CIT237
PROGRAMMING AND ALGORITHMS
NATIONAL OPEN UNIVERSITY OF NIGERIA

FACULTY OF SCIENCES

DEPARTMENT OF COMPUTER SCIENCE

COURSE CODE: CIT237

COURSE TITLE: PROGRAMMING AND ALGORITHMS
COURSE GUIDE

CIT237

PROGRAMMING AND ALGORITHMS

Course Developer    Prof. A.S.Sodiya
                    Dept.of Computer Science
                    Universityof Agriculture
                    Abeokuta

Course Reviewer     Prof Olumide Longe
                    American University, Yola
NATIONAL OPEN UNIVERSITY OF NIGERIA

National Open University of Nigeria
Headquarters
91, Cadastral Zone,
Nnamdi Azikwe Express Way,
Jabi, Abuja.
Nigeria.

National Open University of Nigeria
Lagos Office
14/16 Ahmadu Bello Way
Victoria Island
Lagos

e-mail: centralinfo@nou.edu.ng
URL: www.nou.edu.ng

Published by
National Open University of Nigeria

First Printed 2008
Reprinted 2020


All Rights Reserved
## CONTENTS

<table>
<thead>
<tr>
<th></th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td><strong>What You Will Learn in this</strong></td>
<td>1</td>
</tr>
<tr>
<td>Course</td>
<td>2</td>
</tr>
<tr>
<td>Course Aims</td>
<td>2</td>
</tr>
<tr>
<td>Course Objectives</td>
<td>2</td>
</tr>
<tr>
<td>Working through this Course</td>
<td>3</td>
</tr>
<tr>
<td>Course Materials</td>
<td>3</td>
</tr>
<tr>
<td>Study Units</td>
<td>3</td>
</tr>
<tr>
<td>Textbooks and References</td>
<td>4</td>
</tr>
<tr>
<td>Assignment File</td>
<td>7</td>
</tr>
<tr>
<td>Presentation Schedule</td>
<td>8</td>
</tr>
<tr>
<td><strong>Assessment</strong></td>
<td>8</td>
</tr>
<tr>
<td>Tutor-Marked Assignment</td>
<td>8</td>
</tr>
<tr>
<td>Final Examinations and Grading</td>
<td>9</td>
</tr>
<tr>
<td>Course Marking Scheme</td>
<td>9</td>
</tr>
<tr>
<td><strong>Course Overview</strong></td>
<td>10</td>
</tr>
<tr>
<td><strong>How to Get the Best from this</strong></td>
<td>11</td>
</tr>
<tr>
<td>Course</td>
<td>11</td>
</tr>
<tr>
<td>Facilitators/Tutors and Tutorials</td>
<td>12</td>
</tr>
<tr>
<td>Summary</td>
<td>13</td>
</tr>
</tbody>
</table>
Introduction

CIT237—Programming and Algorithms is a three-credit unit course of twenty-one units. This course presents an overview of the methods and concept of programming and the role of algorithms in programming. It covers aspects on programming concepts such as basic data types, algorithms, performance analysis, fundamental data structures, P, NP, and NP-Complete Problems and some sorting algorithms.

This course is divided into three modules. The first module deals with the basic introduction to the concept of programming and algorithms; such as definition and characteristics of algorithms, basic data types and fundamental data structures, program development life cycle, types of programming languages, language translators and their characteristics, tools for program design, etc.

The second module focuses on the performance analysis of algorithms discussing issues such as efficiency attributes (i.e., time and space efficiency), measuring the running time of an algorithm, measuring input size, worst case, best case and average-case efficiencies, P, NP, and NP-Complete problems, etc.

The third module deals with sorting and some special problems. It introduces you to some sorting and divide-and-conquer algorithms after which it goes on to discuss some sorting techniques such as Merge Sort, Bubble Sort, Selection Sort, etc. giving their algorithms and performance analysis.

The aim of this course is to equip you with the basic knowledge of writing efficient programs through the use of concise and efficient algorithms. By the end of the course, you should be able to confidently tackle any programming problem by breaking it into its component parts, write efficient algorithms to solve the problem and implement the algorithm using any programming language of your choice as well as being able to evaluate and measure the performance efficiency of any algorithm.

This Course Guide gives you a brief overview of the course content, course duration, and course materials.

A course on computers can never be complete because of the existing diversities of the computer systems. Therefore, you are advised to read through the further reading to enhance the basic understanding you will acquire from the course material.
What You will Learn in this Course

The main purpose of this course is to introduce you to concepts relating to problem solving through the efficient use of algorithms and subsequent implementation of the algorithm in any language of choice that is suitable to the application area. This, we intend to achieve through the following:

Course Aims

1. Introduce the basic concepts relating to algorithms and programming;
2. Expose the basic relationship that exist between algorithms and program development.
3. Discuss the basic features of algorithms and components of programs.
4. Discuss the fundamental data structures, datatypes, arithmetic operations, etc.
5. Discuss features of programming languages, programming methodologies and application areas, language translators, programming environment, etc.
6. Expose the basics of measuring the efficiencies of algorithms and how to identify basic operations within an algorithm.

Course Objectives

Certain objectives have been set out to ensure that the course achieves its aims. Apart from the course objectives, every unit of this course has set objectives. In the course, the study, you will need to find out, at the end of each unit, if you have met the objectives set at the beginning of the unit. By the end of this course, you should be able to:

1. Define an algorithm, stating its basic characteristics
2. Enumerate the role of an algorithm in problems solving and how it relates to a program.
3. Define the concept of programming and describe the basic features of a program;
4. Explain the program development life cycle
5. Discuss the concept of order of growth and explain the different asymptotic notations
6. Operate the hill climbing technique and show how hill climbing is used to solve problems.
7. Resolve the Knight's Tour problem, describe and resolve ann*n tour problem
8. Explain the measures of algorithm efficiency
9. Explain the identification of basic operations within an algorithm
10. Distinguish between a polynomial and non-polynomial problem
11. Discuss (extensively) P, NP, NP-complete problems
12. Develop algorithms to perform some basic sorting, such as Merge Sort, Selection Sort, Bubble Sort, Quick Sort, etc., on some data, and evaluate the performance of each algorithm.

Working Through This Course

In order to have a thorough understanding of the course units, you will need to read and understand the contents, and practice the steps by solving some simple problems by breaking them into smaller problems and developing algorithms for each. You may then implement your algorithms using any programming language of your choice that is suitable for the application area.

This course is designed to cover approximately sixteen weeks, and it will require your devoted attention. You should do the exercises in the Tutor-Marked Assignments and submit to your tutors.

Course Materials

These include:

1. The Course Guide
2. Study Units
3. Recommended Texts
4. A file for your assignments and for records to monitor your progress.

Study Units

There are 21 study units in this course:

Module 1 Introduction to Programming and Algorithms

Unit 1 Introduction to Programming
Unit 2 Programming Concepts
Unit 3 Algorithms
Unit 4 Basic Data Types
Unit 5 Fundamental Data Structure
Unit 6 Practical Exercise I
Unit 7 Fundamental Data Structures
Unit 8 Exercise I
### Module 2  Performance Analysis of Algorithms

- **Unit 1**: Performance Analysis Framework
- **Unit 2**: Order of Growth
- **Unit 3**: Worst-case, Best-case and Average-case Efficiencies
- **Unit 4**: P, NP and NP-Complete Problems
- **Unit 5**: Practical Exercise II

### Module 3  Sorting and Some Special Problems

- **Unit 1**: Introduction to Sorting and Divide-and-Conquer Algorithm
- **Unit 2**: Merge Sort
- **Unit 3**: Quick Sort
- **Unit 4**: Binary Search
- **Unit 5**: Selection Sort
- **Unit 6**: Bubble Sort
- **Unit 7**: Special Problems and Algorithms
- **Unit 8**: Practical Exercise IV

### Textbooks and References


[www.doc.ic.ac.uk/~wjk/C++Intro/](http://www.doc.ic.ac.uk/~wjk/C++Intro/)


[www.personal.kent.edu/~muhama/Algorithms](http://www.personal.kent.edu/~muhama/Algorithms)


[www.eslearning.algorithms.com](http://www.eslearning.algorithms.com)


Friedel, F. "The Knight's Tour."


Jelliss, G. "Knight's Tour Notes."


Pegg, E. Jr. "Leapers (Chess Knights and the Like)"


Rose, C. "The Distribution of the Knight."


Roth, A. "The Problem of the Knight: A Fast and Simple Algorithm."

http://library.wolfram.com/infocenter/MathSource/909/.

Ruskey, F. "Information on the $n$ Knight's Tour Problem."
http://www.theory.csc.uvic.ca/~cos/inf/misc/Knight.html.


AssignmentsFile

These are of two types: One for the Self-Assessment Exercises and the other for the Tutor-Marked Assignments. The self-assessment exercises will enable you to monitor your performance by yourself, while the tutor-marked assignments will be supervised. The assignments take a certain percentage of your total score in this course. The tutor-marked assignments will be assessed by your tutor within a specified period. The examination at the end of this course will aim at determining your level of mastery of the subject matter. This course includes 21 tutor-marked assignments and each must be done and submitted as stipulated. Your best scores; however, will be recorded for you. Be sure to send these assignments to your tutor before the deadline to avoid loss of marks.

Presentation Schedule

The Presentation Schedule includes in your course materials gives you the important dates for the completion of tutor-marked assignments and the schedule for attending tutorials. Remember, you are required to submit all your assignments by the due date. You should guard against lagging behind in your work.

Assessment

There are two aspects to the assessment of the course. First are the tutor-marked assignments; second, is a written examination.

In tackling the assignments, you are expected to apply the information and knowledge you acquired during this course. The assignments must be submitted to your tutor for formal assessment in accordance with the deadlines stated in the Assignment File. The work you submit to your tutor for assessment will count for 30% of your total course mark.

At the end of the course, you will need to sit for a final three-hour examination. This will also count for 70% of your total course mark.

Tutor-Marked Assignment

There are 21 tutor-marked assignments in this course. You need to submit all the assignments. The total marks for the best four (4) assignments will be 30% of your total course mark.

Assignment questions for the units in this course are contained in the Assignment File. You should be able to complete your assignments
from the information and materials contained in your set textbooks, reading and study units. However, you may wish to use other references to broaden your viewpoint and provide a deeper understanding of the subject.

When you have completed each assignment, send it together with a form to your tutor. Make sure that each assignment reaches your tutor on or before the deadline given. If however, you cannot complete your work on time, contact your tutor before the assignment is done to discuss the possibility of an extension.

**Final Examinations and Grading**

The final examination for the course will carry 70% of the total marks available for this course. The examination will cover every aspect of the course, so you are advised to revise all your corrected assignments before the examination.

This course endows you with the status of a teacher and that of a learner. This means that you teach yourself and that you learn, as your learning capabilities would allow. It also means that you are in a better position to determine and to ascertain the what, the how, and the when of your learning. No teacher imposes any method of learning on you.

The course units are similarly designed with the introduction following the table of contents, then a set of objectives and then the didactic course and so on.

The objectives guide you as you go through the units to ascertain your knowledge of the required terms and expressions.

**Course Marking Scheme**

This table shows how the actual course marking is broken down.

