we are using a different parsing method. Still, our goal is to reach state q2. For each terminal, add a loop transition on state q0 that reads the terminal in the input and pushes it on the stack. For each production, add a loop transition on state qo that pops the right side of the production and pushes on the left side of the production from the stack. Therefore, for production "S->aABb", we would put transition "lambda, bBAa ; S" on state q0. We can convert all the other productions into transitions. After we have successfully completed them, the window will look like:

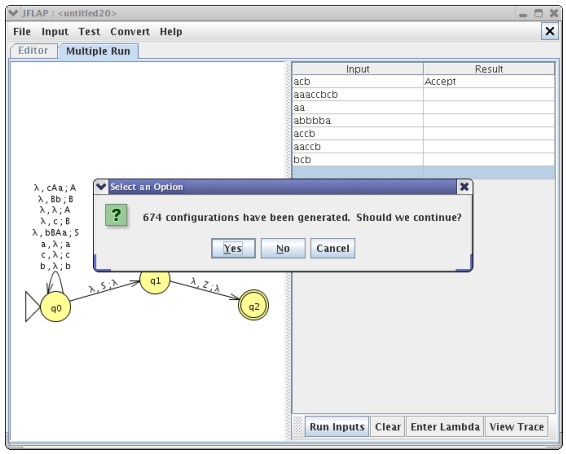
Note: You can click the production on the left side and click on Create Selected to add the corresponding transition in the PDA.



Now, we have successfully transformed the context-free grammar to a pushdown automaton. We can export this PDA and test it on some inputs and compare it to the original CFG. After exporting the PDA, your new JFLAP window should look like:



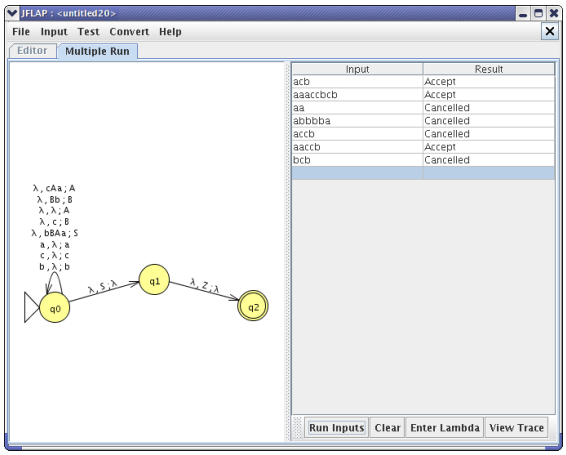
Now let's test many strings. Click on Input, then click on Multiple Run. In the Multiple Run window, type input "acb", "aaaccbcb", "aa", "abbbba", "accb", "aaccb", "bcb". When you click Run Inputs, you might receive a message like this:



Remember

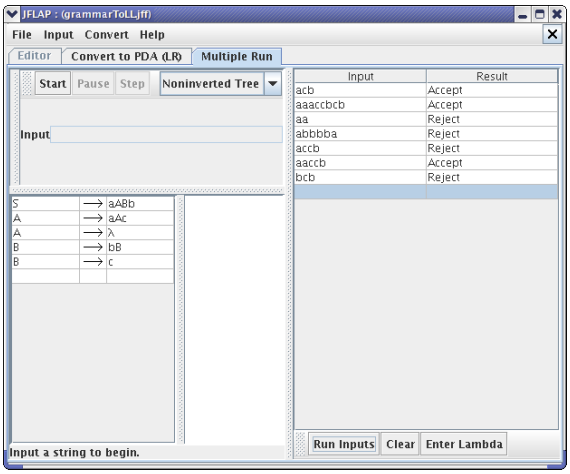
The program must keep track of the size (length) of the stack, using a variable top that records the number of items pushed so far, therefore pointing to the place in the array where the next element is to be inserted (assuming a zero-based index convention).

Notice that string "acb" is already accepted by PDA. However, when PDA is parsing the string "aaaccbcb", it generated 674 configurations and still did not achieve the string yet. This does not necessarily mean that the string is impossible to derive. But, it also implies that it could be the case that the string is impossible to derive. So let's press Yes button and see what happens. After clicking Yes to couple of times, you will notice that the string "aaaccbcb" is accepted eventually.



However, this is not always the case. Notice when you reach string "aa", if you keep clicking on the Yes button, your configurations will get bigger and you will still not achieve the string. In this case, it would be wise not to continue. Select No for this string. After finishing the parsing, your window should look like:

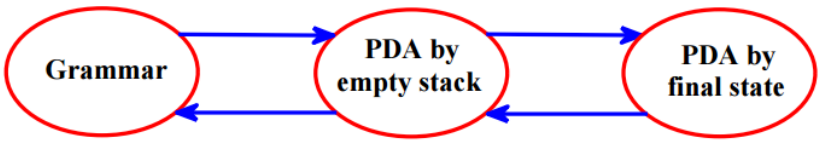
If you run these same inputs on the grammar window using Multiple Run, you will get the same result. Your Multiple Run window should look like this after running:



Equivalence of PDA's and CFG's

The goal is to prove that the following three classes of the languages are all the same class.

- The context-free languages (The language defined by CFG's).
- The languages that are accepted by empty stack by some PDA.
- The languages that are accepted by final state by some PDA.



Algorithm to find PDA corresponding to a given CFG

Input – A CFG, G = (V, T, P, S)

Output – Equivalent PDA, P = (Q, ∑, S, δ, q_0, I, F)

- Step 1 – Convert the productions of the CFG into GNF.
- Step 2 – The PDA will have only one state {q}.
- Step 3 – The start symbol of CFG will be the start symbol in the PDA.
- Step 4 – All non-terminals of the CFG will be the stack symbols of the PDA and all the terminals of the CFG will be the input symbols of the PDA.

- Step 5 – For each production in the form A → aX where a is terminal and A, X are combination of terminal and non-terminals, make a transition δ (q, a, A).

Problem:

Construct a PDA from the following CFG.

G = ({S, X}, {a, b}, P, S)

where the productions are –

S → XS | ε , A → aXb | Ab | ab

Remember

To implement these features in a compiled language, programs must usually be shipped with a runtime library that includes a version of the compiler itself.

Solution:

Let the equivalent PDA,

P = ({q}, {a, b}, {a, b, X, S}, δ, q, S)

where δ –

δ(q, ε , S) = {(q, XS), (q, ε)}

δ(q, ε , X) = {(q, aXb), (q, Xb), (q, ab)}

δ(q, a, a) = {(q, ε)}

δ(q, 1, 1) = {(q, ε)}

3.1.4 Parsing

Parsing is used to derive a string using the production rules of a grammar. It is used to check the acceptability of a string. **Compiler** is used to check whether or not a string is syntactically correct. A parser takes the inputs and builds a parse tree.

A parser can be of two types –

- *Top-Down Parser* – Top-down parsing starts from the top with the start-symbol and derives a string using a parse tree.
- *Bottom-Up Parser* – Bottom-up parsing starts from the bottom with the string and comes to the start symbol using a parse tree.

Design of Top-Down Parser

For top-down parsing, a PDA has the following four types of transitions −

- Pop the non-terminal on the left hand side of the production at the top of the stack and push its right-hand side string.
- If the top symbol of the stack matches with the input symbol being read, pop it.
- Push the start symbol 'S' into the stack.
- If the input string is fully read and the stack is empty, go to the final state 'F'.

Example

Design a top-down parser for the expression "x+y*z" for the grammar G with the following production rules −

P: S → S+X | X, X → X*Y | Y, Y → (S) | id

Solution

If the PDA is (Q, ∑, S, δ, q_0, I, F), then the top-down parsing is −

(x+y*z, I) ⊢(x +y*z, SI) ⊢ (x+y*z, S+XI) ⊢(x+y*z, X+XI)

⊢(x+y*z, Y+X I) ⊢(x+y*z, x+XI) ⊢(+y*z, +XI) ⊢ (y*z, XI)

⊢(y*z, X*YI) ⊢(y*z, y*YI) ⊢(*z,*YI) ⊢(z, YI) ⊢(z, zI) ⊢(ε, I)

Keyword

Compiler is a computer program that transforms source code written in a programming language into another computer language, with the latter often having a binary form known as object code

Design of a Bottom-Up Parser

For bottom-up parsing, a PDA has the following four types of transitions −

- Push the current input symbol into the stack.
- Replace the right-hand side of a production at the top of the stack with its left-hand side.
- If the top of the stack element matches with the current input symbol, pop it.
- If the input string is fully read and only if the start symbol 'S' remains in the stack, pop it and go to the final state 'F'.

Example

Design a top-down parser for the expression "x+y*z" for the grammar G with the following production rules –

P: S → S+X | X, X → X*Y | Y, Y → (S) | id

Solution

If the PDA is (Q, ∑, S, δ, q_0, I, F), then the bottom-up parsing is –

(x+y*z, I) ⊢ (+y*z, xI) ⊢ (+y*z, YI) ⊢ (+y*z, XI) ⊢ (+y*z, SI)

⊢(y*z, +SI) ⊢ (*z, y+SI) ⊢ (*z, Y+SI) ⊢ (*z, X+SI) ⊢ (z, *X+SI)

⊢ (ε, z*X+SI) ⊢ (ε, Y*X+SI) ⊢ (ε, X+SI) ⊢ (ε, SI)

3.2 CONTEXT-FREE GRAMMAR

A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple (N, T, P, S) where

- N is a set of non-terminal symbols.
- T is a set of terminals where N ∩ T = NULL.
- P is a set of rules, P: N → (N ∪ T)*, i.e., the left-hand side of the production rule P does have any right context or left context.
- S is the start symbol.

Example

- The grammar ({A}, {a, b, c}, P, A), P : A → aA, A → abc.
- The grammar ({S, a, b}, {a, b}, P, S), P: S → aSa, S → bSb, S → ε
- The grammar ({S, F}, {0, 1}, P, S), P: S → 00S | 11F, F → 00F | ε

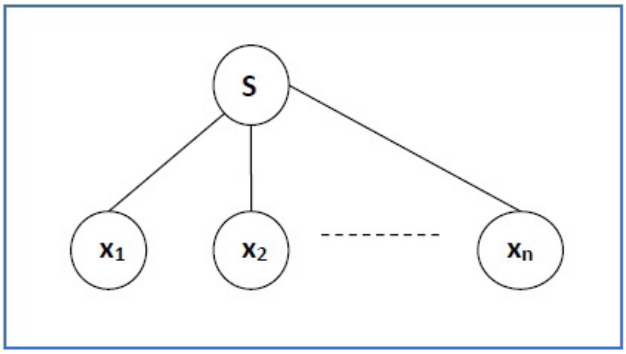
Generation of Derivation Tree

A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information a string derived from a context-free grammar.

Representation Technique

- Root vertex – Must be labeled by the start symbol.
- Vertex – Labeled by a non-terminal symbol.
- Leaves – Labeled by a terminal symbol or ε.

If $S \rightarrow x_1x_2 \ldots\ldots x_n$ is a production rule in a CFG, then the parse tree / derivation tree will be as follows –



There are two different approaches to draw a derivation tree –

Top-down Approach –

- Starts with the starting symbol **S**
- Goes down to tree leaves using productions

Bottom-up Approach –

- Starts from tree leaves
- Proceeds upward to the root which is the starting symbol **S**

Derivation or Yield of a Tree

The derivation or the yield of a parse tree is the final string obtained by concatenating the labels of the leaves of the tree from left to right, ignoring the Nulls. However, if all the leaves are Null, derivation is Null.

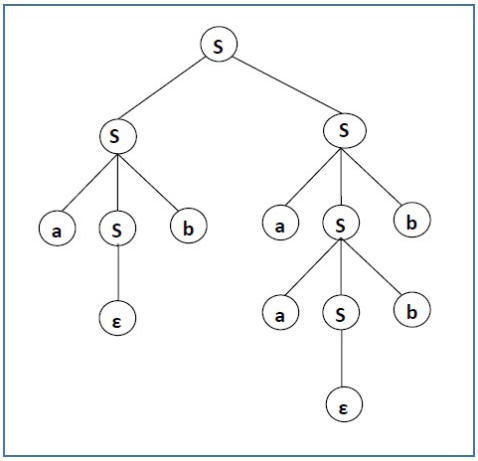
Example

Let a CFG {N,T,P,S} be

N = {S}, T = {a, b}, Starting symbol = S, P = S → SS | aSb | ε

One derivation from the above CFG is "abaabb"

S → SS → aSbS → abS → abaSb → abaaSbb → abaabb



Sentential Form and Partial Derivation Tree

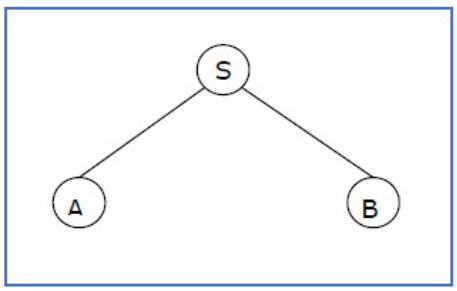
A partial derivation tree is a sub-tree of a derivation tree/parse tree such that either all of its children are in the sub-tree or none of them are in the sub-tree.

Example

If in any CFG the productions are –

$S \rightarrow AB, A \rightarrow aaA \mid \varepsilon, B \rightarrow Bb \mid \varepsilon$

the partial derivation tree can be the following –



If a partial derivation tree contains the root S, it is called a sentential form. The above sub-tree is also in sentential form.

Leftmost and Rightmost Derivation of a String

- Leftmost derivation – A leftmost derivation is obtained by applying production to the leftmost variable in each step.
- Rightmost derivation – A rightmost derivation is obtained by applying production to the rightmost variable in each step.

Example

Let any set of production rules in a CFG be

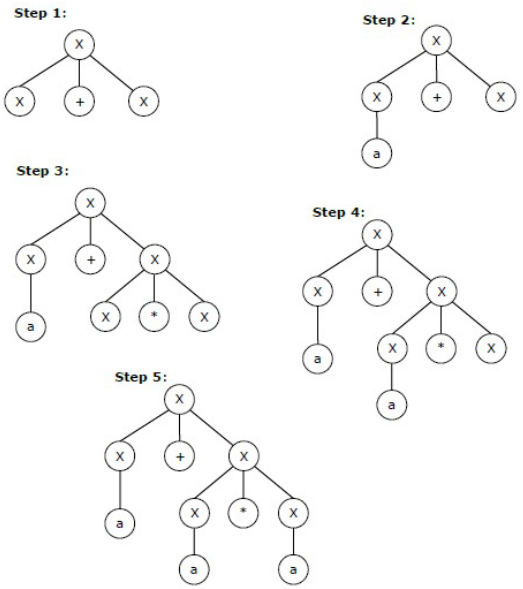
$X \rightarrow X+X \mid X*X \mid X \mid a$

over an alphabet {a}.

The leftmost derivation for the string "a+a*a" may be −

$X \rightarrow X+X \rightarrow a+X \rightarrow a + X*X \rightarrow a+a*X \rightarrow a+a*a$

The stepwise derivation of the above string is shown as below −



The rightmost derivation for the above string **"a+a*a"** may be −

$X \rightarrow X*X \rightarrow X*a \rightarrow X+X*a \rightarrow X+a*a \rightarrow a+a*a$

The stepwise derivation of the above string is shown as below −

Left and Right Recursive Grammars

In a context-free grammar G, if there is a production in the form $X \rightarrow Xa$ where X is a non-terminal and 'a' is a string of terminals, it is called a left recursive production. The grammar having a left recursive production is called a left recursive grammar.

And if in a context-free grammar G, if there is a production is in the form $X \rightarrow aX$ where X is a non-terminal and 'a' is a string of terminals, it is called a right

recursive production. The grammar having a right recursive production is called a right recursive grammar.

3.2.1 Ambiguity in Context-Free Grammars

Keyword

Ambiguous grammar is a context-free grammar for which there exists a string that can have more than one leftmost derivation, while an unambiguous grammar is a context-free grammar for which every valid string has a unique leftmost derivation.

If a context free grammar G has more than one derivation tree for some string w ∈ L(G), it is called an **ambiguous grammar**. There exist multiple right-most or left-most derivations for some string generated from that grammar.

Problem

Check whether the grammar G with production rules –

$$X \rightarrow X+X \mid X*X \mid X \mid a$$

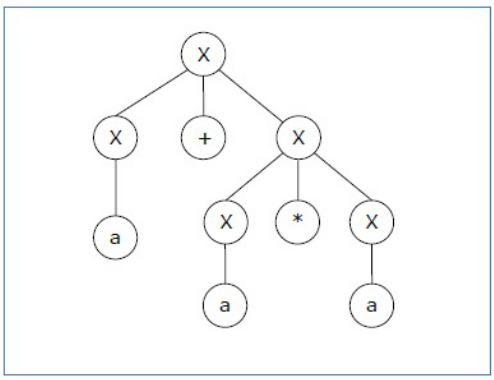
is ambiguous or not.

Solution

Let's find out the derivation tree for the string "a+a*a". It has two leftmost derivations.

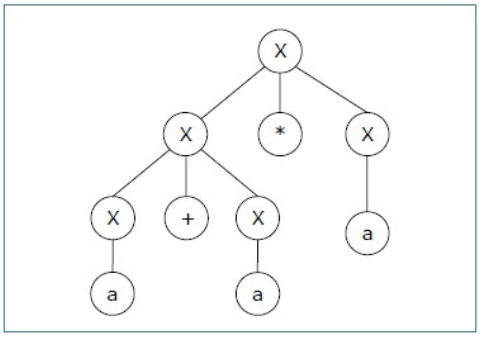
Derivation 1 – X → X+X → a +X → a+ X*X → a+a*X → a+a*a

Parse tree 1 –



Derivation 2 – X → X*X → X+X*X → a+ X*X → a+a*X → a+a*a

Parse tree 2 –



Since there are two parse trees for a single string "a+a*a", the grammar G is ambiguous.

3.2.2 CFL Closure Property

Context-free languages are closed under –

- Union
- Concatenation
- Kleene Star operation

Union

Let L_1 and L_2 be two context free languages. Then $L_1 \cup L_2$ is also context free.

Example

Let $L_1 = \{ a^n b^n , n > 0\}$. Corresponding grammar G_1 will have P: S1 → aAb | ab

Let $L_2 = \{ c^m d^m , n \geq 0\}$. Corresponding grammar G_2 will have P: S2 → cBb | ε

Union of L_1 and L_2, $L = L_1 \cup L_2 = \{ a^n b^n \} \cup \{ c^m d^m \}$

The corresponding grammar G will have the additional production S → S1 | S2

Concatenation

If L_1 and L_2 are context free languages, then L_1L_2 is also context free.

Example

Union of the languages L_1 and L_2, $L = L_1L_2 = \{ a^n b^n c^m d^m \}$

The corresponding grammar G will have the additional production S → S1 S2

Kleene Star

If L is a context free language, then L* is also context free.

Example

Let L = { $a^n b^n$, n ≥ 0}. Corresponding grammar G will have P: S → aAb | ε

Kleene Star L_1 = { $a^n b^n$ }*

The corresponding grammar G_1 will have additional productions S1 → SS_1 | ε

Context-free languages are **not closed** under −

- Intersection − If L1 and L2 are context free languages, then L1 ∩ L2 is not necessarily context free.
- Intersection with Regular Language − If L1 is a regular language and L2 is a context free language, then L1 ∩ L2 is a context free language.
- Complement − If L1 is a context free language, then L1′ may not be context free.

3.2.3 CFG Simplification

In a CFG, it may happen that all the production rules and symbols are not needed for the derivation of strings. Besides, there may be some null productions and unit productions. Elimination of these productions and symbols is called simplification of CFGs. Simplification essentially comprises of the following steps −

- Reduction of CFG
- Removal of Unit Productions
- Removal of Null Productions

Reduction of CFG

CFGs are reduced in two phases −

Phase 1 − Derivation of an equivalent grammar, **G′**, from the CFG, **G**, such that each variable derives some terminal string.

Derivation Procedure −

- Step 1 − Include all symbols, W_1, that derive some terminal and initialize i=1.
- Step 2 − Include all symbols, W_{i+1}, that derive W_i.
- Step 3 − Increment i and repeat Step 2, until $W_{i+1} = W_i$.
- Step 4 − Include all production rules that have W_i in it.

Phase 2 − Derivation of an equivalent grammar, G″, from the CFG, G′, such that each symbol appears in a sentential form.

Derivation Procedure −

- Step 1 − Include the start symbol in Y_1 and initialize i = 1.
- Step 2 − Include all symbols, Y_{i+1}, that can be derived from Y_i and include all production rules that have been applied.
- Step 3 − Increment i and repeat Step 2, until $Y_{i+1} = Y_i$.

Problem:

Find a reduced grammar equivalent to the grammar G, having production rules, P: S → AC | B, A → a, C → c | BC, E → aA | e

Solution by phase 1 −

T = { a, c, e }

W_1 = { A, C, E } from rules A → a, C → c and E → aA

W_2 = { A, C, E } U { S } from rule S → AC

W_3 = { A, C, E, S } U ∅

Since $W_2 = W_3$, we can derive G′ as −

G′ = { { A, C, E, S }, { a, c, e }, P, {S}}

where P: S → AC, A → a, C → c , E → aA | e

Phase 2 −

Y_1 = { S }

Y_2 = { S, A, C } from rule S → AC

Y_3 = { S, A, C, a, c } from rules A → a and C → c

Y_4 = { S, A, C, a, c }

Since $Y_3 = Y_4$, we can derive G″ as −

G″ = { { A, C, S }, { a, c }, P, {S}}

where P: S → AC, A → a, C → c

Removal of Unit Productions

Any production rule in the form A → B where A, B ∈ Non-terminal is called unit production..

Removal Procedure –

- *Step 1* – To remove A → B, add production A → x to the grammar rule whenever B → x occurs in the grammar. [x ∈ Terminal, x can be Null]
- *Step 2* – Delete A → B from the grammar.
- *Step 3* – Repeat from step 1 until all unit productions are removed.

Problem:

Remove unit production from the following –

$$S \rightarrow XY, X \rightarrow a, Y \rightarrow Z \mid b, Z \rightarrow M, M \rightarrow N, N \rightarrow a$$

Solution –

There are three unit productions in the grammar –

$Y \rightarrow Z$, $Z \rightarrow M$, and $M \rightarrow N$

(1) At first, we will remove $M \rightarrow N$.

As $N \rightarrow a$, we add $M \rightarrow a$, and $M \rightarrow N$ is removed.

The production set becomes

$$S \rightarrow XY, X \rightarrow a, Y \rightarrow Z \mid b, Z \rightarrow M, M \rightarrow a, N \rightarrow a$$

(2) Now we will remove $Z \rightarrow M$.

As $M \rightarrow a$, we add $Z \rightarrow a$, and $Z \rightarrow M$ is removed.

The production set becomes

$$S \rightarrow XY, X \rightarrow a, Y \rightarrow Z \mid b, Z \rightarrow a, M \rightarrow a, N \rightarrow a$$

(3) Now we will remove $Y \rightarrow Z$.

As $Z \rightarrow a$, we add $Y \rightarrow a$, and $Y \rightarrow Z$ is removed.

The production set becomes

$$S \rightarrow XY, X \rightarrow a, Y \rightarrow a \mid b, Z \rightarrow a, M \rightarrow a, N \rightarrow a$$

Now Z, M, and N are unreachable, hence we can remove those.

The final CFG is unit production free –

$$S \rightarrow XY, X \rightarrow a, Y \rightarrow a \mid b$$

Removal of Null Productions

In a CFG, a non-terminal symbol 'A' is a null able variable if there is a production A → ε or there is a derivation that starts at A and finally ends up with

ε: A → ..... → ε

Removal Procedure

- Step 1 − Find out nullable non-terminal variables which derive ε.
- Step 2 − For each production A → a, construct all productions A → x where x is obtained from 'a' by removing one or multiple non-terminals from Step 1.
- Step 3 − Combine the original productions with the result of step 2 and remove ε - productions.

Problem:

Remove null production from the following −

S → ASA | aB | b, A → B, B → b | ∈

Solution −

There are two nullable variables − A and B

(1) At first, we will remove B → ε.

After removing **B** → ε, the production set becomes −

S→ASA | aB | b | a, A ε B| b | &epsilon, B → b

(2) Now we will remove A → ε.

After removing **A** → ε, the production set becomes −

S→ASA | aB | b | a | SA | AS | S, A → B| b, B → b

This is the final production set without null transition.

3.2.4 Chomsky Normal Form

A CFG is in Chomsky Normal Form if the Productions are in the following forms −

- A → a
- A → BC

- $S \rightarrow \varepsilon$

where A, B, and C are non-terminals and a is terminal.

Algorithm

Algorithm to Convert into Chomsky Normal Form –

- *Step 1* – If the start symbol S occurs on some right side, create a new start symbol S′ and a new production S′→ S.
- *Step 2* – Remove Null productions. (Using the Null production removal algorithm discussed earlier)
- *Step 3* – Remove unit productions. (Using the Unit production removal algorithm discussed earlier)
- *Step 4* – Replace each production $A \rightarrow B_1 \ldots B_n$ where $n > 2$ with $A \rightarrow B_1C$ where $C \rightarrow B_2 \ldots B_n$. Repeat this step for all productions having two or more symbols in the right side.
- *Step 5* – If the right side of any production is in the form $A \rightarrow aB$ where a is a terminal and A, B are non-terminal, then the production is replaced by $A \rightarrow XB$ and $X \rightarrow a$. Repeat this step for every production which is in the form $A \rightarrow aB$.

Problem

Convert the following CFG into CNF

$S \rightarrow ASA \mid aB, A \rightarrow B \mid S, B \rightarrow b \mid \varepsilon$

Solution

(1) Since S appears in R.H.S, we add a new state S_0 and $S_0 \rightarrow S$ is added to the production set and it becomes –

$$S_0 \rightarrow S, S \rightarrow ASA \mid aB, A \rightarrow B \mid S, B \rightarrow b \mid \in$$

(2) Now we will remove the null productions –

$$B \rightarrow \in \text{ and } A \rightarrow \in$$

After removing $B \rightarrow \varepsilon$, the production set becomes –

$$S_0 \rightarrow S, S \rightarrow ASA \mid aB \mid a, A \rightarrow B \mid S \mid \in, B \rightarrow b$$

After removing $A \rightarrow \in$, the production set becomes –

$$S_0 \rightarrow S, S \rightarrow ASA \mid aB \mid a \mid AS \mid SA \mid S, A \rightarrow B \mid S, B \rightarrow b$$

(3) Now we will remove the unit productions.

After removing S → S, the production set becomes –

$$S_0 \rightarrow S,\ S \rightarrow ASA \mid aB \mid a \mid AS \mid SA,\ A \rightarrow B \mid S,\ B \rightarrow b$$

After removing $S_0 \rightarrow S$, the production set becomes –

$$S_0 \rightarrow ASA \mid aB \mid a \mid AS \mid SA,\ S \rightarrow ASA \mid aB \mid a \mid AS \mid SA$$

$$A \rightarrow B \mid S,\ B \rightarrow b$$

After removing A→ B, the production set becomes –

$$S_0 \rightarrow ASA \mid aB \mid a \mid AS \mid SA,\ S \rightarrow ASA \mid aB \mid a \mid AS \mid SA$$

$$A \rightarrow S \mid b$$

$$B \rightarrow b$$

After removing A→ S, the production set becomes –

$$S_0 \rightarrow ASA \mid aB \mid a \mid AS \mid SA,\ S \rightarrow ASA \mid aB \mid a \mid AS \mid SA$$

$$A \rightarrow b \mid ASA \mid aB \mid a \mid AS \mid SA,\ B \rightarrow b$$

(4) Now we will find out more than two variables in the R.H.S

Here, $S_0 \rightarrow ASA$, $S \rightarrow ASA$, $A \rightarrow ASA$ violates two Non-terminals in R.H.S.

Hence we will apply step 4 and step 5 to get the following final production set which is in CNF –

$$S_0 \rightarrow AX \mid aB \mid a \mid AS \mid SA$$

$$S \rightarrow AX \mid aB \mid a \mid AS \mid SA$$

$$A \rightarrow b \mid AX \mid aB \mid a \mid AS \mid SA$$

$$B \rightarrow b$$

$$X \rightarrow SA$$

(5) We have to change the productions $S_0 \rightarrow aB$, $S \rightarrow aB$, $A \rightarrow aB$

And the final production set becomes –

$$S_0 \rightarrow AX \mid YB \mid a \mid AS \mid SA$$

$$S \rightarrow AX \mid YB \mid a \mid AS \mid SA$$

$$A \rightarrow b\ A \rightarrow b \mid AX \mid YB \mid a \mid AS \mid SA$$

$B \rightarrow b$

$X \rightarrow SA$

$Y \rightarrow a$

3.2.5 Greibach Normal Form

A CFG is in Greibach Normal Form if the Productions are in the following forms –

$A \rightarrow b$

$A \rightarrow bD_1 \ldots D_n$

$S \rightarrow \varepsilon$

where A, D_1,....,D_n are non-terminals and b is a terminal.

Keyword

Algorithm is an effective method that can be expressed within a finite amount of space and time and in a well-defined formal language for calculating a function.

Algorithm

Algorithm to Convert a CFG into Greibach Normal Form

- *Step 1* – If the start symbol S occurs on some right side, create a new start symbol S′ and a new production S′ → S.
- *Step 2* – Remove Null productions. (Using the Null production removal **algorithm** discussed earlier)
- *Step 3* – Remove unit productions. (Using the Unit production removal algorithm discussed earlier)
- *Step 4* – Remove all direct and indirect left-recursion.
- *Step 5* – Do proper substitutions of productions to convert it into the proper form of GNF.

Problem

Convert the following CFG into CNF

$S \rightarrow XY \mid Xn \mid p$

$X \rightarrow mX \mid m$

$Y \rightarrow Xn \mid o$

Solution

Here, **S** does not appear on the right side of any production and there are no unit or null productions in the production rule set. So, we can skip Step 1 to Step 3.

Step 4: Now after replacing

$$X \text{ in } S \rightarrow XY \mid Xo \mid p$$

with

$$mX \mid m$$

we obtain

$$S \rightarrow mXY \mid mY \mid mXo \mid mo \mid p.$$

And after replacing

$$X \text{ in } Y \rightarrow X_n \mid o$$

with the right side of

$$X \rightarrow mX \mid m$$

we obtain

$$Y \rightarrow mXn \mid mn \mid o.$$

Two new productions O → o and P → p are added to the production set and then we came to the final GNF as the following –

$$S \rightarrow mXY \mid mY \mid mXC \mid mC \mid p$$

$$X \rightarrow mX \mid m$$

$$Y \rightarrow mXD \mid mD \mid o$$

$$O \rightarrow o$$

$$P \rightarrow p$$

CASE STUDY

CPU STARVATION

This was a non-data sharing environment in which the total machine CPU time hovered near 100% busy during the production day. The DBAs analyzed the DB2PM Accounting Detail reports to determine whether excessive CPU could be attributed to DB2, to certain applications, to access paths for certain objects, or to something else.

Develop Resource Measurement

For application CPU resource consumption the DBAs developed a "CPU time per SQL statement" measurement. They used the In-DB2 TCB time from the Accounting Detail report as their CPU measure, and created a weighted average measure of the number of SQL statements executed by the application.

Their exact formula is not important here. (For example, they researched the average CPU usage required for various DML statements based on the DB2 V7 Performance Topics Redbook, then modified this to suit their applications.) What is important is that they created a generic measurement for application CPU usage that they could then use to find "CPU hog" programs.

Rank Applications by Resource Consumption

For each application they divided the In-DB2 TCB time by the total number of SQL statements executed. This resulted in numbers in the range of 0.1 to 1.0 milliseconds (msec) per statement. Those applications scoring higher than 1.0 msec were considered to be high CPU consumers.

Determine Alternate Resource Availability

The next step was to determine what other resources were available. In this case, real memory on the box where DB2 was running was not 100% utilized. Also, the DBA team reviewed their overall resource profile and determined that they had some unused DASD resources; in addition, they could afford to sacrifice throughput for some low-priority applications.

Develop Resource Tradeoff Options

In consultation with the systems programming staff, the DBAs were able to get an additional gigabyte of real memory dedicated to DB2. This was implemented in the form of several dataspaces, and several of the most heavily-used tables were allocated

to virtual pools in dataspaces.

The result: an overall decrease in CPU utilization, due to fewer required physical I/Os to access the critical tables. The tradeoff: additional memory usage.

Next, the DBA team analyzed the accounting detail report looking for DB2 subsystem issues. They noticed that the In-DB2 Not Accounted For Time was relatively high (about five percent of In-DB2 Elapsed Time). This indicated that the DB2 address space was waiting for CPU dispatching. This happens most often when the CPU is extremely busy, as DB2 must wait for a CPU to do some synchronous tasks such as log writes. If DB2 must wait for a CPU, then the application waits for DB2.

In this case, reduction of overall CPU time was called for, as this problem tended to occur during times of high CPU busy. They then implemented the following changes:

- Many small and medium-sized tables (under 100,000 rows) that were defined with data compression (COMPRESS YES) were re-defined to remove the compression option. They also concurrently increased the sizes of some of the virtual pools. This increased DASD usage somewhat, and also caused a slight increase in CPU due to the additional I/Os required by SQL statements that accessed large numbers of rows. However, there was a net decrease in CPU time by forgoing compression and decompression of rows for those tables.
- Many distributed applications (i.e., those not originating in z/OS) were invoking DB2 stored procedures. In many cases these stored procedures were being used to execute a single SQL statement that returned one row. The overhead for invoking a stored procedure is several thousand machine instructions; so, for some highly-used stored procedures the DBAs had developers use dynamic SQL instead. In addition, they enlarged the dynamic statement caches. This caused a slight increase in CPU time as dynamic SQL needed to be bound (unless it was in the cache); however, there was a net reduction in overall CPU usage as hundreds of thousands of stored procedure invocations were removed.
- The frequency of DB2 system checkpoints was changed from every 5 minutes to every 12 minutes. This reduced CPU usage somewhat, as DB2 did fewer checkpoints. However, this had the effect of increasing recovery time in the event of a system outage.

SUMMARY

- The term "pushdown" refers to the fact that the stack can be regarded as being "pushed down" like a tray dispenser at a cafeteria, since the operations never work on elements other than the top element. A stack automaton, by contrast, does allow access to and operations on deeper elements.
- A pushdown automaton (PDA) is a finite automaton equipped with a stack-based memory.
- A pushdown automaton is a way to implement a context-free grammar in a similar way we design DFA for a regular grammar.
- A pushdown automaton is a way to implement a context-free grammar in a similar way we design DFA for a regular grammar. A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.
- Parsing is used to derive a string using the production rules of a grammar. It is used to check the acceptability of a string. Compiler is used to check whether or not a string is syntactically correct.
- A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information a string derived from a context-free grammar.

KNOWLEDGE CHECK

1. **PDA is more powerful than**
   a. Turing machine
   b. Finite automata
   c. Both (a) and (b)
   d. None of these
2. **Which operation can be applied on stack?**
   a. PUSH
   b. POP
   c. Both (a) and (b)
   d. None of these
3. **PDA can be represented with the help of**
   a. Instantaneous description
   b. Transition diagram
   c. Transition table
   d. All of these
4. **A push down automata is different than finite automata by**
   a. Its memory (stack)
   b. Number of states
   c. Both (a) and (b)
   d. None of these
5. **The push down automata indicate the acceptance of input string in terms of**
   a. Finial state
   b. Empty store
   c. Both (a) and (b)
   d. None of these
6. **The transition a Push down automaton makes is additionally dependent upon the:**
   a. stack
   b. input tape
   c. terminals
   d. none of the mentioned

7. **A PDA machine configuration (p, w, y) can be correctly represented as:**
   a. (current state, unprocessed input, stack content)
   b. (unprocessed input, stack content, current state)
   c. (current state, stack content, unprocessed input)
   d. none of the mentioned
8. **A DPDA is a PDA in which:**
   a. No state p has two outgoing transitions
   b. More than one state can have two or more outgoing transitions
   c. Atleast one state has more than one transitions
   d. None of the mentioned
9. **A language accepted by Deterministic Push down automata is closed under which of the following?**
   a. Complement
   b. Union
   c. All of the mentioned
   d. None of the mentioned
10. **Which of the following correctly recognize the symbol '|-' in context to PDA?**
   a. Moves
   b. transition function
   c. or/not symbol
   d. none of the mentioned

REVIEW QUESTIONS

1. What do you understand by pushdown automata?
2. Describe the basic structure of PDA.
3. Explain the context-free grammar.
4. Discuss about the CFG simplification.
5. Write the short note on "Parsing".

Check Your Result

1. (b)	2. (c)	3. (d)	4. (a)	5. (c)
6. (a)	7. (a)	8. (a)	9. (a)	10. (a)

REFERENCES

1. Aizikowitz, Tamar; Kaminski, Michael (2011). "LR(0) Conjunctive Grammars and Deterministic Synchronized Alternating Pushdown Automata". Computer Science – Theory and Applications. Lecture Notes in Computer Science. Vol. 6651. pp. 345–358.
2. Hromkovic, J., Schnitger, G.: On probabilistic pushdown automata. Information and Computation 208(8), 982–995 (2010).
3. Jacques Chabin and Pierre Réty. "Visibly Pushdown Languages and Term Rewriting". In: Frontiers of Combining Systems, 6th International Symposium, FroCoS 2007, Liverpool, UK, September 10-12, 2007, Proceedings. Ed. by Boris Konev and Frank Wolter. Vol. 4720. Lecture Notes in Computer Science. Springer, 2007, pp. 252–266.
4. John E. Hopcroft; Rajeev Motwani; Jeffrey D. Ullman (2003). Introduction to Automata Theory, Languages, and Computation. Addison Wesley. Here: Sect.6.4.3, p.249.
5. Maria Paola Cabasino, Alessandro Giua, and Carla Seatzu. "Diagnosability of DiscreteEvent Systems Using Labeled Petri Nets". In: IEEE Transactions on Automation Science and Engineering 11.1 (2014), pp. 144–153.
6. Maria Paola Cabasino, Alessandro Giua, Laura Marcias, and Carla Seatzu. "A comparison among tools for the diagnosability of discrete event systems". In: 2012 IEEE International Conference on Automation Science and Engineering, CASE 2012, Seoul, Korea (South), August 20-24, 2012. IEEE, 2012, pp. 218–223.
7. Wujie Chao, Yongmei Gan, Zhaoan Wang, and Walter Murray Wonham. "Modular supervisory control and coordination of state tree structures". In: International Journal of Control 86.1 (2013), pp. 9–21.
8. Yakaryılmaz, A., Cem Say, A.C.: Succinctness of two-way probabilistic and quantum finite automata. DMTCS 12(4), 19–40 (2010).
9. Yakaryılmaz, A., Cem Say, A.C.: Unbounded-error quantum computation with small space bounds. Technical Report arXiv:1007.3624 (2010).

CHAPTER

4

TURING MACHINES

Learning Objectives

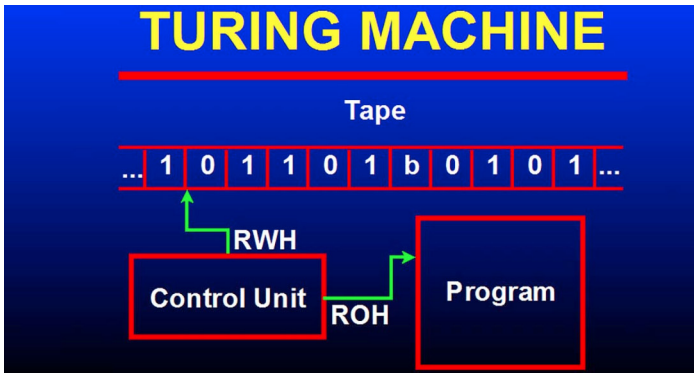
After studying this chapter, you will be able to:

1. Describe the Turing machines: basic
2. Explain the computing functions

INTRODUCTION

A Turing machine consists of an infinite tape (as the memory), a tape head (a pointer to the currently inspected cell of memory), and a state transition table (to govern the behavior of the machine). Each cell of the tape can have one of a predetermined finite set of symbols, one of which is the blank symbol.

A Turing machine, like an algorithm, executes on an input string of bits. At the beginning, the input string is written on the tape, the tape head points to the first cell of the string, and all other cells are blank.

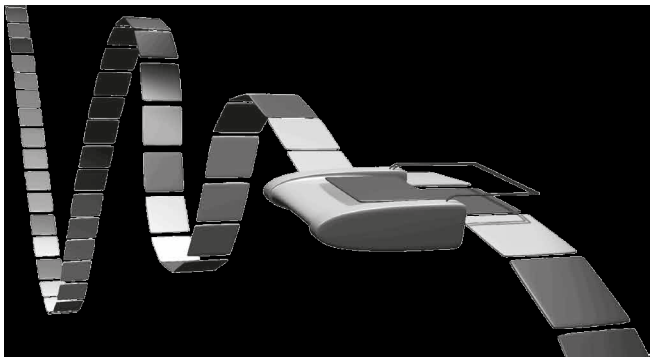


During operation, the tape head is in a certain state. Every step, it consults the table (the set of transition functions), based on only the state it's in and the symbol underneath the head, to obtain its next choice: either halt (end the operation), or resume by writing a symbol to the current cell, changing its state, and moving to the left or the right. This way, A Turing machine can simulate the fact that a program is made of many lines and thus it depends on what line a program is executing, and it can also simulate the fact that a program can react differently with different data in memory.

A Turing machine can thus either halt at some point or run forever. If it halts, the contents of the tape are the output.

4.1 TURING MACHINES: BASIC

A Turing machine is a hypothetical device that manipulates symbols on a strip of tape according to a table of rules. Despite its simplicity, a Turing machine can be adapted to simulate the logic of any computer algorithm, and is particularly useful in explaining the functions of a CPU inside a computer.



A logical device that called "automatic machine" (or "a-machine"), can read, write and erase binary symbols on a tape of unlimited length and divided by squares of equal size. A head read / write would move in any direction along the tape, one

square at a time, and a control unit could play a list of simple instructions, moving to the right or left. The rule executed determines the so-called state of the machine.

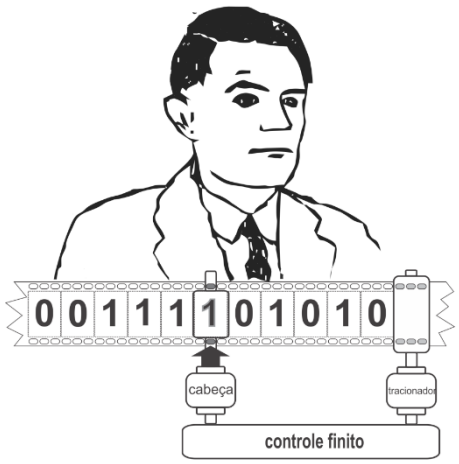


Figure 4.1: Conceptual model of the Turing machine.

Turing machine is similar to a formula or equation. Thus, there is an infinity of possible Turing machines, each corresponding to a defined method or algorithm. Turing proposed that each algorithm, formalized as a finite set of well-defined instructions, could be interpreted and executed by a mechanical process. Formally a Turing machine can be defined as a machine that contains:

- A finite set of states Q with a distinguished initial state,
- A finite set of symbols Σ.

The interpretation and implementation of the algorithms are carried out by states and a transition function determines the new contents of the tape. In this way, a restriction imposed on the algorithm, you can change the contents of only one square at a time or move the head at most one cell in any direction. It also permitted the use of any finite set of symbols for the alphabet Σ, even if the original definition has insisted on Σ = {0,1}. This change has no impact on defining the set of functions computable by the machine.

Turing machine capable of performing a task is the table of transition rules that comprise the machine and a given initial

DID YOU KNOW?

The Turing machine is capable of processing an unrestricted grammar, which further implies that it is capable of robustly evaluating first-order logic in an infinite number of ways.

state. The instruction set known and processed by control module finite, illustrated in figure 4.1, is referenced below:

- In print 0 passing by square head
- In print 1 passing by square head
- Go to square one left
- Go to square one right
- i go to step up the head by passing square contains 0
- Go to step j if the head by passing square box 1
- Stop

For example, a computation can be represented by three states, named s_0, s_1, s_2 and some instructions formalized in algorithm 4.1:

1. $\langle s_0, 1, s_0, \rangle$
2. $\langle s_0, 0, s_1, 1\rangle$
3. $\langle s_1, 1, s_1, \langle$
4. $\langle s_1, 0, s_2, \rangle\rangle$

Algorithm 4.1: Example of instructions between three states s_0, s_1 and s_2.

Keyword

Computer is a device that can be instructed to carry out an arbitrary set of arithmetic or logical operations automatically.

The first two statements (lines 1 and 2) describe what happens in state s_0. There are two possibilities: first, the machine reads a digit '1 ', movimentará head to the right and will remain in state s_0. In the second, if read in a digit '0 'the machine will leave the state s_0, s_1 enter the state and write the digit '1' in this transition. The instructions on lines 3 and 4 show what happens in state s_1, that is, if read in the digit '1 ', the machine movimentará head to the left and remain in state s_1. If you read the digit '0 ', the head will be moved to the right and the machine moves to state s_2. Since there are no instructions defined by the algorithm in state s_2, the machine stops its execution (stop condition) to reach this state. The representation of the machine is effective using a state diagram. In the figure 4.2 represented the set of instructions in this visual format.

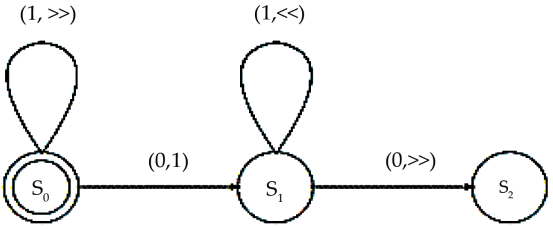
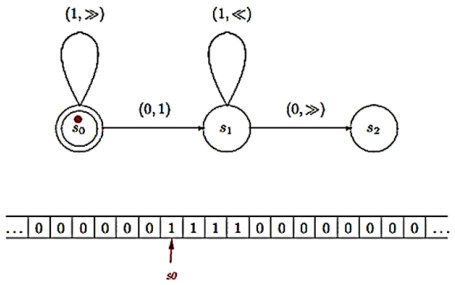


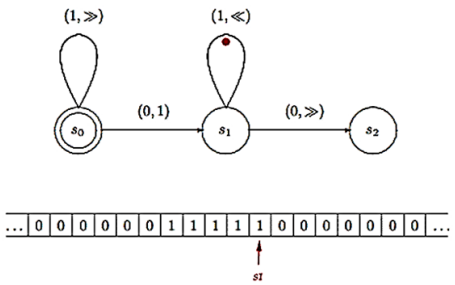
Figure 4.2: A state diagram representing the algorithm 1.

The states are represented by circles, with a double circle identifying the initial state. A transition is represented by an edge or arc from one state to another or to the same state. Edges are labeled by pairs (symbol, action), the first consisting of the symbol should be read and then the action that should be performed with the transition. The symbols belong to the alphabet Σ and the action is the symbol to be written, or even ‹or›, indicating a movement to the left or right.

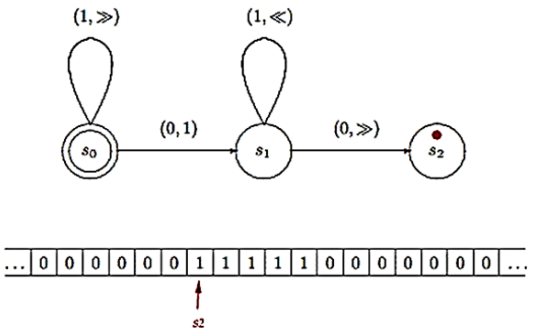
Figure 4.2 illustrates a machine that calculates the successor of n, as a simple execution for purposes merely demonstrative. Thus, the initial state of the ribbon represents the number n after executing the sequence of steps, it will stop the default configuration that represents the number n +1. A possible implementation is demonstrated in figure 4.3 (ac).



(a)



(b)



(c)

Figure 4.3: Sequence implementation of algorithm 1, from a initial state.

Today, it is easy to relate a computer program with a Turing machine and the mechanical task of interpretation and implementation obeying the algorithm. Thus, the Universal Turing machine embodies the essential principle of the **computer**: a simple machine that can perform any task well defined, provided that specified as an appropriate program.

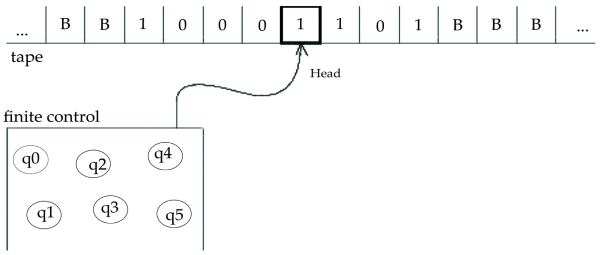
4.1.1 Formal definition of Turing machine

A Turing machine consists of a finite program, called the finite control, capable of manipulating a linear list of *cells*, called the *tape*, using one access pointer, called the *head*. We refer to the two directions on the tape as `right' and `left.' The finite control can be in any one of a finite set of states Q, and each tape cell can contain a 0, a 1, or a `blank' $B$. Time is discrete and the time instants are ordered 0,1,2,…, with 0 the time at which the machine starts its computation. At any time, the head is positioned over a particular cell, which it is said to `scan.' At time 0 the head is situated on a distinguished cell on the tape called the `start cell,' and the finite control is in a distinguished state $q_0$. At time 0 all cells contain $B$'s, except for a contiguous finite sequence of cells, extending from the start cell to the right, which contain 0's and 1's. This binary sequence is called the `input.' The device can perform the following basic operations:

- it can write an element from A={0,1,B} in the cell it scans; and
- it can shift the head one cell left or right.

When the device is active it executes these operations at the rate of one operation per time unit (a `step'). At the conclusion of each step, the finite control takes on a state from Q. The device is constructed so that it behaves according to a finite list of rules. These rules determine, from the current state of the finite control and the

symbol contained in the cell under scan, the operation to be performed next and the state to enter at the end of the next operation execution.



The rules have format (p,s,a,q) : p is the current state of the finite control; s is the symbol under scan; a is the next operation to be executed of type (1) or (2) designated in the obvious sense by an element from $S=\{0,1,B,L,R\}$; and q is the state of the finite control to be entered at the end of this step. For now, we assume that there are no two distinct quadruples that have their first two elements identical, the device is *deterministic*. Not every possible combination of the first two elements has to be in the set; in this way we permit the device to perform `no' operation. In this case we say that the device *halts*. Hence, we can define a Turing machine by a mapping from a finite subset of $Q\times A$ into $S\times Q$. Given a Turing machine and an input, the Turing machine carries out a uniquely determined succession of operations, which may or may not terminate in a finite number of steps.

Keyword

Binary number is a number expressed in the binary numeral system or base-2 numeral system which represents numeric values using two different symbols: typically 0 (zero) and 1 (one).

Strings and natural numbers are occasionally identified according to the pairing

$$(\epsilon,0),(0,1),(1,2),(00,3),(01,4),(10,5),(11,6),\ldots,$$

where ϵ denotes the empty string (with no bits). In the following we need the notion of a ‹self-delimiting› code of a binary string. If $x=x_1\ldots x_n$ is a string of n bits, then its *self-delimiting code* is $\bar{x} = 1^n 0x$. Clearly, the length $|\bar{x}| = 2|x| + 1$. Encoding a **binary** string self-delimitingly enables a machine to determine where the string ends reading it from left to right in a single pass and without reading past the last bit of the code.

Computable functions

We can associate a partial function with each Turing machine in the following way: The input to the Turing machine is presented as an n-tuple $(x_1,\ldots,x_n)$ consisting of self-delimiting versions of the x_i's. The integer represented by the maximal binary string (bordered by blanks) of which some bit is scanned, or 0 if a blank is scanned, by the time the machine halts, is called the `output' of the computation. Under this convention for inputs and outputs, each Turing machine defines a partial function from n-tuples of integers onto the integers, $n \geq 1$. We call such a function partial computable. If the Turing machine halts for all inputs, then the function computed is defined for all arguments and we call it total computable. (Instead of `computable' the more ambiguous `recursive' has also been used.) We call a function with range {0,1} a `predicate', with the interpretation that the predicate of an $n$-tuple of values is `true' if the corresponding function assumes value 1 for that n-tuple of values for its arguments and is `false' or `undefined' otherwise. Hence, we can talk about `partial (total) computable predicates'.

Remember

Universal Turing machine is far from obvious that a universal Turing machine exists. In particular, such a machine must have a finite number of states, yet it must be able to simulate a computation performed by a Turing machine with a much greater number of states.

Computability thesis

The class of algorithmically computable numerical functions (in the intuitive sense) coincides with the class of partial computable functions. Originally intended as a proposal to henceforth supply intuitive terms such as `computable' and `effective procedure' with a precise meaning, the Computability thesis has come into use as shorthand for a claim that from a given description of a procedure in terms of an informal set of instructions we can derive a formal one in terms of Turing machines.

4.1.2 Universal Turing machine

It is possible to give an effective (computable) one-to-one pairing between natural numbers and Turing machines. This is called an *effective enumeration*. One way to do this is to encode the table of rules of each Turing machine in binary, in a canonical way.

The only thing we have to do for every Turing machine is to encode the defining mapping T from $Q \times A$ into $S \times Q$. Giving each element of $Q \cup S$ a unique binary code requires s bits for

each such element, with $s=\lceil \log(|Q|+5) \rceil$. Denote the encoding function by e . Then the quadruple $(p,0,B,q)$ is encoded as $e(p)e(0)e(B)e(q)$. If the number of rules is r , then $r \leq 3|Q|$. We agree to consider the state of the first rule as the start state. The entire list of quadruples,

$$T = (p_1, t_1, s_1, q_1), (p_2, t_2, s_2, q_2), \ldots, (p_r, t_r, s_r, q_r)$$

is encoded as

$$E(T) = \bar{s}\bar{r}e(p_1)e(t_1)e(s_1)e(q_1)\ldots e(p_r)e(t_r)e(s_r)e(q_r).$$

Note that $l(E(T)) \leq 4rs+2\log rs+4$. (Moreover, E is self-delimiting, which is convenient in situations in which we want to recognize the substring $E(T)$ as prefix of a larger string.)

We order the resulting binary strings lexicographically (according to increasing length). We assign an index, or Gödel number, $n(T)$ to each Turing machine T by defining $n(T)=i$ if $E(T)$ is the ith element in the lexicographic order of Turing machine codes. This yields a sequence of Turing machines $T_1, T_2, \ldots$ that constitutes the effective enumeration. One can construct a Turing machine to decide whether a given binary string x encodes a Turing machine, by checking whether it can be decoded according to the scheme above, that the tuple elements belong to $Q \times A \times S \times Q$, followed by a check whether any two different rules start with the same two elements. This observation enables us to construct `universal' Turing machines.

A *universal* Turing machine U is a Turing machine that can imitate the behavior of any other Turing machine T. It is a fundamental result that such machines exist and can be constructed effectively. Only a suitable description of T's finite program and input needs to be entered on U's tape initially. To execute the consecutive actions that T would perform on its own tape, U uses T's description to simulate T's actions on a representation of T's tape contents. Such a machine U is also called `computation universal.' In fact, there are infinitely many such U's.

We focus on a universal Turing machine U that uses the encoding above. It is not difficult, but tedious, to define a Turing machine in quadruple format that expects inputs of the format $E(T)p$ and is undefined for inputs not of that form. The machine U starts to execute the successive operations of T using p as input and the description $E(T)$ of T it has found so that $U(E(T)p)=T(p)$ for every T and p . We omit the explicit construction of U .

For the contemporary reader there should be nothing mysterious in the concept of a general-purpose computer which can perform any computation when supplied with an appropriate program. The surprising thing is that a general-purpose computer can

be very simple: Marvin Minsky has been shown that four tape symbols and seven states suffice easily in the above scheme. This machine can be changed to, in the sense of being simulated by, our format using tape symbols {0,1,*B*} at the cost of an increase in the number of states. The last reference contains an excellent discussion of Turing machines, their computations, and related machines. The effective enumeration of Turing machines $T_1, T_2, \ldots$ determines an effective enumeration of partial computable functions $\phi_1, \phi_2, \ldots$ such that ϕ_i is the function computed by Ti , for all i . It is important to distinguish between a function ψ and a name for ψ . A name for ψ can be an algorithm that computes ψ, in the form of a Turing machine T. It can also be a natural number i such that ψ equals ϕ_i in the above list. We call i an index for ψ. Thus, each partial computable ψ occurs many times in the given effective enumeration, that is, it has many indices.



We must explain how a single Turing machine with bounded alphabet size can read the description of a Turing machine with a much larger alphabet.

Universal partial computable function

The partial computable function $v^{(2)}(i,x)$ computed by the universal Turing machine U is called the *universal partial computable function*. The generalization to n-place functions is straightforward. A partial computable function $v^{(n+1)}(i,x_1,\ldots,x_n)$ is *universal* for all n-place partial computable functions if for each partial computable function $\phi^{(n)}(x_1,\ldots,x_n)$ there exists an i such that the mapping $v^{(n+1)}$) with the first argument fixed to i is identical to the mapping $\phi^{(n)}$. Here i is an index of $\phi^{(n)}$ with respect to $v^{(n+1)}$. For each n , we fix a partial computable $(n+1)$-place function that is universal for all n-place partial computable functions.

Enumeration theorem for n-place partial computable functions

Here z is the index of the universal function. For each n there exists an index z such that for all i and $x_1,\ldots,x_n$, if $\phi_i^{(n)}(x_1,\ldots,x_n)$ is defined, then $\phi_z^{(n+1)}(i,x_1,\ldots,x_n) = \phi_i^{(n)}(x_1,\ldots,x_n)$, and

$\phi_z^{(n+1)}(i, x_1, \ldots, x_n)$ is undefined otherwise. ($\phi_z^{(n+1)}$ is a universal partial computable function that enumerates the partial computable functions of n variables.)

Computably enumerable sets

A set A is computably enumerable if it is empty or the range of some total computable function f .We say that f `enumerates' A . The intuition behind this definition is that there is a Turing machine for listing the elements of A in some arbitrary order with repetitions allowed. An equivalent definition is that A is computably enumerable if it is accepted by a Turing machine. That is, for each element in A , the Turing machine halts in a distinguished accepting state, and for each element not in A the machine either halts in a no accepting state or computes forever.

A set A is *computable* if it possesses a computable characteristic function. That is, A is computable iff there exists a computable function f such that for all x , if $x \in A$, then $f(x)=1$, and if $x \in \bar{A}$, then $f(x)=0$ ($\bar{A}$ is the complement of A). An equivalent definition is that A is computable if A is accepted by a Turing machine that always halts. Obviously, all computable sets are computably enumerable.

Remember

Note that the limitation to variables with a predefined finite set of values is forced on us by the requirement that the Turing machine should have a finite set of states.

The following sets are computable:

- the set of odd integers;
- the set of natural numbers;
- the empty set;
- the set of primes;
- every finite set;
- every set with a finite complement.

The following sets are computably enumerable:

- every computable set;
- the set of indices i such that the range of ϕ_i is nonempty;
- the set $\{x$: a run of at least x consecutive 0's occurs in $\pi\}$, where π=3.1415….

Theorem

- A set A is computable iff both A and its complement A^- are computably enumerable.
- An infinite set A is computable iff it is computably enumerable in increasing order. (Here we have postulated a total order on the elements of A . For instance, if A (a subset of the natural numbers) with the usual order, then ϕ enumerates A in increasing order if $\phi(i) < \phi(i+1)$, for all i .)
- Every infinite computably enumerable set contains an infinite computable subset.

The equivalent statements hold for computable and computably enumerable sets of n-tuples.

Remember

To define a universal Turing machine, we must first explain what it means to give a "description" of one Turing machine as the input to another one.

4.1.3 Undecidability of the Halting Problem

Turing's paper, and more so Kurt Gödel's paper, where such a result first appeared, are celebrated for showing that certain well-defined questions in the mathematical domain cannot be settled by any effective procedure for answering questions. The following 'machine form' of this undecidability result is due to Turing and Church: `which machine computations eventually terminate with a definite result, and which machine computations go on forever without a definite conclusion?' This is sometimes called the *halting problem*.

Since all machines can be simulated by the universal Turing machine U , this question cannot be decided in the case of the single machine U , or more generally for any other individual universal machine. The following theorem due to Turing in 1936, formalizes this discussion. Let ϕ_1, ϕ_2,... be the standard enumeration of partial computable functions and write $\phi(x)<\infty$ if $\phi(x)$ is defined and write $\phi(x)=\infty$otherwise. Define $K_0 = \{\langle x, y\rangle : \phi_x(y) < \infty\}$ as the *halting set*.

Theorem. The halting set K_0 is not computable.

The trick used in the proof is called *diagonalization* to show that there is no computable function g such that for all x,y , we have $g(x,y)=1$ if $\phi_x(y)$ is defined, and $g(x,y)=0$ otherwise.

It is easy to see that K_0 is computably enumerable. The halting set is so ubiquitous that it merits the standard notation K_0 . We shall also use the *diagonal halting set* $K = \{x : \phi_x(x) < \infty\}$. Just like K_0 , the diagonal halting set is computably enumerable; and K is also not a computable set.

The theorem of Turing on the in computability of the halting set was preceded by (and was intended as an alternative way to show) the famous (first) incompleteness theorem of Kurt Gödel in 1931. Recall that a formal theory T consists of a set of well-formed formulas, formulas for short. For convenience these formulas are taken to be finite binary strings. Invariably, the formulas are specified in such a way that an effective procedure exists that decides which strings are formulas and which strings are not.

Remember

A variable with an unbounded number of possible values cannot be represented using a state set of some predefined, finite size.

The formulas are the objects of interest of the theory and constitute the meaningful statements. With each theory we associate a set of *true* formulas and a set of *provable* formulas. The set of true formulas is `true' according to some (often nonconstructive) criterion of truth. The set of provable formulas is `provable' according to some (usually effective) syntactic notion of proof.

A theory T is simply any set of formulas. A theory is *axiomatizable* if it can be effectively enumerated. For instance, its axioms (initial formulas) can be effectively enumerated and there is an effective procedure that enumerates all proofs for formulas in T from the axioms. A theory is *decidable* if it is a recursive set. A theory T is *consistent* if not both formula x and and its negation $\neg x$ are in T . A theory T is *sound* if each formula x in T is true (with respect to the standard model of the natural numbers).

Hence, soundness implies consistency. A particularly important example of an axiomatizable theory is *Peano arithmetic*, which axiomatizes the standard elementary theory of the natural numbers.

Theorem: There is a computably enumerable set, say the set K_0 defined above, such that for every axiomatizable theory T that is sound and extends Peano arithmetic, there is a number n such that the formula `$n \notin K_0$' is true but not provable in T .

In his original proof, Gödel uses diagonalization to prove the incompleteness of any sufficiently rich logical theory T with a computably enumerable axiom system, such as Peano arithmetic. By his technique he exhibits for such a theory an explicit construction of an undecidable statement y that says of itself `I am unprovable in T .' The formulation in terms of computable function theory is due to A. Church and S.C. Kleene.