<table>
<thead>
<tr>
<th>Assessment</th>
<th>Marks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment 1-4</td>
<td>Four assignments, best three mark of the four count at 30% of course marks</td>
</tr>
<tr>
<td>Final Examination</td>
<td>70% of overall course marks</td>
</tr>
<tr>
<td>Total</td>
<td>100% of course marks</td>
</tr>
</tbody>
</table>

*Table 1: The course marking scheme*
## Course Overview

<table>
<thead>
<tr>
<th>Unit</th>
<th>Title of Work</th>
<th>Weeks Activity</th>
<th>Assessment (End of Unit)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Course Guide</td>
<td>Week1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Module 1: Introduction to Programming and Algorithm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Introduction to Programming</td>
<td>Week1</td>
<td>Assignment 1</td>
</tr>
<tr>
<td>2</td>
<td>Programming Concepts</td>
<td>Week1</td>
<td>Assignment 2</td>
</tr>
<tr>
<td>3</td>
<td>Algorithm</td>
<td>Week2</td>
<td>Assignment 3</td>
</tr>
<tr>
<td>4</td>
<td>Basic DataTypes</td>
<td>Week2</td>
<td>Assignment 4</td>
</tr>
<tr>
<td>5</td>
<td>Fundamental Data Structure</td>
<td>Week2</td>
<td>Assignment 5</td>
</tr>
<tr>
<td>6</td>
<td>Practical Exercises I</td>
<td>Week3</td>
<td>Assignment 6</td>
</tr>
<tr>
<td>7</td>
<td>Fundamental Data Structures</td>
<td>Week3</td>
<td>Assignment 7</td>
</tr>
<tr>
<td>8</td>
<td>Exercises I</td>
<td>Week3</td>
<td>Assignment 8</td>
</tr>
<tr>
<td></td>
<td><strong>Module 2: Performance Analysis of Algorithms</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Performance Analysis Framework</td>
<td>Week4</td>
<td>Assignment 9</td>
</tr>
<tr>
<td>2</td>
<td>Order of Growth</td>
<td>Week4</td>
<td>Assignment 10</td>
</tr>
<tr>
<td>3</td>
<td>Worst-case, Best-case and Average-case Efficiencies</td>
<td>Week5</td>
<td>Assignment 11</td>
</tr>
<tr>
<td>4</td>
<td>P, NP and NP-Complete Problems</td>
<td>Week 6-7</td>
<td>Assignment 12</td>
</tr>
<tr>
<td>5</td>
<td>Practical Exercise II</td>
<td>Week8</td>
<td>Assignment 13</td>
</tr>
<tr>
<td></td>
<td><strong>Module 3: Sorting and Some Special Problems</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Introduction to Sorting and Divide-and-Conquer Algorithm</td>
<td>Week 9</td>
<td>Assignment 14</td>
</tr>
<tr>
<td>2</td>
<td>Merge Sort</td>
<td>Week10</td>
<td>Assignment 15</td>
</tr>
<tr>
<td>3</td>
<td>Quick Sort</td>
<td>Week10</td>
<td>Assignment 16</td>
</tr>
<tr>
<td>4</td>
<td>Binary Search</td>
<td>Week11</td>
<td>Assignment 17</td>
</tr>
<tr>
<td>5</td>
<td>Selection Sort</td>
<td>Week12</td>
<td>Assignment 18</td>
</tr>
<tr>
<td>6</td>
<td>Bubble Sort</td>
<td>Week13</td>
<td>Assignment 19</td>
</tr>
<tr>
<td>7</td>
<td>Special Problems and Algorithms</td>
<td>Week14</td>
<td>Assignment 20</td>
</tr>
<tr>
<td>8</td>
<td>Practical Exercise IV</td>
<td>Week15</td>
<td>Assignment 21</td>
</tr>
<tr>
<td></td>
<td>Revision</td>
<td>Week16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Examination</td>
<td>Week17</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>17 weeks</td>
<td></td>
</tr>
</tbody>
</table>
How to Get the Best from this Course

In distance learning the study units replace the university lecturer. This is one of the great advantages of distance learning: you can read and work through specially designed study materials at your own pace, and at a time and place that suit you best. Think of it as reading the lecture instead of listening to a lecturer. The same way that a lecturer might set you some reading to do, the study unit tells you when to read your setbooks or other materials. Just as a lecturer might give you an in-class exercise, your study units provide exercises for you to do at appropriate points.

Each of the study units follows a common format. The first item is a common description of the subject matter of the unit and how the particular unit is integrated with the other units and the course as a whole. Next is a set of learning objectives. These objectives enable you to know what you should be able to do by the time you have completed the unit. You should use these objectives to guide your study. When you have finished the units, you must go back and check whether you have achieved the objectives. If you make a habit of doing this, you will significantly improve your chances of passing the course.

Remember that your tutor’s job is to assist you. When you need help, don’t hesitate to call and ask him other topics that you provide.

1. Read this Course Guide thoroughly.

2. Organize a study schedule. Refer to the Course Overview for more details. Note the time you are expected to spend on each unit and how the assignments relate to the units. Whatever method you choose to use, you should decide on it and write in your own dates for working on each unit.

3. Once you have created your own study schedule, do everything you cannot stick to it. The major reason students fail is that they lag behind in their course work.

4. Turn to Unit 1 and read the introduction and the objectives for the unit.

5. Assemble the study materials. Information about what you need for a unit is given in the Overview at the beginning of each unit. You will almost always need both the study unit you are working on and one of your set of books on your desk at the same time.
6. Work through the unit. The content of the unit itself has been arrangedtoprovideasequenceforyoufollow. Asyouwork throughtheunit youwillbe instructedtoreadsectionsfromyourset books or other articles. Use theunittoguideyour reading.

7. Review the objectivesfor each study unit to confirm that you have achieved them. If you feel unsure about any of the objectives, review the study material or consult your tutor.

8. When you are confident that you have achieved a unit’s objectives, you can then start on the next unit. Proceed unit by unit through the course and try to pace your study so that you keep yourself on schedule.

9. When you have submitted an assignment to your tutor for marking, do not wait for its return before starting on the next unit. Keep to yours schedule. When the assignment is returned, pay particular attention to your tutor’s comments, both on the tutor-marked assignment form and on the assignment. Consult your tutor as soon as possible if you have any questions or problems.

10. After completing the last unit, review the course and prepare yourself for the final examination. Check that you have achieved the unit objectives (listed at the beginning of each unit) and the course objectives (listed in this Course Guide).

**Facilitators/Tutors and Tutorials**

There are 15 hours of tutorials provided in support of this course. You will be notified of the dates, times and location of these tutorials, together with the name and phone number of your tutor, as soon as you are allocated a tutorial group.

Your tutor will mark and comment on your assignments, keep a close watch on your progress and mark any difficulties you might encounter and provide assistance for you during the course. You must mail or submit your tutor-marked assignment to your tutor well before the due date (at least two working days are required). They will be marked by your tutor and returned to you as soon as possible.

Do not hesitate to contact your tutor by telephone, or e-mail if you need help. The following might be circumstances in which you would find help necessary. Contact your tutor if:

i. You do not understand any part of the study units or the assigned readings,
ii. You have difficulty with the self-tests or exercises,

iii. You have a question or problem with an assignment, with your tutor’s comments on an assignment or with the grading of an assignment.

You should try your best to attend the tutorials. This is the only chance to have a face-to-face contact with your tutor and to ask questions which are answered instantly. You can raise any problem encountered in the course of your study. To gain the maximum benefit from course tutorials, prepare a question list before attending the classes. You will learn a lot from participating in discussions actively.

**Summary**

Programming and Algorithms, as the title implies, will take you through the fundamental concepts of problem-solving through the use of algorithms and efficient programming. Therefore, you should acquire the basic knowledge of the principles of algorithm development and program writing in this course. The content of the course material was planned and written to ensure that you acquire the proper knowledge and skills in order to be able to write efficient algorithms and implement them using applicable programming languages. For that area of application, the essence is to get you to acquire the necessary knowledge and competence and equip you with the necessary tools.

I wish you success with the course and hope that you will find it interesting and useful.
<table>
<thead>
<tr>
<th>Module 1</th>
<th>Introduction to Programming and Algorithms</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit 1</td>
<td>Introduction to Programming</td>
<td>1</td>
</tr>
<tr>
<td>Unit 2</td>
<td>Programming Concepts</td>
<td>9</td>
</tr>
<tr>
<td>Unit 3</td>
<td>Algorithms</td>
<td>16</td>
</tr>
<tr>
<td>Unit 4</td>
<td>Basic Data Types</td>
<td>22</td>
</tr>
<tr>
<td>Unit 5</td>
<td>Fundamental Data Structure</td>
<td>27</td>
</tr>
<tr>
<td>Unit 6</td>
<td>Practical Exercise I</td>
<td>32</td>
</tr>
<tr>
<td>Unit 7</td>
<td>Fundamental Data Structures</td>
<td>36</td>
</tr>
<tr>
<td>Unit 8</td>
<td>Exercise I</td>
<td>41</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Module 2</th>
<th>Performance Analysis of Algorithms</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit 1</td>
<td>Performance Analysis Framework</td>
<td>44</td>
</tr>
<tr>
<td>Unit 2</td>
<td>Order of Growth</td>
<td>48</td>
</tr>
<tr>
<td>Unit 3</td>
<td>Worst-case, Best-case and Average-case</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>Efficiencies</td>
<td></td>
</tr>
<tr>
<td>Unit 4</td>
<td>P, NP and NP-Complete Problems</td>
<td>58</td>
</tr>
<tr>
<td>Unit 5</td>
<td>Practical Exercise II</td>
<td>64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Module 3</th>
<th>Sorting and Some Special Problems</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit 1</td>
<td>Introduction to Sorting and Divide-and-Conquer Algorithm</td>
<td>66</td>
</tr>
<tr>
<td>Unit 2</td>
<td>Merge Sort</td>
<td>71</td>
</tr>
<tr>
<td>Unit 3</td>
<td>Quick Sort</td>
<td>75</td>
</tr>
<tr>
<td>Unit 4</td>
<td>Binary Search</td>
<td>79</td>
</tr>
<tr>
<td>Unit 5</td>
<td>Selection Sort</td>
<td>82</td>
</tr>
<tr>
<td>Unit 6</td>
<td>Bubble Sort</td>
<td>85</td>
</tr>
<tr>
<td>Unit 7</td>
<td>Special Problems and Algorithms</td>
<td>88</td>
</tr>
<tr>
<td>Unit 8</td>
<td>Practical Exercise IV</td>
<td>96</td>
</tr>
</tbody>
</table>
MODULE 1  INTRODUCTION TO PROGRAMMING AND ALGORITHM

Unit 1  Introduction to Programming
Unit 2  Programming Concepts
Unit 3  Algorithm
Unit 4  Basic Data Types
Unit 5  Fundamental Data Structure
Unit 6  Practical Exercise I
Unit 7  Fundamental Data Structures
Unit 8  Exercise I

UNIT 1  INTRODUCTION TO PROGRAMMING

CONTENTS

1.0  Introduction
2.0  Objectives
3.0  Main Content
   3.1  Meaning and Significance of Programming
   3.2  Levels of Programming Languages
   3.3  Features of Programming Languages
   3.4  Programming Methodologies and Application Areas
   3.5  Language Translators
   3.6  Elements of programming languages
   3.7  Language Evaluation Criteria
   3.8  The Programming Environment
4.0  Conclusion
5.0  Summary
6.0  Tutor-Marked Assignment
7.0  References/Further Readings

1.0  INTRODUCTION

This unit introduces methods and concepts of programming. It also explains how programs are executed by the compilers.

2.0  OBJECTIVES

By the end of this unit you should be able to:

- list programs and programming languages
- outline the different levels of programming languages and their characteristics
- outline the conventional features of programming languages
- outline the method of programming and its application areas
- understand language evaluation criteria
explain language translators, types, and their characteristics outline and explain the environments of programming.

3.0 MAIN CONTENT

3.1 Meaning and Significance of Programming Languages

Programming languages are formal languages through which we can instruct the computer to carry out some processes or tasks in order to produce a more accurate and meaningful results called outputs. Programming language consists of words whose letters are taken from set of alphabets called character set and obey a well-defined set of rules called syntax. In this way, programming languages are used to communicate explicit instruction between human beings and computer systems. These explicit instructions which are often expressed in a computer implementable notation are called algorithms. Programming languages can be used to execute a wide range of algorithms, that is, an instruction could be executed through more than a procedure of execution. The full concept of algorithm will be explained later.

A computer program is a set of instructions (i.e. notations or codes) that can be executed by a computer to perform a particular task or process. The process of writing a computer program is called programming. Programming often involves a number of steps through which computer instructions are transformed into usable computer applications.

3.2 Levels of Programming Languages

Programs and programming languages have been in existence since the invention of computers, and there are three levels of programming languages. These are:

- **Machine Language:** Machine language is a set of binary coded instructions, which consist of zeros (0) and ones (1). Machine language is specific to each type of computer. The first generation of computers was coded in machine language that was specific to each model of computer. Some of the shortcomings of machine language were:
1. Coding in machine language was a very tedious and boring job
2. Machine language was not user-friendly. That is the user had to remember along list of codes, numbers or operation codes and know where instructions were stored in computer memory.
3. Debugging any set of codes is a very difficult task since it requires going through the program instruction from the beginning to the end.
The major advantage of machine language is that it requires no translation since it is already in machine language and is therefore faster to execute.

- **Low Level Language**: This is a level of programming language which is different from the machine language. That is, the instructions are not entirely in binary-coded form. It also consists of some symbolic codes, which are easier to remember than machine codes. In assembly language, memory addresses are referenced by symbols rather than addresses in machine language. Low-level programming language is also called **assembly language**, because it makes use of an assembler to translate codes into machine language. An example of an assembly language statement is:

```
MOVE A1, A2   Move the content of Register A2 to A1
JMP Pb       Go to the process with label Pb
```

The disadvantages of assembly language are that:

- It is specific to particular machines
- It requires a translator called an assembler.

The major advantage of the assembly language is that programs written initially are easier to read and more user-friendly than those written in machine language, especially when comments are inserted in the codes.