Turing's idea was to give a formal meaning to the notion of `giving a proof.' Intuitively, a proof is a sort of computation where every step follows (and follows logically) from the previous one, starting from the input. To put everything as broad as possible, Turing analyses the notion of `computation' from an `input' to an `output' and uses this to give an alternative proof of Gödel's theorem.

Semicomputable functions

The notion of computable functions can be extended from integer functions to real valued functions of rational arguments, and to weaker types of computability, using the framework explained above. We consider partial computable functions $g(\langle\langle y, z\rangle, k\rangle) = \langle p, q\rangle$ and write $g(y/z, k) = p/q$, with y,z,p,q,k nonnegative integers. The extension to negative arguments and values is straightforward. The interpretation is that g is a rational-valued function of a rational argument and a nonnegative integer argument.

A partial function f from the rational numbers to the real numbers is *upper semicomputable* if it is defined by a rational-valued partial computable function $\phi(x,k)$ with x a rational number and k a nonnegative integer such that $\phi(x,k+1) \leq \phi(x,k)$ for every k and $\lim_{k\to\infty}\phi(x,k) = f(x)$. This means that f can be computably approximated from above. A function f is *lower semicomputable* if $-f$ is upper semicomputable. A function is called *semicomputable* if it is either upper semicomputable or lower semicomputable or both. If a function f is both upper semicomputable and lower semicomputable on its domain, then we call f *computable* (equivalent to computable if the domain is integer or rational). The total function versions are defined similarly.

Thus, a total real function f is computable iff there is a total computable function $g(x,k)$such that $|f(x)-g(x,k)|<1/k$. In this way, we extend the notion of integer computable functions to real-valued computable functions with rational arguments, and to real-valued semicomputable functions with rational arguments. The idea is that a semicomputable function can be approximated from one side by a computable function with rational values, but we may never know how close we are to the real value. A computable real function can be approximated to any degree of precision by a computable function with rational values. A function f is lower semicomputable iff the set $\{(x,r): r \leq f(x), r \text{ rational}\}$ is computably enumerable.

4.2 COMPUTING FUNCTIONS

Turing machines can also be used to compute functions from strings to strings, instead of just accepting or rejecting strings. Since we do not care about acceptance or rejection, we replace the explicit accept and reject states with a single halt state, and we define the output of the Turing machine to be the contents of the tape when the machine halts, after removing the infinite sequence of trailing blanks. More formally, for any Turing machine M, any string w ∈ Σ^*, and any string x ∈ Γ^* that does not end with a blank, we write M(w) = x if and only if $(w,s,0) \Rightarrow_M^* (x, \text{halt}, i)$ for some integer i. If M does not halt on input w, then we write $M(w)\nearrow$, which can be read either "M diverges on w" or "M(w) is undefined." We say that M computes the function f: $\Sigma^* \rightarrow \Sigma^*$ if and only if M(w) = f (w) for every string w.

Keyword

Turing machine is an abstract machine that manipulates symbols on a strip of tape according to a table of rules; to be more exact, it is a mathematical model of computation that defines such a device

4.2.1 Shifting

One basic operation that is used in many Turing machine constructions is shifting the input string a constant number of steps to the right or to the left.

For example, given any input string $w \in \{0,1\}^*$, we can compute the string $0w$ using a Turing machine with tape alphabet $\Gamma = \{0,1,\square\}$, state set Q = {0, 1,halt}, start state 0, and the following transition function:

$$\begin{aligned}
\delta(p,\ a) &= (\ q\ ,\ b,\ \Delta\ ) \\
\hline
\delta(0,\ 0) &= (\ 0\ ,\ 0,\ +1) \\
\delta(0,\ 1) &= (\ 1\ ,\ 0,\ +1) \\
\delta(0,\ \square) &= (\text{halt},\ 0,\ +1) \\
\hdashline
\delta(1,\ 0) &= (\ 0\ ,\ 1,\ +1) \\
\delta(1,\ 1) &= (\ 1\ ,\ 1,\ +1) \\
\delta(1,\ \square) &= (\text{halt},\ 1,\ +1)
\end{aligned}$$

By increasing the number of states, we can build a **Turing machine** that shifts the input string any fixed number of steps in either direction.

For example, a machine that shifts its input to the left by five steps might read the string from right to left, storing the five most recently read symbols in its internal state. A typical transition for such a machine would be δ(12345, 0) = (01234, 5,−1).

4.2.2 Binary Addition

With a more complex Turing machine, we can implement binary addition. The input is a string of the form w+x, where $w, x \in \{0, 1\}^n$, representing two numbers in binary; the output is the binary representation of w+ x. To simplify our presentation, we assume that |w| = |x| > 0; however, this restrictions can be removed with the addition of a few more states. The following figure shows the entire Turing machine at a glance. The machine uses the tape alphabet $\Gamma = \{\square, 0, 1, +, \underline{0}, \underline{1}\}$; the start state is shift0. All missing transitions go to a fail state, indicating that the input was badly formed.

Execution of this Turing machine proceeds in several phases, each with its own subset of states, as indicated in the figure. The initialization phase scans the entire input, shifting it to the right to make room for the output string, marking the rightmost bit of w, and reading and erasing the last bit of x.

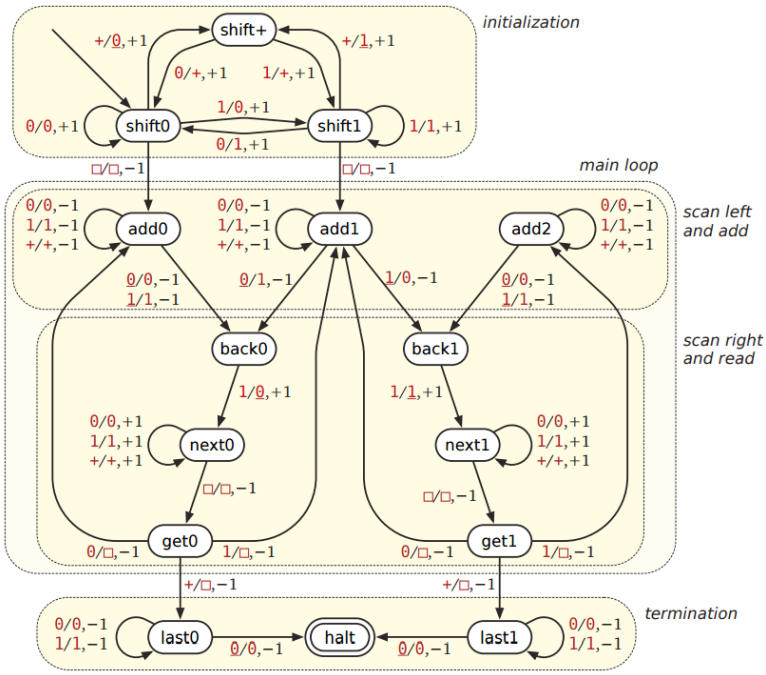


Figure 4.4: A Turing machine that adds two binary numbers of the same length.

$$
\begin{array}{l}
\delta(\ p\ ,\ a) = (\ q\ ,\ b,\ \Delta) \\
\hline
\delta(\text{shift0},\ 0) = (\text{shift0},\ 0,\ +1) \\
\delta(\text{shift0},\ 1) = (\text{shift1},\ 0,\ +1) \\
\delta(\text{shift0},\ +) = (\text{shift+},\ \underline{0},\ +1) \\
\delta(\text{shift0},\ \square) = (\text{add0},\ \square,\ -1) \\
\hdashline
\delta(\text{shift1},\ 0) = (\text{shift0},\ 1,\ +1) \\
\delta(\text{shift1},\ 1) = (\text{shift1},\ 1,\ +1) \\
\delta(\text{shift1},\ +) = (\text{shift+},\ \underline{1},\ +1) \\
\delta(\text{shift1},\ \square) = (\text{add1},\ \square,\ -1) \\
\hdashline
\delta(\text{shift+},\ 0) = (\text{shift0},\ +,\ +1) \\
\delta(\text{shift+},\ 1) = (\text{shift1},\ +,\ +1)
\end{array}
$$

The first part of the main loop scans left to the marked bit of w, adds the bit of x that was just erased plus the carry bit from the previous iteration, and records the carry bit for the next iteration in the machines internal state.

$$
\begin{array}{l}
\delta(\ p\ ,\ a) = (\ q\ ,\ b,\ \Delta) \\
\hline
\delta(\text{add0},\ 0) = (\text{add0},\ 0,\ -1) \\
\delta(\text{add0},\ 1) = (\text{add0},\ 0,\ -1) \\
\delta(\text{add0},\ +) = (\text{add0},\ 0,\ -1) \\
\delta(\text{add0},\ \underline{0}) = (\text{back0},\ 0,\ -1) \\
\delta(\text{add0},\ \underline{1}) = (\text{back0},\ 1,\ -1)
\end{array}
\qquad
\begin{array}{l}
\delta(\ p\ ,\ a) = (\ q\ ,\ b,\ \Delta) \\
\hline
\delta(\text{add1},\ 0) = (\text{add1},\ 0,\ -1) \\
\delta(\text{add1},\ 1) = (\text{add1},\ 0,\ -1) \\
\delta(\text{add1},\ +) = (\text{add1},\ 0,\ -1) \\
\delta(\text{add1},\ \underline{0}) = (\text{back0},\ 1,\ -1) \\
\delta(\text{add1},\ \underline{1}) = (\text{back1},\ 0,\ -1)
\end{array}
\qquad
\begin{array}{l}
\delta(\ p\ ,\ a) = (\ q\ ,\ b,\ \Delta) \\
\hline
\delta(\text{add2},\ 0) = (\text{add2},\ 0,\ -1) \\
\delta(\text{add2},\ 1) = (\text{add2},\ 0,\ -1) \\
\delta(\text{add2},\ +) = (\text{add2},\ 0,\ -1) \\
\delta(\text{add2},\ \underline{0}) = (\text{back1},\ 0,\ -1) \\
\delta(\text{add2},\ \underline{1}) = (\text{back1},\ 1,\ -1)
\end{array}
$$

The second part of the main loop marks the previous bit of w, scans right to the end of x, and then reads and erases the last bit of x, all while maintaining the carry bit.

$$
\begin{array}{l}
\delta(\ p\ ,\ a) = (\ q\ ,\ b,\ \Delta) \\
\hline
\delta(\text{back0},\ 0) = (\text{next0},\ \underline{0},\ +1) \\
\delta(\text{back0},\ 1) = (\text{next0},\ \underline{1},\ +1) \\
\hdashline
\delta(\text{next0},\ 0) = (\text{next0},\ 0,\ +1) \\
\delta(\text{next0},\ 1) = (\text{next0},\ 0,\ +1) \\
\delta(\text{next0},\ +) = (\text{next0},\ 0,\ +1) \\
\delta(\text{next0},\ \square) = (\text{get0},\ \square,\ -1) \\
\hdashline
\delta(\text{get0},\ 0) = (\text{add0},\ \square,\ -1) \\
\delta(\text{get0},\ 1) = (\text{add1},\ \square,\ -1) \\
\delta(\text{get0},\ +) = (\text{last0},\ \square,\ -1)
\end{array}
\qquad
\begin{array}{l}
\delta(\ p\ ,\ a) = (\ q\ ,\ b,\ \Delta) \\
\hline
\delta(\text{back1},\ 0) = (\text{next1},\ \underline{0},\ +1) \\
\delta(\text{back1},\ 1) = (\text{next1},\ \underline{1},\ +1) \\
\hdashline
\delta(\text{next1},\ 0) = (\text{next1},\ 0,\ +1) \\
\delta(\text{next1},\ 1) = (\text{next1},\ 0,\ +1) \\
\delta(\text{next1},\ +) = (\text{next1},\ 0,\ +1) \\
\delta(\text{next1},\ \square) = (\text{get1},\ \square,\ -1) \\
\hdashline
\delta(\text{get1},\ 0) = (\text{add1},\ \square,\ -1) \\
\delta(\text{get1},\ 1) = (\text{add2},\ \square,\ -1) \\
\delta(\text{get1},\ +) = (\text{last1},\ \square,\ -1)
\end{array}
$$

Finally, after erasing the + in the last iteration of the main loop, the termination phase adds the last carry bit to the leftmost output bit and halts.

$$\begin{array}{l}\delta(\ p\ ,\ a) = (\ q\ ,\ b,\ \Delta\ ) \\ \hline \delta(\text{last0},\ 0) = (\text{last0},\ 0,\ -1) \\ \delta(\text{last0},\ 1) = (\text{last0},\ 0,\ -1) \\ \delta(\text{last0},\ \underline{0}) = (\ \text{halt}\ ,\ 0,\ \ \ ) \\ \hdashline \delta(\text{last1},\ 0) = (\text{last1},\ 0,\ -1) \\ \delta(\text{last1},\ 1) = (\text{last1},\ 0,\ -1) \\ \delta(\text{last1},\ \underline{0}) = (\ \text{halt}\ ,\ 1,\ \ \ )\end{array}$$

Keyword

Symbol is a mark, sign, or word that indicates, signifies, or is understood as representing an idea, object, or relationship.

4.2.3 Variations

There are actually several formal models that all fall under the name "Turing machine", each with small variations on the definition we have given. Although we do need to be explicit about which variant we want to use for any particular problem, the differences between the variants are relatively unimportant. For any machine defined in one model, there is an equivalent machine in each of the other models; in particular, all of these variants recognize the same languages and decide the same languages.

Halting conditions

Some models allow multiple accept and reject states, which (depending on the precise model) trigger acceptance or rejection either when the machine enters the state, or when the machine has no valid transitions out of such a state. Others include only explicit accept states, and either equate crashing with rejection or do not define a rejection mechanism at all. Still other models include halting as one of the possible actions of the machine, in addition to moving left or moving right; in these models, the machine accepts/rejects its input if and only if it halts in an accepting/non-accepting state.

Actions

Some Turing machine models allow transitions that do not move the head, or that move the head by more than one cell in a single step. Others insist that a single step of the machine either writes a new **symbol** onto the tape or moves the head

one step. Finally, as mentioned above, some models include halting as one of the available actions.

Transition function

Some models of Turing machines, including Turing's original definition, allow the transition function to be undefined on some state-symbol pairs. In this formulation, the transition function is given by a set $\delta \subset Q \times \Gamma \times Q \times \Gamma \times \{+1,-1\}$, such that for each state q and symbol a, there is at most one transition $(q, a, \cdot, \cdot, \cdot) \in \delta$. If the machine enters a configuration from which there is no transition, it halts and (depending on the precise model) either crashes or rejects. Others define the transition function as $\delta: Q \times \Gamma \to Q \times (\Gamma \cup \{-1,+1\})$, allowing the machine to either write a symbol to the tape or move the head in each step.

Beginning of the tape

Some models forbid the head to move past the beginning of the tape, either by starting the tape with a special symbol that cannot be overwritten and that forces a rightward transition, or by declaring that a leftward transition at position 0 leaves the head in position 0, or even by pure fiat—declaring any machine that performs a leftward move at position 0 to be invalid.

To prove that any two of these variant "species" of Turing machine are equivalent, we must show how to transform a machine of one species into a machine of the other species that accepts and rejects the same strings.

For example, let $M = (\Gamma, \square, \Sigma, Q, s, \text{accept}, \text{reject}, \delta)$ be a Turing machine with explicit accept and reject states. We can define an equivalent Turing machine M' that halts only when it moves left from position 0, and accepts only by halting while in an accepting state, as follows. We define the set of accepting states for M' as A = {accept} and define a new transition function

$$\delta'(q,a) := \begin{cases} (\text{accept}, a, -1) & \text{if } q = \text{accept} \\ (\text{reject}, a, -1) & \text{if } q = \text{reject} \\ \delta(q,a) & \text{otherwise} \end{cases}$$

Similarly, suppose someone gives us a Turing machine $M = (\Gamma, \square, \Sigma, Q, s, \text{accept}, \text{reject}, \delta)$ whose transition function $\delta: Q\times\Gamma \to Q\times\Gamma \times\{-1, 0,+1\}$ allows the machine to transition without moving its head. We can construct an equivalent Turing machine $M' = (\Gamma, \square, \Sigma, Q', s, \text{accept}, \text{reject}, \delta')$ that moves its head at every transition by defining

$Q' := Q \times \{0,1\}$ and

$$\delta'((p,0),a) := \begin{cases} ((q,1),b,+1) & \text{if } \delta(p,a) = (q,b,0), \\ ((q,0),b,\Delta) & \text{if } \delta(p,a) = (q,b,\Delta) \text{ and } \Delta \neq 0, \end{cases}$$
$$\delta'((p,1),a) := ((p,0),a,-1).$$

Variations on Tracks, Heads, and Tapes

There are some variations on tracks, heads, and tapes, such as:

Multiple Tracks

It is sometimes convenient to endow the Turing machine tape with multiple tracks, each with its own tape alphabet, and allow the machine to read from and write to the same position on all tracks simultaneously.

For example, to define a Turing machine with three tracks, we need three tape alphabets Γ_1 , Γ_2 , and Γ_3 , each with its own blank symbol, where (say) Γ_1 contains the input alphabet Σ as a subset; we also need a transition function of the form

$$\delta : Q \times \Gamma_1 \times \Gamma_2 \times \Gamma_3 \to Q \times \Gamma_1 \times \Gamma_2 \times \Gamma_3 \times \{-1,+1\}$$

Describing a configuration of this machine requires a quintuple (q, x_1 , x_2 , x_3 , i), indicating that each track i contains the string xi followed by an infinite sequence of blanks. The initial configuration is (start, w, ε, ε, 0), with the input string written on the first track, and the other two tracks completely blank.

But any such machine is equivalent (if not identical) to a single-track Turing machine with the (still finite!) tape alphabet $\Gamma := \Gamma_1 \times \Gamma_2 \times \Gamma_3$. Instead of thinking of the tape as three infinite sequences of symbols, we think of it as a single infinite sequence of "records", each containing three symbols. Moreover, there's nothing special about the number three in this construction; a Turing machine with any constant number of tracks is equivalent to a single-track machine.

Remember

A variable with an unbounded number of possible values cannot be represented using a state set of some predefined, finite size. More subtly, a recursive algorithm must maintain a stack of program counters; if this stack can potentially grow to unbounded depth, then once again it is impossible to represent it using a finite state set.

Doubly-Infinite Tape

It is also sometimes convenient to allow the tape to be infinite in both directions, for example, to avoid boundary conditions. There are several ways to simulate a doubly-infinite tape on a machine with only a semi-infinite tape. Perhaps the simplest method is to use a semi-infinite tape with two tracks, one containing the cells with positive index and the other containing the cells with negative index in reverse order, with a special marker symbol at position zero to indicate the transition.

0	+1	+2	+3	+4	⋯
▶	−1	−2	−3	−4	⋯

Another method is to shuffle the positive-index and negative-index cells onto a single track, and add additional states to allow the Turing machine to move two steps in a single transition. Again, we need a special symbol at the left end of the tape to indicate the transition:

▶	0	−1	+1	−2	+2	−3	+3	⋯

A third method maintains two sentinel symbols ▶ and ▶ that surround all other non-blank symbols on the tape. Whenever the machine reads the right sentinel ▶ we write a blank, move right, write ▶, move left, and then proceed as if we had just read a blank. On the other hand, when the machine reads the left sentinel ▶, we shift the entire contents of the tape (up to and including the right sentinel) one step to the right, then move back to the left sentinel, move right, write a blank, and finally proceed as if we had just read a blank. Since the Turing machine does not actually have access to the position of the head as an integer, shifting the head and the tape contents one step right has no effect on its future evolution.

▶	−3	−2	−1	0	+1	+2	+3	+4	+5	◀	⋯

Using either of the first two methods, we can simulate t steps of an arbitrary Turing machine with a doubly-infinite tape using only O(t) steps on a standard Turing machine. The third method, unfortunately, requires $\Theta(t^2)$ steps in the worst case.

Insertion and Deletion

We can also allow Turing machines to insert and delete cells on the tape, in addition to simply overwriting existing symbols. We have already seen how to insert a new cell: Leave a special mark on the tape (perhaps in a second track), shift everything to the right of this mark one cell to the right, scan left to the mark, erase the mark, and finally write the correct character into the new cell. Deletion is similar: Mark

the cell to be deleted, shift everything to the right of the mark one step to the left, scan left to the mark, and erase the mark. We may also need to maintain a mark in some cell to the right every non-blank symbol, indicating that all cells further to the right are blank, so that we know when to stop shifting left or right.

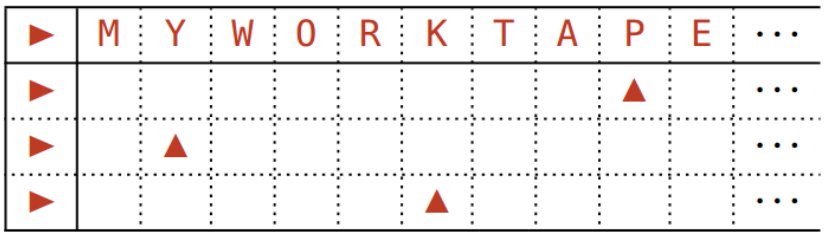
Multiple Heads

Another convenient extension is to allow machines simultaneous access to more than one position on the tape. *For example*, to define a Turing machine with three heads, we need a transition function of the form

$$\delta : Q \times \Gamma^3 \rightarrow Q \times \Gamma^3 \times \{-1, +1\}^3.$$

Describing a configuration of such a machine requires a quintuple (q, x, i, j, k), indicating that the machine is in state q, the tape contains string x, and the three heads are at positions i, j, k. The transition function tells us, given q and the three symbols x[i], x[j], x[k], which three symbols to write on the tape and which direction to move each of the heads.

We can simulate this behavior with a single head by adding additional tracks to the tape that record the positions of each head. To simulate a machine M with three heads, we use a tape with four tracks: track 0 is the actual work tape; each of the remaining tracks has a single non-blank symbol recording the position of one of the heads. We also insert a special marker symbols at the left end of the tape.



We can simulate any single transition of M, starting with our single head at the left end of the tape, as follows. Throughout the simulation, we maintain the internal state of M as one of the components of our current state. First, for each i, we read the symbol under the ith head of M as follows:

Scan to the right to find the mark on track i, read the corresponding symbol from track 0 into our internal state, and then return to the left end of the tape.

At this point, our internal state records M's current internal state and the three symbols under M's heads. After one more transition (using M's transition function),

our internal state records M's next state, the symbol to be written by each head, and the direction to move each head. Then, for each i, we write with and move the ith head of M as follows:

Scan to the right to find the mark on track i, write the correct symbol onto on track 0, move the mark on track i one step left or right, and then return to the left end of the tape.

Again, there is nothing special about the number three here; we can simulate machines with any fixed number of heads.

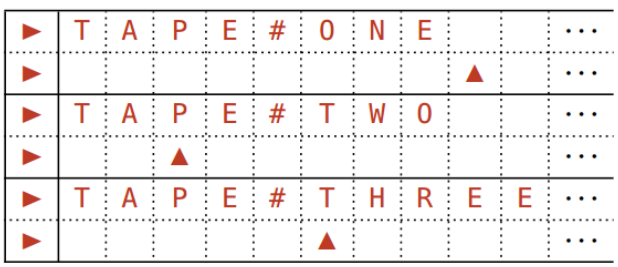
Careful analysis of this technique implies that for any integer k, we can simulate t steps of an arbitrary Turing machine with k independent heads in $\Theta(t^2)$ time on a standard Turing machine with only one head. Unfortunately, this quadratic blowup is unavoidable. It is relatively easy to recognize the language of marked palindromes $\{w \bullet w^R \mid w \in \{0,1\}^*\}$ in O(n) time using a Turing machine with two heads, but recognizing this language provably requires $\Omega(n^2)$ time on a standard machine with only one head. On the other hand, with much more sophisticated techniques, it is possible to simulate t steps of a Turing machine with k head, for any fixed **integer** k, using only O(t log t) steps on a Turing machine with just two heads.

Keyword

Integer is a number that can be written without a fractional component.

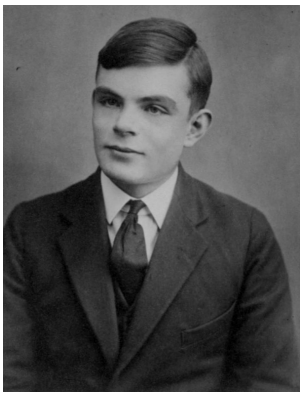
Multiple Tapes

We can also allow machines with multiple independent tapes, each with its own head. To simulate such a machine with a single tape, we simply maintain each tape as an independent track with its own head. Equivalently, we can simulate a machine with k tapes using a single tape with 2k tracks, half storing the contents of the k tapes and half storing the positions of the k heads.



Just as for multiple tracks, for any constant k, we can simulate t steps of an arbitrary Turing machine with k independent tapes in $\Theta(t^2)$ steps on a standard Turing machine with one tape, and this quadratic blowup is unavoidable. Moreover, it is possible to simulate t steps on a k-tape Turing machine using only O(t log t) steps on a two-tape Turing machine using more sophisticated techniques. (This faster simulation is easier to obtain for multiple independent tapes than for multiple heads on the same tape.) By combining these tricks, we can simulate a Turing machine with any fixed number of tapes, each of which may be infinite in one or both directions, each with any fixed number of heads and any fixed number of tracks, with at most a quadratic blowup in the running time.

ROLE MODEL



ALAN TURING: ENGLISH COMPUTER SCIENTIST, MATHEMATICIAN

Alan Turing was born on June 23, 1912, in London. In his seminal 1936 paper, he proved that there cannot exist any universal algorithmic method of determining truth in mathematics, and that mathematics will always contain undecidable propositions. That paper also introduced the "Turing machine. His papers on the subject are widely acknowledged as the foundation of research in artificial intelligence.

Early Life

English scientist Alan Turing was born Alan Mathison Turing on June 23, 1912, in Maida Vale, London, England. At a young age, he displayed signs of high intelligence, which some of his teachers recognized, but did not necessarily respect. When Turing attended the well-known independent Sherborne School at the age of 13, he became particularly interested in math and science.

After Sherborne, Turing enrolled at King's College (University of Cambridge) in Cambridge, England, studying there from 1931 to 1934. As a result of his dissertation, in which he proved the central limit theorem, Turing was elected a fellow at the school upon his graduation.

In 1936, Turing delivered a paper, "On Computable Numbers, with an Application to the Entscheidungsproblem," in which he presented the notion of a universal machine (later called the "Universal Turing Machine," and then the "Turing machine") capable of computing anything that is computable: The central concept of the modern computer was based on Turing's paper.

Over the next two years, Turing studied mathematics and cryptology at the Institute for Advanced Study in Princeton, New Jersey. After receiving his Ph.D. from Princeton University in 1938, he returned to Cambridge, and then took a part-time position with the Government Code and Cypher School, a British code-breaking organization.

Cryptanalysis and Early Computers

During World War II, Turing was a leading participant in wartime code-breaking particularly that of German ciphers. He worked at Bletchley Park, the GCCS wartime

station, where he made five major advances in the field of cryptanalysis, including specifying the bombe, an electromechanical device used to help decipher German Enigma encrypted signals. Turing's contributions to the code-breaking process did not stop there: He also wrote two papers about mathematical approaches to code-breaking, which became such important assets to the Code and Cypher School (later known as the Government Communications Headquarters) that the GCHQ waited until April 2012 to release them to the National Archives of the United Kingdom.

Turing moved to London in the mid-1940s, and began working for the National Physical Laboratory. Among his most notable contributions while working at the facility, Turing led the design work for the Automatic Computing Engine and ultimately created a groundbreaking blueprint for store-program computers. Though a complete version of the ACE was never built, its concept has been used as a model by tech corporations worldwide for several years, influencing the design of the English Electric DEUCE and the American Bendix G-15—credited by many in the tech industry as the world's first personal computer—among other computer models.

Turing went on to hold high-ranking positions in the mathematics department and later the computing laboratory at the University of Manchester in the late 1940s. He first addressed the issue of artificial intelligence in his 1950 paper, "Computing machinery and intelligence," and proposed an experiment known as the "Turing Test"—an effort to create an intelligence design standard for the tech industry. Over the past several decades, the test has significantly influenced debates over artificial intelligence.

Homosexuality, Conviction and Death

Homosexuality was illegal in the United Kingdom in the early 1950s, so when Turing admitted to police—who he called to his house after a break-in—in January, 1952, that he had had a sexual relationship with the perpetrator, 19-year-old Arnold Murray, he was charged with gross indecency. Following his arrest, Turing was forced to choose between temporary probation on the condition that he receive hormonal treatment for libido reduction, or imprisonment. He chose the former, and soon underwent chemical castration through injections of a synthetic estrogen hormone for a year, which eventually rendered him impotent.

As a result of his conviction, Turing's security clearance was removed and he was barred from continuing his work with cryptography at the GCCS, which had become the GCHQ in 1946.

Turing died on June 7, 1954. Following a postmortem exam, it was determined that the cause of death was cyanide poisoning. The remains of an apple were found next to the body, though no apple parts were found in his stomach. The autopsy

reported that "four ounces of fluid which smelled strongly of bitter almonds, as does a solution of cyanide" was found in the stomach. Trace smell of bitter almonds was also reported in vital organs. The autopsy concluded that the cause of death was asphyxia due to cyanide poisoning and ruled a suicide.

In a June 2012 BBC article, philosophy professor and Turing expert Jack Copeland argued that Turing's death may have been an accident: The apple was never tested for cyanide, nothing in the accounts of Turing's last days suggested he was suicidal and Turing had cyanide in his house for chemical experiments he conducted in his spare room.

Awards, Recognition and Royal Pardon

Shortly after World War II, Alan Turing was awarded an Order of the British Empire for his work. For what would have been his 86th birthday, Turing biographer Andrew Hodges unveiled an official English Heritage blue plaque at his childhood home. In June 2007, a life-size statue of Turing was unveiled at Bletchley Park, in Buckinghamshire, England. A bronze statue of Turing was unveiled at the University of Surrey on October 28, 2004, to mark the 50th anniversary of his death. Additionally, the *Princeton University Alumni Weekly* named Turing the second most significant alumnus in the history of the school - James Madison held the number 1 position.

Turing was honored in a number of other ways, particularly in the city of Manchester, where he worked toward the end of his life. In 1999, *T*ime magazine named him one of its "100 Most Important People of the 20th century," saying, "The fact remains that everyone who taps at a keyboard, opening a spreadsheet or a word-processing program, is working on an incarnation of a Turing machine." Turing was also ranked 21st on the BBC nationwide poll of the "100 Greatest Britons" in 2002. By and large, Turing has been recognized for his impact on computer science, with many crediting him as the "founder" of the field.

Following a petition started by John Graham-Cumming, then-Prime Minister Gordon Brown released a statement on September 10, 2009 on behalf of the British government, which posthumously apologized to Turing for prosecuting him as a homosexual. "Thousands of people have come together to demand justice for Alan Turing and recognition of the appalling way he was treated," Brown wrote in the statement. "While Turing was dealt with under the law of the time and we cannot put the clock back, his treatment was of course utterly unfair and I am pleased to have the chance to say how deeply sorry I and we all are for what happened to him.

"This recognition of Alan's status as one of Britain's most famous victims of homophobia is another step towards equality and long overdue. But even more than that, Alan deserves recognition for his contribution to humankind," Brown stated.

"It is thanks to men and women who were totally committed to fighting fascism, people like Alan Turing, that the horrors of the Holocaust and of total war are part of Europe's history and not Europe's present. So on behalf of the British government, and all those who live freely thanks to Alan's work I am very proud to say: we are sorry, you deserved so much better." In 2013, Queen Elizabeth II posthumously granted Turing a rare royal pardon almost 60 years after he committed suicide. Three years later, on October 20, 2016, the British government announced "Turing's Law" to posthumously pardon thousands of gay and bisexual men who were convicted for homosexual acts when it was considered a crime. According to a statement issued by Justice Minister Sam Gyimah, the law also automatically pardons living people who were "convicted of historical sexual offenses who would be innocent of any crime today.

SUMMARY

- A Turing machine is a hypothetical device that manipulates symbols on a strip of tape according to a table of rules. A Turing machine can be adapted to simulate the logic of any computer algorithm, and is particularly useful in explaining the functions of a CPU inside a computer.
- A Turing machine consists of a finite program, called the finite control, capable of manipulating a linear list of cells, called the tape, using one access pointer, called the head.
- A universal Turing machine U is a Turing machine that can imitate the behavior of any other Turing machine T. It is a fundamental result that such machines exist and can be constructed effectively.
- A head read / write would move in any direction along the tape, one square at a time, and a control unit could play a list of simple instructions, moving to the right or left. The rule executed determines the so-called state of the machine.
- The interpretation and implementation of the algorithms are carried out by states and a transition function determines the new contents of the tape.
- Edges are labeled by pairs (symbol, action), the first consisting of the symbol should be read and then the action that should be performed with the transition. The symbols belong to the alphabet Σ and the action is the symbol to be written, or even 'or', indicating a movement to the left or right.
- The trick used in the proof is called diagonalization to show that there is no computable function g such that for all x,y , we have g(x,y)=1 if ϕx(y) is defined, and g(x,y)=0 otherwise.

KNOWLEDGE CHECK

1. **Turing machine was invented by:**
   a. Alan Turing
   b. Turing man
   c. Turing taring
   d. None of these
2. **Turing machine is more powerful than:**
   a. Finite automata
   b. Push down automata
   c. Both (a) and (b)
   d. None of these

3. **Turing machine can be represented using:**
   a. Transition table
   b. Transition diagram
   c. Instantaneous description
   d. All of these
4. **Which of the following is an extension to the basic model of turing machine:**
   a. Multitude turing machine
   b. Multi head turing machine
   c. Offline turing machine
   d. All of the above
5. **Which of the following is the restricted model of turing machines**
   a. Turing machine with semi-infinite tape
   b. Multi stack machine
   c. Offline turing machine
   d. Both (a) and (b)
6. **A turing machine that is able to simulate other turing machines:**
   a. Nested Turing machines
   b. Universal Turing machine
   c. Counter machine
   d. None of the mentioned
7. **Which of the following a turing machine does not consist of?**
   a. input tape
   b. head
   c. state register
   d. none of the mentioned
8. **Which of the problems are unsolvable?**
   a. Halting problem
   b. Boolean Satisfiability problem
   c. Halting problem & Boolean Satisfiability problem
   d. None of the mentioned
9. **The value of n if turing machine is defined using n-tuples:**
   a. 6
   b. 7

c. 8

d. 5

10. If d is not defined on the current state and the current tape symbol, then the machine _____

a. does not halts

b. halts

c. goes into loop forever

d. none of the mentioned

REVIEW QUESTIONS

1. What is a Turing machine (TM)?
2. Describe the partial computable function
3. Give (in our abbreviated notation) a random access Turing machine program that decides the language {wcw : w ϵ {a, b}*}.
4. Describe (in an extension of our notation for k-tape Turing machines) a 2-head Turing machine that compute the function f(w) = ww.
5. Give a Turing machine that semidecides the language a* ba* b.

Check Your Result

1. (a)	2. (c)	3. (d)	4. (d)	5. (d)
6. (b)	7. (d)	8. (c)	9. (b)	10. (b)

REFERENCES

1. Bullynck, Maarten, Edgar G. Daylight, and Liesbeth De Mol, 2015, "Why Did Computer Science Make a Hero Out of Turing?", Communications of the ACM, 58(3): 37–39.
2. Cooper, S. Barry and Jan Van Leeuwen, 2013, Alan Turing: His Work and Impact, Amsterdam: Elsevier.
3. Copeland, B. Jack, 2002, "Accelerating Turing Machines", Minds and Machines, 12(2): 281–301.
4. De Mol, Liesbeth, 2013, "Generating, Solving and the Mathematics of Homo Sapiens. Emil Post's Views On computation", Hector Zenil (ed.), A Computable Universe. Understanding Computation & Exploring Nature As Computation, Hackensack, NJ: World Scientific, pp. 45–62.
5. De Mol, Liesbeth, Maarten Bullynck, and Edgar G. Daylight, 2018, "Less is More in the Fifties: Encounters between Logical Minimalism and Computer Design during the 1950s", IEEE Annals of the History of Computing, 40(1): 19–45.
6. Dershowitz, Nachum and Yuri Gurevich, 2008, " A Natural Axiomatization of Computability and Proof of Church's Thesis", Bulletin of Symbolic Logic, 14(3): 299–350.
7. Haigh, Thomas, 2013, "'Stored Program Concept' Considered Harmful: History and Historiography", in Paola Bonizzoni, Vasco Brattka, and Benedikt Löwe, The Nature of Computation. Logic, Algorithms, Applications: 9th Conference on Computability in Europe, CiE 2013, Milan, Italy, July 1–5, 2013 Proceedings, (Lecture Notes in Computer Science, 7921), Berlin: Springer, pp. 241–251.
8. M. Li and P. Vitanyi, An Introduction to Kolmogorov Complexity and Its Applications, Springer-Verlag, New York, Third edition, 2008.
9. Mounier-Kuhn, Pierre, 2012, "Logic and Computing in France: A Late Convergence", in AISB/IACAP World Congress 2012: History and Philosophy of Programming, University of Birmingham, 2-6 July 2012.
10. Shapiro, Stewart, 2007, "Computability, Proof, and Open-Texture", in Adam Olszewski, Jan Wolenski, and Robert Janusz (eds.), Church's Thesis After 70 years, Berlin: Ontos Verlag, pp. 420–455.

CHAPTER

5

ALGORITHMS

Learning Objectives

After studying this chapter, you will be able to:

1. Discuss about the sorting algorithms
2. Explain the numeric algorithm

INTRODUCTION

An algorithm is a procedure used for solving a problem or performing a computation. Algorithms act as an exact list of instructions that conduct specified actions step by step in either hardware- or software-based routines.

Algorithms are widely used throughout all areas of IT. In mathematics and computer science, an algorithm usually refers to a small procedure that solves a recurrent problem. Algorithms are also used as specifications for performing data processing and play a major role in automated systems.

Algorithms can be expressed as natural languages, programming languages, pseudocode, flowcharts and control tables. Natural language expressions are rare, as they are more ambiguous. Programming languages are normally used for expressing algorithms executed by a computer.

Algorithms use an initial input along with a set of instructions. The input is the initial data needed to make

decisions and can be represented in the form of numbers or words. The input data gets put through a set of instructions, or computations, which can include arithmetic and decision-making processes. The output is the last step in an algorithm and is normally expressed as more data.

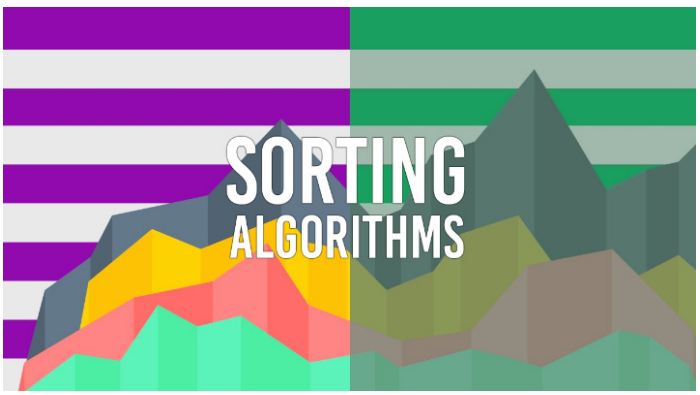
For example, a search algorithm takes a search query as input and runs it through a set of instructions for searching through a database for relevant items to the query. Automation software acts as another example of algorithms, as automation follows a set of rules to complete tasks. Many algorithms make up automation software, and they all work to automate a given process.

Keyword

Number is a mathematical object used to count, measure, and label. The original examples are the natural numbers 1, 2, 3, and so forth.

5.1 SORTING ALGORITHMS

In computer programming, there are often many different ways -- algorithms -- to accomplish any given task. Each algorithm has advantages and disadvantages in different situations. Sorting is one place where a lot of research has been done, because computers spend a lot of time sorting lists. Here are five different algorithms that are used in sorting: All the sorting algorithms use data structures of a specific type to demonstrate sorting, *for example*, a 32 bit integer is often used as its associated operations (*for example*, <, >, etc.) are clear in their behavior. The algorithms discussed can easily be translated into generic sorting algorithms within respective language of choice.



5.1.1 Bubble Sort

A bubble sort, also called a sinking sort or exchange sort, is a sorting algorithm that compares adjacent pairs and swaps them if necessary, causing the items to "bubble" up toward their proper position. The process continues until no swaps are necessary. A bubble sort always brings the highest value to the top during the first pass through the list, the second highest value to the second topmost position during the second pass, and so forth for each successive pass. This pattern enables a common optimization of bubble sort: not all elements need to be processed in every pass. The **number** of elements processed during a particular pass may be reduced to n – *i*, where *i* is the number of passes previously completed.



For example, consider the array:

45, 67, 12, 34, 25, 39

In the first step, the focus is on the first two elements (in bold) which are compared and swapped, if necessary. In this case, since the element at index 1 is larger than the one at index 0, no swap takes place.

45, 67, 12, 34, 25, 39

Then the focus move to the elements at index 1 and 2 which are compared and swapped, if necessary. In our example, 67 is larger than 12 so the two elements are swapped. The result is that the largest of the first three elements is now at index 2.

45, 12, 67, 34, 25, 39

The process is repeated until the focus moves to the end of the array, at which point the largest of all the elements ends up at the highest possible index. The remaining steps and result are:

45, 12, 34, 67, 25, 39

45, 12, 34, 25, 67, 39

45, 12, 34, 25, 39, 67

Working of Bubble sort algorithm

Bubble sort algorithm starts by comparing the first two elements of an array and swapping if necessary, i.e., if you want to sort the elements of array in ascending order and if the first element is greater than second then, you need to swap the elements but, if the first element is smaller than second, you must not swap the element. Then, again second and third elements are

compared and swapped if it is necessary and this process go on until last and second last element is compared and swapped. This completes the first step of bubble sort.

If there are n elements to be sorted then, the process mentioned above should be repeated n-1 times to get required result. But, for better performance, in second step, last and second last elements are not compared because, the proper element is automatically placed at last after first step. Similarly, in third step, last and second last and second last and third last elements are not compared and so on.

Keyword

Memory refers to the computer hardware devices used to store information for immediate use in a computer; it is synonymous with the term "primary storage".

One of the most simple forms of sorting is that of comparing each item with every other item in some list, however as the description may imply this form of sorting is not particularly efficient $O(n^2)$. In its most simple form bubble sort can be implemented as two loops.

```
1) algorithm BubbleSort(list)
2)     Pre: list ≠ ∅
3)     Post: list has been sorted into values of ascending order
4)     for i ← 0 to list.Count − 1
5)         for j ← 0 to list.Count − 1
6)             if list[i] < list[j]
7)                 Swap(list[i], list[j])
8)             end if
9)         end for
10)    end for
11)    return list
12) end BubbleSort
```

Advantages of bubble sort

The Bubble sort basically just runs through an array and swaps adjacent elements if they are out of order until the list is sorted.

Its most common usage is in algorithms courses and tutorials, because it is probably the simplest sorting algorithm after the Insertion sort (which just runs through an array and pulls out the lowest item and appends it to the results list) and has a few other interesting properties:

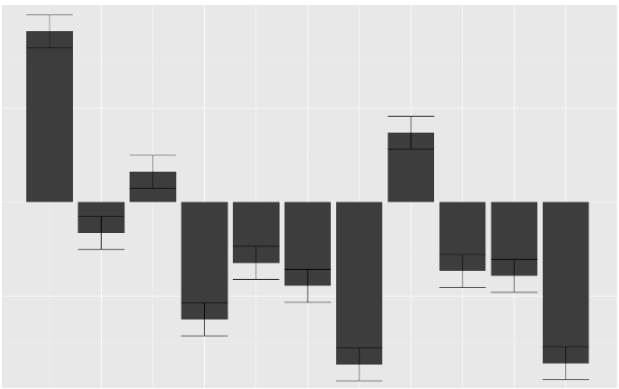
- *It is time-inefficient* - It runs in $\mathcal{O}(n^2)$ time, versus the $\mathcal{O}(n \log(n))$ of most efficient sorting algorithms.
- *It is very memory-efficient* - It requires only $\mathcal{O}(1)$ space, as the only extra memory it requires is that of the temporary value created when swapping adjacent elements.

- *It is stable* - The order of identical items does not change while the array is being sorted.

Outside of educational material, it is pretty much only used to be made fun of. There's almost no reason why anyone would want to use Bubble Sort for any practical purpose (unless there is some weird limitation that every element is only allowed to move one space per move). However, it is worth noting that in the extremely unlikely case that you have only have a few bytes of free **memory** left, an $\mathcal{O}(n^2)$ sorting algorithm is pretty much the best you can get.

5.1.2 Merge Sort

Merge sort is an efficient, general-purpose sorting algorithm which produces a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output. Merge sort is a comparison sort, i.e. it can sort items of any type for which a less-than relation is defined. Merge sort's most common implementation does not sort in-place.



Working of Merge sort algorithm

Merge sort is a divide and conquer algorithm. Like all divide and conquer algorithms, merge sort first divides a large array into two smaller sub-arrays and then recursively sort the sub-arrays. Basically, there are two steps are involved in whole process –

- Divide the unsorted list into n sub lists, each containing 1 element (a list of 1 element is considered sorted).
- Repeatedly merge sub lists to produce new sorted sub lists until there is only 1 sub list remaining. This will be the sorted list.

Merge sort is an that has a fairly efficient space time complexity -O(n log n) and is fairly trivial to implement. The algorithm is based on splitting a list, into two

similar sized lists (left, and right) and sorting each list and then merging the sorted lists back together.

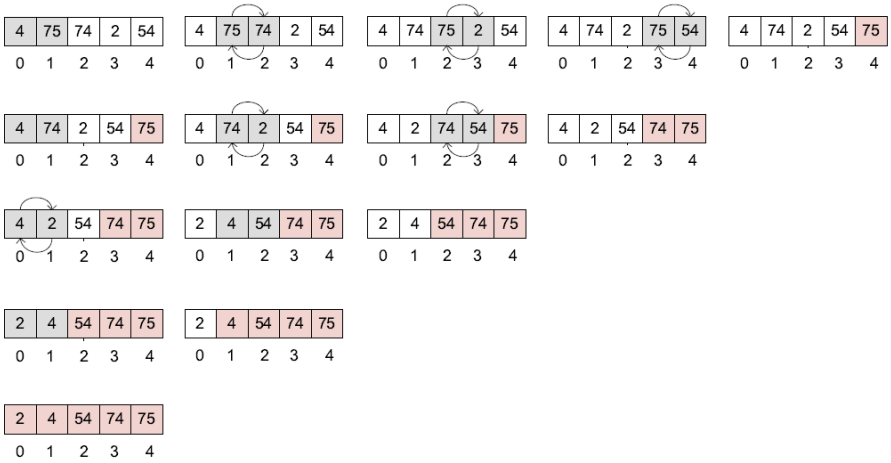


Figure 5.1: Bubble sort iterations.

```
1) algorithm Mergesort(list)
2)     Pre: list ≠ ∅
3)     Post: list has been sorted into values of ascending order
4)     if list.Count = 1 // already sorted
5)         return list
6)     end if
7)     m ← list.Count / 2
8)     left ← list(m)
9)     right ← list(list.Count − m)
10)    for i ← 0 to left.Count−1
11)        left[i] ← list[i]
12)    end for
13)    for i ← 0 to right.Count−1
14)        right[i] ← list[i]
15)    end for
16)    left ← Mergesort(left)
17)    right ← Mergesort(right)
18)    return MergeOrdered(left, right)
19) end Mergesort
```

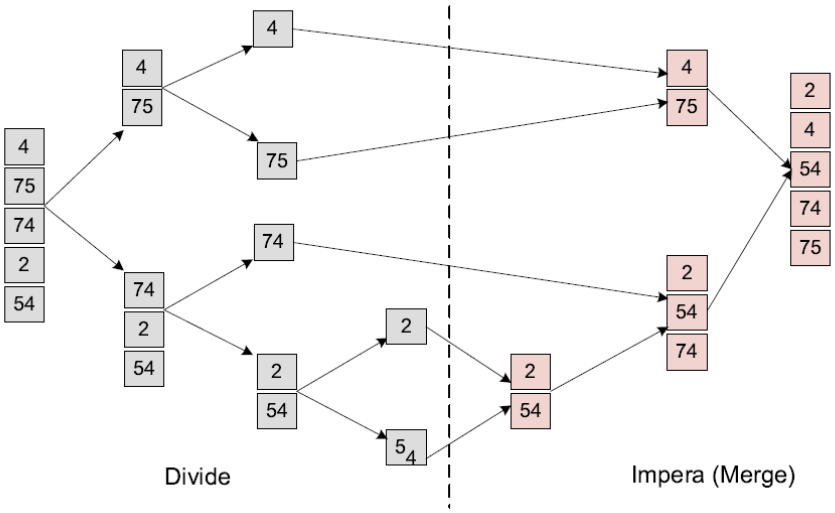


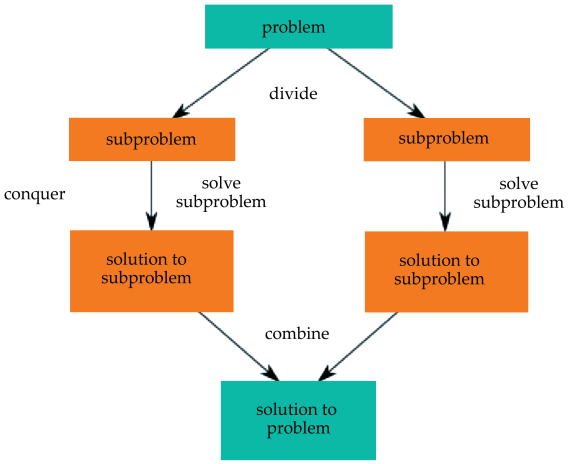
Figure 5.2: Merge Sort Divide et Impera Approach.

Divide-and-conquer

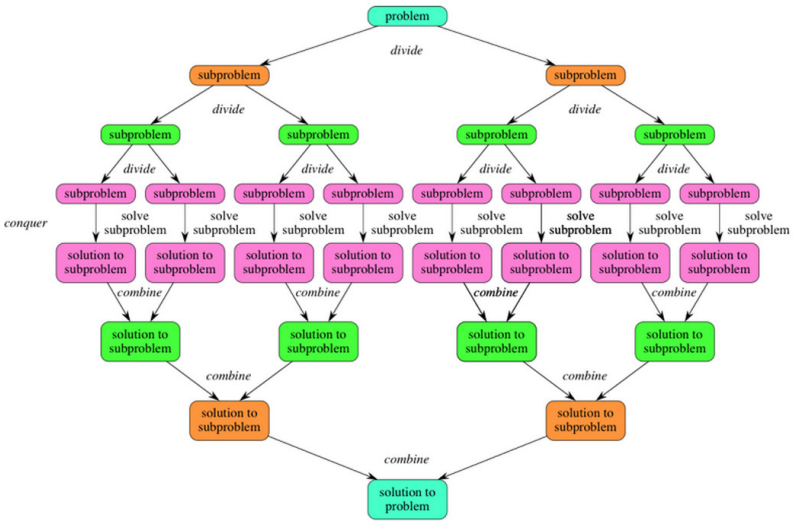
Both merge sort and quicksort employ a common algorithmic paradigm based on recursion. This paradigm, divide-and-conquer, breaks a problem into sub problems that are similar to the original problem, recursively solves the sub problems, and finally combines the solutions to the sub problems to solve the original problem. Because divide-and-conquer solves sub problems recursively, each sub problem must be smaller than the original problem, and there must be a base case for sub problems. You should think of a divide-and-conquer algorithm as having three parts:

- Divide the problem into a number of sub problems that are smaller instances of the same problem.
- Conquer the sub problems by solving them recursively. If they are small enough, solve the sub problems as base cases.
- Combine the solutions to the sub problems into the solution for the original problem.

You can easily remember the steps of a divide-and-conquer algorithm as *divide, conquer, combine*. Here's how to view one step, assuming that each divide step creates two sub problems (though some divide-and-conquer algorithms create more than two):



If we expand out two more recursive steps, it looks like this:



Analysis of merge sort

Given that the merge function runs in $\Theta(n)$ time when merging n elements, how do we get to showing that merge Sort runs in $\Theta(n\lg n)$ time? We start by thinking about the three parts of divide-and-conquer and how to account for their running times. We assume that we are sorting a total of n elements in the entire array.

- The divide step takes constant time, regardless of the sub array size. After all, the divide step just computes the midpoint q of the indices p and r. Recall that in big-Θ notation, we indicate constant time by $\Theta(1)$.
- The conquer step, where we recursively sort two sub **arrays** of approximately

$n/2$ elements each, takes some amount of time, but we shall account for that time when we consider the sub problems.

- The combine step merges a total of n elements, taking $\Theta(n)$ time.

Keyword

Array is a systematic arrangement of similar objects, usually in rows and columns.

5.1.3 Quick Sort

Quicksort is popular because it is not difficult to implement, works well for a variety of different kinds of input data, and is substantially faster than any other sorting method in typical applications.

The quick sort uses divide and conquer to gain the same advantages as the merge sort, while not using additional storage. As a trade-off, however, it is possible that the list may not be divided in half. When this happens, we will see that performance is diminished.

A quick sort first selects a value, which is called the pivot value. Although there are many different ways to choose the pivot value, we will simply use the first item in the list. The role of the pivot value is to assist with splitting the list. The actual position where the pivot value belongs in the final sorted list, commonly called the split point, will be used to divide the list for subsequent calls to the quick sort.

Quicksort has a couple of other differences from merge sort. Quicksort works in place. And its worst-case running time is as bad as selection sort's and insertion sort's: $\Theta(n^2)$. But its average-case running time is as good as merge sort's: $\Theta(n \lg n)$. So why think about quicksort when merge sort is at least as good? That's because the constant factor hidden in the big-Θ notation for quicksort is quite good. In practice, quicksort outperforms merge sort, and it significantly outperforms selection sort and insertion sort.

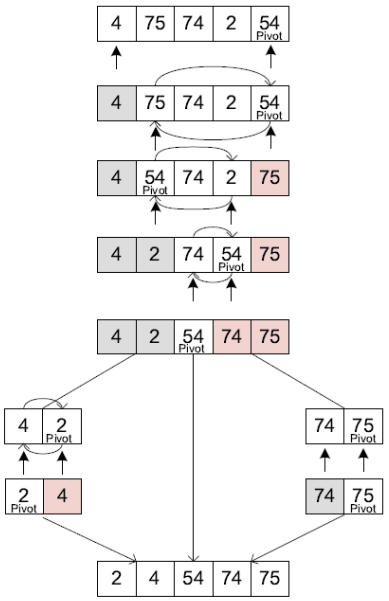


Figure 5.3: Quick sort Example (pivot median strategy).

```
1) algorithm QuickSort(list)
2)    Pre: list ≠ ∅
3)    Post: list has been sorted into values of ascending order
4)    if list.Count = 1 // already sorted
5)       return list
6)    end if
7)    pivot ←MedianValue(list)
8)    for i ← 0 to list.Count−1
9)       if list[i] = pivot
10)         equal.Insert(list[i])
11)      end if
12)      if list[i] < pivot
13)         less.Insert(list[i])
14)      end if
15)      if list[i] > pivot
16)         greater.Insert(list[i])
17)      end if
18)   end for
19)   return Concatenate(QuickSort(less), equal, QuickSort(greater))
20) end Quicksort
```

Here is how quicksort uses divide-and-conquer. As with merge sort, think of sorting a sub array array[p..r], where initially the sub array is array[0..n-1].

- Divide by choosing any element in the subarray array[p..r]. Call this element the pivot. Rearrange the elements in array[p..r] so that all other elements in array[p..r] that are less than or equal to the pivot are to its left and all elements in array[p..r] are to the pivot's right. We call this procedure partitioning. At this point, it does not matter what order the elements to the left of the

pivot are in relative to each other, and the same holds for the elements to the right of the pivot. We just care that each element is somewhere on the correct side of the pivot.

As a matter of practice, we shall always choose the rightmost element in the subarray, array[r], as the pivot. So, *for example*, if the subarray consists of [9, 7, 5, 11, 12, 2, 14, 3, 10, 6], then we choose 6 as the pivot. After partitioning, the subarray might look like [5, 2, 3, 6, 12, 7, 14, 9, 10, 11]. Let qbe the index of where the pivot ends up.

- Conquer by recursively sorting the subarrays array[p..q-1] (all elements to the left of the pivot, which must be less than or equal to the pivot) and array[q+1..r] (all elements to the right of the pivot, which must be greater than the pivot).
- Combine by doing nothing. Once the conquer step recursively sorts, we are done. Why? All elements to the left of the pivot, in array[p..q-1], are less than or equal to the pivot and are sorted, and all elements to the right of the pivot, in array[q+1..r], are greater than the pivot and are sorted. The elements in array[p..r] cannot help but be sorted!

For example, after recursively sorting the subarrays to the left and right of the pivot, the subarray to the left of the pivot is [2, 3, 5], and the subarray to the right of the pivot is [7, 9, 10, 11, 12, 14]. So the subarray has [2, 3, 5], followed by 6, followed by [7, 9, 10, 11, 12, 14]. The subarray is sorted.

5.1.4 Insertion Sort

Insertion sort is a somewhat interesting algorithm with an expensive runtime of $O(n^2)$. It can be best thought of as a sorting scheme similar to that of sorting a hand of playing cards that is you take one card and then look at the rest with the intent of building up an ordered set of cards in hand.

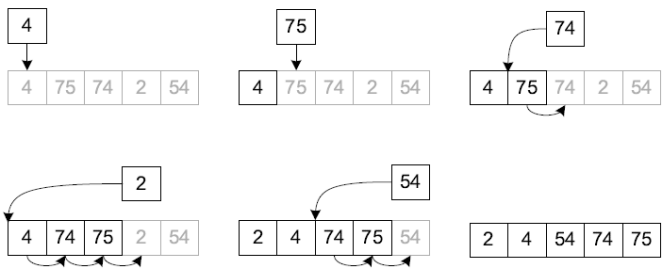


Figure 5.4: Insertion sort iterations.

```
1) algorithm Insertionsort(list)
2)     Pre:   list ≠ ∅
3)     Post: list has been sorted into values of ascending order
4)     unsorted ← 1
5)     while unsorted < list.Count
6)         hold ← list[unsorted]
7)         i ← unsorted − 1
8)         while i ≥ 0 and hold < list[i]
9)             list[i + 1] ← list[i]
10)            i ← i − 1
11)        end while
12)        list[i + 1] ← hold
13)        unsorted ← unsorted + 1
14)    end while
15)    return list
16) end Insertionsort
```

Keyword

Algorithm is a self-contained sequence of actions to be performed. Algorithms perform calculation, data processing, and/or automated reasoning tasks.

If the first few objects are already sorted, an unsorted object can be inserted in the sorted set in proper place. This is called insertion sort. An algorithm consider the elements one at a time, inserting each in its suitable place among those already considered. Insertion sort is an example of an incremental algorithm; it builds the sorted sequence one number at a time. This is perhaps the simplest example of the incremental insertion technique, where we build up a complicated structure on n items by first building it on n − 1 items and then making the necessary changes to fix things in adding the last item. The given sequences are typically stored in arrays. We also refer the numbers as keys. Along with each key may be additional information, known as satellite data

Sorting **algorithms** that perform swaps are inherently unstable. Items are always moving around and so you cannot guarantee any previous ordering will be maintained. Insertion sort does not perform swaps. Instead, it picks out individual items and inserts them into the correct spot in an array.

How insertion sort Algorithm works?

An insertion sort works by separating an array into two sections, a sorted section and an unsorted section. Initially, of course, the entire array is unsorted. The sorted section is then considered to be empty. The first step is to add a value to the sorted section, so the first item in the array is used. Then at each item in the unsorted section:

- If the item value goes after the last item in the sorted section, then do nothing.
- If the item value goes before the last item in the sorted section, remove the item value from the array and shift the last sorted item into the now-vacant spot.
- Compare the item value to the previous value (second to last) in the sorted section.
- If the item value goes after the previous value and before the last value, then place the item into the open spot between them, otherwise, continue this process until the start of the array is reached.

For example, suppose, you want to sort elements in ascending as in below figure. Then,

- *Step 1*: The second element of an array is compared with the elements that appears before it (only first element in this case). If the second element is smaller than first element, second element is inserted in the position of first element. After first step, first two elements of an array will be sorted.
- *Step 2*: The third element of an array is compared with the elements that appears before it (first and second element). If third element is smaller than first element, it is inserted in the position of first element. If third element is larger than first element but, smaller than second element, it is inserted in the position of second element. If third element is larger than both the elements, it is kept in the position as it is. After second step, first three elements of an array will be sorted.
- *Step 3*: Similar, the fourth element of an array is compared with the elements that appears before it (first, second and third element) and the same procedure is applied and that element is inserted in the proper position. After third step, first four elements of an array will be sorted.

If there are n elements to be sorted. Then, this procedure is repeated n-1 times to get sorted list of array.

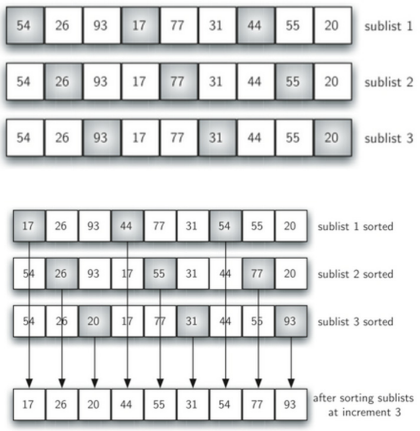
5.1.5 Shell Sort

The shell sort, sometimes called the "diminishing increment sort," improves on the insertion sort by breaking the original list into a number of smaller sublists, each of which is sorted using an insertion sort. The unique way that these sublists are chosen is the key to the shell sort. Instead of breaking the list into sublists of contiguous items, the shell sort uses an increment i, sometimes called the gap, to create a sublist by choosing all items that are i items apart.

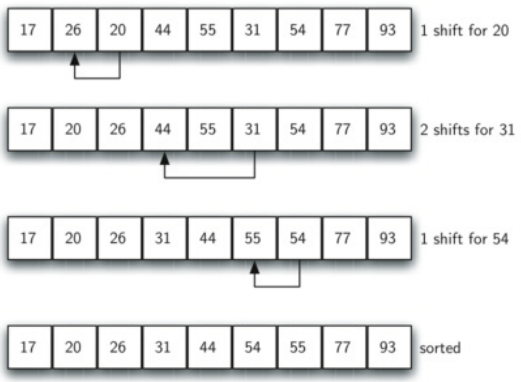
This can be seen in below figure. This list has nine items. If we use an increment of three, there are three sublists, each of which can be sorted by an insertion sort. After completing these sorts, we get the list. Although this list is not completely sorted, something very interesting has happened. By sorting the sublists, we have moved the items closer to where they actually belong.