- **High Level Language**: This programming language consists of English-like codes. High-level languages are independent of the computer because the programmer only needs to pay attention to the steps or procedures involved in solving the problem for which the program is to be used to execute the problem. High-level languages are usually broken into one or more states such as: Main programs, sub-programs, classes, blocks, functions, procedures, etc. The name given to each component differs from one language to the other.

**Some advantages of high-level language:**

- It is more user-friendly, that is, easy to learn and write
- It is very portable, that is, it can be used on almost any computer
- It saves much time and effort when used compared to any other programming level language.
3.3 Features of Programming Languages

There are some conventional features which a programming language must possess, these features are:

- It must have syntactic rules for forming statements.
- It must have a vocabulary that consists of letters of the alphabet.
- It must have a language structure, which consists of keywords, expressions and statements.
- It may require a translator before it can be understood by a computer.
- Programming languages are written and processed by the computer for the purpose of communicating data between the human being and the computer.

3.4 Programming Methodologies and Application Domain

3.4.1 Methodologies

Some programming methodologies are stated below:

- **Procedural Programming**: A procedural program is a series of steps, each of which performs a calculation, retrieves input, or produces output. Concepts like assignments, loops, sequences and conditional statements are the building blocks of procedural programming. Major procedural programming languages are COBOL, FORTRAN, C, ANDC++.

- **Object-Oriented (OO) Programming**: The OO program is a collection of objects that interact with each other by passing messages that transform their state. The fundamental building blocks of OO programming are object modeling, classification and inheritance. Major object-oriented languages are C++, Java etc.

- **Functional Programming**: A functional program is a collection of mathematical functions, each with an input (domain) and an output (range). Interaction and combination of functions is carried out by functional compositions, conditionals and recursion. Major functional programming languages are Lisp, Scheme, Haskell, and ML.
- **Logic(Declarative)Programming:** A logic programme is a collection of logical declarations about what outcome a function should accomplish rather than how that outcome should be accomplished. Logic programming provides a natural vehicle for expressing non-determinism, since the solution to many problems are often not unique but manifold. The major logic programming language is Prolog.

- **EventDrivenProgramming:** An event-driven program is a continuous loop that responds to events that are generated in an unpredictable order. These events originate from user actions on the screen (mouse clicks or keystrokes, for example), or else from other sources (like readings from sensors or a robot). Major event-driven programming languages include Visual Basic and Java.

- **ConcurrentProgramming:** A concurrent program is a collection of cooperating processes, sharing information with each other from time to time but generally operating asynchronously. Concurrent programming languages include SR, Linda, and High Performance FORTRAN.

### 3.4.2 Application Areas

The programming communities that represent distinct application areas can be grouped in the following ways:

- **Scientific Computing:** It is concerned with making complex calculations very fast and very accurately. The calculations are defined by mathematical models, which represent scientific phenomena. Examples of scientific programming languages include Fortran 90, C, and High Performance Fortran.

- **Management Information System (MIS):** Programs for use by institutions to manage their information systems are probably the most prolific in the world. These systems include an organization’s payroll system, online sales and marketing systems, inventory and manufacturing systems, and so forth. Traditionally, MIS have been developed in programming languages like COBOL, RPG, and SQL.

- **Artificial Intelligence:** The artificial intelligence programming community has been active since the early 1960s. This community is concerned about developing
program that model human intelligent behaviour, logical deduction, and cognition. Examples of AI programming languages are prominent functional and logic programming languages like Prolog, CLP, ML, Lisp, Scheme and Haskell.

- Systems: System programmers are those who design and maintain the basic software that runs systems—operating system components, network software, programming language compilers and debuggers, virtual machines and interpreters, and so on. Some of these programs are written in the assembly language of the machine, while many others are written in a language specifically designed for systems programming. The primary example of a system programming language is C.

- Web-centric: The most dynamic area of new programming community growth is the Worldwide Web, which is the enabling vehicle for electronic commerce and a wide range of applications in academia, government, and industry. The notion of Web-centric computing, and then Web-centric programming, is motivated by an interactive model, in which a program remains in an infinite loop waiting for the next request to eventuate, responding to that event, and returning to its looping state. Programming languages that support Web-centric computing require a paradigm that encourages system-user interaction, event-driven programming. Programming languages that support Web-centric computing include Perl, Tcl/Tk, Visual Basic, and Java.

- Mobile Computing: This is one of the newer areas of programming technology. Mobile computing often involves technology which allow for transport of data, voice and video over a network via a mobile device. This new area of programming paradigm allows for connectivity, personalization and social engagement via availability of different applications or apps. Mobile computing supports a number of devices such as Tablets, smart phones, hand held gaming devices, wearable devices etc. The programming languages for mobile computing include Python, Kotlin, SWIFT, Go etc.

- Cloud Computing: This is one of the newer emerging technologies that is rapidly changing the face of internet and programming. It is a programming technology in
which programs are designed to provide on-demand availability of computer system resources over data storage without direct active management by the subscriber or user. Businesses are now employing cloud computing in different ways to maintain users’ information on private servers or public servers on the internet. The most popular examples of cloud computing infrastructures include Google cloud, IBM cloud, Amazon web services etc. Most common programming languages for cloud computing include Python, Clojure, Erlang, Haskell etc.

3.5 Translators

A translator is a program that translates another program written in any programming language other than the machine language to an understandable set of codes for the computer and insodoing produces a program that may be executed on the computer. The need for a translator arises because only a program that is directly executable on a computer is the machine language. Examples of a translator are:

1. **Assembler:** This is a program that converts programs written in assembly or low-level language to machine language.

2. **Interpreters and Compilers:** These consist of programs that convert programs in high-level programming language into machine language. The major difference between interpreters and compilers is that a compiler converts the entire resource program
into object code before the entire program is executed while the interpreter translates the source instructions line by line. In the former, the computer immediately executes one instruction before translating the next instruction.

### 3.5.1 Features of Translators

They exist to make programs understandable by the computer. There exist different translators for different levels and types of programming languages. Without them, the programs cannot be executed.
3.6 Element of Programming languages
All programming languages have some basic building blocks for describing data and processes or transformations applied to them. There are two main elements of all programming languages namely syntax and semantics.

- Syntax: The syntax of a programming language describes the possible combinations of symbols from the language’s character set that form a syntactically correct program. Programming language syntax is usually defined using a combination of regular expressions and Backus Naur Form.

- Semantics: The semantic of a programming language describes the meaning of languages. Semantics comes in different forms. For instance, the static semantics defines restrictions on the structure of valid texts that are hard or impossible to express in standard syntactic formalisms while the dynamic semantics defines how and when the various constructs of a language should produce a program behavior.

3.7 Language Evaluation Criteria
The following are the criteria that influences the evaluation of computer languages:

- Readability: This describes the ease with which programs can be read and understood and it is one of the most important criteria for evaluating a programming language.

- Simplicity: This describes the ability of programming languages to use familiar symbols for their basic operations and computations.

- Reliability: This describes the ability of a programming language to perform to its specification under all conditions by providing for exception handling, type checking, error overthrow etc.

- Expressivity: This describes the ability of a programming language to clearly reflect the meaning intended by a programmer by using notations which are consistent with those used in the field for which the language is designed.
- Pedagogy: This is the ability to teach and learn programming language which is usually evaluated by its clarity and simplicity of instructions and programming constructs.

### 3.8 The Programming Environment

**The Editor:** An editor allows a program to be retrieved from the disk and amended as necessary. In order to type any program on the keyboard and save the program on a disk, it will be necessary to run a program called an editor.

**The Compiler:** This will translate a program written in high-level language stored in a text mode on a disk into a program stored in a machine-oriented language on a disk.

**The Linker/Loader:** A linker/loader picks up the machine-oriented program and combines it with any necessary software (already in machine-oriented form) to enable the program to be run. Before a compiled program can be run or executed by the computer, it must be converted into an executable form.

### 4.0 CONCLUSION

In the course of these units you were introduced to the concept of programming. You also learned about the idea of programming languages and the various types and methodologies involved in writing a program. Conclusively, you have learnt about the various fields in which programming language could be implemented. We finished this course by looking at various interpreters and the various features of a programming environment.

### 5.0 SUMMARY

In this unit you learnt that:

- Programming languages are languages through which we can instruct the computer to carry out processes and tasks.
- A program is a set of codes that instruct the computer to carry out some processes while programming is the act of writing programs.
- There are four levels of the programming language-machine language, low-level language, assembly language and high level language.
- There are various programming methodologies, of which we have procedural programming, object-
oriented programming, functional, logic (declarative), event-driven and concurrent programming.

- There are basically three types of translators - assembler, interpreters and compilers.
- Programming environment consists of the editor, translator and the linker/loader.
- Language evaluation criteria are Readability, Simplicity, Reliability, Expressivity and Pedagogy.
6.0 TUTOR-MARKED ASSIGNMENT

a. Write out any ten programming languages stating the application areas
b. How does pedagogy affect language selection?
c. Using a practical example, differentiate between syntax and semantic
d. State ten distinct programming application areas

7.0 REFERENCES/FURTHER READINGS


Pascal Programming Continuum (2nd ed).


Published by Morgan Kaufmann.

UNIT2 PROGRAMMINGCONCEPTS

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 MainContent
   3.1 ProgramDevelopment CycleC 7,D4
   3.2 ProgramExecution Stages
   3.3 PrinciplesofGoodProgramming Style
   3.4 Programming paradigms
   3.5 Object-oriented modeling
4.0 Conclusion
5.0 Summary
6.0 Tutor-MarkedAssignment
7.0 References/FurtherReadings

1.0 INTRODUCTION

Thisunitintroducesyouto programming concepts. These includethe programmingdevelopment cyclewhichconsistsofthestagesinvolved in developing an efficient program. It also introduces you to the program execution stages, as well as the conventional principles of good programming

2.0 OBJECTIVES

Havinggonethroughthisunit, youshouldbeableto:

✔ explainthefivemajorstepsinvolvedindevelopinganefficient program
✔ outlinethefourstagesinvolvedintheexecutionofanormal program
✔ outlinetheprinciplesof a goodprogramming style.
✔ understand programming paradigms

3.0 MAINCONTENT

3.1 TheProgramDevelopmentCycle

Program Development cycle depicts the various stages involved in the lifespan of a computer program from its origin until completion or closure. Program development cycle addresses three main attributes namely stakeholders of the program (i.e. organization requesting the computer application), benefits acquired from the program (i.e. the
deliverables) and rules governing the development of the program lifecycle (i.e. expectation and documentations). In program development, a program is constituted by two fundamental parts namely the objects and the operations. The object is a representation of the data relative to the domain of interest while the operations describe how the objects are to be manipulated in such a way to realize the desired outputs. The various stages of program development lifecycle are discussed relative to these two fundamental parts.

The major five stages involved in developing an efficient program are:-

- **Problem Analysis**: This is where the clear statement of the problem is stated. The programmer must be sure that he understands the problem and how to solve it. He must know what is expected of the problem, i.e. what the program should do, the nature of the output and the input to consider so as to get the output. He must also understand
the ways of solving the problem and the relationship between input and expected output.

- **Design:** The planning of the solution of the problem in the first stage takes place in this stage. The planning consists of the process of finding a logical sequence of precise steps that solve the problem. Such a sequence of steps is called an **algorithm**. Every detail, including obvious steps, should appear in the algorithm. The three popular methods used to develop the logic plan are: flowcharts, pseudo code, and a top-down chart. These tools help the programmer break down a problem into a sequence of small tasks the computer can perform to solve the problem. Planning may also involve using representative data to test the logic of the algorithm by hand to ensure that it is correct.

- **Coding:** Translation of the algorithm into stage two into a programming language is called **coding**. The programmer uses the algorithm devised in the design stage along with the choice of the programming language he got from stage three.

- **Testing and Debugging:** The process involves the location and removal of errors in the program if any. Testing is the process of checking if the program is working as expected and finding errors in the program, and debugging is the process of correcting errors that are found (An error in a program is called a bug.).

- **Documentation:** This is the final stage of program development. It consists of organizing all the material that describes the program. The documentation of the program is intended to allow another person or the programmer at a later date to understand the program. Internal documentation remarks consist of statements in the program that are not executed, but point out the purpose of various parts of the program. Documentation might also consist of a detailed description of what the program does and how to use the program. Other types of documentation are flowchart and pseudo code that were used to construct the program. Although documentation is listed as the last step in the program development cycle, it should take place as the program is being coded. It is sometimes the first step during program execution because the programmer can use another program documentation in
developing a new program by just improving on the previous work.

### 3.2 Program Execution Stages

The normal program execution consists of four stages (See figure 2.1), though some programming languages like BASIC combine two or three of these in one single process. The program execution stages are explained below:

- **The Program (Source Code):** This is the code given to the computer in a particular programming language in order to accomplish a given task. The source code must obey the syntactic and semantic rules of the source programming language.

- **The Compilation Process:** The source code is supplied to the compiler, which converts the object code. The process of compilation involves reading the source code, checking for errors in the source code and converting it to an
executable format(machinecode) if noerror isdetected,
else the process of compilation is aborted and an error is
reported.

- **TheObjectCode**: The object code is the result of the
  compilation process and it is also called the target code.
The object code is dependent on the programming
language chosen. For instance, the object code of JAVA
compilation is a bytecode, that of Fortran is an executable
statement of the target machine, while that of BASIC is that of
the target machine language, but it is not written to any
file like that of Fortran and JAVA.

- **TheOutput**: The last stage is for the computer to give the
  result. The computer executes the object code in order to
  present the desired output. It is important to note that a valid
  ordered output might not be given if the logic of
  the program is not correct.

### 3.3 Principles of a Good Programming Style

The following represent the major considerations in writing good
programs:

1. **Naming Conventions**: It is very important to give
   meaningful names to all your constructs. A name like
   `get_Height()` or `get_avg_height()` gives us much
more information than `ctunde()` . Also, a variable name `-total`
for addition is more meaningful than `pen`. The name of a class
should communicate its purpose. Classnames should start with uppercase letter, e.g. `class AddPrime`.

Major variables, which are shared by multiple functions and/or modules should be identified and named at the design stage itself. Variable name should start with a lowercase letter, e.g. `firstQuad`.

Function should be named similar to variables. We can always distinguish between them because of the parenthesis associated with functions.

2. **File Naming and Organisation**: Files should be organised into directories in a module-wise fashion instead of having a monolithic structure where all source code files and all header files are in a single directory. This should be part of the design process.

3. **Formatting and Indentation**: The lines within the code should be clearly organised in a way that it will be easy to read and understand even for the writer. Proper indentation should be used to show subordinates and lines.

4. **Comments and Documentation**: Introducing comments and proper explanations (documentation) of the program aid in understanding the code. They help in following the program flow and skipping parts for which we are not interested in details. This allows for program amendment and extensibility.

5. **Classes**: Ensure that all the classes in your application have a default constructor, copy constructor, and overloaded operators. Also ensure that all the class data items are appropriately initialised in the constructor and assigned to each member of the class.

6. **Functions**: A function should normally do only one job and do it well. Avoid generic functions with lots of conditional branches to do everything. If a function is supposed to do multiple jobs, then create helper functions and delegate responsibilities to them. Make functions simple and small. The ideal size of functions is around 35-40 lines.

7. **Using STL**: Use Standard Template Library (STL) instead of creating your own container data structures. Do not use hash maps in STL; they are not portable across platforms.
8. **Pointers and References**: User references, especially if coding in C++, encourage the use of references instead of pointers. In fact, a pointer should typically be passed to a function only in cases where you need to execute something on the pointer being an null condition.

9. **Minimising Bugs by Testing**: Testing is an integral part of software development. Testing helps not only in making sure that what we have written is correct, but also in finding out if someone breaks the code later. So, it is a good programming style to thoroughly test a program.

3.4 **Programming paradigms**

There are several programming paradigms used in program life cycle development. Each programming paradigm differs in the emphasis put on the two fundamental aspects of programming which are objects and operations. The objects are entities that receive instructions to perform a particular method or actions while the operations are the events that direct the behavior of an object. The three main programming paradigms are:

- **Imperative**: This paradigm places emphasis on the operations intended as actions that change the state of the computations. The objects are functional to the computation.

- **Functional**: This paradigm puts emphasis or development on the operations intended as functions that compute results. In this case, the objects are functional to the computation like in the Imperative type.

- **Object-oriented**: This paradigm places emphasis or development on the objects which serves as the domain of interest in the overall picture of the system. In this case, the operations are functional to the representation.
Imperative/Functional paradigm  Object-oriented paradigm

Usually, every programming languages provide support for these three programming paradigms as a program may use different paradigms within the development life cycle of a program depending on the ease and functionality intended at hand.

3.5 Object-oriented Modeling
Object-oriented modeling (OOM) is a common approach to modeling programs by using object-oriented paradigm throughout the entire development life cycle. This is the main technique used in modern software engineering. The OOM typically divides programming life cycle into two aspects:
- Modeling of the dynamic behaviors like processes and use cases
- Modeling the static structures like classes and components

The advantages of using OOM are
- Efficient and effective communications between the system and the real world
- Useful and stable abstractions that define essential structures and behavior within the system under development

4.0 CONCLUSION
In the course of the program, the students should be able to write a good program following a good programming convention, apart from learning the cycle of program development. The program execution stages are also not left out of this unit.

5.0 SUMMARY
- Problem Analysis - This is where the clear statement of the problem is stated.
- Design - The planning of the solution to the problem in the first stage takes place in this stage.
- Planning may also involve using representative data to test the logic of the algorithm by hand to ensure that it is correct.
- Coding - Translation of the algorithm in stage two into a programming language takes place.
- The process for writing the program is called coding.
- Testing and debugging - The process involves the location and removal of errors (if any) in the program.
- Documentation - This is the final stage of program development; it consists of organizing all the material that describes the program.
- The normal program execution consists of four (4) stages.
• The programme (source code) – This is the set of coded instructions given to the computer to perform a particular task. The process of compilation involves reading the source code and checking for errors.
• The object code – The object code is the result of the compilation process and is also called the target code.
• It is very important to give meaningful names to all your constructs. A name like `get_Height()` or `get_avg_height()` gives us much more information than `ctunde()`.
The variable name should start with a lowercase letter, e.g. 
\texttt{firstQuad};
The name of a class should communicate its purpose. 
A class name should start with an uppercase letter. 
Files should be organised into directories in a module-wise fashion. 
Introducing comments and proper explanations (documentation) 
of the program helps in understanding the code. 
Ensure that all the classes in your application have a default constructor. 
Avoid generic functions with lots of conditional branches - todo everything. 
Use the Standard Template Library (STL) instead of creating your own container data structures. 
Programming paradigms are generally classified as imperative, functional and object-oriented

\section*{6.0 TUTOR-MARKED ASSIGNMENT}

\begin{enumerate}
\item What are the major five stages involved in developing an efficient program?
\item What is the final stage of program development?
\item Is the process of checking if the program is working or not working alright.
\item Draw the program execution chart.
\item What do you understand by compilation?
\item What do you understand by logical error?
\item The computer executes the object code in order to present the desired _
\item What are the principles of a good programming language?
\item Which of these is a good variable name following a good programming? 
\hspace{1cm} conventions: 
\hspace{1cm} (a) variable  \hspace{0.5cm} (b) variable2  \hspace{0.5cm} (c) 2variable 
\hspace{1cm} (d) math()
\item Differentiate between a function and a variable.
\item How do you make a clumsy code look neat and readable?
\item What is the importance of comments?
\item Is an integral part of software development.
\item What do you understand by a good programming style?
\item State the main goal of object-oriented technology
\item What do you understand by object in object-oriented?
\end{enumerate}
7.0 REFERENCES/FURTHER READINGS


www.doc.ic.ac.uk/~wjk/C++Intro/

www.personal.kent.edu/~muhama/Algorithms


https://en.wikipedia.org/wiki/Object-oriented_analysis_and_design
UNIT3   ALGORITHMS

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 MainContent
   3.1 Introduction to Algorithms Page 3
   3.2 Computational Problems and Algorithms
   3.3 Characteristics of Algorithm C
   3.4 Algorithm Design and Analysis Stages Page 9
   3.5 Relationship between computers and algorithms
   3.6 Types of Major Computing Problems
4.0 Conclusion
5.0 Summary
6.0 Tutor-Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

In this unit, you will be introduced to algorithms as well as the necessary conditions to design a good algorithm. The unit also highlights important stages in designing an algorithm. The characteristics of a good algorithm are also outlined, among which is the fact that a good algorithm must have a beginning and an end.

2.0 OBJECTIVES

By the end of this unit, you should be able to:

- explain what an algorithm is
- differentiate between computational problems and algorithms outlining the characteristics of an algorithm
- explain the stages in the design of an algorithm
- understand the relationship between computer and algorithm
- explain types of algorithm

3.0 MAIN CONTENT

3.1 Introduction to Algorithms

What is an algorithm? Although there is no universally agreed-on wording to describe this notion, there is general agreement about what the concept means:

...
An algorithm is a finite sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.
As shown in the above figure, after a problem has been identified, the problem is then carefully analysed in order to present a suitable algorithm. An algorithm is then designed and presented to the computer in a particular programming language. The computer will then generate the output, based on the input. Hence, an algorithm presents a well-defined computational procedure that takes some values as input and produces some value as output.

In general, an effective algorithm has three main characteristics:
- Explicit, complete and precise initial conditions (input)
- A finite, complete but not necessarily linear (i.e. looping, recursive, sequential) series of steps to arrive at the desired results (process)
- Explicit, complete and precise terminal (stopping) conditions (output)

Example: Converting Fahrenheit to Celsius
The example describe the relationship between the Celsius and Fahrenheit scales as follows:

\[ 5(F - 32) = 9C \]  

(1)

where \( F \) = temperature in degrees Fahrenheit, and \( C \) = temperature in degrees Celsius.

Formula (1) defines the relationship between temperatures in Celsius and Fahrenheit, but it doesn't give us an explicit algorithm for converting from one to the other. Fortunately, our understanding of algebra can easily allow us write such an algorithm.
First, we can use the basic operations of algebra to convert (1) into a form that expresses \( C \) as a function of \( F \):

\[
C = \frac{5(F - 32)}{9} \tag{2}
\]

Now it's a straightforward task to write an algorithm (based on the standard order of arithmetic operations) to convert from Fahrenheit to Celsius.

1. Start with a given temperature in degrees Fahrenheit.
2. Subtract 32 from the value used in step #1.
3. Multiply the result of step #2 by 5.
4. Divide the result of step #3 by 9.
5. The result of step #4 is the temperature in degrees Celsius.

Algorithm 1: Conversion from Fahrenheit to Celsius

The skills required to effectively design and analyze algorithms are entangled with the skills required to effectively describe algorithms. At least, a complete description of any algorithm has four components:
- **What**: A precise specification of the problem that the algorithm solves.
- **How**: A precise description of the algorithm itself.
- **Why**: A proof that the algorithm solves the problem it is supposed to solve.
- **How fast**: An analysis of the running time of the algorithm.

It is not necessary (or even advisable) to develop these four components in this particular order. Problem specifications, algorithm descriptions, correctness proofs, and time analyses usually evolve simultaneously, with the development of each component informing the development of the others.

### 3.2 Computational Problems and Algorithms

**Definition 1**: A computational problem is a specification of the desired input-output relationship.

**Definition 2**: An instance of a problem is all the inputs needed to compute a solution to the problem.

**Definition 3**: An algorithm is a well-defined computational procedure that transforms inputs into outputs, achieving the desired input-output relationship.

**Definition 4**: A correct algorithm halts with the correct output for every input instance. We can then simply say that an algorithm is a procedure for solving computational problems.

### 3.3 Characteristics of Algorithms

30 30
The following are the major considerations in the design of algorithms:

An algorithm must have a beginning and an end.
Then, non-ambiguity requirement for each step of an algorithm cannot be compromised.
The range of inputs for which an algorithm works has to be specified carefully. The same algorithm can be represented in several different ways. Several algorithms for solving the same problem may exist. Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds. It must terminate at a reasonable period of time.

3.4 Algorithm Design and Analysis Stages

The diagram below represents the stages in algorithm design.

![Algorithm Design Diagram]

**Problem Definition**

Conventionally, when providing solutions for any given problem, the problem solver must fully have the understanding of the problem, that is, he/she must think about the problem's exceptional cases and must ask questions again and again in order to avoid doubt(s), and fully understand the subject matter.
Identifying Computational Requirement(s)

After the programmer has fully understood the problem, all he/she needs do is to identify the computational requirement(s) needed to solve the problem.

Algorithmic Decisions (Pre-Design Decisions)

Before a programmer designs an algorithm, he/she decides the method to implement in solving the problem, whether its exact or approximate, which are called exact algorithm or approximate algorithm respectively. Also during this stage, the programmer decides and chooses the appropriate data structure needed to represent the inputs.

Algorithm Design

In this phase, the programmer battles with the problem of how he or she should design an algorithm to solve the given problem. Also, the programmer specifies the fashion in which the algorithm will follow, either pseudocode algorithm fashion or Euclid algorithm fashion.