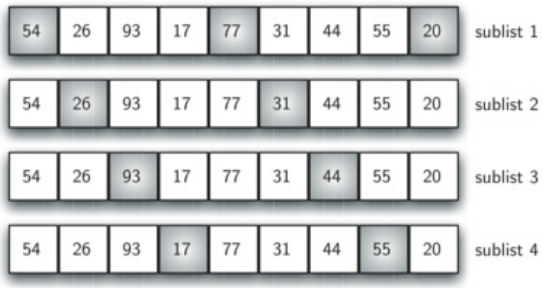
Remember

An optimal algorithm, even running in old hardware, would produce faster results than a non-optimal (higher time complexity) algorithm for the same purpose, running in more efficient hardware; that is why algorithms, like computer hardware, are considered technology.



A final insertion sort using an increment of one; in other words, a standard insertion sort. Note that by performing the earlier sublist sorts, we have now reduced the total number of shifting operations necessary to put the list in its final order. For this case, we need only four more shifts to complete the process.






```
public class ShellSortEx
  {
    public static void main(String a[])
      {
        //Numbers which are to be sorted
        int n[] = {88,33,77,55,99,22,11,66,44};
        //Displays the numbers before sorting
        System.out.print("Before sorting, numbers are ");
        for(int i = 0; i < n.length; i++)
          {
            System.out.print(n[i]+" ");
          }
        System.out.println();
        //Sorting in ascending order using Shell Sort
        initializemergeSort(n);;
        //Displaying the numbers after sorting
        System.out.print("After sorting, numbers are ");
        for(int i = 0; i < n.length; i++)
          {
            System.out.print(n[i]+" ");
```

```
    }
  }
public static void initializemergeSort(int n[])
  {
    int i1,i,j,increment,temp, no_of_elem=n.length;
    /* Shell Sort Program */
    for (increment=no_of_elem/2;increment>0;increment/=2)
      {
        for(i=increment;i<no_of_elem;i++)
          {
            temp=n[i];
            for(j=i;j>=increment;j-=increment)
              {
                if(temp<n[j-increment])
                  {
                    n[j]=n[j-increment];
                  }
                else
                  {
                    break;
                  }
              }
            n[j] = temp;
          }
      }
  }
```

```
}
```

Output

Time Complexity

Time complexity of above implementation of shellsort is $O(n^2)$. In the above implementation gap is reduce by half in every iteration. There are many other ways to reduce gap which lead to better time complexity.

5.1.6 Radix Sort

Radix sort is one of the linear sorting algorithms for integers. It functions by sorting the input numbers on each digit, for each of the digits in the numbers. However, the process adopted by this sort method is somewhat counterintuitive, in the sense that the numbers are sorted on the least-significant digit first, followed by the second-least significant digit and so on till the most significant digit.

For example, suppose that we wish to sort a deck of 52 playing cards (the different suits can be given suitable values, *for example* 1 for Diamonds, 2 for Clubs, 3 for Hearts and 4 for Spades). The 'natural' thing to do would be to first sort the cards according to suits, then sort each of the four seperate piles, and finally combine the four in order. This approach, however, has an inherent disadvantage. When each of the piles is being sorted, the other piles have to be kept aside and kept track of. If, instead, we follow the 'counterintuitive' aproach of first sorting the cards by value, this problem is eliminated. After the first step, the four seperate piles are combined in order and then sorted by suit. If a stable **sorting algorithm** (i.e. one which resolves a tie by keeping the number obtained first in the input as the first in the output) it can be easily seen that correct final results are obtained.

Keyword

Sorting Algorithm is used to rearrange a given array or list of elements according to a comparison operator on the elements.

As has been mentioned, the sorting of numbers proceeds by sorting the least significant to most significant digit. For sorting each of these digit groups, a stable sorting algorithm is needed. Also, the elements in this group to be sorted are in the fixed

range of 0 to 9. Both of these characteristics point towards the use of Counting Sort as the sorting algorithm of choice for sorting on each digit.

Radix Sort is an algorithm that sorts a list of numbers and comes under the category of distribution sort.

This sorting algorithm does not compare the numbers but distributes them, it works as follows:

- Sorting takes place by distributing the list of number into a bucket by passing through the individual digits of a given number one-by-one beginning with the least significant part. Here, the number of buckets are a total of ten, which bare key values starting from 0 to 9.
- After each pass, the numbers are collected from the buckets, keeping the numbers in order
- Now, recursively redistribute the numbers as in the above step '1' but with a following reconsideration: take into account next most significant part of the number, which is then followed by above step '2'.

The number used as an example has in total three keys:

1. Ones
2. Tens
3. Hundreds

For further clarification what if we wanted to determine how many thousands the number 102 has? Clearly there are none, but often looking at a number as final like we often do it is not so obvious so when asked the question how many thousands does 102 have you should simply pad the number with a zero in that location, *for example*, 0102 here it is more obvious that the key value at the thousands location is zero. The last thing to identify before we actually show you a simple implementation of radix sort that works on only positive integers, and requires you to specify the maximum key size in the list is that we need a way to isolate a specific key at any one time. The solution is actually very simple, but it is not often you want to isolate a key in a number so we will spell it out clearly here. A key can be accessed from any integer with the following expression: key Ã (number = key To Access) % 10. As a simple example let us say that we want to access the tens key of the number 1290, the tens column is key 10 and so after substitution yields key Ã (1290 = 10) % 10 = 9. The next key to look at for a number can be attained by multiplying the last key by ten working left to right in a sequential manner. The value of key is used in the following algorithm to work out the index of an array of queues to enqueue the item into.

```
1) algorithm Radix(list, maxKeySize)
2)     Pre: list ≠ ∅
3)          maxKeySize ≥ 0 and represents the largest key size in the list
4)     Post: list has been sorted
5)     queues ← Queue[10]
6)     indexOfKey ← 1
7)     fori ← 0 to maxKeySize − 1
8)         foreach item in list
9)             queues[GetQueueIndex(item, indexOfKey)].Enqueue(item)
10)        end foreach
11)        list ← CollapseQueues(queues)
12)        ClearQueues(queues)
13)        indexOfKey ← indexOfKey ∗ 10
14)    end for
15)    return list
16) end Radix
```

Figure 5.5 shows the members of queues from the algorithm described operating on the list whose members are 90; 12; 8; 791; 123; and 61, the key we are interested in for each number is highlighted. Omitted queues in figure 5.5 mean that they contain no items.

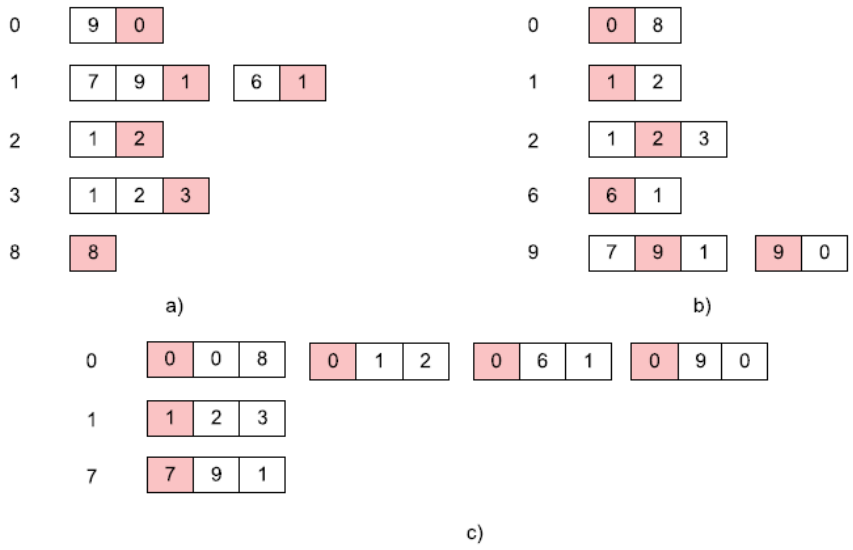


Figure 5.5: Radix sort base 10 algorithm.

Time complexity

The time complexity of the algorithm is as follows: Suppose that the n input numbers have maximum k digits. Then the Counting Sort procedure is called a total of k times. Counting Sort is a linear, or O(n) algorithm. So the entire Radix Sort procedure takes O(kn) time. If the numbers are of finite size, the algorithm runs in O(n) asymptotic time.

Disadvantages

There are some tradeoffs for Radix Sort that can make it less preferable than other sorts.

- The speed of Radix Sort largely depends on the inner basic operations, and if the operations are not efficient enough, **Radix Sort** *can* be slower than some other algorithms such as Quick Sort and Merge Sort. These operations include the insert and delete functions of the sublists and the process of isolating the digit you want.
- In the example above, the numbers were all of equal length, but many times, this is not the case. If the numbers are not of the same length, then a test is needed to check for additional digits that need sorting. This can be one of the slowest parts of Radix Sort, and it is one of the hardest to make efficient.
- Radix Sort can also take up more space than other sorting algorithms, since in addition to the array that will be sorted, you need to have a sublist for each of the possible digits or letters. If you are sorting pure English words, you will need at least 26 different sublists, and if you are sorting alphanumeric words or sentences, you will probably need more than 40 sublists in all!
- Since Radix Sort depends on the digits or letters, Radix Sort is also *much* less flexible than other sorts. For every different type of data, Radix Sort needs to be rewritten, and if the sorting order changes, the sort needs to be rewritten again. In short, Radix Sort takes more time to write, and it is very difficult to write a general purpose Radix Sort that can handle all kinds of data.
- For many programs that need a fast sort, Radix Sort is a good choice. Still, there are faster sorts, which is one reason why Radix Sort is not used as much as some other sorts.

Keyword

Radix sort is a non-comparative sorting algorithm. It avoids comparison by creating and distributing elements into buckets according to their radix.

5.2 NUMERIC ALGORITHM

Numerical Algorithms introduces the skills necessary to be both clients and designers of numerical methods for computer science applications. A numeric algorithm does some computation given

one or more numeric values. For the purposes of measuring complexity, the size of a number is the *number of bits* (or digits) in the numbers, not the value of the numbers themselves!

Note the base of the numerals *does not matter when computing asymptotic complexity*. There is always a linear relationship between the number of digits required to represent a number in base a and base b.

DID YOU KNOW?

'Algorithm' stems from the name of a Latin translation of a book written by al-Khwārizmī, a Persian mathematician, astronomer and geographer.

5.2.1 Primarily Test

Primality testing of a number is perhaps the most common problem concerning number theory that top coders deal with. A prime number is a natural number which has exactly two distinct natural number divisors: 1 and itself.

The problem of detecting whether a given number is a prime number has been studied extensively but nonetheless, it turns out that all the deterministic algorithms for this problem are too slow to be used in real life situations and the better ones amongst them are tedious to code. But, there are some probabilistic methods which are very fast and very easy to code. Moreover, the probability of getting a wrong result with these algorithms is so low that it can be neglected in normal situations.

A simple algorithm that determines whether or not a given integer is a prime number, *for example*, 2, 5, 7, and 13 are all prime numbers, however 6 is not as it can be the result of the product of two numbers that are < 6.

In an attempt to slow down the inner loop the $\sqrt{n}$ is used as the upper bound.

```
1) algorithm IsPrime(n)
2)     Post: n is determined to be a prime or not
3)     for i ← 2 to n do
4)         for j ← 1 to sqrt(n) do
5)             if i * j = n
6)                 return false
7)             end if
8)         end for
9)     end for
10) end IsPrime
```

Remember

The concept of algorithm has existed for centuries; however, a partial formalization of what would become the modern algorithm began with attempts to solve the Entscheidungsproblem posed by David Hilbert in 1928.

5.2.2 Base Conversions

Base conversion of whole numbers is fairly easy when we use remainders. DSA contains a number of algorithms that convert a base 10 number to its equivalent binary, octal or **hexadecimal** form. *For example*, 7810 has a binary representation of 10011102. Table 5.1 shows the algorithm trace when the number to convert to binary is 74210.

```
1) algorithm ToBinary(n)
2)     Pre:  n ≥ 0
3)     Post: n has been converted into its base 2 representation
4)     while n > 0
5)         list.Add(n % 2)
6)         n ← n/2
7)     end while
8)     return Reverse(list)
9) end ToBinary
```

Table 5.1: Algorithm trace of two binary

Keyword

Hexadecimal is a positional numeral system with a radix, or base, of 16.

n	List
742	{0}
371	{0, 1}
185	{0, 1, 1}
92	{0, 1, 1, 0}
46	{0, 1, 1, 0, 1}
23	{0, 1, 1, 0, 1, 1}
11	{0, 1, 1, 0, 1, 1, 1}
5	{0, 1, 1, 0, 1, 1, 1, 1}
2	{0, 1, 1, 0, 1, 1, 1, 1, 0}
1	{0, 1, 1, 0, 1, 1; 1, 1; 0, 1}

Attaining the Greatest Common Denominator of Two Numbers

A fairly routine problem in mathematics is that of finding the greatest common denominator of two integers, what we are essentially after is the greatest number which is a multiple of both, *for example*, the greatest common denominator of 9, and 15 is 3. One of the most elegant solutions to this problem is based on Euclid's algorithm that has a run time complexity of $O(n^2)$.

1) Algorithm Greatest Common Denominator (m, n)

2) Pre: m and n are integers

3) Post: the greatest common denominator of the two integers is calculated

4) if n = 0

5) return m

6) end if

7) return GreatestCommonDenominator(n, m % n)

8) end GreatestCommonDenominator

Computing the Maximum Value for a Number of a Specific base consisting of N Digits

This algorithm computes the maximum value of a number for a given number of digits, *for example*, using the base 10 system the maximum number we can have made up of 4 digits is the number 9999_{10}. Similarly the maximum number that consists of 4 digits for a base 2 number is 1111_2 which is 15_{10}.

The expression by which we can compute this maximum value for N digits is: B^N-1. In the expression B is the number base, and N is the number of digits. As an example if we wanted to determine the maximum value for a hexadecimal number (base 16) consisting of 6 digits the expression would be as follows: 16^6-1. The maximum value of the example would be represented as $FFFFFF_{16}$ which yields 16777215_{10}.

In the following algorithm numberBase should be considered restricted to the values of 2, 8, 9, and 16. For this reason in our actual implementation numberBase has an enumeration type.

The Base enumeration type is defined as:

$$Base = \{Binary \leftarrow 2, Octal \leftarrow 8, Decimal \leftarrow 10, Hexadecimal \leftarrow 16\}$$

The reason we provide the definition of Base is to give you an idea how this algorithm can be modeled in a more readable manner rather than using various checks to determine the correct base to use. For our implementation we cast the value of numberBase to an integer, as such we extract the value associated with the relevant option in the Base enumeration. As an example if we were to cast the option Octal to an integer we would get the value 8. In the algorithm listed the cast is implicit so we just use the actual argument numberBase.

Keyword

Factorial is the product of all positive integers less than or equal to n.

1) Algorithm MaxValue (numberBase, n)

2) Pre: number Base is the number system to use, n is the number of digits

3) Post: the maximum value for numberBase consisting of n digits is computed

4) return Power(numberBase; n)-1

5) end MaxValue

5.2.3 Factorial of a Number

The factorial number system, also called factoradic, is a mixed radix numeral system adapted to numbering permutations. It is also called factorial base, although factorials do not function as base, but as place value of digits.

Attaining the factorial of a number is a primitive mathematical operation. Many implementations of the **factorial** algorithm are recursive as the problem is recursive in nature, however here we present an iterative solution. The iterative solution is presented because it too is trivial to implement and does not suffer from the use of recursion.

The factorial of 0 and 1 is 0. The aforementioned acts as a base case that we will build upon. The factorial of 2 is 2¤ the factorial of 1, similarly the factorial of 3 is 3¤ the factorial of 2 and so on. We can indicate that we are after the factorial of a number using the form N! where N is the number we wish to attain the factorial of. Algorithm does not use such notation but it is handy to know.

```
1) algorithm Factorial(n)
2)     Pre:  n ≥ 0, n is the number to compute the factorial of
3)     Post: the factorial of n is computed
4)     if n < 2
5)         return 1
6)     end if
7)     factorial ← 1
8)     for i ← 2 to n
9)         factorial ← factorial * i
10)    end for
11)    return factorial
12) end Factorial
```

ROLE MODEL



ANDERS HEJLSBERG: A MAN WHO HAS FUNDAMENTALLY CHANGED HOW DEVELOPERS BUILD SOFTWARE, GIVING THEM MORE PRODUCTIVE AND POWERFUL FRAMEWORKS

When we say the world has been transformed by the magic of software, we are really tipping our hats to the millions of developers working behind the scenes as the drivers of that change. And driving those drivers is Microsoft Technical Fellow Anders Hejlsberg, a man who has fundamentally changed how developers build software, giving them more productive and powerful frameworks, more elegant and higher-level languages, and a more tightly integrated toolset.

Hejlsberg is no stranger to the Technical Recognition Awards

He and his C# team shared the Outstanding Technical Achievement Award in 2007. But in awarding him the 2010 Career Achievement Award, Microsoft is now recognizing his entire body of work, one that has been profoundly important in improving developer productivity over the last 20 years.

Hejlsberg had already made his mark on the developer world before joining Microsoft in 1996. His achievements in PolyPascal, Turbo Pascal, and Borland Delphi were all groundbreaking; Turbo Pascal in particular set a productivity standard for its day, and served as the "first IDE" (Integrated Development Environment) for a generation of developers.

After joining Microsoft, Hejlsberg led the company's efforts in Visual J++ and Windows Foundation Classes (WFC). By pairing a highly productive Java development environment with a Windows class library without the least common denominator problem of other Java frameworks, Hejlsberg helped make Visual J++ the most popular and most productive Java IDE in the late '90s.

It was then that Hejlsberg proposed designing a new language that would be delivered synchronously with the .NET framework. "At the time there were two camps, debating what we were going to do," recalls Hejlsberg. "There was sort of the evolutionaries or the revolutionaries. I was pretty much in the revolutionaries' camp. I felt that it was not good enough to just modernize what we already had, or try to fix some of the problems. I thought we were at an inflection point, and the

time was right for us to clean out the closet and build new, and get rid of all the old ghosts, if you will. Of course there was a lot of back and forth on that, and I was by no means the only person advocating this, but certainly I was part of that.

"I felt there was a need for us to build a new programming language. I also had come to see that Microsoft functions best when it controls its own destiny. This was an opportunity for us to build a language that would best suit our customers, as opposed to trying to build on someone else's technology, or trying to reinvent an existing thing."

Hejlsberg led this idea from its conception, starting with his "What is Cool" paper that defined the essential elements, and then working through delivery of Visual C# .NET 2002 and Visual C# 2003, 2005, and 2008. Hejlsberg's recent work on Language Integrated Query (LINQ), including support in multiple languages (C# and VB) and multiple data domains (objects, relational, XML, DataSet, and others) has helped establish Microsoft as a thought leader in programming language, object-relational mapping, and data/language integration. This work is breaking down the barriers separating the worlds of object-oriented programming languages, relational databases, and XML documents. Hejlsberg and his team's work on C# and .NET helped both gain immense popularity.

Today, the number of .NET developers worldwide surpasses the number of Java developers, and C# is one of the most popular and fastest growing languages in the industry. Quite an achievement for a man who began his career 30 years ago in a Copenhagen computer store, coding on 8-bit kit computers.

"It gives you great pleasure to know that millions of developers, day to day, make their living using the software that you created," says Hejlsberg. "That's about as good as it gets in this industry, in terms of job satisfaction. It makes you feel that your work is highly relevant to a lot of people. That is very satisfying."

SUMMARY

- Algorithms are widely used throughout all areas of IT. In mathematics and computer science, an algorithm usually refers to a small procedure that solves a recurrent problem. Algorithms are also used as specifications for performing data processing and play a major role in automated systems.
- A bubble sort, also called a sinking sort or exchange sort, is a sorting algorithm that compares adjacent pairs and swaps them if necessary, causing the items to "bubble" up toward their proper position.
- Merge sort is an efficient, general-purpose sorting algorithm which produces a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output.
- Quicksort is popular because it is not difficult to implement, works well for a variety of different kinds of input data, and is substantially faster than any other sorting method in typical applications.
- Insertion sort is a somewhat interesting algorithm with an expensive runtime of O(n2). It can be best thought of as a sorting scheme similar to that of sorting a hand of playing cards that is you take one card and then look at the rest with the intent of building up an ordered set of cards in hand.
- The shell sort, sometimes called the "diminishing increment sort," improves on the insertion sort by breaking the original list into a number of smaller sublists, each of which is sorted using an insertion sort.
- Radix sort is one of the linear sorting algorithms for integers. It functions by sorting the input numbers on each digit, for each of the digits in the numbers. The process adopted by this sort method is somewhat counterintuitive, in the sense that the numbers are sorted on the least-significant digit first, followed by the second-least significant digit and so on till the most significant digit.
- Numerical Algorithms introduces the skills necessary to be both clients and designers of numerical methods for computer science applications. A numeric algorithm does some computation given one or more numeric values.
- The factorial number system, also called factoradic, is a mixed radix numeral system adapted to numbering permutations. It is also called factorial base, although factorials do not function as base, but as place value of digits.

KNOWLEDGE CHECK

1. **Which of the following sorting algorithm is of priority queue sorting type?**
   a. Bubble sort
   b. Insertion sort

c. Merge sort
d. Selection sort

2. Partition and exchange sort is.....

a. quick sort
b. tree sort
c. heap sort
d. bubble sort

3. A sorting algorithm which can prove to be a best time algorithm in one case and a worst time algorithm at other time?

a. Selection sort
b. Heap sort
c. Quick sort
d. All of the above

4. The complexity of merge sort algorithm is:

a. O(n)
b. O(log n)
c. $O(n^2)$
d. O(n log n)

5. The complexity of Bubble sort algorithm is:

a. O(n)
b. O(log n)
c. $O(n^2)$
d. O(n log n)

6. What is an algorithm?

a. A flowchart
b. A flowchart or pseudocode
c. A decision
d. Step by step instructions used to solve a problem

7. What is an internal sorting algorithm?

a. Algorithm that uses tape or disk during the sort
b. Algorithm that uses main memory during the sort
c. Algorithm that involves swapping
d. Algorithm that are considered 'in place'

8. **What is the worst case complexity of bubble sort?**
   a. O(nlogn)
   b. O(logn)
   c. O(n)
   d. O(n2)
9. **The given array is arr = {1, 2, 4, 3}. Bubble sort is used to sort the array elements. How many iterations will be done to sort the array?**
   a. 4
   b. 2
   c. 1
   d. 0
10. **A system wherein items are added from one and removed from the other end.**
   a. Stack
   b. Queue
   c. Linked List
   d. Array

REVIEW QUESTIONS

1. What is sorting algorithms?
2. What do you understand by bubble sort? Explain its advantages.
3. Describe the analysis of merge sort.
4. Explain the numeric algorithm.
5. What is quick sort?
6. How insertion sort algorithm works?
7. Write short notes on:
   a. Radix Sort
   b. Shell Sort

Check Your Result

1 (d)	2. (a)	3. (c)	4. (d)	5. (c)
6. (d)	7. (b)	8. (d)	9. (a)	10. (b)

REFERENCES

1. Gaifman, Haim, 2012, "On Ontology and Realism in Mathematics," Review of Symbolic Logic, 5(3): 480–512.
2. Gigerenzer, G., and Selten, R., 2002, Bounded Rationality: The Adaptive Toolbox, Cambridge, Massachusetts: MIT Press.
3. Hemaspaandra, L., and Ogihara, M., 2002, The Complexity Theory Companion, Berlin: Springer Verlag.
4. Kahneman, D., 2003, "A Perspective on Judgment and Choice: Mapping Bounded Rationality." American Psychologist, 58(9): 697–720.
5. Kathy Sierra and Bert Bates, OCA/OCP Java 7 SE Programmer I & II Study Guide (Exams 1Z0-803 & 1Z0-804), McGraw-Hill Education, 2015.
6. Kuznets, R., 2000, "On the Complexity of Explicit Modal Logics," in Computer Science Logic, pp. 371–383, Springer.
7. Marx, M., 2007, "Complexity of Modal Logic," in P. Blackburn, J. van Benthem, & F. Wolter (eds.), Handbook of Modal Logic, pp. 140–179, Amsterdam: Elsevier.
8. Mulmuley, K., and Sohoni, M., 2001, "Geometric complexity theory I: An approach to P vs. NP and related problems," SIAM Journal on Computing, 31(2): 496–526.
9. Pudlák, Pavel, 2013, Logical Foundations of Mathematics and Computational Complexity: A Gentle Introduction, Berlin: Springer.
10. Robert Lafore, Data Structures & Algorithms in Java, Second Edition, Sams Publishing, 2003.
11. Scott, A., and Sorkin, G., 2006, "Solving sparse random instances of MAX CUT and MSC 2-CSP in linear expected time," Combinatorics, Probability and Computing, 15(1-2): 281–315.
12. Sieg, W., 2009, "On Computability," in Andrew Irvine (ed.), Philosophy of Mathematics, Vol. 4, pp. 549–630, Amsterdam: North-Holland.

CHAPTER

6

DISTRIBUTED COMPUTING

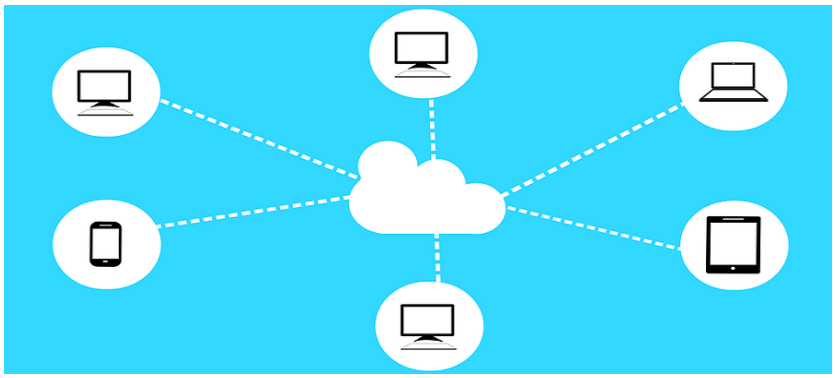
Learning Objectives

After studying this chapter, you will be able to:

1. Describe the distributed computing systems
2. Explain the remote procedure calls

INTRODUCTION

Distributed computing is the field in computer science that studies the design and behavior of systems that involve many loosely-coupled components. The components of such distributed systems may be multiple threads in a single program, multiple processes on a single machine, or multiple processors connected through a shared memory or a network. Distributed systems are unusually vulnerable to nondeterminism, where the behavior of the system as a whole or of individual components is hard to predict. Such unpredictability requires a wide range of new techniques beyond those used in traditional computing.



Like other areas in computer science, distributed computing spans a wide range of subjects from the applied to the very theoretical. On the theory side, distributed computing is a rich source of mathematically interesting problems in which an algorithm is pitted against an adversary representing the unpredictable elements of the system. Analysis of distributed algorithms often has a strong game-theoretic flavor, because executions involve a complex interaction between the algorithm's behavior and the system's responses.

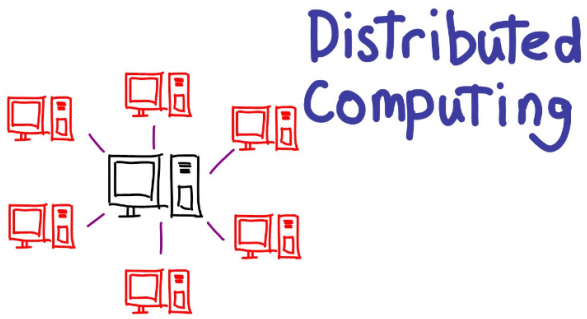


Like other areas in computer science, distributed computing spans a wide range of subjects from the applied to the very theoretical. On the theory side, distributed computing is a rich source of mathematically interesting problems in which an algorithm is pitted against an adversary representing the unpredictable elements of the system. Analysis of distributed algorithms often has a strong game-theoretic flavor, because executions involve a complex interaction between the algorithm's behavior and the system's responses.

6.1 DISTRIBUTED COMPUTING SYSTEMS: BASIC

Distributed computing is a computing concept that, in its most general sense, refers to multiple computer systems working on a single problem. In distributed computing, a single problem is divided into many parts, and each part is solved by different computers. As long as the computers are networked, they can communicate with each other to solve the problem. If done properly, the computers perform like a single entity.

The ultimate goal of distributed computing is to maximize performance by connecting users and IT resources in a cost-effective, transparent and reliable manner. It also ensures fault tolerance and enables resource accessibility in the event that one of the components fails.



A distributed system is a network of autonomous computers that communicate with each other in order to achieve a goal. The computers in a distributed system are independent and do not physically share memory or processors. They communicate with each other using *messages*, pieces of information transferred from one computer to another over a network. Messages can communicate many things: computers can tell other computers to execute a procedures with particular arguments, they can send and receive packets of data, or send signals that tell other computers to behave a certain way. Computers in a distributed system can have different roles. A computer's role depends on the goal of the system and the computer's own hardware and software properties. There are two predominant ways of organizing computers in a distributed system. The first is the client-server architecture, and the second is the **peer-to-peer** architecture.

Keyword

Peer-to-peer is a decentralized communications model in which each party has the same capabilities and either party can initiate a communication session.

Computer architectures consisting of interconnected, multiple processors are basically of two types:

Tightly coupled systems

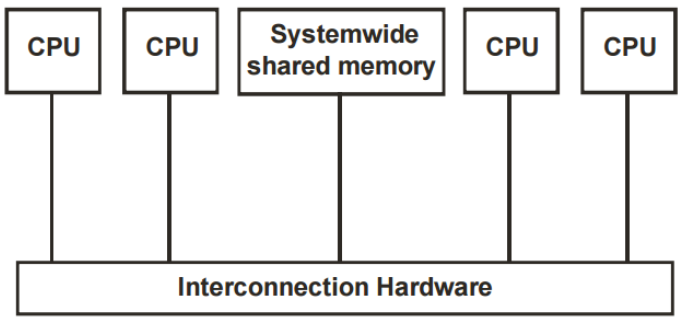
In these systems, there is a single system wide primary memory (address space) that is shared by all the processors. If any processor writes, *for example*, the value 100 to the memory location x, any other processor subsequently reading from location x will get the value 100. Therefore, in these systems, any communication between the processors usually takes place through the shared memory.

Keyword

Server is a computer program or a device that provides functionality for other programs or devices, called "clients".

Loosely coupled systems

In these systems, the processors do not share memory, and each processor has its own local memory. If a processor writes the value 100 to the memory location x, this write operation will only change the contents of its local memory and will not affect the contents of the memory. In these systems, all physical communication between the processors is done by passing messages across the network that interconnects the processors.



(a)

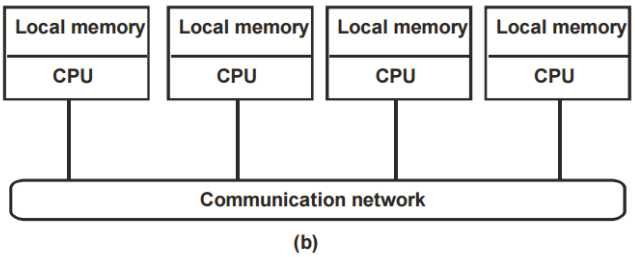


Figure 6.1: Difference between tightly coupled and loosely coupled multiprocessor systems (a) a tightly coupled multiprocessor system; (b) a loosely coupled multiprocessor system

- Tightly coupled systems are referred to as parallel processing systems, and loosely coupled systems are referred to as distributed computing systems, or simply distributed systems.
- In contrast to the tightly coupled systems, the processor of distributed computing systems can be located far from each other to cover a wider geographical area. Furthermore, in tightly coupled systems, the number of processors that can be usefully deployed is usually small and limited by the bandwidth of the shared memory. This is not the case with distributed computing systems that are more freely expandable and can have an almost unlimited number of processors.
- In short, a distributed computing system is basically a collection of processors interconnected by a communication network in which each processor has its own local memory and other peripherals, and the communication between any two processors of the system takes place by message passing over the communication network.
- For a particular processor, its own resources are local, whereas the other processors and their resources are remote. Together, a processor and its resources are usually referred to as a node or site or machine of the distributed computing system.

6.1.1 Distributed Computing System Models

Various models are used for building distributed computing systems. These models can be broadly classified into five categories – minicomputer, workstation, workstation-**server**, processor pool, and hybrid. They are briefly described below.

Minicomputer Model

The minicomputer model is a simple extension of the centralized time sharing system as shown in figure 6.2, a distributed computing system based on this model consists of a few minicomputers interconnected by a communication network. Each

minicomputer usually has multiple users simultaneously logged on to it. For this, several interactive terminals are connected to each minicomputer. Each user is logged on to one specific minicomputer, with remote access to other minicomputers. The network allows a user to access remote resources that are available on some machine other than the one on to which the user is currently logged.

- The minicomputer model may be used when resource sharing (Such as sharing of information databases of different types, with each type of database located on a different machine) with remote users is desired.
- The early ARPAnet is an example of a distributed computing system based on the minicomputer model.

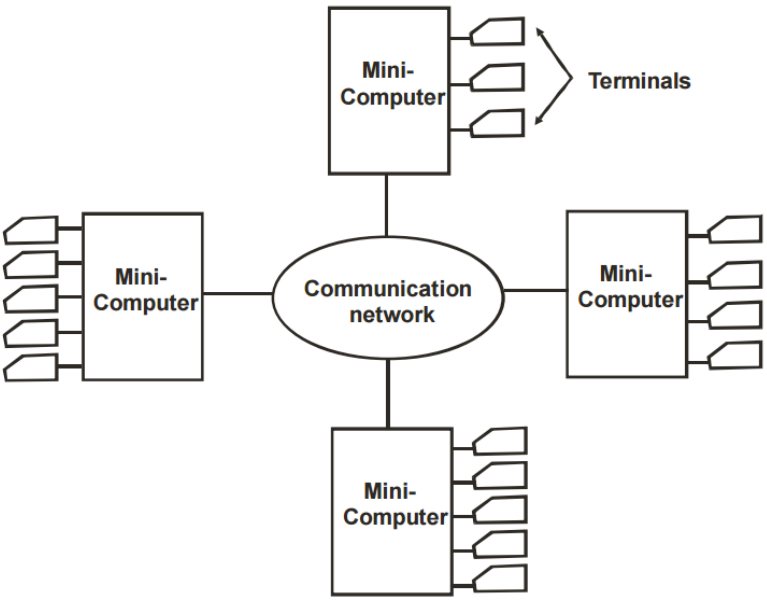


Figure 6.2: A distributed computing system based on the minicomputer model

Workstation Model

A distributed computing system based on the workstation model consists of several workstations interconnected by a communication network. A company's office or a university department may have several workstations scattered throughout a building or campus, each workstation equipped with its own disk and serving as a single-user computer.

It has been often found that in such an environment, at any one time (especially at night), a significant proportion of the workstations are idle (not being used), resulting in the waste of large amounts of CPU time. Therefore, the idea of the workstation model is to interconnect all these workstations by a high speed LAN so

that idle workstations may be used to process jobs of users who are logged onto other workstations and do not have sufficient processing power at their own workstations to get their jobs processed efficiently.

A user logs onto one of the workstations called his or her "home" workstation and submits jobs for execution. When the system finds that the user's workstation does not have sufficient processing power for executing the processes of the submitted jobs efficiently, it transfers one or more of the process from the user's workstation to some other workstation that is currently idle and gets the process executed there, and finally the result of execution is returned to the user's workstation.

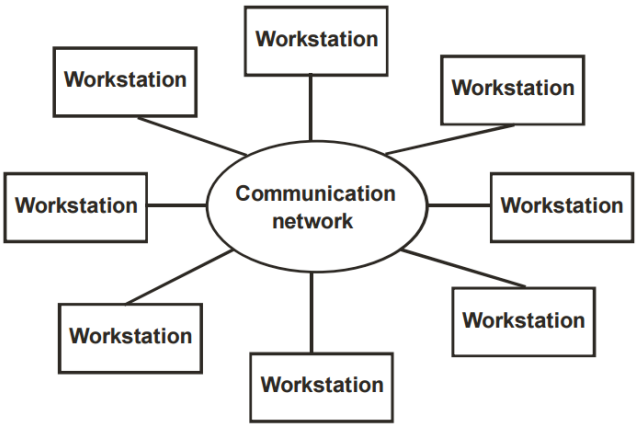


Figure 6.3: A distributed computing system based on the workstation model

Approach

Three commonly used approaches for handling the third issue are as follows:

- The first approach is to allow the remote process share the resources of the workstation along with its own logged-on user's processes. This method is easy to implement, but it defeats the main idea of workstations serving as personal computers, because if remote processes are allowed to execute simultaneously with the logged on user's own processes, the logged-on user does not get his or her guaranteed response.
- The second approach is to kill the remote process. The main drawbacks of this method are that all processing done for the remote process gets lost and the file system may be left in an inconsistent state, making this method unattractive.
- The third approach is to migrate the remote process back to its home workstation, so that its execution can be continued there. This method is

difficult to implement because it requires the system to support preemptive process migration facility.

For a number of reasons, such as higher reliability and better scalability, multiple servers are often used for managing the resources of a particular type in a distributed computing system.

For example, there may be multiple file servers, each running on a separate minicomputer and cooperating via the network, for managing the files of all the users in the system. Due to this reason, a distinction is often made between the services that are provided to clients and the servers that provide them. That is, a service is an abstract entity that is provided by one or more servers.



One or more file servers may be used in a distributed computing system to provide file service to the users.

Advantage

As compared to the workstation model, the workstation – server model has several advantages:

- In general, it is much cheaper to use a few minicomputers equipped with large, fast disks that are accessed over the network than a large number of diskful workstations, with each workstation having a small, slow disk.
- Diskless workstations are also preferred to diskful workstations from a system maintenance point of view. Backup and hardware maintenance are easier to perform with a few large disks than with many small disks scattered all over a building or campus. Installing new releases of software is easier when the software is to be installed on a few file server machines than on every workstations.
- In the workstation server model, since all files are managed by the file servers, user have the flexibility to use any workstation and access the files in the same manner irrespective of which workstation the user is currently logged on. Note that this is not true with the workstation model, in which each workstation has

its local file system, because different mechanisms are needed to access local and remote files.

- In the workstation server model, the request response protocol described above is mainly used to access the services of the server machines. Therefore, unlike the workstation model, this model does not need a process migration facility, which is difficult to implement.

  The request response protocol is known as the client-server model of communication. In this model, a client process sends a request to a server process for getting some service such as a block of a file. The server executes the request and sends back a reply to the client that contains the result of request processing.

  The client-server model provides an effective general – purpose approach to the sharing of information and resources in distributed computing systems. It is not only meant for use with the workstation – server model but also can be implemented in a variety of hardware and software environments. The computers used to run the client and server processes need not necessarily be workstations and minicomputers. They can be of many types and there is no need to distinguish between them. It is even possible for both the client and server processes to be run on the same computer.

- A user has guaranteed response time because workstations are not used for executing remote processes. However, the model does not utilize the processing capability of idle **workstations**.

Keyword

Workstation is a special computer designed for technical or scientific applications.

Processor Pool Model

The processor pool model is based on the observation that most of the time a user does not need any computing power but once in a while he or she may need a very large amount of computing power for a short time. (e.g., when recompiling a program consisting of a large number of files after changing a basic shared declaration). Therefore, unlike the workstation server model in which a processor is allocated to each user, in the processor-pool model the processors are pooled together to be shared by the users as needed. The pool of processors consists of a large number of microcomputers and minicomputers attached

to the network. Each processor in the pool has its own memory to load and run a system program or an application program of the distributed computing system.

Hybrid Model

Out of the four models described above, the workstation server model, is the most widely used model for building distributed computing systems. This is because a large number of computer users only perform simple interactive tasks such as editing jobs, sending electronic mails, and executing small programs. The workstation-server model is ideal for such simple usage. However, in a working environment that has groups of users who often perform jobs needing massive computation, the processor-pool model is more attractive and suitable.

To continue the advantages of both the workstation-server and processor-pool models, a hybrid model may be used to build a distributed computing system. The hybrid model is based on the workstation-server model but with the addition of a pool of processors. The processors in the pool can be allocated dynamically for computations that are too large for workstations or that requires several computers concurrently for efficient execution. In addition to efficient execution of computation-intensive jobs, the hybrid model gives guaranteed response to interactive jobs by allowing them to be processed on local workstations of the users. However, the hybrid model is more expensive to implement than the workstation – server model or the processor-pool model.

6.1.2 Transparency

A distributed system that is able to present itself to user and application as if it were only a single computer system is said to be transparent. There are eight types of transparencies in a distributed system:

Access Transparency

It hides differences in data representation and how a resource is accessed by a user. Example, a distributed system may have a computer system that runs different operating systems, each having their own file naming conventions. Differences in naming conventions as well as how files can be manipulated should be hidden from the users and applications.

Location Transparency

Hides where exactly the resource is located physically. Example, by assigning logical names to resources like yahoo.com, one cannot get an idea of the location of the web page's main server.

Migration Transparency

Distributed system in which resources can be moved without affecting how the resource can be accessed are said to provide migration transparency. It hides that the resource may move from one location to another.

Relocation Transparency

This transparency deals with the fact that resources can be relocated while it is being accessed without the user who is using the application to know anything. Example: using a Wi-Fi system on laptops while moving from place to place without getting disconnected.

Replication Transparency

Hides the fact that multiple copies of a resource could exist simultaneously. To hide replication, it is essential that the replicas have the same name. Consequently, as system that supports replication should also support location transparency.

Concurrency Transparency

It hides the fact that the resource may be shared by several competitive users. Example, two independent users may each have stored their file on the same server and may be accessing the same table in a shared database. In such cases, it is important that each user does not notice that the others are making use of the same resource.

Failure Transparency

Hides failure and recovery of the resources. It is the most difficult task of a distributed system and is even impossible when certain apparently realistic assumptions are made. Example: A

Keyword

Network is defined as a group of two or more computer systems linked together.

user cannot distinguish between a very slow or dead resource. Same error message come when a server is down or when the network is overloaded of when the connection from the client side is lost. So here, the user is unable to understand what has to be done, either the user should wait for the **network** to clear up, or try again later when the server is working again.

Persistence Transparency

It hides if the resource is in memory or disk. Example, Object oriented database provides facilities for directly invoking methods on storage objects. First the database server copies the object states from the disk i.e. main memory performs the operation and writes the state back to the disk. The user does not know that the server is moving between primary and secondary memory.



In 1965, IBM introduced the IBM 1130 scientific computer, which was meant as the successor to the 1620. Both of these systems came with the ability to run programs written in Fortran and other languages.

Persistence	Hide whether a (software) resource is in memory or on disk
Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource may be shared by several competitive users
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource

Performance Transparency

The aim of performance transparency is to allow the system to be automatically reconfigured to improve performance, as loads vary dynamically in the system. As far as practicable, a situation in which one processor of the system is overloaded with jobs while another processor is idle should not be allowed to occur. That is, the processing capability of the system should be uniformly distributed among the currently available jobs in the system.

This requirements calls for the support of intelligent resource allocation and process migration facilities in distributed operating systems.

Scaling Transparency

The aim of scaling transparency is to allow the system to expand in scale without disrupting the activities of the users. This requirement calls for open-system architecture and the use of scalable algorithms for designing the distributed operating system components.

6.1.3 Reliability

In general, distributed systems are expected to be more reliable than centralized systems due to the existence of multiple instances of resources. However, the existence of multiple instances of the resources alone cannot increase the system's reliability. Rather, the distributed operating system, which manages these resources must be designed properly to increase the system's reliability by taking full advantage of this characteristic feature of a distributed system.

A fault is a mechanical or algorithmic defect that may generate an error. A fault in a system causes system failure. Depending on the manner in which a failed system behaves, system failures are of two types – fail stop and Byzantine. In the case of fail-step failure, the system stops functioning after changing to a state in which its failure can be detected. On the other hand, in the case of Byzantine failure, the system continues to function but produces wrong results. Undetected software bugs often cause Byzantine failure of a system. Obviously, Byzantine failures are much more difficult to deal with than fail-stop failures.

Commonly used methods for dealing with these issues are briefly described text.

Remember

For higher reliability, the fault-handling mechanisms of a distributed operating system must be designed properly to avoid faults, to tolerate faults, and to detect and recover form faults.

Fault Avoidance

Fault avoidance deals with designing the components of the system in such a way that the occurrence of faults in minimized. Conservative design practice such as using high reliability

components are often employed for improving the system's reliability based on the idea of fault avoidance. Although a distributed operating system often has little or no role to play in improving the fault avoidance capability of a hardware component, the designers of the various software components of the distributed operating system must test them thoroughly to make these components highly reliable.

Keyword

Operating system is system software that manages computer hardware and software resources and provides common services for computer programs.

Fault Tolerance

Fault tolerance is the ability of a system to continue functioning in the event of partial system failure. The performance of the system might be degraded due to partial failure, but otherwise the system functions properly. Some of the important concepts that may be used to improve the fault tolerance ability of a distributed **operating system** are as follows:

Redundancy techniques

The basic idea behind redundancy techniques is to avoid single points of failure by replicating critical hardware and software components, so that if one of them fails, the others can be used to continue. Obviously, having two or more copies of a critical component makes it possible, at least in principle, to continue operations in spite of occasional partial failures.

For example, a critical process can be simultaneously executed on two nodes so that if one of the two nodes fails, the execution of the process can be completed at the other node. Similarly, a critical file may be replicated on two or more storage devices for better reliability.

Notice that with redundancy techniques additional system overhead is needed to maintain two or more copies of a replicated resource and to keep all the copies of a resource consistent.

For example, if a file is replicated on two or more nodes of a distributed system, additional disk storage space is required and for correct functioning, it is often necessary that all the copies of the file are mutually consistent. In general, the larger is the number of copies kept, the better is the reliability but the incurred overhead involved. This raises an important question: How much replication is enough? For an answer to this question,

note that a system is said to be k-fault tolerant if it can continue to function even in the event of the failure of k components. Therefore, if the system is to be designed to tolerance k fail – stop failures, k + 1 replicas are needed. If k replicas are lost due to failures, the remaining one replica can be used for continued functioning of the system. On the other hand, if the system is to be designed to tolerance k Byzantine failures, a minimum of 2k + 1 replicas are needed. This is because a voting mechanism can be used to believe the majority k + 1 of the replicas when k replicas behave abnormally.

Another application of redundancy technique is in the design of a stable storage device, which is a virtual storage device that can even withstand transient I/O faults and decay of the storage media. The reliability of a critical file may be improved by storing it on a stable storage device.

Remember

A distributed operating system must be designed to maintain a proper balance between the degree of reliability and the incurred overhead.

Distributed control

For better reliability, many of the particular algorithms or protocols used in a distributed operating system must employ a distributed control mechanism to avoid single points of failure.

For example, a highly available distributed file system should have multiple and independent file servers controlling multiple and independent storage devices. In addition to file servers, a distributed control technique could also be used for name servers, scheduling algorithms, and other executive control functions. It is important to note here that when multiple distributed servers are used in a distributed system to provide a particular type of service, the servers must be independent. That is, the design must not require simultaneous functioning of the servers; otherwise, the reliability will become worse instead of getting better.

Fault Detection and Recovery

The faulty detection and recovery method of improving reliability deals with the use of hardware and software mechanisms to determine the occurrence of a failure and then to correct the system to a state acceptable for continued operation. Some of the commonly used techniques for implementing this method in a distributed operating system are as follows.

Atomic transactions

An atomic transaction is a computation consisting of a collection of operation that take place indivisibly in the presence of failures and concurrent computations. That is, either all of the operations are performed successfully or none of their effects prevails, other processes executing concurrently cannot modify or observe intermediate states of the computation. Transactions help to preserve the consistency of a set of shared date objects (e.g. files) in the face of failures and concurrent access. They make crash recovery much easier, because transactions can only end in two states: Either all the operations of the transaction are performed or none of the operations of the transaction is performed. In a system with transaction facility, if a process halts unexpectedly due to a hardware error before a transaction is completed, the system subsequently restores any data objects that were undergoing modification to their original states. Notice that if a system does not support a transaction mechanism, unexpected failure of a process during the processing of an operation may leave the data objects that were undergoing modification in an inconsistent state. Therefore, without transaction facility, it may be difficult or even impossible in some cases to roll back (recover) the data objects from their current inconsistent states to their original states.

Remember

Although stateful service becomes necessary in some cases, to simplify the failure detection and recovery actions, the stateless service paradigm must be used, wherever possible.

Stateless servers

The client-server model is frequently used in distributed systems to service user requests. In this model, a server may be implemented by using any one of the following two service paradigms – stateful or stateless. The two paradigms are distinguished by one aspect of the client – server relationship, whether or not the history of the serviced requests between a client and a server affects the execution of the next service request. The stateful approach does depend on the history of the serviced requests, but the stateless approach does not depend on it. Stateless servers have a distinct advantage over stateful servers in the event of a failure. That is, the stateless service paradigm makes crash recovery very easy because no client state information is maintained by the server. On the other hand, the stateful service paradigm requires complex crash recovery procedures. Both the client and server need to reliably detect

crashes. The server needs to detect client crashes so that it can discard any state it is holding for the client, and the client must detect server crashes so that it can perform necessary error – handling activities.

Acknowledgments and timeout-based retransmission of messages

In a distributed system, events such as a node crash or a communication link failure may interrupt a communication that was in progress between two processes, resulting in the loss of a message. Therefore, a reliable interposes communication mechanism must have ways to detect lost messages so that they can be retransmitted. Handling of lost messages usually involves return of acknowledgment messages and retransmissions on the basis of timeouts. That is, the receiver must return an acknowledgment message for every message received, and if the sender does not receive any acknowledgement for a message within a fixed timeout period, it assumes that the message was lost and retransmits the message. A problem associated with this approach is that of duplicate message. Duplicates messages may be sent in the event of failures or because of timeouts. Therefore, a reliable interposes communication mechanism should also be capable of detecting and handling duplicate messages. Handling of duplicate messages usually involves a mechanism for automatically generating and assigning appropriate sequence numbers to messages. Use of acknowledgement messages, timeout-based retransmissions of messages, and handling of duplicate request messages for reliable communication.

6.1.4 Flexibility

Another important issue in the design of distributed operating systems is flexibility. Flexibility is the most important features for open distributed systems. The design of a distributed operating system should be flexible due to the following reasons:

Ease of modification

From the experience of system designers, it has been found that some parts of the design often need to be replaced / modified either because some bug is detected in the design or because the design is no longer suitable for the changed system environment or new-user requirements. Therefore, it should be easy to incorporate changes in the system in a user-transparent manner or with minimum interruption caused to the users.

Ease of enhancement

In every system, new functionalities have to be added from time to time it more

powerful and easy to use. Therefore, it should be easy to add new services to the system. Furthermore, if a group of users do not like the style in which a particular service is provided by the operating system, they should have the flexibility to add and use their own service that works in the style with which the users of that group are more familiar and feel more comfortable.

The most important design factor that influences the flexibility of a distributed operating system is the model used for designing its kernel. The kernel of an operating system is its central controlling part that provides basic system facilities. It operates in a separate address space that is inaccessible to user processes. It is the only part of an operating system that a user cannot replace or modify. We saw that in the case of a distributed operating system identical kernels are run on all the nodes of the distributed system.

Keyword

File management system is used for file maintenance (or management) operations. It is is a type of software that manages data files in a computer system.

The two commonly used models for kernel design in distributed operating systems are the monolithic kernel and the microkernel. In the monolithic kernel model, most operating system services such as process management, memory management, device management, **file management,** name management, and inter-process communication are provided by the kernel. As a result, the kernel has a large, monolithic structure. Many distributed operating systems that are extensions or limitations of the UNIX operating system use the monolithic kernel model. This is mainly because UNIX itself has a large, monolithic kernel.

6.1.5 Performance

If a distributed system is to be used its performance must be at least as good as a centralized system. That is, when a particular application is run on a distributed system, its overall performance should be better than or at least equal to that of running the same application on a single processor system. However, to achieve his goal, it is important that the various components of the operating system of a distributed system be designed properly; otherwise, the overall performance of the distributed system may turn out to be worse than a centralized system. Some design principles considered useful for better performance are as follows:

- Batch if possible, Batching often helps in improving performance greatly. *For example,* transfer of data across the network in large chunks rather than as individual pages is much more efficient. Similarly, piggybacking of acknowledgement of previous messages with the next message during a series of messages exchanged between two communicating entities also improves performance.
- Cache whenever possible: Caching of data at clients' sites frequently improves overall system performance because it makes data available wherever it is being currently used, thus saving a large amount of computing time and network bandwidth. In addition, caching reduces contention on centralized resources.
- Minimize copying of data: Data copying overhead (e.g. moving data in and out of buffers) involves a substantial CPU cost of many operations. *For example*, while being transferred from its sender to its receiver, a message data may take the following path on the sending side
  - From sender's stack to its message buffer
  - From the message buffer in the sender's address space to the message buffer in the kernel's address space
  - Finally, from the kernel to the network interface board

  On the receiving side, the data probably takes a similar path in the reverse direction. Therefore, a total of six copy operations are involved in the message transfer operation. Similarly, in several systems, the data copying overhead is also large for read and write operations on block I/O devices. Therefore, for better performance, it is desirable to avoid copying of data, although this is not always simple to achieve. Making optimal use of memory management often helps in eliminating much data movement between the kernel, block I/O devices, clients, and servers.
- Minimize network traffic: System performance may also be improved by reducing internode communication costs. *For example*, accesses to remote resources require communication, possibly through intermediate nodes. Therefore, migrating a process closer to the resources it is using most heavily may be helpful in reducing network traffic in the system if the decreased cost of accessing its favorite resource offsets the possible increased post of accessing its less favored ones. Another way to reduce network traffic is to use the process migration facility to cluster two or more processes that frequently communicate with each other on the same node of the system. Avoiding the collection of global state information for making some decision also helps in reducing network traffic.
- Take advantage of fine-grain parallelism for multiprocessing. Performance

can also be improved by taking advantage of finegiam parallelism for multiprocessing. *For example*, threads are often used for structuring server processes. Servers structured as a group of threads can operate efficiently, because they can simultaneously service requests from several clients. Finegrained concurrency control of simultaneous accesses by multiple processes, to a shared resource is another example of application of this principle for better performance.

6.2 REMOTE PROCEDURE CALLS

A remote procedure call (RPC) is an inter-process communication that allows a computer program to cause a procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction.

- It further aims at hiding most of the intricacies of message passing and is idle for client-server application.
- Information can be transported in the form of parameters and can come back in procedure result. No message passing is visible to the programmer. As calling and called procedures exist on different machines, they execute in different address spaces, the parameters and result should be identical and if machines crash during communication, it causes problems.

6.2.1 RPC Operations

RPC allows programs to call procedures located on other machines. But the procedures 'send' and 'receive' do not conceal the communication which leads to achieving access transparence in distributed systems.

Conventional procedure call

For a call of a program, an empty stack is present to make the call, the caller pushes the parameters onto the stack (last one first order). After the read has finished running, it puts the return values in a register and removes the return address and transfers controls back to the caller. Parameters can be called by value or reference.

Call by Value

Here the parameters are copied into the stack. The value parameter is just an initialized local variable. The called procedure may modify the variable, but such changes do not affect the original value at the calling side.

Call by reference

It is a pointer to the variable. In the call to Read, the second parameter is a reference parameter. It does not modify the array in the calling procedure.

Call-by-copy

Another parameter passing mechanism exists along with the above two, it is called call-by-copy or Restore. Here the caller copies the variable to the stack and then copies the variable to the stack and then copies it back after the call, overwriting the caller's original values. The decision of which parameter passing mechanism to use is normally made by the language designers and is a fixed property of the language. Sometimes it depends on the data type being passed.

Client and Server Stubs

There are some client and server stubs, such as:

- A stub in distributed computing is a piece of code used for converting parameters passed during a Remote Procedure Call.
- The main idea of an RPC is to allow a local computer (client) to remotely call procedures on a remote computer (server). The client and server use different address spaces, so conversion of parameters used in a function call have to be performed; otherwise the values of those parameters could not be used, because of pointers to the computer's memory pointing to different data on each machine.
- The client and server may also use different data representations even for simple parameters. Stubs are used to perform the conversion of the parameters, so a Remote Function Call looks like a local function call for the remote computer.

For transparency of RPC, the calling procedure should not know that the called procedure is executing on a different machine.

- Client Stub: Used when read is a remote procedure. Client stub is put into a library and is called using a calling sequence. It calls for the local operating system. It does not ask for the local operating system to give data, it asks the server and then blocks itself till the reply comes.
- Server Stub: when a message arrives, it directly goes to the server stub. Server stub has the same functions as the **client stub**. The stub here unpacks the parameters from the message and then calls the server procedure in the usual way.

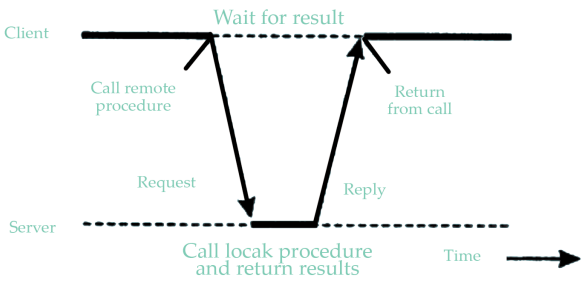


Figure 6.4: Principle of RPC between a client and server program.

Keyword

Client stub (sometimes called proxy) is responsible for conversion (marshalling) of parameters used in a function call and deconversion of results passed from the server after execution of the function.

6.2.2 Transparency of RPC

A major issue in the design of an RPC facility is its transparency property. A transparent RPC mechanism is one in which local procedures and remote procedures are (effectively) indistinguishable to programmers. This requires the following two types of transparencies:

- Syntactic transparency means that a remote procedures call should have exactly the same syntax as a local procedure call.
- Semantic transparency means that the semantics of a remote procedure call are identical to those of a local procedure call.

It is not very difficult to achieve syntactic transparency of an RPC mechanism, and we have seen that the semantics of remote procedure calls are also analogous to that of local procedure calls for most parts:

- The calling process is suspended until the called procedure returns.
- The caller can pass arguments to the called procedure (remote procedure).
- The called procedure (remote procedure) can return results to the caller.

Unfortunately, achieving exactly the same semantics for remote procedure calls as for local procedure calls is close to impossible. This is mainly because of the following differences between remote procedure calls and local procedure calls.

- Unlike local procedure calls, with remote procedure calls the called procedure is executed in an address space that is disjoint from the calling program's address space. Due to this reason, the called (remote) procedure cannot have access to any variables or data values in the calling program's environment. Thus in the absence of shared memory, it is meaningless to pass addresses in arguments, making call-by-reference pointers highly unattractive. Similarly, it is meaningless to pass argument values containing pointer structures (e.g., linked lists), since pointers are normally represented by memory addresses.

  An alternative implementation is to send a copy of the value pointed at the receiver, but this has subtly different semantics and may be difficult to implement if the pointer points into the middle of a complex data structure, such as a directed graph. Similarly, call by reference can be replaced by copy in / copy out, but at the cost of slightly different semantics.
- Remote procedure calls are more vulnerable to failure than local procedure calls, since they involve two different processes and possibly a network and two different computers. Therefore programs that make use of remote procedure calls must have the capability of handling even those errors that cannot occur in local procedure calls. The need for the ability to take care of the possibility of processor crashes and communication problems of a network makes it even more difficult to obtain the same semantics for remote procedure calls as for local procedure calls.
- Remote procedure calls consume much more time than local procedure calls. This is mainly due to the involvement of a communication network in RPCs. Therefore applications using RPCs must also have the capability to handle the long delays that may possibly occur due to network congestion.

Because of these difficulties in achieving normal call semantics for remote procedure calls, some researchers feel that the RPC facility should be nontransparent. *For example*, Hamilton [1984] argues that remote procedures should be treated differently from local procedures from the start, resulting in a nontransparent RPC mechanism. Similarly, the designers of RPC were of the opinion that although the RPC system should hide low-level details of message passing from the users, failures and long delays should not be hidden from the caller. That is, the caller should have the flexibility of handling failures and long delays in an application – dependent manner. Although in most environments total semantic transparency is impossible, enough can be done to ensure that distributed application programmers feel comfortable.

6.2.3 Implementing RPC Mechanism

To achieve the goal of semantic transparency, the implementation of an RPC mechanism is based on the concept of stubs, which provide a perfectly normal (local) procedure

call abstraction by concealing from programs the interface to the underlying RPC system. We saw that an RPC involves a client process and a server process.

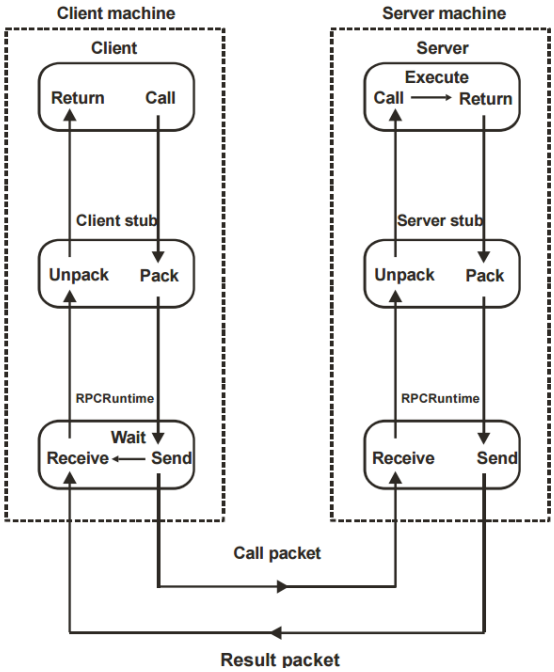


Figure 6.5: Implementation of RPC mechanism

Therefore, to conceal the interface of the underlying RPC system from both the client and server processes, a separate stub procedure is associated with each of the two processes. Moreover, to hide the existence and functional details of the underlying network, an RPC communication package is used on both the client and server sides. Thus, implementation of an RPC mechanism usually involves the following five elements of program:

- The client
- The client stub
- The RPCRuntime
- The server stub
- The server

Client

The client is a user process that intitiates a remote procedure call. To make a remote procedure call, the client makes a perfectly normal local call that invokes a corresponding procedure in the client stub.

Client Stub

The client stub is responsible for carrying out the following two tasks:

- On receipt of a call request from the client, it packs a specification of the target procedure and the arguments into a message and then asks the local RPC Runtime to send it to the server stub.
- On receipt of the result of procedure execution, it unpacks the result and passes it to the client.

RPCRuntime

The RPCRuntime handles transmission of messages across the network between client and server machines. It is responsible for retransmissions, acknowledgements, packet routing, and encryption. The RPCRuntime on the client machine receives the call request message from the client stub and sends it to the server machine. It also receives the message containing the result of procedure execution from the server machine and passes it to the client stub. On the other hand, the RPCRuntime on the server machine receives the message containing the result of procedure execution from the server stub and sends it to the client machine. It also receives the call request message from the client machine and passes it to the server stub.

Server Stub

The job of the server stub is very similar to that of the client stub. It performs the following two tasks:

- On the receipt of the call request message from the local RPCRuntime, the server stub unpacks it and makes a perfectly normal call to invoke the appropriate procedure in the server.
- On receipt of the result of procedure execution from the server, the server stub packs the result into a message and then asks the local RPCRuntime to send it to the client stub.