Algorithm Evaluation

The algorithm evaluation phase tests the testing phase whereby the programmer confirms that the algorithm yields the desired result for the right input that is in a reasonable amount of time. The programmer proves the correctness of the algorithm.

Proving an Algorithm’s Correctness

Since an algorithm has been specified, you have to prove its correctness. That is, you have to prove that the algorithm yields the desired result for every legitimate input in a finite amount of time.

Algorithm Analysis

In this phase, we check the efficiency of the algorithm in terms of time and space which are termed as time efficiency and space efficiency respectively.

Coding the Algorithm

Most algorithms finally transit into computer programs. The transition (coding of an algorithm) involves a challenge and an opportunity. The challenge is the development of the algorithm into a program, either incorrectly or inefficiently, while the opportunity is that the coded
algorithm eventually becomes an automated solution to the given problem.

3.5 **Relationship between computer and algorithms**

Computers are electronic devices whose amazing feats of calculation and memory depend largely on its internal and external programs. These programs consist of step-by-step procedures for the computer to follow to produce specific results. In other words, computer programs is almost about algorithms. For instance, when we write a line of Java or Python code to compose a formula for a cell in a spreadsheet, we are using algorithms that others have written as building blocks for performing the required operations.

3.6 **Types of Algorithms**

1. **Recursive Algorithms**

These are algorithms that have the same function calling themselves. For example, the recursive algorithm for the Fibonacci example is:

   Algorithm f(n)
   
   F(0)=0;
   F(1)= 1;
   For I = 2 to N
   
   F(I) = F(I-1) + F(I-2);
   RETURN F(N);

In this algorithm, we can see functions like F(I-1), F(I), F(I-2) calling / referring to the same algorithm. This case is also referred to as a **recursion**. Factorial problem is another example of such algorithm.

2. **Non-recursive Algorithms**

These are algorithms that do not call back the same algorithm or function. For example, write a program to generate Fibonacci sequence.

   M = 1
   N = 2
   I = 2
   WRITEM
   WRITEN
   30 L = N
   N = N + M
   WRITEN
   M = L
   I = I + 1
   IF I <= 30 GOTO 30
END

4.0 CONCLUSION

In this unit, you have learned how to design an efficient algorithm. You were also shown the difference between computational problems and algorithms. This unit also explained the stages in the design of an efficient algorithm.
5.0 SUMMARY

This unit has explained what an algorithm is and the necessary considerations to design a good algorithm. It has also examined the important stages to take in the design of an algorithm, as well as the characteristics of algorithms.

6.0 TUTOR-MARKED ASSIGNMENT

1. In one sentence, define an algorithm.
2. List and explain five (5) stages involved in the design of an algorithm.
3. Write a recursive algorithm for finding the factorial of any given number.

7.0 REFERENCES/FURTHER READINGS


www.doc.ic.ac.uk/~wjk/C++Intro/

UNIT 4  PROGRAM DESIGN TOOLS

CONTENTS

1.0  Introduction
2.0  Objectives
3.0  MainContent
   3.1  Introduction
   3.2  Flowcharts
   3.3  Pseudocodes
4.0  Conclusion
5.0  Summary
6.0  Tutor-marked Assignment
7.0  References/Further Readings

1.0  INTRODUCTION

In this unit, you will be introduced to algorithms as well as the necessary conditions to design a good algorithm. The unit highlights important stages to design an algorithm and also outlines the characteristics of a good algorithm, one of which is that a good algorithm must have a beginning and an end.

2.0  OBJECTIVES

At the end of this unit, you should be able to:

✓ identify various tools used to represent an algorithm otherwise known as programming tools
✓ understand the symbols and functions of each pictorial component of a flow-chart
✓ explain what pseudocodes are and their advantages
   - differentiate between flowcharts and pseudocodes.

3.0  MAIN CONTENT

3.1 Introduction to Programming Tools

It has been stated earlier that an algorithm is a set of procedures for solving a problem. The tools used to clearly represent an algorithm are programming tools.

Example: Problem: Design an algorithm to find the average of two numbers.
**Discussion:** Since an algorithm is just the solution steps for a problem, it can be represented by ordinary English expressions.

**Solution:**

1. Start
2. Get the first number
3. Get the second number
4. Add the two numbers together
5. Show the result
6. Stop

**3.2 Flowcharts**

A flowchart consists of special geometric symbols connected by arrows. Within each symbol is a phrase representing the activity at that step. The shape of the symbol indicates the type of operation that is to occur. For instance, the parallelogram denotes input or output. The arrows connecting the symbols, called **flow lines**, show the progression in which the steps take place. Flowchart should “flow” from the top of the page to the bottom. Although the symbols used in flowcharts are standardised, no standard exists for the amount of detail required within each symbol.

A table of the flowchart symbols adopted by the American National Standards Institute (ANSI) follows (Figure 4). Figure 5 shows the flowchart for the postage stamp problem.

The main advantage of using a flowchart to plan a task is that it provides a pictorial representation of the task, which makes the logic easier to follow. We can clearly see every step and how each step is connected to the next. The major disadvantage with flowcharts is that when a program is very large, the flowchart may continue for many pages, making them difficult to follow and modify.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>➔</td>
<td>Flow line</td>
<td>Used to connect symbols and indicate the flow of logic.</td>
</tr>
<tr>
<td></td>
<td>Terminal</td>
<td>Used to represent the beginning (start) or the end (end) of a task.</td>
</tr>
<tr>
<td></td>
<td>Input/Output</td>
<td>Used for input and output operations, such as reading and printing. The data to be read or printed are described inside.</td>
</tr>
<tr>
<td></td>
<td>Processing</td>
<td>Used for arithmetic and data-manipulation operations. The instructions are listed inside the symbol.</td>
</tr>
<tr>
<td></td>
<td>Decision</td>
<td>Used for any logic or comparison operations. Unlike the input/output and processing symbols, which have one entry and one exit flowline, the decision symbol has one entry and two exit paths. The path chosen depends on whether the answer to a question is “yes” or “no”.</td>
</tr>
<tr>
<td></td>
<td>Off page</td>
<td>Used to indicate that the flowchart continues on a second page.</td>
</tr>
<tr>
<td></td>
<td>Connector</td>
<td>Used to join different flowlines.</td>
</tr>
<tr>
<td></td>
<td>Predefined</td>
<td>Used to represent a group of statements that perform one processing task.</td>
</tr>
<tr>
<td></td>
<td>Annotation</td>
<td>Used to provide additional information about another flowchart symbol.</td>
</tr>
</tbody>
</table>

*Figure 4: Table of the flowchart symbols adopted by the American National Standards Institute (ANSI)*
Add the two numbers \((C = A + B)\)

Stop

Figure 5: The flowchart for the postage stamp problem

### 3.3 Pseudocodes

A pseudo code is an abbreviated version of an actual computer code (hence, the term pseudo code). The geometric symbols used in flowcharts are replaced by English-like statements that outline the process. As a result, a pseudocode looks more like a computer code than a flowchart does. The pseudocode allows the programmer to focus on the steps required to solve a problem rather than on how to use the computer language. The programmer can describe the algorithm in Visual Basic-like form without being restricted by the rules of Visual Basic. When the pseudocode is completed, it can be easily translated into the Visual Basic language.

The pseudocode has several advantages. It is compact and probably will not extend from many pages as a flowchart would. Also, the plan looks like the code to be written and so is preferred by many programmers.
The pseudocode for the example in 3.2 is given below:

Step 1: Start  Step 2
Input A  Step 3
Input B  Step 4
C = A + B  Step 5
Print C  Step 6: Stop

4.0 CONCLUSION

The unit introduced the tools used to represent an algorithm, known as programming tools, such as flowcharts and pseudocodes. You were also introduced to the symbol names and meaning of pictorial components of a flowchart.

5.0 SUMMARY

This unit has dealt with programming tools (flow-charts and pseudo code). It has also showed the diagrams used and the English-like statements used to represent an algorithm. The unit also stated the advantages of both pseudocodes and flowcharts.

6.0 TUTOR-MARKED ASSIGNMENT

a. Using the flowchart only, design an algorithm to find the mean of five numbers.

b. Write the pseudocode of the flowchart in (a) above.

7.0 REFERENCES/FURTHER READINGS


www.doc.ic.ac.uk/~wjk/C++Intro/

www.personal.kent.edu/~muhama/Algorithms
UNIT5   PROGRAM TESTING, DOCUMENTATION & MAINTENANCE

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 MainContent
   3.1 ProgramTesting
   3.2 ProgramDocumentation
   3.3 Program Maintenance
4.0 Conclusion
5.0 Summary
6.0 Tutor-MarkedAssignment
7.0 References/FurtherReadings

1.0 INTRODUCTION

Thisunitintroducesyou testing, documentation, and maintenance of programs indetails.

2.0 OBJECTIVES

After completing thisunit you should beableto:

 ✓ outline what is meant by program testing and the reason why it may be labor intensive
 ✓ explain program documentation and the two major or reasons for program documentation
 ✓ explain why program maintenance is important.

3.0 MAINCONTENT

3.1 Program Testing

Program testing is an integral component of software development and it is performed to determine the existence, quality, and genuineness of the attributes of the program of application.

Program testing is done in the way that the program is run on some test cases and the result of the program’s performance are examined to check whether the program is working as expected. It is also important to perform test processes every condition or attribute that determines the effective/correct functionality of the system. The testing process
normally begins with selecting the test factor(s). The test factors determine whether the program is working correctly and efficiently.
Testing is generally focused on two areas: internal efficiency and external effectiveness. The goal of external effectiveness testing is to verify that the software is functioning according to system design, and that it is performing all the necessary functions or sub-functions. The goal of internal testing is to make sure that the computer code is efficient, standardized, and well documented. Testing can be a labor-intensive process, due to its iterative nature.

1. **Structural System Testing**: This is designed to verify that the developed system and programs work correctly. Its components include:

   - Stress testing
   - Recovery testing
   - Compliance testing
   - Execution testing
   - Operation testing
   - Security testing

<table>
<thead>
<tr>
<th>TECHNIQUE</th>
<th>DESCRIPTION</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRESS</td>
<td>Determinethat the system still performs with expected volumes</td>
<td>Sufficient disk space allocation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Communication lines adequate</td>
</tr>
<tr>
<td>EXECUTION</td>
<td>System achieves desired level of proficiency</td>
<td>Transaction turnaround</td>
</tr>
<tr>
<td></td>
<td></td>
<td>time adequate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Software/hardware use optimized</td>
</tr>
<tr>
<td>RECOVERY</td>
<td>System can be returned to an operational status after a failure</td>
<td>Induce failure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Evaluate adequacy of backup data</td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>System can be executed in a normal operational status</td>
<td>Determine systems can run using document</td>
</tr>
<tr>
<td></td>
<td></td>
<td>JCL adequate</td>
</tr>
<tr>
<td>COMPLIANCE (TO PROCESS)</td>
<td>System is developed in accordance with standards and procedures</td>
<td>Standards followed Documentation complete</td>
</tr>
<tr>
<td>SECURITY</td>
<td>System is protected in accordance with importance to organisation</td>
<td>Access denied</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Procedures in place</td>
</tr>
</tbody>
</table>
2. **Functional System Testing**: This is designed to ensure that the system requirements and specifications are achieved. Its components include:

- Requirement testing
- Error-handling testing
- Inter-system testing
- Parallel testing
- Regressing testing
- Manual-support testing
- Control testing

<table>
<thead>
<tr>
<th>TECHNIQUE</th>
<th>DESCRIPTION</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQUIREMENTS</td>
<td>System performs as specified</td>
<td>Prove system requirements</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compliance to policies regulations</td>
</tr>
<tr>
<td>REGRESSION</td>
<td>Verifies that anything unchanged still performs correctly</td>
<td>Unchanged system segments function</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unchanged manual procedures correct</td>
</tr>
<tr>
<td>ERROR HANDLING</td>
<td>Errors can be prevented or detected, and then corrected</td>
<td>Error introduced into test</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Errors re-entered</td>
</tr>
<tr>
<td>MANUAL SUPPORT</td>
<td>The people-computer interaction works</td>
<td>Manual procedures developed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>People trained</td>
</tr>
<tr>
<td>INTER-SYSTEMS</td>
<td>Data is correctly passed from system to system</td>
<td>Intersystem parameters changed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intersystem documentation updated</td>
</tr>
<tr>
<td>CONTROL</td>
<td>Controls reduce system risk to an acceptable level</td>
<td>File reconciliation procedures work</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Manual controls in place</td>
</tr>
<tr>
<td>PARALLEL</td>
<td>Old system and new system are run and differences compared</td>
<td>Old and new system can reconciled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Operational status of old system maintained</td>
</tr>
</tbody>
</table>
3.2 Program Documentation

This is the procedure of including illustrations or comments to explain lines or segments within the program. This is necessary so as to understand the program especially when the program is long.