Server

On receiving a call request from the server stub, the server executes the appropriate procedure and returns the result of procedure execution to the server stub.

Note here that the beauty of the whole scheme is the total ignorance on the part of the client that the work was done remotely instead of by the local kernel. When the client gets control following the procedure call that it made, all it knows is that the results of the procedure execution are available to it. Therefore, as far

as the client is concerned, remote services are accessed by making ordinary (local) procedure calls, not by using the send and receive primitives. All the details of the message passing are hidden in the client and server stubs, making the steps involved in message passing invisible to both the client and the server.

6.2.4 STUB Generation

Stubs can be generated in one of the following two ways:

- *Manually:* In this method, the RPC implement or provides a set of translation functions from which a user can construct his or her own stubs. This method is simple to implement and can handle very complex parameter types.
- *Automatically*: This is the more commonly used method for stub generation. It uses Interface Definition Language (IDL) that is used to define the interface between a client and a server. An interface definition is mainly a list of procedure names supported by the interface, together with the types of their arguments and results. This is sufficient information for the client and server to independently perform compile-time type checking and to generate appropriate calling sequences. However, an interface definition also contains other information that helps RPC reduce data storage and the amount of data transferred over the network. For example, an interface definition has information to indicate whether each argument is input, output, or both – only input arguments need be copied from client to server and only output arguments need be copied from server to client. Similarly, an interface definition also has information about type definitions, enumerated types, and defined constants that each side uses to manipulate data from RPC calls making it unnecessary for both the client and the server to store this information separately.

A server program that implements procedures in an interface is said to export the interface and a client program that calls procedures from an interface is said to import the interface. When writing a distributed application, a programmer first writes an interface definition using the IDL. He or she can then write the client program that imports the interface and the server program that exports the interface. The interface definition is processed using an IDL computer to generate components that can be combined with client and server programs, without making any changes to the existing compliers.

In particular, from an interface definition, an IDL complier generate a client stub procedure and a server such procedure for each procedure is the interface, the appropriate marshaling and un-marshaling operations in each stub procedure, and a header file that supports the data types in the interface definition. The header file is included in the source files of both the client and server programs, the client stub

procedures are compiled and linked with the client program, and the server stub procedures are compiled and linked with the server program. An IDL compiler can be designed to process interface definitions for use with different languages, enabling clients and servers written in different languages, to communicate by using remote procedure calls.

6.2.5 RPC Messages

Any remote procedure call involves a client process and a server process that are possibly located on different computers. The mode of interaction between the client and server is that the client asks the server to execute a remote procedure and the server returns the result of execution of the concerned procedure to the client. Based on this mode of interaction, the two types of messages involved in the implementation of an RPC system are as follows:

- Call messages that are sent by the client to the server for requesting execution of a particular remote procedure.
- Reply messages that are sent by the server to the client for returning the result of remote procedure execution.

The protocol of the concerned RPC system defines the format of these two types of message. Normally, an RPC protocol is independent of transport protocols. That is, RPC does not care how a message is passed from one process to another. Therefore an RPC protocol deals only with the specification and interpretation of these two types of messages.

Call Messages

Since a call message is used to request execution of a particular remote procedure the two basic components necessary in a call message are as follows:

- The identification information of the remote procedure to be executed.
- The arguments necessary for the execution of the procedure. In addition to these two fields, a call message normally has the following fields.
- A message identification field that consists of a sequence number. This field is useful of two ways – for identifying lost messages and duplicate messages in case of system failures and for properly matching reply messages to outstanding call messages, especially in those cases when the replies of several outstanding call messages arrive out of order.
- A message type field that is used to distinguish call messages from reply messages. For example, in an RPC system, this field may be set to 0 for all call messages and set to 1 for all reply messages.

- A client identification field that may be used for two purposes – to allow the server of the RPC to identify the client to whom the reply message has to be returned and to allow the server to check the authentication of the client process for executing the concerned procedure.

Reply Messages

When the server of an RPC receives a call message from a client, it could be faced with one of the following conditions. It is assumed for a particular condition that no problem was detected by the server for any of the previously listed conditions:

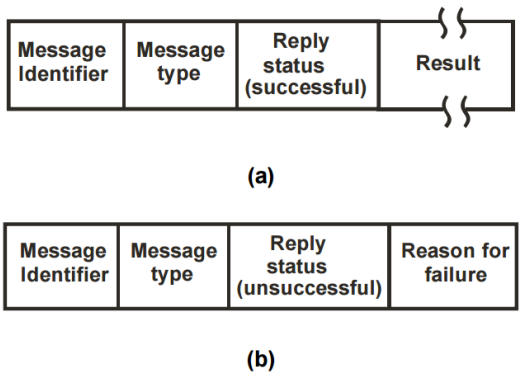


Figure 6.6: A typical RPC reply message format: (a) a successful reply message format; (b) an unsuccessful reply message format

Remember

The marshaling process must reflect the structure of all types of program objects used in the concerned language.

6.2.6 Marshaling Arguments and Results

Implementation of remote procedure calls involves the transfer of arguments from the client process to the server process and the transfer of results from the server process to the client process. These arguments and results are basically language-level data structures (program objects), which are transferred in the form of message data between the two computers involved in the call. The transfer of message data between two computers requires encoding and decoding of the message data. For RPC this operation is known as marshaling and basically involves the following actions.

- Taking the arguments (of a client process) or the result (of a server process) that will form the message data to be set to the remote process.
- Encoding the message data of step 1 above on the sender's computer. This encoding process involves the conversion of program objects into a stream form that is suitable for transmission and placing them into a message buffer.
- Decoding of the message data on the receiver's computer. This decoding process involves the reconstruction of program objects from the message data that was received in stream form.

In order that encoding and decoding of an RPC message can be performed successfully, the order and the representation method (tagged or untagged) used to marshal arguments and results must be known to both the client and the server of the RPC. This provides a degree of type safety between a clients a server because the server will not accept a call from a client until the client uses the same interface definition as the server. Type safety is of particular importance to servers since it allows them to survive against corrupt call requests. These include primitive types, structured types, and user defined types. Marshaling procedures may be classified into two groups:

- Those provided as a part of the RPC software. Normally marshaling procedures for scalar data types, together with procedures to marshal compound types built from the scalar ones, fall in this group.
- Those that are defined by the users of the RPC system. This group contains marshaling procedures for user – defined data types and data types that include pointers. *For example*, in Concurrent CLU, developed for use in the Cambridge Distributed Computer System, for user-defined types, the type definition must contain procedures for marshaling.

A good RPC system should always generate in-line marshaling code for every remote call so that the users are relieved of the burden of writing their own marshaling procedures. However, practically it is difficult to achieve this goal because of the unacceptable large amounts of code that may have to be generated for handling all possible data types.

6.2.7 Server Management

In RPC based applications, two important issues that need to be considered for every management are server implementation and server creation.

Server Implementation

Based on the style of implementation used, servers may be of two types: stateful and stateless.

Stateful Servers

A stateful server maintains clients' state information from one remote procedure call to the next. That is, in case of two subsequent calls by a client to a stateful server, some state information pertaining to the service performed for the client as a result of the first call execution is stored by the server process. These clients' state information is subsequently used at the time of executing the second call.

For example, let us consider a server for byte-stream files that allows the following operations on files:

- *Open (filename, mode):* This operation is used to open a file identified by filename in the specified mode. When the server executes this operation, it creates an entry for this file in a file-table that it uses for maintaining the file state information of all the open files. The file state information normally consists of the identifier of the file, the open mode, and the current position of a nonnegative integer pointer, called the read write pointer. When a file is opened, its read-write pointer is set to zero and the server returns to the client a file identifier (fid), which is used by the client for subsequent accesses to that file.
- *Read (fid, n, buffer):* This operation is used to get n bytes of data from the file identified by fid into the buffer named buffer. When the server executes this operation, it returns to the client n bytes of file data starting from the byte currently addressed by the read – write pointer and then increments the read – write pointer by n.
- *Write (fid, n, buffer):* On execution of this operation, the server takes n bytes of data from the specified buffer, writes it into the file identified by fid at the byte position currently addressed by the read – write pointer, and then increments the read – write pointer by n.
- *Seek (fid, position):* This operation causes the server to change the value of the read write pointer of the file identified by fid to the new value specified as position.
- *Close (fid):* This statement causes the server to delete from its file table the file state information of the file identified by fid.

The file server mentioned above is stateful because it maintains the current state information for a file that has been opened for use by a client. Therefore, as shown in figure 6.7, after opening a file, if a client makes two subsequent Read (fig, 100, buf), calls, the first call will return the first 100 bytes (bytes 0 – 99) and the second call will return the next 100 bytes (bytes 100 – 199).

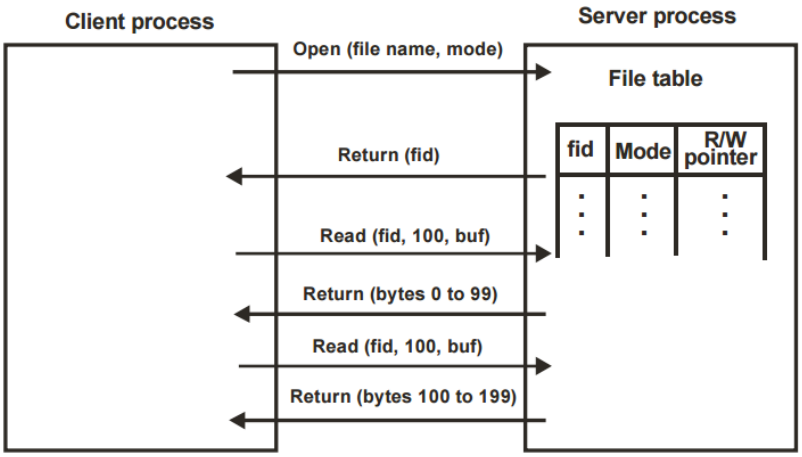


Figure 6.7: An example of a stateful file server

To keep track of the current record position for each client that has opened the file for accessing. Therefore to design an idempotent interface for reading the next record from the file, it is important that each client keeps track of its own current record position and the server is made stateless, that is, no client state should be maintained on the server side. Based on this idea, an idempotent procedure for reading the next record from a sequential file is

ReadRecordN (Filename, N)

which returns the Nth record from the specified file. In this case, the client has to correctly specify the value of n to get desired record from the file.

However, not all non-idempotent interfaces can be so easily transformed to an idempotent form.

For example, consider the following procedure for appending a new record to the same sequential file.

AppendRecord (Filename, Record)

It is clearly not idempotent since repeated execution will add further copies of the same record to the file. This interface may be converted into an idempotent interface by using the following two procedures instead of the one defined above:

GetLastRecordNo (Filename)
WriteRecordN (Filename, Record, N)

The first procedure returns the record number of the last record currently in the file, and the second procedure writes a record at specified in the file. Now, for appending a record, the client will have to use the following two procedures:

```
Last = GetLastRecordNo (Filename)
WriteRecordN (Filename, Record, Last)
```

For exactly-once semantics, the programmer is relieved of the burden of implementing the server procedure in an idempotent manner because the call semantics itself takes care of executing the procedure only once. The implementation of exactly-once call semantics is based on the use of timeouts, retransmissions, call identifiers with the same identifier for repeated calls, a reply cache associated with the callee.

SUMMARY

- Distributed computing is a rich source of mathematically interesting problems in which an algorithm is pitted against an adversary representing the unpredictable elements of the system. Analysis of distributed algorithms often has a strong game-theoretic flavor, because executions involve a complex interaction between the algorithm's behavior and the system's responses.
- The ultimate goal of distributed computing is to maximize performance by connecting users and IT resources in a cost-effective, transparent and reliable manner. It also ensures fault tolerance and enables resource accessibility in the event that one of the components fails.
- A distributed computing system based on the workstation model consists of several workstations interconnected by a communication network. A company's office or a university department may have several workstations scattered throughout a building or campus, each workstation equipped with its own disk and serving as a single-user computer.
- A distributed system that is able to present itself to user and application as if it were only a single computer system is said to be transparent. There are eight types of transparencies in a distributed system:
- Fault tolerance is the ability of a system to continue functioning in the event of partial system failure. The performance of the system might be degraded due to partial failure, but otherwise the system functions properly.
- Decoding of the message data on the receiver's computer. This decoding process involves the reconstruction of program objects from the message data that was received in stream form.
- Decoding of the message data on the receiver's computer. This decoding process involves the reconstruction of program objects from the message data that was received in stream form.

KNOWLEDGE CHECK

1. **In distributed system each processor has its own**
   a. local memory
   b. clock
   c. both (a) and (b)
   d. none of the mentioned
2. **Network operating system runs on**
   a. server
   b. every system in the network

c. both (a) and (b)
d. none of the mentioned

3. Which technique is based on compile-time program transformation for accessing remote data in a distributed-memory parallel system.

a. cache coherence scheme
b. computation migration
c. remote procedure call
d. message passing

4. In distributed systems, a logical clock is associated with

a. each instruction
b. each process
c. each register
d. none of the mentioned

5. In distributed systems, transaction coordinator

a. starts the execution of transaction
b. breaks the transaction into number of sub transactions
c. coordinates the termination of the transaction
d. all of the mentioned

6. In distributed systems, link and site failure is detected by ……..

a. polling
b. handshaking
c. token passing
d. none of the mentioned

7. Which routing technique is used in a distributed system?

a. fixed routing
b. virtual routing
c. dynamic routing
d. all of the mentioned

8. Logical extension of computation migration is ……..

a. process migration
b. system migration
c. thread migration
d. data migration

9. **To improve performance, many distributed systems……..**
   a. combine check pointing with message logging
   b. combine check pointing with recovery oriented computing
   c. All of above
   d. None of above
10. **The dynamic replication algorithm takes into account……..**
   a. files on server can be migrated anywhere
   b. schedule process migration
   c. to reduce load on server
   d. resource sharing

REVIEW QUESTIONS

1. What is distributed computing with example?
2. What are the types of distributed computing system models?
3. What is remote procedure call?
4. Explain the RPC messages.
5. What is a stateful system?
6. Write the short note on:
   a. Fault tolerance
   b. Fault avoidance

Check Your Result

1. (c)	2. (a)	3. (b)	4. (b)	5. (d)
6. (b)	7. (d)	8. (a)	9. (a)	10. (c)

REFERENCES

1. Tanenbaum and M. Van Steen, Distributed Systems: Principles and Paradigms, Upper Saddle River, NJ, Prentice-Hall, 2003.
2. Sundararaman, U. Buy, and A. D. Kshemkalyani, Clock synchronization in wireless sensor networks: a survey, Ad-Hoc Networks, 3(3), 2005, 281–323.
3. Lea, S. Vinoski, and W. Vogels, Guest editors' introduction: asynchronous middleware and services, IEEE Internet Computing, 10(1), 2006, 14–17.
4. G. Coulouris, J. Dollimore, and T. Kindberg, Distributed Systems Concepts and Design, Harlow, UK, 3rd edn, Addison-Wesley, 2001.
5. H. Attiya and J. Welch, Distributed Computing Fundamentals, Simulations, and Advanced Topics, 2nd edn, Hoboken, NJ, Wiley Inter-Science, 2004.
6. Hieu., Vu, Quang (2010). Peer-to-peer computing : principles and applications. Lupu, Mihai., Ooi, Beng Chin, 1961-. Heidelberg: Springer. p. 16.
7. Foster, The Grid: a new infrastructure for 21st century science, Physics Today, 55(2), 2002, 42–47.
8. Ohlídal, M.; Jaroš, J.; Schwarz, J.; et al. (2006). "Evolutionary Design of OAB and AAB Communication Schedules for Interconnection Networks". In Rothlauf, F.; Branke, J.; Cagnoni, S. (eds.). Applications of Evolutionary Computing. Springer Science & Business Media. pp. 267–78.
9. Svozil, K. (2011). "Indeterminism and Randomness Through Physics". In Hector, Z. (ed.). Randomness Through Computation: Some Answers, More Questions. World Scientific. pp. 112–3.
10. V. Kumar, A. Grama, A. Gupta, and G. Karypis, Introduction to Parallel Computing, 2nd edn, Harlow, UK, Pearson Education 2003.

CHAPTER

7

COMPUTATIONAL COMPLEXITY

Learning Objectives

After studying this chapter, you will be able to:

1. Understand the basic of computational complexity
2. Define the time complexity of a turing machine, and the set p
3. Explain the set np and polynomial verifiability
4. Understand the polynomial-time reductions and np-completeness

INTRODUCTION

Computational complexity, a measure of the amount of computing resources (time and space) that a particular algorithm consumes when it runs. Computer scientists use mathematical measures of complexity that allow them to predict, before writing the code, how fast an algorithm will run and how much memory it will require. Such predictions are important guides for programmers implementing and selecting algorithms for real-world applications.

Computational complexity is a continuum, in that some algorithms require linear time (that is, the time required increases directly with the number of items or nodes in the list, graph, or network being processed), whereas others require quadratic or even exponential time to complete (that is, the time required increases with the number of items squared or with the exponential of that number). At the far end of this continuum lie intractable problems—those whose solutions cannot be efficiently implemented. For those

problems, computer scientists seek to find heuristic algorithms that can almost solve the problem and run in a reasonable amount of time.

7.1 THE BASIC OF COMPUTATIONAL COMPLEXITY

Computational complexity is a computer science concept that focuses on the amount of computing resources needed for particular kinds of tasks. In computational **complexity theory**, researchers assess the kinds of resources that will be needed for a given type or class of task in order to classify different kinds of tasks into various levels of complexity.

Keyword

Computational complexity theory is a mathematical research area in which the goal is to quantify the resources required to solve computational problems.

Although computational complexity is in some ways similar to the analysis of algorithms, it is essentially its own branch of mathematical theory. Some think of this approach as a measurement of how much work it would take to solve a particular problem or to achieve a particular task. Different kinds of analysts use computational complexity research to find which parts of a task may be most difficult for a computing system, or to figure out how to most efficiently complete some project. Although some developers might consider computational complexity to be irrelevant to their work, others have pointed out that successfully changing tasks or algorithms from a higher complexity class to a lower complexity class can make them work much better. Programmers and developers who use computational complexity theory on items like nested loops, logic trees or other kinds of rhythms can build more efficient systems with a better understanding of how to create less resource-hungry processes.

7.1.1 Big-O notation

Big-O notation is a simple mathematical formula which roughly approximates or places an upper bound on the resource requirements for an algorithm based on the size of the input — the complexity of the algorithm, its algorithmic complexity.

Technically, algorithmic complexity can and should apply to both time and space (memory and storage) resource requirements,

but commonly people are primarily interested in running time of an algorithm. Algorithmic complexity is couched with the language "on the order of", indicating the rough or approximate cost of the algorithm in terms of resource requirements. How rough? Within an order of magnitude. Precision is not attempted. A rough estimate is usually good enough.

The text "*on the order of*" is shortened as a capital (big) "O", hence the name *Big-O notation*.

The form of Big-O notation is simply:

O(formula)

Where the *formula* is any mathematical expression based on the input parameters or size of the input data for the algorithm. The input size is commonly expressed in the formula using a lower-case (little) "n".

For example,

$O(n^2)$

The main takeaway is that algorithmic complexity can get out of hand very quickly, especially for advanced AI or simulation algorithms or any algorithm handling lots of data or performing complex calculations, so great care is needed to try to minimize the algorithmic complexity, or to at least realize the costs being incurred.

A short summary of the common forms of Big O notation, in increasing order of cost and declining order of performance, include:

- O(1) or O(k). Constant. Great, easy, fastest, trivial, simple.
- O(log log n). Double logarithmic. Relatively trivial except for very large input.
- O(log n). Logarithmic. Non-trivial but very manageable.
- O(sqrt(n)). Quadratic speedup. Faster than linear, but not as fast as logarithmic. For example, Grover's algorithm for searching unstructured data on a quantum computer.
- O(n). Linear. Still manageable and predictable, but not optimal for large amounts of data.
- O(n log n). Linearithmic. Not quite as good as linear, but not so bad.
- O(n^2). Quadratic. Now it's getting somewhat expensive, but not terrible.
- O(n^k). Polynomial. Worse than quadratic (for k > 2), but not as bad as exponential.

- O(2^n) or O(k^n). Exponential. Try to avoid this. This is bad, except for very small amounts of data.
- O(n!). Factorial. Only if you really need it and don't need real-time performance. Or your input data is very small.

7.1.2 Polynomial Complexity

The concept of a polynomial refers to a set of parameters or input values which are used as bases to which constant exponents are applied.

Polynomial complexity refers to algorithmic complexity which can be expressed as a polynomial.

Some examples:

- $O(n^2)$
- $O(n^3)$
- $O(n^4)$
- $O(n^3 + m^2)$

Polynomial complexity also tends to include algorithmic complexity which is simpler than a polynomial, such as *linear* and *logarithmic*, although they are better described as having subpolynomial complexity.

7.1.3 Brute Force

Programmers are expected to design clever and efficient algorithms, but oftentimes the best — or only feasible — approach is simply brute force. A brute force algorithm is an algorithm which exhaustively evaluates all possible solutions without any heuristics or cleverness to intelligently guide its selection of candidate solutions to evaluate.

A brute force algorithm may proceed sequentially through all data or all possible solutions, starting at the first and continuing until the last, without any heuristics or intelligence to change or guide the order of processing.

Whether a brute force algorithm can stop as soon as it finds a workable solution, or whether the algorithm must proceed through all possible solutions before a final decision can be made will depend on the algorithm and its requirements.

Generally brute force is the least attractive approach to finding a solution, except in four situations:

- The data size or prospective solution count is small enough that the time and resources to evaluate all possible solutions is reasonably small or acceptable.
- There is no known algorithm to achieve a solution other than exhaustive processing.
- Known algorithms are not guaranteed to achieve a sufficiently optimal or accurate solution.
- A quantum computer can be used to simultaneously evaluate all possible solutions in parallel in a single step using quantum parallelism.

Keyword

Quantum parallelism is used to perform a large number of computations at the same time, and quantum interference is used to combine their results into something that is both meaningful and can be measured according to the laws of quantum mechanics.

7.1.4 Random and Statistical Methods

Another type of approach to computation are statistical methods such as using random sampling to approximate a deterministic result. This requires generating lots of random numbers and then evaluating the values of functions based on these random input values. Repeated enough times, results can be fairly accurate, but repetitions have a cost.

Fewer repetitions cost less but produces less accurate results. More repetitions produce more accurate results but cost more. It's a classic cost/benefit tradeoff, which affects performance — and the algorithmic complexity of the algorithm.

The number of repetitions must be factored into the algorithmic complexity, but in fact the repetition count might be an input parameter so that different runs of the same algorithm may use a different number of repetitions, resulting in different performance even though the main input data may be unchanged.

Generally, changing the repetition count will only result in a linear impact on the performance, which won't change the overall, rough algorithmic complexity of the algorithm, but it can depend on other factors as well. In short, it may have no impact, or it could have a large impact, depending on the details of the algorithm.

A common example is the Monte Carlo method for simulations.

Simulations of complex processes, including phenomena of the physical world and business and industrial processes, commonly use such methods, where there is no single deterministic result, but the result depends on input values which tend to be random rather than predictable.

Input parameters for a simulation could have a multi-dimensional effect, so that the impact on performance is not a simple linear scaling, and may result in a change in the overall algorithmic complexity — or not, depending on exactly how the parameters are used. It can get complicated.

The good news is that if performance is unacceptable, the granularity (precision) of the simulation can be reduced to make performance more acceptable. But this does not always work out as well as expected. It all depends, on many factors. As a result, performance — algorithmic complexity — can be problematic.

In short, modeling the algorithmic complexity for computations requiring randomness and statistical methods can be quite challenging.

7.1.5 Space Complexity

Technically, algorithmic complexity is concerned not only with time, but also with space — memory. The amount of space or memory required to execute an algorithm for a given amount of input is referred to as *space complexity*.

There are a few key points:

- An algorithm might generate a large amount of data, even for a small amount of input data.
- Even a relatively simple algorithm can generate a large amount of data.
- Even if an algorithm doesn't generate a large amount of data for final results, a significant amount of data may be needed for *intermediate results*, which may be discarded when the algorithm completes, but still consume resources while the algorithm is running.
- Space or memory includes both main memory — commonly RAM or **random access memory** — and storage — commonly on a spinning disk or flash memory. Memory tends to be transient while storage tends to be permanent or persistent, although those distinctions are not necessarily always the case.
- Some algorithms may use very little space, while others, especially those which are data-intensive, such as data analysis, logging, real-time data, and databases, can require dramatic amounts of space.
- Memory and storage requirements are frequently an overlooked aspect of

the design, development, evaluation, and deployment of algorithms and applications. Deployment of a poorly-characterized algorithm can lead to very unpleasant surprises.

7.2 THE TIME COMPLEXITY OF A TURING MACHINE, AND THE SET P

A Turing machine deciding a language $L \subseteq \Sigma^*$ can be thought of as solving a decision problem: Given $x \in \Sigma^*$, is $x \in L$? A natural measure of the size of the problem instance is the length of the input string. In the case of a decision problem with other kinds of instances, we usually expect that any integer we might use to measure the size of the instance will be closely related to the length of the string that encodes that instance—although we will return to this issue later in this section to pin down this relationship a little more carefully. The first step in describing and categorizing the complexity of computational problems is to define the time complexity function of a Turing machine.

Keyword

Random-access memory is a form of computer memory that can be read and changed in any order, typically used to store working data and machine code

Definition 1. The Time Complexity of a Turing Machine

Suppose T is a Turing machine with input alphabet Σ that eventually halts on every input string. The time complexity of T is the function $\tau_T : N \rightarrow N$, where τ_T (n) is defined by considering, for every input string of length n in Σ^*, the number of moves T makes on that string before halting, and letting τ_T (n) be the maximum of these numbers. When we refer to a TM with a certain time complexity, it will be understood that it halts on every input.

Example 1: Calculating the Time Complexity of a Simple TM

The computation for a string of length n = 2k has three parts. First, T finds the middle, changing the symbols to uppercase as it goes; second, it changes the first half back to lowercase as it moves the tape head back to the beginning; finally, it compares the two halves.

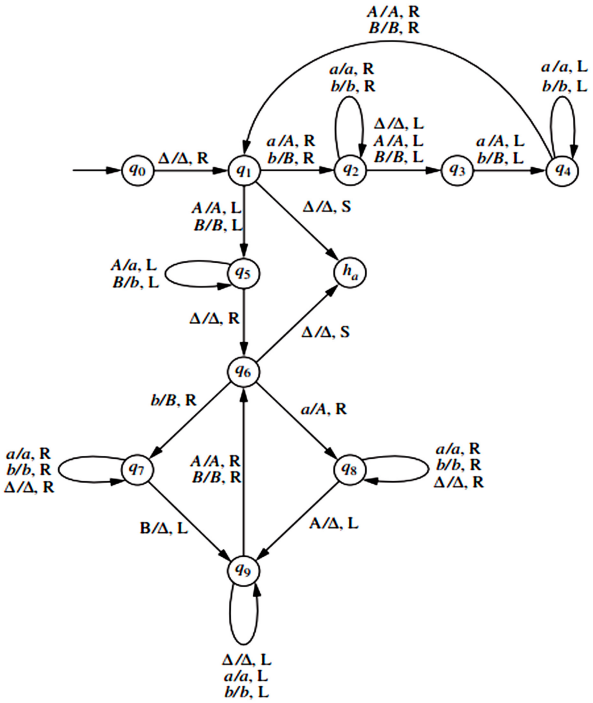


Figure 7.1. A Turing machine accepting {xx | x ∈ {a, b}*}.

In the first part, it takes one move to move the tape head to the first input symbol, 2k + 1 moves to change the first symbol and find the rightmost symbol, and 2k more to change the rightmost symbol and return the tape head to the leftmost lowercase symbol. Each subsequent pass requires four fewer moves, and the total number of moves is therefore

$$1 + \sum_{i=0}^{k}(4i+1) = 1 + (k+1) + 4\sum_{i=0}^{k} i = k + 2 + 4\frac{k(k+1)}{2} = 2k^2 + 3k + 2$$

The number of moves in the second part is k + 1 and also depends only on the length of the input string. The length of the third part depends on the actual string, but the strings requiring the most moves are the ones in L. The last part of the computation for a string in L involves one pass for each of the k symbols in the first half, and each pass contains k moves to the right, k to the left, and one more to the right. The maximum number of moves in this part is $2k^2 + k$, and we conclude that

$$\tau_T(n) = \tau_T(2k) = 4k^2 + 5k + 4 = n^2 + 5n/2 + 4$$

For each symbol of the input string, there are one or two moves that change the symbol to the opposite case, and one move that compares it to another symbol. The formula for $\tau_T(n)$ is quadratic because of the other moves, which involve repeated

back-and-forth movements from one end of the string to the other. It is possible to reduce the number of these by adding more states. If we had the TM remember two symbols at a time, we could come close to halving the number of back-and-forth passes, and going from two to a larger number would improve the performance even more, at least for large values of n. Techniques like these can, in fact, be used to reduce the time complexity of an arbitrary TM, by as large a constant factor as we wish. However, the back-and-forth movement in this example can't be eliminated completely, and it is possible to show that every TM accepting this language has time complexity that is at least quadratic.

In this example, the significant feature of τ_T is the fact that the formula is quadratic. The next definition makes it a little easier, when discussing the time complexity of a TM, to focus on the general growth rate of the function rather than the value at any specific point.

Definition 2. Big-Oh Notation

If f and g are partial functions from N to R^+ (that is, both functions have values that are nonnegative real numbers wherever they are defined), we say that f = O(g), or f (n) = O(g(n)) (which we read "f is big-oh of g," or "f (n) is big-oh of g(n)"), if for some positive numbers C and N,

f (n) ≤ Cg(n) for every n ≥ N

We can paraphrase the statement f = O(g) by saying that except perhaps for a few values of n, f is no larger than g except for some constant factor. The statement f (n) = O(g(n)) starts out looking like a statement about the number f (n), but it is not. (The notation is standard but is confusing at first.) It means the same thing as f = O(g). Even though it tells us nothing about the values of f (n) and g(n) for any specific n, it is a way of comparing the long-term growth rates of the two functions

If f and g are total functions with f = O(g), and g(n) is positive for every n ∈ N , then it is possible to say that for some positive C, f (n) ≤ Cg(n) for every n; the way the definition is stated allows us to write f = O(g) even if f or g is undefined in a few places, or if g(n) = 0 for a few values of n. For example,

$n^2 + 5n/2 + 4 = O(n^2)$

As a reason in this case, we could use the fact that for every n ≥ 1, $n^2 + 5n/2 + 4 \le n^2 + 5n^2/2 + 4n^2 = 7.5n^2$. Or, if we preferred, we could say that $n^2 + 5n/2 + 4 \le 2n^2$ for every n ≥ 4, because $5n/2 + 4 \le n^2$ for those values of n.

Similarly, we can consider a polynomial f of degree k, with the formula

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

where the coefficients a_i are real numbers. As long as the leading coefficient a_k is positive, f (n) will be positive for all sufficiently large values of n (and everywhere it's negative we can simply think of it as undefined). An argument similar to the one for the quadratic polynomial shows that f (n) = $O(n_k)$.

Although we haven't gotten very far yet toward our goal of categorizing decision problems in terms of their complexity, we would probably say that the problem of deciding the language in Example 1 is relatively simple. There is no shortage of decision problems that appear to be much harder. An easy way to find examples is to consider combinatorial problems for which finding the answer might involve considering a large number of cases.