The two major reasons for documentation are:

- **Clarity:** It makes the program to be clear and understandable to the programmers. Even, the program writer will find it difficult understanding some part of the program if it is not properly documented.

- **Extensibility:** Documentation allows for easy amendment, extension or upgrade of the program. Documentation allows other programmers (apart from the writer) to be able to work on the program. We all know that a programmer might not be available every time.

However, it is important to note that program documentation must be efficient. This means that correct descriptions should be attached to the lines and segments within the program. This is necessary so as not to mislead other programmers that might want to work on the program in the future.

3.3 Program Maintenance

Program development does not really end after implementation; it is still important to still monitor the system so as to continually check whether the program is still working according to earlier specifications. It is also important to check whether the program still meets current needs of the user. Program maintenance is the act of ensuring the smooth and continuous working of the program in the nature of business and dynamics of operation. The following are the reasons why program maintenance is necessary:

1. Changes in nature of business
2. Dynamic of operation
3. Changes in technology
4. Improving the size and efficiency of code – refactoring

4.0 CONCLUSION

This unit has explained some of the reasons why you should maintain your program. It has further explained two major reasons for program
documentation. It introduced program testing as well as the reasons for carrying out our program.
5.0 SUMMARY

What this unit explains is the reason why you should maintain your programs and why program testing is a labour-intensive task. It also explained and gave instances of types of program testing and the techniques used in carrying them out.

6.0 TUTOR-MARKED ASSIGNMENT

1. Explain briefly what you understand by Program Testing.
2. Enumerate the components of Structural System Testing.
3. What do you understand by System Recovery?
4. What is the function of Functional System Testing?
5. Give two examples of Error Handling.
6. What are the two major reasons for documentation?
7. Why is program maintenance necessary?

7.0 REFERENCES/FURTHER READINGS


www.doc.ic.ac.uk/~wjk/C++Intro/

www.personal.kent.edu/~muhama/Algorithms
UNIT 6	BASICDATATYPES

CONTENTS

1.0	Introduction
2.0	Objectives
3.0	MainContent
  3.1	Data and Programming
  3.2	NumericData Types
  3.3	Non-numericDataTypes
4.0	Conclusion
5.0	Summary
6.0	Tutor-MarkedAssignment
7.0	References/FurtherReadings

1.0	INTRODUCTION

In this unit, you will be introduced to the various forms in which data can be represented (Data Structures). You will also be introduced to the different datatypes, like integers, real numbers, character data type and string datatype.

2.0	OBJECTIVES

By the time you must have completed this unit, you should be able to:

✓ explain the fact that data exists in a variety of forms, outlined the datatypes, which include numeric and non-numeric datatypes
✓ outline the constituents of an integer, real numbers, character datatype and string datatype.

3.0	MAINCONTENT

3.1 Data and Programming

Most programs are designed to manipulate data in order to get an output. Data exist in a variety of forms. Examples are 20,000,000, which might be a day’s sales, simplified, limited, name of an organization and soon.

Data serve as input to most programs. The formatter procedure for input specification within a program depends on the nature of data.
3.2 Numeric Data Types

They consist of wholenumeric values. Examples are:

1. **Integers:** Integers consist of positive and negative whole values.

   Examples, are 500, -112, 77 etc.

   Major standard integer data types are:

   - Bytes
   - Shortint
   - Integer
   - Word
   - Longint

   You can find out about all these in programming languages.

2. **Real Numbers:** These consist of values with fractional parts. Examples are 257.29, 20.10, 11.00, etc. Floating-point numbers normally have two parts: the mantissa (the fractional part) and an exponent (the power to which the base of the number is raised to in order to give the correct value of the number).

   For example: The floating-point representation of 49234.5 is mantissa

   \[
   \text{Mantissa} = 0.4923425 \\
   \text{Exponent} = 5 \\
   \text{So we have} \ 0.4923425 \times 10^5.
   \]

   The standard real data types are:

   - Real
   - Single
   - Double
   - Extended

   Also read more about these
3.3 Non-Numeric Data Types

These are values that are not numbers in nature. Examples are

1. CharacterDataType

This consists of representation of individual characters using the American Standard Code for Information Interchange (ASCII). ASCII uses 7-bit to represent each character.

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>65</td>
</tr>
<tr>
<td>B</td>
<td>66</td>
</tr>
<tr>
<td>C</td>
<td>67</td>
</tr>
</tbody>
</table>

2. String Data Type

A string consists of a sequence of characters enclosed in single or double quotation marks depending on the programming language.

For example

- “Abiola”
- “Iama man”
- “1999”

4.0 CONCLUSION

This unit has examined in detail the types of data in programming languages. Also, you have been able to know more about the numeric and non-numeric data types, the standard real data types, etc.

5.0 SUMMARY

There are basically two types of data types, which are numeric and non-numeric data types.

Numeric data types are either integers or real numbers.

Non-numeric data types are either character or string.

6.0 TUTOR-MARKED ASSIGNMENT

Write and explain any example of a numeric and non-numeric data type.
7.0 REFERENCES/FURTHERREADINGS


www.doc.ic.ac.uk/~wjk/C++Intro/

www.doc.ic.ac.uk/~wjk/C++Intro/

www.personal.kent.edu/~muhama/Algorithms
UNIT7  FUNDAMENTAL DATA STRUCTURES

CONTENTS

1.0  Introduction
2.0  Objectives
3.0  MainContent
    3.1  Introduction to Data Structure
    3.2  Linear Data Structures
    3.3  Graphs
    3.4  Trees
4.0  Conclusion
5.0  Summary
6.0  Tutor-Marked Assignment
7.0  References/Further Readings

1.0  INTRODUCTION

This unit is intended to show you the various ways of representing data in an algorithm and in programming as a whole.

2.0  OBJECTIVES

At the end of this unit you should be able to:

✓ explain data structures and give related examples
✓ outline the different types of data structures
✓ explain the different types of linear data structures and when they are used in the design of algorithms
✓ explain the operations of the different types of data structures and differentiate between trees and graphs.

3.0  MAIN CONTENT

3.1  Introduction to Data Structure

Data structure is a means of organising related data items. Data structures became necessary to learn the design of algorithms since most algorithms operate on data. Therefore, it is important to understand the ways of organising data in the design and analysis of algorithms. The data structure to be used is determined by the problem at hand. Data structure is a way to store and organize data in order to facilitate access and modifications. No single data structure works well for all purposes and so it is important to know the features of some of them
For instance, if you have a list of data, you will need an array in the design of the algorithm. There are two basic types of data structures; these are linear data structures and non-linear data structures. Examples
of data structures are arrays (one-dimensional or multidimensional), queues, stack, trees, linked list etc.

3.2 Linear Data Structures

Array: An array can be defined as sequences of objects all of which are of the same type that are collectively referred to by the same name. Each individual array element (that is each of the data items), can be referred to by specifying the array name, followed by an index (also called a subscript), enclosed in parenthesis. For instance, LIST(1) is the name of the array while 1 is the index pointing to the data item on LIST. There are two types of arrays: one-dimensional array (list or column) and multi-dimensional arrays (table, matrix etc).

Arrays are used to implement mathematical vectors and matrices, as well as other kinds of rectangular tables. Many databases, small and large, consist of (or include) one-dimensional arrays whose elements are records. Arrays are used to implement other data structures, such as heaps, hash tables, queues, stacks, strings etc.

There are three ways in which the elements of an array can be indexed:

- 0 (zero-based indexing): The first element of the array is indexed by subscript of 0.
- 1 (one-based indexing): The first element of the array is indexed by subscript of 1.
- n (n-based indexing): The base index of an array can be freely chosen. Usually programming languages allowing n-based indexing also allow negative index values and other scalar data types like enumerations, or characters may be used as an array index.

Linkedlist: A linked list is a sequence of zero or more elements called nodes, each containing two kinds of information: some data and one or more links called pointers to other nodes of the linked list. In every linked list, there is a special pointer called the null which is used to indicate the absence of a node successor. Also, it contains a special node called the header; this node contains the information about the linked list such as its current length.

Stack: A stack is a data structure in which insertion and deletion can only be done at one end (called the top). In a stack, there are two major processes called PUSH and POP. PUSH is the process of adding elements to the stack while POP is the process of deleting elements from the stack. This stack's scheme is referred to as Last-In-First-Out (LIFO) scheme. Atypical/physical illustration of a stack is a pile of plates in a container. Stacks are used in implementing recursive algorithms.
**Queue:** Unlike the stack, a queue is a data structure with two ends, in which an insertion is made at the end (REAR) and a deletion is done at the other end (FRONT). A queue operates a First-In-First-Out (FIFO) scheme. A typical and practical illustration of this data structure is a queue in a modern entry. Queues are used for several graph problems.

**Heap:** It is a partially ordered data structure that is used in implementing priority queues. A priority queue is a set of items with an orderable characteristic called an item's priority. A heap can also be defined as a binary tree with keys assigned to its nodes.
3.3 Non-Linear Data Structures

A graph consists of two things:

1. A set whose elements are called vertices, points or nodes and
2. A set of unordered pairs of distinct vertices, called edges.

A graph is denoted by $G(V, E)$ when we want to emphasize the two parts of the graph $G$.

Therefore, a graph can be pictorially (imaginarily) defined as a connection of points in a plane called vertices or edges, some of which are connected by line segments called edges or arcs. It is formally defined by a pair of two sets.

Graph $G = (V, E)$.

It is more convenient to label the vertices of a graph with letters, integer numbers or character strings.

The figure below represents the graph $G$ with four vertices $A, B, C$ & $D$ and five edges $e_1 = (A, B), e_2 = (B, C), e_3 = (C, D), e_4 = (A, C), e_5 = (B, D)$.

We usually denote a graph by drawing its diagram rather than explicitly listing its vertices and edges.

3.4 TREES

A graph is said to be acyclic or cycle-free, since it contains no cycle, while a tree is a connected acyclic graph. A forest is a graph with no cycle hence, each of its connected components is a tree.

There are some properties possessed by trees which graphs do not have; for instance, the number of edges in a tree is always one less than the number of its vertices.

$$\|E\| = |V| - 1.$$
The figures below are examples of trees with different numbers of vertices.

4.0 CONCLUSION

In the course of this unit you were introduced to the concept of data structure and the various data structures that are available.

5.0 SUMMARY

This unit has shown that:

Data structure is a means of organizing related data items.
A data structure could be either linear or non-linear.
The basic linear data structures available are array, linked list, stack, queue and heap.
The basic non-linear data structures are graph and trees.

6.0 TUTOR-MARKED ASSIGNMENT

1. Describe how one can implement each of the following operations on an array so that the time it takes does not depend on the array’s size n.
   a. Delete the ith element of an array (1 ≤ i ≤ n).
   b. Delete the ith element of a sorted array (the remaining array has to stay sorted, of course).
2. If you have to solve the searching problem for a list of n numbers, how can you take advantage of the fact that the list is known to be sorted? Give separate answers for
   a. Lists represented as arrays
   b. Lists represented as linked lists.

3. a. Show the stack after each operation of the following sequence that starts with the empty stack.
      Push(a), push(b), pop, push(c), push(d), pop
   b. Show the queue after each operation of the following sequence that starts with the empty queue: enqueue(a),
      enqueue(b), dequeue, enqueue(c), enqueue(d), dequeue

7.0 REFERENCES/FURTHER READING


www.doc.ic.ac.uk/~wjk/C++Intro/

www.personal.kent.edu/~muhama/Algorithms
UNIT 8  EXERCISE I

CONTENTS

1.0  Introduction
2.0  Objectives
3.0  MainContent
    3.1  Exercise
    3.2  Solution
4.0  Conclusion
5.0  Summary
6.0  Tutor-MarkedAssignment
7.0  References/FurtherReadings

1.0  INTRODUCTION

This unit is a recap of the module because it will expose you to the practical aspect of what the module has taught.

2.0  OBJECTIVES

At the end of this unit you should be able to develop a working algorithm and a corresponding flowchart for the algorithm.

3.0  MAINCONTENT

3.1  The Problem

Use pseudocodes and a flowchart to represent an algorithm to generate prime numbers between 1 and 200.

3.2  Solution

a. Pseudocodes

\[
\begin{align*}
I &= 1 \\
&\text{WRITE I} \\
&\text{FORI} = 2, I-1 \\
&\quad \text{FORJ} = 2, \text{I-1} \\
&\quad \quad \text{IF MOD (I,J) = 0} \\
&\quad \quad \quad \text{GOTO 20} \\
&\quad \quad \text{ENDIF} \\
&\quad \text{NEXT J} \\
&\text{WRITE "I"} \\
&20 \quad \text{NEXT I}
\end{align*}
\]}
Note

- The mod function that is used in the pseudocode is used in most programming languages to get the remainder when a number is divided by another number.
- The facilitator should explain the pseudo codes to the students.

b. Flowchart
Start

Write 1

I = 1

J = 2

Is \( \text{mod} (I, J) = 0 \)?