An instance of the satisfiability problem is an expression involving Boolean variables x_1, x_2, ... , x_n (whose values are true or false) and the logical connectives ∧, ∨, and ¬. The question is whether there is an assignment of truth values to the variables that satisfies the expression, or makes its value true. There is no doubt that this problem is decidable, because a brute-force algorithm will work: Try possible assignments of truth values, until one has been found that satisfies the expression or until all 2n assignments have been tried.

The traveling salesman problem considers n cities that a salesman must visit, with a distance specified for every pair of cities. It's simplest to formulate this as an optimization problem, and to ask in which order the salesman should visit the cities in order to minimize the total distance traveled. We can turn the problem into a decision problem by introducing another number k that represents a constraint on the total distance: Is there an order in which the salesman can visit the cities so that the total distance traveled is no more than k? There is a brute-force solution here also, to consider possible orders one at a time. The decision problem might be answered before all n! permutations have been considered, but the brute-force approach for either formulation might require looking at all of them.

For every decision problem, if the number of steps required to decide an instance of size n is proportional to n, computer hardware currently available makes it possible to perform the computation for instances that are very large indeed. The TM in Example 1 has time complexity that is roughly proportional to n^2, and that computation is still feasible for large values of n; many standard algorithms involving matrices or graphs take time proportional to n^2 or n^3. On the other hand, a decision problem for which the required time grows exponentially with n is undecidable in practice except for small instances. If an instance of size n required time 2n, and if we could just manage an instance of size 20, doubling the computing speed would only allow us to go to n = 21, and a thousand-fold increase would still not be quite sufficient for n = 30. An instance of size 100 would be out of reach. It's not hard to see that if the time is proportional to n!, the situation is considerably worse.

Showing that a brute-force approach to solving a problem takes a long time does not in itself mean that the problem is complex. The satisfiability problem and the traveling salesman problem are assumed to be hard, not because the bruteforce approach takes at least exponential time in both cases, but because no one has found a way of solving either problem that doesn't take at least exponential time.

If we have determined that problems, or languages, requiring only quadratic time to decide should be considered relatively easy, and those that require exponential time or more should be considered very hard, what is a reasonable criterion that can be used to distinguish the tractable problems, those for which reasonablesize instances can be decided in a reasonable amount of time? The most common answer, though it is not perfect, is that the tractable problems are those with a polynomial-time solution; in particular, the problem of deciding a language L may be considered tractable if L can be accepted by a Turing machine T for which $\tau_T(n) = O(n^k)$ for some $k \in N$.

Remember

We must qualify this last statement only in the sense that the encoding we use should be reasonable, and that another requirement for an encoding method to be reasonable is that both encoding an instance and decoding a string should take no more than polynomial time (in the first case as a function of the instance size, and in the second as a function of the string length).

One reason this criterion is convenient is that the class of tractable problems seems to be invariant among various models of computation. There is obviously a difference between the length of a computation on a Turing machine and the length of a corresponding computation on another model, whether it is a multitape TM or an electronic computer, but the difference is likely to be no more than a polynomial factor. As a general rule, if a problem can be solved in polynomial time on some computer, then the same is true on any other computer. Similarly, using this criterion makes it unnecessary to distinguish between "the time required to answer an instance of the problem," as a function of the size of the instance, and the number of steps a Turing machine needs to make on an input string representing that instance.

Not only is the polynomial-time criterion convenient in these ways, but to a large extent it seems to agree with people's ideas of which problems are tractable. Most interesting problems with polynomial-time solutions seem to require time proportional to n^2, or n^3, or n^4, rather than n^{50}; and in the case of problems for which no polynomial-time solutions are known, the only solutions that are known typically require time proportional to 2^n or worse.

Definition 3. The Set P

P is the set of languages L such that for some Turing machine T deciding L and some $k \in N$, $\tau_T(n) = O(n^k)$.

Because of our comments above about decision problems and reasonable encodings, we can speak informally about a decision problem being in P if for some reasonable way of encoding instances, the resulting language of encoded yes-instances is in P.

We have seen a language that is, in Example 1. We can construct "artificial" languages that are definitely not. The satisfiability problem and the traveling salesman problem seem to be good candidates for real-life problems that are not in P. Most people believe that the satisfiability problem is not in P and therefore that $P \neq NP$., but no one has proved this, and the $P \stackrel{?}{=} NP$ question remains very perplexing.

7.3 THE SET NP AND POLYNOMIAL VERIFIABILITY

The satisfiability problem, seems like a hard problem because no one has found a decision algorithm that is significantly better than testing possible assignments one at a time. Testing is not hard; for any reasonable way of describing a Boolean expression with n distinct variables, we can easily find out whether a given truth assignment satisfies the expression. The difficulty is that the number of possible assignments is exponential in n. Even if we could test each one in constant time, independent of n, the total time required would not be polynomial.

When we were interested only in whether a language was recursively enumerable and not with how long it might take to test an input string, we used nondeterminism in several examples to simplify the construction of a TM. We can consider a nondeterministic TM here as well, which accepts Satisfiable by guessing an assignment and then testing (deterministically) whether it satisfies the expression. This approach doesn't obviously address the issue of complexity. The steps involved in guessing and testing an assignment can easily be carried out in polynomial time, as we will see in more detail below, but because the nondeterminism eliminates the time-consuming part of the brute-force approach, the approach doesn't seem to shed much light on how hard the problem is.

Nevertheless, at least it provides a simple way of formulating the question posed by problems like Sat: If we have a language L that can be accepted in polynomial time by a nondeterministic TM, is there any reason to suspect that L is in P?

Definition 4. The Time Complexity of a Nondeterministic Turing Machine

If T is an NTM with input alphabet Σ such that, for every $x \in \Sigma^*$, every possible sequence of moves of T on input x eventually halts, the time complexity $\tau_T : N \rightarrow N$ is defined by letting $\tau_T(n)$ be the maximum number of moves T can possibly make on any input string of length n before halting. As before, if we speak of an NTM as having a time complexity, we are assuming implicitly that no input string can cause it to loop forever.

Definition 5. The Set NP

NP is the set of languages L such that for some NTM T that cannot loop forever on any input, and some integer k, T accepts L and $\tau_T(n) = O(n^k)$. We say that a language in NP can be accepted in nondeterministic polynomial time.

As before, we can speak of a decision problem being in NP if, for some reasonable way of **encoding** instances, the corresponding language is. Because an ordinary Turing machine can be considered an NTM, it is clear that $P \subseteq NP$. Before starting to discuss the opposite inclusion, we consider two examples, including a more detailed discussion of the satisfiability problem.

Keyword

Encoding is the process of converting data into a format required for a number of information processing needs, including: Program compiling and execution

Example 2. The Satisfiability Problem

We let Sat denote the satisfiability problem. An instance is an expression involving occurrences of the Boolean variables $x_1, x_2, \ldots, x_\nu$, for some $\nu \geq 1$. We use the term literal to mean either an occurrence of a variable x_i or an occurrence of $\overline{x}_i$, which represents the negation $\neg x_i$ of x_i. The expression is assumed to be in conjunctive normal form, or CNF: It is the conjunction

$$C_1 \wedge C_2 \wedge \ldots C_c$$

of clauses, or conjuncts, C_i, each of which is a disjunction (formed with $\vee$'s) of literals. For an expression of this type, Sat asks whether the expression can be satisfied by some truth assignment to the variables.

For example,

$$(x_1 \vee x_3 \vee x_4) \wedge (\overline{x}_1 \vee x_3) \wedge (\overline{x}_1 \vee x_4 \vee \overline{x}_2) \wedge \overline{x}_3 \wedge (x_2 \vee \overline{x}_4)$$

is an expression with five conjuncts that is satisfied by the truth assignment

$x_2 = x_4 =$ true, $x_1 = x_3 =$ false

We will encode instances of Sat by omitting parentheses and ∨'s and using unary notation for subscripts. For example, the expression

$$(x_1 \vee \overline{x}_2) \wedge (x_2 \vee x_3 \vee \overline{x}_1) \wedge (\overline{x}_4 \vee x_2)$$

will be represented by the string

$$\wedge x1\overline{x}11 \wedge x11x111\overline{x}1 \wedge \overline{x}1111x11$$

Satisfiable will be the language over $\Sigma = \{\wedge, x, \overline{x}, 1\}$ containing all the encoded yesinstances.

If the size of a problem instance is taken to be the number k of literals (not necessarily distinct), and n is the length of the string encoding the instance, then $k \leq n$, and it is easy to see that n is bounded by a quadratic function of k. Therefore, if Satisfiable ∈ NP, it makes sense to say that the decision problem Sat is in NP.

The first step carried out by an NTM T accepting Satisfiable is to verify that the input string represents a valid CNF expression and that for some ν, the variables are $x_1, x_2, \ldots, x_\nu$. If this is the case, T attempts to satisfy the expression, keeping track as it proceeds which conjuncts have been satisfied so far and which variables within the unsatisfied conjuncts have been assigned values. The iterative step consists of finding the first conjunct not yet satisfied; choosing a literal within that conjunct that has not been assigned a value (this is the only occurrence of nondeterminism); giving the variable in that literal the value that satisfies the conjunct; marking the conjunct as satisfied; and giving the same value to all subsequent occurrences of that variable in unsatisfied conjuncts, marking any conjuncts that are satisfied as a result. The loop terminates in one of two ways. Either all conjuncts are eventually satisfied, or the literals in the first unsatisfied conjunct are all found to have been falsified. In the first case T accepts, and in the second it rejects. If the expression is satisfiable, some choice of moves causes T to guess a truth assignment that works, and otherwise no choice of moves leads to acceptance.

The TM T can be constructed so that, except for a few steps that take time proportional to n, all its actions are minor variations of the following operation: Begin with a string of 1's in the input, delimited at both ends by some symbol other than 1, and locate some or all of the other occurrences of this string that are similarly delimited. We leave it to you to convince yourself that a single operation of this type can be done in polynomial time, and that the number of such operations that must be performed is also no more than a polynomial. Our conclusion is that Satisfiable, and therefore Sat, are in NP.

Example 3. Composites and Primes

A decision problem that is more familiar and almost obviously in NP is the problem to which we will give the name Comp: Given a natural number n, is it a composite number—that is, is there some smaller number other than 1 that divides it evenly? The way the problem is stated makes it easy to apply the guess-and-test strategy, just as in Example 2. An NTM can examine the input n, guess a positive integer p strictly between 1 and n, divide n by p, and determine whether the remainder is 0. Or, taking even greater advantage of nondeterminism, it could guess two posive integers p and q, both between 1 and n, and multiply them to determine whether n = pq.

One subtlety related to any problem whose instances are numbers has to do with how we measure the size of an instance. It might seem that an obvious measure of the size of the instance n is n, which would be the length of an input string if we used unary notation. However, people don't do it this way: Using binary notation would allow an input string with approximately log n digits, and it would be a little unreasonable to say that an algorithm required polynomial time if that meant a polynomial function of n but an exponential function of the input length.

It is easy to see that in the case of Comp, an NTM can carry out the steps described above in polynomial time, even as a function of log n. Therefore, Comp is indeed in NP. Without nondeterminism, the problem appears intimidating at first. The most obvious brute-force approach, testing natural numbers one at a time as possible divisors, certainly does not produce a polynomial-time solution. We can improve on this, at least to the extent that only primes less than n need to be considered as possible divisors, but the improvement is not enough to make the time a polynomial in log n. However, this problem does not seem to have quite the combinatorial complexity of the assignment problem, and there are additional mathematical tools available to be applied, if only because properties of primes and composites have been studied for centuries. In any case, the problem Comp is in a different category than Sat, because it has actually been proven to be in P.

This is slightly curious. The complementary problem to Comp is Prime: Given a natural number n, is n prime? If Comp is in P, then Prime is also, but it's not even clear at first that Prime is in NP. Although the two problems ask essentially the same question, a nondeterministic approach is available for one version (Does there exist a number that is a factor of n?) and not obviously available for the other (Is p a nondivisor of n for every p?). In fact, nondeterminism can be used for the problem Prime, as suggested in the next paragraph, but the corresponding question is difficult in general: It is not known whether the complementary problem of every problem in NP is itself in NP.

Although the proof that Comp or Prime is in P is complicated, here is a way to approach the problem Prime nondeterministically. It involves a characterization of primes due to Fermat: An integer n greater than 3 is prime if and only if there is an integer x, with 1 <x < n − 1, satisfying.

$$x^{n-1} \equiv_n 1, \text{ and for every } m \text{ with } 1 < m < n-1, \; x^m \not\equiv_n 1$$

The fact that the characterization begins "there is an integer x" is what suggests some hope. It's still not apparent that this approach will be useful, because deciding that the statement is true seems to require performing a test on every number between 1 and n − 1. However, this turns out not to be too serious an obstacle.

Let us return to the satisfiability problem in order to obtain a slightly different characterization of NP, based on the guess-and-verify strategy we have mentioned in all our examples. We consider a Boolean expression e, and an assignment a of truth values to the variables in e; it may help to clarify things if we identify e and a with the strings that represent them.

The NTM we described that accepts Satisfiable takes the input string e and guesses an assignment a. This is the nondeterministic part of its operation. The remainder of the computation is deterministic and consists of checking whether a satisfies e. If the expression is satisfiable and the guesses in the first part are correct, the string a can be interpreted as a certificate for e, proving, or attesting to the fact, that e is satisfiable. By definition, every string e possessing a certificate is in Satisfiable. A certificate may be difficult to obtain in the first place, but the process of verifying that it is a certificate for e is straightforward.

We can isolate the second portion of the computation by formulating the definition of a verifier for Satisfiable: a deterministic algorithm that operates on a pair (e, a) (we take this to mean a Turing machine operating on the input string e$a, for some symbol $ not in the alphabet) and either accepts it or rejects it. We interpret acceptance to mean that a is a certificate for e and that e is therefore in Satisfiable; rejection is interpreted to mean, not that e fails to be in Satisfiable—there may be a string different from a that is a certificate for e—but only that a is not a certificate for e. An expression (string) e is an element of Satisfiable if and only if there is a string a such that the string e$a is accepted by the verifier.

The issue of polynomial time arises when we consider how many steps are required by a verifier to accept or reject the string e$a. However, it is more appropriate to describe this number in terms of |e|, the size of the instance of the original problem, than in terms of |e$a|, the length of the entire string.

Definition 6. A Verifier for a Language

If $L \subseteq \Sigma^*$, we say that a TM T is a verifier for L if T accepts a language $L_1 \subseteq \Sigma^*\{\$\}\ \Sigma^*$, T halts on every input, and

$$L = \{x \in \Sigma^* \mid \text{for some } a \in *, x\$a \in L_1\}$$

A verifier T is a polynomial-time verifier if there is a polynomial p such that for every x and every a in Σ^*, the number of moves T makes on the input string x$a is no more than $p(|x|)$.

Theorem 1. For every language $L \in \Sigma^*$, $L \in NP$ if and only if L is polynomially verifiable—i.e., there is a polynomial-time verifier for L.

Keyword

Verification algorithm is a two-argument algorithm A, where one argument is an ordinary input string x, and the other argument is a binary string y called a certificate.

Proof

By definition, if $L \in NP$, there is an NTM T_1 accepting L in nondeterministic polynomial time. This means that the function τ_{T1} is bounded by some polynomial q. We can obtain a verifier T for L by considering strings m representing sequences of moves of T_1. The **verification algorithm** takes an input x$m, executes the moves of T_1 corresponding to m on the input x (or, if m represents a sequence of more than $q(|x|)$ moves, executes the first $q(|x|)$), and accepts if and only if this sequence of moves causes T_1 to accept. It is clear that the verifier operates in polynomial time as a function of $|x|$.

On the other hand, if T is a polynomial-time verifier for L, and p is the associated polynomial, then we can construct an NTM T_1 that operates as follows: On input x, T_1 places the symbol $ after the input string, followed by a sequence of symbols, with length no more than $p(|x|)$, that it generates nondeterministically; it then executes the verifier T. T_1 accepts the language L, because the strings in L are precisely those that allow a sequence of moves leading to acceptance by the verifier. Furthermore, it is easy to see that the number of steps in the computation of T_1 is bounded by a polynomial function of $|x|$.

7.4 POLYNOMIAL-TIME REDUCTIONS AND NP-COMPLETENESS

Just as we can show that a problem is decidable by reducing it to another one that is, we can show that languages are in P by reducing them to others that are. (If we have languages we know are not in P, we can also use reductions to find others that are not.) Reducing a language L_1 to another language L_2 required only that we find a computable function f such that deciding whether $x \in L_1$ is equivalent to deciding whether $f(x) \in L_2$. We said that deciding the first language is no harder than deciding the second, because we considered only two degrees of hardness, decidable and undecidable. To see whether $x \in L_1$, all we have to do is compute f (x) and see whether it is in L_2; and f is computable. Because one decidable language can be harder (i.e., take longer) to decide than another, a reduction should now satisfy some additional conditions. Not surprisingly, an appropriate choice is to assume that the function be computable in polynomial time.

Definition 7. Polynomial-Time Reductions

If L_1 and L_2 are languages over respective alphabets Σ_1 and Σ_2, a polynomial-time reduction from L_1 to L_2 is a function $f : \Sigma_1^* \rightarrow \Sigma_2^*$ satisfying two conditions: First, for every $x \in \Sigma_1^*$, $x \in L_1$ if and only if $f(x) \in L_2$; and second, f can be computed in polynomial time—that is, there is a TM with polynomial time complexity that computes f . If there is a polynomial-time reduction from L_1 to L_2, we write $L_1 \leq_p L_2$ and say that L_1 is polynomial-time reducible to L_2.

If we interpret $\leq_p$ to mean "is no harder than," then the statements in the following theorem are not surprising.

Example 4. A Reduction from Sat to the Complete Subgraph Problem

We begin with some terminology. An undirected graph is a pair G = (V, E), where V is a finite nonempty set of vertices and E is a set of edges. An edge is an unordered pair (v_1, v_2) of vertices ("unordered" means that (v_1, v_2) and (v_2, v_1) are the same). If e is the edge (v_1, v_2), then v_1 and v_2 are the end points of e, are said to be joined by e, and are said to be adjacent. A complete subgraph of G is a subgraph (a graph whose vertex set and edge set are subsets of the respective sets of G) in which every two vertices are adjacent. It is clear that if a graph has a complete subgraph with k vertices, then for every $n < k$, it has a complete subgraph with n vertices. Figure 7.2 shows a graph G with seven vertices and ten edges. There is a complete subgraph with the vertices 3, 5, 6, and 7, and no subgraph of G with more than 4 vertices is complete.

The complete subgraph problem is the problem: Given a graph G and a positive integer k, does G have a complete subgraph with k vertices? We can represent the graph by a string of edges followed by the integer k, where we represent the vertices by integers 1, 2, ... , n and use unary notation for each vertex and for the number k. For example, the instance consisting of the graph in Figure 7.2 and the integer 4 might be represented by the string

(1, 11)(11, 111)(111, 1111)(111, 11111)...(111111, 1111111)1111

A graph with n vertices has no more than n(n − 1)/2 edges, and it is reasonable to take the number n as the size of the instance.

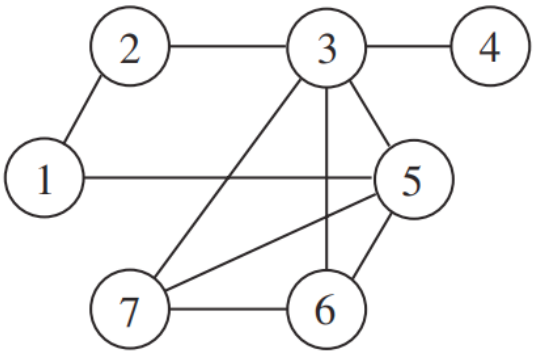


Figure 7.2. A graph with 7 vertices and 10 edges.

The complete subgraph problem is in NP, because an NTM can take a string encoding an instance (G, k), nondeterministically select k of the vertices, and examine the edges to see whether every pair among those k is adjacent. We will construct a reduction from the satisfiability problem Sat to the complete subgraph problem. It is easiest to discuss the reduction by considering the instances themselves, rather than the strings representing them. Suppose x is the CNF expression

$$x = \bigwedge_{i=1}^{c} \bigvee_{j=1}^{d_i} a_{i,j}$$

where each $a_{i,j}$ is a literal. We must describe an instance f (x) = (G_x , k_x) of the complete subgraph problem so that x is satisfiable if and only if G_x has a complete subgraph with k_x vertices. We construct G_x = (V_x , E_x) so that vertices correspond to occurrences of literals in x:

$V_x = \{(i, j) \mid 1 \le i \le c \text{ and } 1 \le j \le d_i\}$

The edge set E_x is specified by saying that (i, j) is adjacent to (l, m) if and only if the corresponding literals are in different conjuncts of x and there is a truth assignment that makes them both true. In other words,

$$E_x = \{((i, j), (l, m)) \mid i \neq l \text{ and } a_{i,j} \neq \neg a_{l,m}\}$$

Finally, we define k_x to to be c, the number of conjuncts in x.

If x is satisfiable, then there is a truth assignment Θ such that for every i there is a literal a_{i,j_i} that is given the value true by Θ. The vertices

$$(1, j_1), (2, j_2), \ldots, (k, j_k)$$

determine a complete subgraph of G_x , because we have chosen the edges of G_x so that every two of these vertices are adjacent. On the other hand, suppose there is a complete subgraph of G_x with k_x vertices. Because none of the corresponding literals is the negation of another, some truth assignment makes them all true. Because these literals must be in distinct conjuncts, the assignment makes at least one literal in each conjunct true and therefore satisfies x.

In order to construct the string representing (G_x, k_x) from the string representing x, a number of steps are necessary: associating the occurrences of literals in x with integers from 1 through n; for each one, finding every other one occurring in a different conjunct that is not the negation of that one; and constructing the string representing that edge. Each of these steps can evidently be carried out in a number of steps that is no more than a polynomial in |x|, and the overall time is still no more than polynomial.

SUMMARY

- Computational complexity is a computer science concept that focuses on the amount of computing resources needed for particular kinds of tasks. In computational complexity theory, researchers assess the kinds of resources that will be needed for a given type or class of task in order to classify different kinds of tasks into various levels of complexity.
- Programmers and developers who use computational complexity theory on items like nested loops, logic trees or other kinds of rhythms can build more efficient systems with a better understanding of how to create less resource-hungry processes.
- Big-O notation is a simple mathematical formula which roughly approximates or places an upper bound on the resource requirements for an algorithm based on the size of the input — the complexity of the algorithm, its algorithmic complexity.
- Polynomial complexity also tends to include algorithmic complexity which is simpler than a polynomial, such as linear and logarithmic, although they are better described as having subpolynomial complexity.
- Programmers are expected to design *clever* and *efficient* algorithms, but oftentimes the best — or only feasible — approach is simply *brute force*. A *brute force algorithm* is an algorithm which exhaustively evaluates all possible solutions without any *heuristics* or *cleverness* to intelligently guide its selection of candidate solutions to evaluate.
- A brute force algorithm may proceed sequentially through all data or all possible solutions, starting at the first and continuing until the last, without any heuristics or intelligence to change or guide the order of processing.
- Simulations of complex processes, including phenomena of the physical world and business and industrial processes, commonly use such methods, where there is no single deterministic result, but the result depends on input values which tend to be random rather than predictable.

KNOWLEDGE CHECK

1. The complexity of linear search algorithm is .....

a. O(n)

b. O(log n)

c. $O(n^2)$

d. O(n log n)

2. **The worst case occurs in quick sort when .....**
   a. Pivot is the median of the array
   b. Pivot is the smallest element
   c. Pivot is the middle element
   d. None of the mentioned
3. **Which of the following case does not exist in Complexity Theory?**
   a. Beat case
   b. Worst case
   c. Average case
   d. Null case
4. **The complexity of Binary search algorithm is .....**
   a. O(n)
   b. O(log)
   c. $O(n^2)$
   d. O(n log n)
5. **The worst case complexity of quick sort is .....**
   a. O(n)
   b. O(log n)
   c. $O(n^2)$
   d. O(n log n)

REVIEW QUESTIONS

1. Suppose f, g : N → N . Show that if f + g = O(max(f, g)).
2. Show that for any decidable decision problem, there is a way to encode instances of the problem so that the corresponding language can be accepted by a TM with linear time complexity.
3. Show that if L can be accepted by a multitape TM with time complexity f , then L can be accepted by a one-tape machine with time complexity $O(f^2)$.
4. Show that the general satisfiability problem, Given an arbitrary Boolean expression, not necessarily in conjunctive normal form, involving the variables $x_1, x_2, \ldots, x_v$, is it satisfiable? is NP-complete.
5. Explain why it is appropriate to insist on binary notation when encoding instances of the primality problem, but not necessary to do this when encoding subscripts in instances of the satisfiability problem.
6. Show that both of the following decision problems are in P.

a. DNF-Sat: Given a Boolean expression in disjunctive normal form (the disjunction of clauses, each of which is a conjunction of literals), is it satisfiable?

b. CNF-Tautology: Given a Boolean expression in CNF, is it a tautology (i.e., satisfied by every possible truth assignment)?

Check Your Result

1. (a) 2. (b) 3. (d) 4. (b) 5. (c)

REFERENCES

1. Arora, S., and Barak, B., 2009, Computational Complexity: A Modern Approach, Cambridge, England: Cambridge University Press.
2. Artemov, S., and Kuznets, R., 2014, "Logical Omniscience as infeasibility," Annals of Pure and Applied Logic, 165: 6–25.
3. Cook, S., and Kolokolova, A., 2001, "A second-order system for polytime reasoning using Gradel's theorem," in Logic in Computer Science, 2001. Proceedings. 16th Annual IEEE Symposium on, pp. 177–186.
4. Cook, W., 2012, In pursuit of the Traveling Salesman: mathematics at the limits of computation, Princeton: Princeton University Press.
5. Cormen, T., Leiserson, C., and Rivest, R., 2005, Introduction to Algorithms (Second Ed.), Cambridge, Massachusetts: MIT Press.
6. Crandall, R., and Pomerance, C., 2005, Prime Numbers (Second Ed.), Berlin: Springer.
7. Du, D., and Ko, K., 2000, Theory of Computational Complexity, New York: John Wiley.
8. Fortnow, L., 2003, "One Complexity Theorist's View of Quantum Computing," Theoretical Computer Science, 292(3): 597–610.
9. Gaifman, H., 2010, "Vagueness, tolerance and contextual logic," Synthese, 174: 1–42.
10. Gaifman, Haim, 2012, "On Ontology and Realism in Mathematics," Review of Symbolic Logic, 5(3): 480–512.
11. Gigerenzer, G., and Selten, R., 2002, Bounded Rationality: The Adaptive Toolbox, Cambridge, Massachusetts: MIT Press.
12. Goldreich, O., 2010, P, NP, and NP-completeness: The basics of computational complexity, Cambridge, England: Cambridge University Press.
13. Hemaspaandra, L., and Ogihara, M., 2002, The Complexity Theory Companion, Berlin: Springer Verlag.
14. Homer, S., and Selman, A., 2011, Turing and the Development of Computational Complexity (No. 2011-06), SUNY Buffalo Department of Computer Science.
15. Iwan, S., 2000, "On the Untenability of Nelson's Predicativism," Erkenntnis, 53(1-2): 147–154.
16. Kahneman, D., 2003, "A Perspective on Judgment and Choice: Mapping Bounded Rationality." American Psychologist, 58(9): 697–720.
17. Magidor, O., 2011, "Strict Finitism and the Happy Sorites," Journal of Philosophical Logic, 41: 1–21.

18. Milnikel, R., 2007, "Derivability in certain subsystems of the Logic of Proofs is ΠP2-complete," Annals of Pure and Applied Logic, 145(3): 223–239.
19. Mulmuley, K., and Sohoni, M., 2001, "Geometric complexity theory I: An approach to P vs. NP and related problems," SIAM Journal on Computing, 31(2): 496–526.
20. Nielsen, M. A., and Chuang, I., 2000, Quantum Computation and Quantum Information, Cambridge, England: Cambridge University Press.
21. Scott, A., and Sorkin, G., 2006, "Solving sparse random instances of MAX CUT and MSC 2-CSP in linear expected time," Combinatorics, Probability and Computing, 15(1-2): 281–315.

INDEX

A

Accepting branch 68
Algorithm 163, 164, 165, 166, 167, 169, 173, 174, 176, 179, 180, 181, 182, 183, 184, 185, 186, 189, 190, 191
Algorithmic complexity 230, 231, 232, 233, 234, 249
American national standards institute (ANSI) 8
Analysis of distributed algorithms 194
Automata 59, 60, 61, 62, 64, 65, 66, 68, 69, 77, 86, 90, 92, 93, 94

B

Big-O notation 230, 231, 249
Brute force algorithm 232, 249
Bubble sort 165, 166, 190, 191

C

Client machines 19
Compiler 62, 63
Computably enumerable 141, 142, 143, 144
Computational complexity 229, 230, 249
Compute functions 145
Computer program 23, 47, 48
Computer programming 164
Computer science 157, 194, 230, 249
Computer system 4, 11, 19, 53
Context-free grammar 97, 99, 103, 104, 105, 110, 113, 114, 128
Control unit 98, 99
Critical rule 61

D

Data analysis 234
Distributed computing 195
Distributed system 195, 202, 203, 205, 206, 207, 209, 210, 225

E

Electronic device 3

F

Finite automata 127
Flash memory 234

G

Graphical user interface (GUI) 11

I

Implementation of the algorithms 133
Input device 22
Instantaneous description (ID) 100

M

Memory 98, 100, 124, 125, 127
Merge function 170
Merge sort 167, 189, 190

Microprocessor 10, 12, 17
Minicomputer 197, 198, 200
Monte Carlo method 233

N

Nondeterministic finite automaton (NFA) 67, 92

O

Operating system 6, 8, 9, 10, 11

P

Polynomial-time solution 239, 243
Primary memory 196
Processor pool model 201
Pushdown automaton (PDA) 98

Q

Quick sort 171, 190

S

Semiconductor 5, 11
Shell sort 175

T

Turing machine 132, 133, 136, 137, 138, 139, 140, 141, 142, 145, 146, 148, 149, 150, 151, 152, 153, 154, 155, 157, 159, 160
Turing's paper 142, 155

V

Very large scale integration (VLSI) 10

W

Wide area networks (WAN) 11

Level: Beginner to Advanced
Subject: Computer and Information Science

Introduction to Computer Theory

2nd Edition

Theory of Computation or Theory of Automata is the core area of computer science and engineering. It is the branch that aims to try the deep understanding of computational processes by effectively solving the problems with mathematical models, tools and techniques. This understanding is important for its applications, which span various computational models such as algorithms, compilers, and VLSI design to the creation of intelligent technology, cognitive psychology, and philosophy. Automata enables scientists to understand how machines compute functions and solve problems. The main motivation behind the development of automata theory was the development of methods for describing and analyzing the dynamic behavior of discrete systems. To conduct a thorough computational study, computer scientist's work with a mathematical abstraction of computers called a computational model. Several models are used, but the most studied is the Turing machine. Computer scientists study the Turing machine because it is easy to formulate, can be analyzed and used to prove results, and because it represents what many consider to be the most powerful possible reasonable computational model. It may seem that the potentially infinite memory capacity is an unrealizable attribute, but any decidable problem solved by a Turing machine always requires a finite amount of memory. So, in principle, any problem that can be solved by a Turing machine can be solved by a computer that has a finite amount of memory.

The content coverage in this edition is organized into seven chapters. This edition is completely revised and new topics are added. The book addresses the fundamental questions about the nature of computation while at the same time helping we better understand the ways in which we interact with the computer.

ISBN 978-1-98468-255-0

BIBLIOTEX
Digital Library

www.3ge-learning.com
email: info@3ge-learning.com