Yes

No

J = J + 1

Is J < I - 1?

No

Print I

I = I + 1

Is I \leq 200?

Yes

No

Stop
4.0 CONCLUSION

The unit is a practical approach to the module and it is necessary for you to have a solid experience in the development of an algorithm and a flow chart as a pre-requisite for the remaining part of the course.

5.0 SUMMARY

You should be able to develop an algorithm and a flow chart for the algorithm.

6.0 TUTOR-MARKED ASSIGNMENT

Use pseudocode and a flowchart to represent an algorithm to find the average of the first 100 numbers.

7.0 REFERENCES/FURTHER READINGS


www.personal.kent.edu/~muhama/Algorithms
MODULE 2 PERFORMANCE ANALYSIS OF ALGORITHMS

Unit 1    Performance Analysis Framework
Unit 2    Order of Growth
Unit 3    Worst-Case, Best-Case and Average-Case Efficiencies
Unit 4    P, NP and NP-Complete Problems
Unit 5    Practical Exercise II

UNIT 1 PERFORMANCE ANALYSIS FRAMEWORK

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
   3.1 Efficiency Attributes
   3.2 Measuring Input Size
   3.3 Units for Measuring Running Time
4.0 Conclusion
5.0 Summary
6.0 Tutor-Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

This unit will introduce you to the criteria used in analysing the performance of an algorithm. As you will see, this unit is an introduction to other units in this module.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- state the criteria for estimating the running time of an algorithm
- estimate the running time of an algorithm
- list the efficiency attributes of an algorithm
- describe how time efficiency of an algorithm is measured.
3.0 MAINCONTENT

3.1 EfficiencyAttributes

The efficiency attributes are used to analyse the performance of algorithms. There are two types of algorithm efficiency attributes.

**Time Efficiency:** This indicates how fast an algorithm runs.

**Space Efficiency:** It deals with the space required for an algorithm to run efficiently. In the early computing days, both resources—time and space—were limited.

3.2 Measuring Input Size

It is certain that all algorithms donot run at time on the same number of input(s). For example, it takes longer to sort larger arrays, multiply larger matrices, and so on. It is then necessary to investigate an algorithm’s efficiency, as a function of input size parameter n. Selecting such a parameter is not difficult in most problems. For example, it will be the size of the list for problems of sorting, searching, finding the list’s smallest element, and most other problems dealing with lists. For the problem of evaluating a polynomial $P(x) = a_n x^n + \ldots + a_0$ of degree $n$, it will be the polynomial’s degree or the number of coefficients, which is larger by one than its degree.

Of course, there are situations where the choice of a parameter indicating an input size is not really a factor. An example is computing the product of two $n$-by-$n$ matrices. There are two natural measures of size for this problem. The first is the most frequently used, the natural matrix order $n$. But there are other natural contenders, the total number of elements $N$ in the matrices being multiplied.

The choice of an appropriate size metric can be influenced by operations of the algorithm in question. For example, how should we measure an input’s size for a spell-checking algorithm? If the algorithm examines individual characters of its input, then we should measure the size by the number of characters; if it works by processing words, we should count their number in the input.

3.3 Units for Measuring Running Time

In measuring the running time of an algorithm, it is necessary to identify the basic operations within the algorithm. The basic operations are the most important operations of the algorithm. After identifying the basic
operation, which then computes the number of times the basic operation is executed.

As a rule, it is not difficult to identify the basic operation of an algorithm: it is usually the most time-consuming operation in the algorithm's innermost loop. For example, most sorting algorithms work by comparing elements (keys) of a list being sorted with each other; for such algorithms, the basic operation is a key comparison. As another example, algorithms for matrix multiplication and polynomial evaluation require two arithmetic operations: multiplication and addition. On most computers, multiplication of two numbers takes longer than addition, making the former an unquestionable choice for the basic operation.

Conclusively, an algorithm’s time efficiency can be measured by counting the number of times the algorithm’s basic operation is executed on input of size $n$. This will be fully treated in Unit Three of this module.

### 4.0 CONCLUSION

In this unit, you have been introduced to the fundamental concepts of analyzing the performance of an algorithm.

### 5.0 SUMMARY

Having gone through this unit, you are expected to have learnt the following:

- The efficiency attributes of an algorithm are time efficiency and space efficiency.
- The time efficiency is measured as a function of the input size $n$.
- The time efficiency is measured as a function of the number of times basic operations were executed on an input.
- The running time of an algorithm is estimated based on the basic operations.

### 6.0 TUTOR-MARKED ASSIGNMENT

1. What do you understand by the term “Running time of an algorithm”? How is it measured?
2. What are the units for measuring the running time of an algorithm?
3. Describe space efficiency with respect to the running of an algorithm.

7.0 REFERENCES/FURTHER READINGS


[www.eslearning.algorith.com](http://www.eslearning.algorith.com)


[www.personal,kent.edu/wmuhama/algorithms](http://www.personal,kent.edu/wmuhama/algorithms).
UNIT2 ORDER OF GROWTH

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
   3.1 Informal Notation
   3.2 O-Notation
   3.3 O-Notation
   3.4 - Notation
   3.5 - Notation
   3.6 - Notation
4.0 Conclusion
5.0 Summary
6.0 Tutor-Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

This unit introduces and explains order of growth and the different notation that are used to represent order of growth of algorithms.

2.0 OBJECTIVES

By the end of this unit you should be able to:

- explain order of growth

- explain the different asymptotic notations.

3.0 MAIN CONTENT

3.1 Informal Introduction

Let us consider the table below to informally describe the growth of algorithms based on some standard functions.
<table>
<thead>
<tr>
<th>N</th>
<th>log₂⁰</th>
<th>n</th>
<th>nlog₂⁰</th>
<th>n²</th>
<th>n³</th>
<th>2^n</th>
<th>n!</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3.3</td>
<td>10³</td>
<td>3.3.10³</td>
<td>10²</td>
<td>10³</td>
<td>10⁴</td>
<td>3.6.10⁶</td>
</tr>
<tr>
<td>10²</td>
<td>6.6</td>
<td>10²</td>
<td>6.6.10²</td>
<td>10⁴</td>
<td>10⁶</td>
<td>1.3.10³⁰</td>
<td>9.3.10¹⁵⁷</td>
</tr>
<tr>
<td>10³</td>
<td>10</td>
<td>10³</td>
<td>1.0.10⁴</td>
<td>10⁶</td>
<td>10⁹</td>
<td>10⁶</td>
<td></td>
</tr>
<tr>
<td>10⁴</td>
<td>13</td>
<td>10⁴</td>
<td>1.3.10⁵</td>
<td>10⁸</td>
<td>10¹²</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10⁵</td>
<td>17</td>
<td>10⁵</td>
<td>1.7.10⁶</td>
<td>10¹⁰</td>
<td>10¹⁵</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10⁶</td>
<td>20</td>
<td>10⁶</td>
<td>2.0.10⁷</td>
<td>10¹²</td>
<td>10¹⁸</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All the functions are based on input size. We can see that the function growing the slowest among these is the logarithmic function. It grows so slowly, in fact, that we should expect a program implementing an algorithm with a logarithmic basic-operation count to run practically instantaneously on input of all realistic sizes.

On the other end of the spectrum are the exponential function $2^n$ and the factorial function $n!$. Both these functions grow so fast that their values become astronomically large, even for rather small values of $n$.

Algorithms performance is mostly represented by these functions because these functions describe the performance of the most algorithmic on input size $n$.

### 3.1 $o$-Notation

This is pronounced as “little oh”. Let $(x)\text{and}(g(x))$ be two functions of $x$. Each of the five symbols above is intended to compare the rapidity of growth of $f(x)$ and $g(x)$. If we say that $(x)=o (g(x))$, then informally we are saying that $f(x)$ grows more slowly than $g(x)$ when $x$ is very large.

**Definition**

We say that $f(x) = o (g(x))$ ($f(x) \to \infty$) if $\lim_{x \to \infty} f(x)/g(x) \to 0$. If $\lim_{x \to \infty} f(x)/g(x) \to \infty$, then $f(x)$ grows faster than $g(x)$.

Here are some examples:

1. $X^2 = o(x^5)$
2. $\sin x = o(x)$
3. $14.709 \sqrt{x} = o(x^{2/7} + \cos x)$

### 3.2 $O$-Notation

This is pronounced as “big oh”. The second symbol of the asymptotic vocabulary is the ‘$O’$. When we say that $(x) = O(g(x))$, we mean, informally, that $f(x)$ grows at a rate no faster than $g(x)$. It might
grow at the same rate or it might grow more slowly; both are possibilities that the ‘O’ permits.

**Definition**

We say that \( f(x) = O(g(x)) \) \( (x \to \infty) \) if \( \exists C, x_0 \) such that \( |f(x)| < Cg(x) \) \( (\forall x > x_0) \).

The qualifier ‘\( x \to \infty \)’ will usually be omitted, since it will be understood that we will most often be interested in large values of the variable that are involved.

For example, it is certainly true that \( \sin x = O(x) \), but even more can be said, namely that \( \sin x = O(1) \). Also \( x^{\alpha} + 5x^2 + 77\cos x = O(x^5) \) and \( 1/\left(1 + x^2\right) = O(1) \). Now we can see how the ‘\( O \)’ gives more precise information than the ‘\( O \)’, for we can sharpen the last example by saying that \( 1/(1+x^2) = o(1) \). This is sharper because not only does it tell us that the function is bounded when \( x \) is large, we learn that the function actually approaches 0 as \( x \).

This is typical of the relationship between \( O \) and \( o \). It often happens that a ‘\( O \)’ result is insufficient for an application. However, that may not be the case, and we may need the more precise ‘\( o \)’ estimate.

3.3 **\( \Theta \) -Notation**

The third symbol of the language of asymptotic is the ‘\( \Theta \)’. This is pronounced as ‘is theta of’.

**Definition**

We say that \( f(x) = \Theta(g(x)) \) if there are constants \( c_1, c_2 > 0, x_0 \) such that for all \( x > x_0 \), it is true that \( c_1 g(x) < f(x) < c_2 g(x) \).

We might then say that \( f \) and \( g \) are of the same rate of growth, only the multiplicative constants are uncertain. Some examples of the ‘\( \Theta \)’ at work are:

\[
(X + 1)^2 = \Theta(3X^2)
\]

\[
(x^2 + 5x + 7)/(5x^3 + 7x + 2) = \Theta(1/x)
\]

\[
3 + 2x = \Theta(x^{1/4})
\]

\[
(1 + 3/x)^5 = \Theta(1).
\]

The ‘\( \Theta \)’ is much more precise than either the ‘\( O \)’ or the ‘\( o \)’. If we know that \( (x) = \Theta(x^2) \), then we know that \( f(x)/x^2 \) stays between two nonzero constants for all sufficiently large values of \( x \). The rate of growth of \( \Theta \) is established: it grows quadratically with \( x \).
3.4 ~-Notation

This is pronounced as “is asymptotically equal to”. The most precise of the symbols of asymptotic is the ~. It tells us that not only do we grow at the same rate, but that in fact /g approaches 1 as x→∞.

**Definition**

We say that \( f(x) ∼ g(x) \) if \( \lim_{x \to \infty} \frac{f(x)}{g(x)} = 1 \)

Here are some examples.

\[ X^2 + x \sim x^2 \]
\[ (3x + 1)^4 \sim 81x^4 \]
\[ \sin \frac{1}{x} \sim \frac{1}{x} \]
\[ \frac{(2x^3 + 5x + 7)}{(x^2 + 4)} \sim 2x \]
\[ 2^x + 7 \log x + \cos x \sim 2^x \]

Observe the importance of getting the multiplicative constant exactly right when the ~” is symbolised. While it is true that \( x^2 = \Theta(x^2) \), it is not true that \( 1x^2 = \theta(17x^2) \), but to make such an assertion is to use a bad style of asymptotic notation.

3.5 \( \Omega \)-Notation

This is pronounced as “is of order”. The last symbol in the asymptotic set that we will need is the \( \Omega \) “which is the negation of ‘o’”. That is to say, \( f(x) = \Omega(g(x)) \) means that it is not true that \( (x) = o(g(x)) \). In the study of algorithms for computers, the \( \Omega \) is used when we want to express the thought that a calculation takes at least so-and-so long to do. For instance, we can multiply together two \( n \times n \) matrices in time \( \Omega(n^3) \).

4.0 CONCLUSION

In this unit, you have learnt the five asymptotic symbols for representing order of growth of algorithms.

5.0 SUMMARY

This unit has explained the five functions for comparing the growth order of an algorithm. These functions are in ascending order are \( O \).

6.0 TUTOR-MARKED ASSIGNMENT
Question: Use the asymptotic symbols to compare the following functions. You might need to write small programs before you can get the order of growth effectively.

1. \( f(x) \) and \( g(x) \) where \( f(a) = 4a^2 + a + 7 \) and \( g(a) = a! \)
2. \( f(x) \) and \( g(x) \) where \( f(x) = 4^x \) and \( g(x) = x^4 \)
3. \( h(x) \) and \( i(x) \) where \( h(x) = \cos(x) \) and \( i(x) = \sin(x) \)
4. \( f(a) \) and \( g(b) \) where \( f(x) = (x^2)^2 \) and \( g(x) = x^4 \)

### 7.0 REFERENCES/FURTHER READINGS


[www.eslearning.algorith.com](http://www.eslearning.algorith.com)


[www.personal.kent.edu/wmuhama/algorithms](http://www.personal.kent.edu/wmuhama/algorithms)

UNIT3 WORST-CASE, BEST-CASE AND AVERAGE-CASE EFFICIENCIES

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 MainContent
   3.1 Worst-Case
   3.2 Best-Case
   3.3 Average-Case
4.0 Conclusion
5.0 Summary
6.0 Tutor-MarkedAssignment
7.0 References/FurtherReadings

1.0 INTRODUCTION

This unit extends what we learnt in Unit 1 of this module. It discusses the different techniques that can be used to measure the efficiencies of algorithms.

2.0 OBJECTIVES

At the end of this unit you should be able to:

- explain the three methods that are used to measure the efficiencies of algorithms
- explain how to identify basic operations within an algorithm.

3.0 MAINCONTENT

3.1 Worst-Case

An algorithm's worst-case efficiency for the worst-case input of size \( n \), which is an input (or inputs) of size \( n \) for which the algorithm runs the longest among all possible inputs of that size. The way to determine the worst-case efficiency of an algorithm, in principle, is quite straightforward:
We analyse the algorithm to see what kind of inputs yield the largest value of the basic operations’ count $C(n)$ among all possible inputs of size $n$ and then compute this worst-case value $C_{\text{worst}}(n)$. Clearly, the worst-case analysis provides very important information about an algorithm’s efficiency by bounding its running time from above. In other words, it guarantees that for any instance of size $n$, the running time will not exceed $C_{\text{worst}}(n)$, its running time on the worst-case inputs.

As it was mentioned in Unit 1 of this module, we need to count the number of basic operations performed by the algorithm on the worst-case input.

A basic operation could be:

- An assignment
- A comparison between two variables
- An arithmetic operation between two variables. The worst-case input is that input assignment for which the most basic operations are performed.

**Example 1**

```plaintext
n := 5;
loop
    get(m);
    n := n - 1;
until (m = 0 or n = 0)
```

Worst-case: 5 iterations

**Example 2**

```plaintext
get(n);
loop
    get(m);
    n := n - 1;
until (m = 0 or n = 0)
```

Worst-case: $n$ iterations

**Examples 3 of “input size”:**

a. Sorting:
n== Thenumber of itemstobesorted;
Basicoperation: Comparison.
b. **Multiplication** (of \(x\) and \(y\)):

\(n = \) The number of digits in \(x\) plus the number of digits in \(y\).

Basic operations: single-digit arithmetic.

c. **Graph “searching”**:

\(n = \) the number of nodes in the graph thenumber of edges in the graph.

Counting the number of basic operations

**Example 4**  Sequence: \(P \) and \(Q\) are two algorithm sections:

Time( \(P \); \(Q\)) = Time( \(P\)) + Time( \(Q\))

**Iteration**:
while <condition > loop
  \(P\);
end loop;
or
forii in 1..nloop
  \(P\);
end loop

Time = Time( \(P\)) *( Worst-case number of iterations)

**Conditional**
if <condition > then
  \(P\);
else
  \(Q\);
end if;

Time = Time( \(P\)) if <condition > = true
      Time( \(Q\)) if <condition > = false

**Example 5**

forii in 1..nloop
  forj in 1..nloop
    if i< j then
      swop (a(i,j), a(j,i)); -- Basic operation
    end if;
  end loop;
end loop;
Time \leq n^2

3.2 Best-Case

The best-case efficiency of an algorithm is an input (or inputs) of size $n$ for which the algorithm run the fastest among all possible inputs of that size. Accordingly, we can analyse the best-case efficiency as follows: First, we determine the kind of inputs for which the count $C(n)$ will be the smallest among all possible inputs of size $n$. (Note: the best case does not mean the smallest input; it means the input of size $n$ for which the algorithm runs the fastest). Then we should ascertain the value of $C(n)$ on these most convenient inputs. For example, for sequential search, best-case inputs will be list of size $n$ with their first elements equal to a search key; accordingly, $C_{bea}(n) = 1$.

The analysis of the best-case efficiency is not as important as that of the worst-case efficiency. But it is not completely useless either.

3.3 Average-Case

The average-case efficiency seeks to provide information on random input. It is calculated by dividing all instances of size $n$ into several classes so that for each instance of the class, the number of times the algorithm’s basic operation is executed is the same. This then means that the probability distribution of inputs need to be assumed or obtained so that the expected value of the basic operation’s count can be derived. Estimating average-case efficiency is not an easy task and it is difficult for this level. Students can just get familiar with known average case results.

4.0 CONCLUSION

This unit teaches the three methods of analysing the efficiency of algorithms.

5.0 SUMMARY

This unit has explained how to measure the efficiencies of algorithms.

6.0 TUTOR-MARKED ASSIGNMENT
Write the pseudocode for the algorithm to find the largest out of n integers and use worst-case to determine the efficiency.

7.0 REFERENCES/FURTHER READINGS


[www.personal.kent.edu/wmuhama/algorithms](http://www.personal.kent.edu/wmuhama/algorithms)

[www.eslearning.algorithim.com](http://www.eslearning.algorithim.com)
UNIT 4 P, NPANDNP-COMPLETE PROBLEMS

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 MainContent
   3.1 BasicDefinitions
   3.2 P and NP Problems
   3.3 NP-Complete Problems
4.0 Conclusion
5.0 Summary
6.0 Tutor-Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

This unit will discuss the different categories of computational problems.

2.0 OBJECTIVES

At the end of this unit you should be able to:

- distinguish between a polynomial and non-polynomial problem
- identify a non-polynomial–complete problem
- understand some basic issues in algorithm time efficiencies
- understand P, NP, NP–complete problems.

3.0 MAINCONTENT

3.1 Basic Definitions

We say that an algorithm solves a problem in polynomial time if its worst-case time efficiency belongs to $O(p(n))$ here $p(n)$ is a polynomial of the problem’s inputs’ size. (Notethat since we are using big-oh notation here, problems solvable in, say, logarithmic time are solvable in polynomial time as well). Problems that can be solved in polynomial time are called tractable; problems that cannot be solved in polynomial time are called intractable.

Computational complexity classifies problems according to their inherent difficulty.
3.2 Pandora Problem

Most problems discussed in this unit can be solved in polynomial time by some algorithms. They include computing the product and the greatest common divisor of two integers, sorting, searching (for a particular key in a list), for a given pattern in a text string, checking connectivity and acyclicity of a graph, finding a minimum spanning tree, and finding the shortest path in a weighted graph. (You are invited to add more examples to this list.)

Informally, we can think about problems that can be solved in polynomial time as the set that computer scientists call \( P \). A more formal definition of \( P \) is that it includes only decision problems, which are problems with yes/no answers.

Class \( P \) is a class of decision problems that can be solved in polynomial time by (deterministic) algorithms. This class of problems is called polynomial.

The restriction of \( P \) to decision problems can be justified by the following reasons. First, it is sensible to exclude problems that are not solvable in polynomial time because of their exponentially large output. Such problems arise naturally, e.g., generating subsets of a given set, all the permutations of a distinct items but is apparent from the output set that they cannot be solved in polynomial time. Second, many important problems that are not decision problems in their most natural formulation can be reduced to a series of decision problems that are easier to study. For example, instead of asking about the smallest number of colours needed to colour a graph’s vertices so that no two adjacent vertices are coloured the same colour, we can ask whether there exist such a colouring of the graph’s vertices with no more than \( m \) colours for some \( m \). (The problem of vertex colouring with \( m \) colours is called the \textbf{m-colouring problem}.) The first value of \( m \) in this series for which the decision problem of \( m \)-colouring has a solution solves the optimisation version of the graph colouring problem as well.

It is natural to wonder whether every decision problem can be solved in polynomial time. The answer to this question is: it is not known. In fact, some decision problems cannot be solved at all by any algorithm. Such problems are called \textbf{undecidable}. Alan Turing gave a famous example in 1936. The problem in question is called the \textbf{halting problem}: given a computer program and an input to it, determine whether the program will halt on that input or continue working indefinitely on it.

Are there decidable but intractable problems? Yes, there are, but the number of known examples is small, especially of those that arise...
naturally rather than being constructed for the sake of theoretical argument.

There are a large number of important problems, however, for which no polynomial-time algorithm has been found, nor has the impossibility of such an algorithm been proved. The classic monograph by M. Garey and D. Johnson (GJ79) contains a list of several hundreds of such problems from different areas of computer science, mathematics, and operations research. Here is just a small sample of some of the best-known problems that fall into this category:

**Hamiltonian circuit:** Determine whether a given graph has a Hamiltonian circuit (a path that starts and ends at the same vertex and passes through all the other vertices exactly once).

**Traveling salesman:** Find the shortest tour through n cities with known positive integer distances between them (find the shortest Hamiltonian circuit in a complete graph with positive integer weights).

**Knapsack problem:** Find the most valuable subset of n items of given positive integer weights and values that fit into a knapsack of a given positive integer capacity.

**Partition problem:** Given n positive integers, determine whether it is possible to partition them into two disjoint subsets with the same sum.

**Bin packing:** Given n items whose sizes are positive rational numbers not larger than 1, put them into the smallest number of bins of size 1.

**Graph colouring:** For a given graph, find its chromatic number (the smallest number of colours that need to be assigned to the graph’s vertices so that no adjacent vertices are assigned the same colour).

**Integer linear programming:** Find the maximum (or minimum) value of a linear function of several integer-valued variables subject to a finite set of constraints in the form of linear equalities and/or inequalities.

An nondeterministic algorithm is a two-stage procedure that takes as its input an instance I of a decision problem and does the following:

**Nondeterministic ("guessing") stage:** An arbitrary string S is generated that can be thought of as a candidate solution to the given instance I (but may be complete gibberish as well).

**Deterministic ("verification") stage:** A deterministic algorithm takes both I and S as its input and outputs yes if S represents a solution to
instance \(I\). (If \(S\) is not a solution to instance \(I\), the algorithm neither returns nor is allowed to halt at all.)

We say that a nondeterministic algorithm solves a decision problem if and only if for every yes instance of the problem it returns yes on some execution. (In other words, we require a nondeterministic algorithm to be capable of "guessing" a solution at least once and to be able to verify its validity. And, of course, we don't want it to ever output any other answer on an instance for which the answer should be no.) Finally, a nondeterministic algorithm is said to be \textbf{nondeterministic polynomial} if the time efficiency of its certification stage is polynomial.

Now we can define the class of \textbf{NP} problems.

Class \textbf{NP} is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called \textbf{nondeterministic polynomial}.

Most decision problems are in \textbf{NP}. First of all, this class includes all the problems in \(P\):

\[
P \subseteq \text{NP},
\]

This is true because, if a problem is in \(P\), we can use the deterministic polynomial time algorithm that solves it in the verification stage of a nondeterministic algorithm that simply ignores string \(S\) generated in its nondeterministic ("guessing") stage. But \textbf{NP} also contains the Hamiltonian circuit problem, the partition problem, as well as decision versions of the traveling salesman, the knapsack, graph colouring and many hundred problems of other difficult combinatorial optimisation problems catalogued in (GJ79). The halting problem, on the other hand, is in \textbf{NP} among the rare examples of decision problems that are known not to be in \textbf{NP}.

\subsection*{3.3 \textbf{NP}-Complete Problems}

Let us introduce another important notion in the computational complexity theory that of \textbf{NP} completeness.

A decision problem \(D\) is said to be \textbf{NP}-complete if

1. it belongs to class \(NP\);
2. every problem in \textbf{NP} is polynomially reducible to \(D\).

The fact that closely related decision problems are polynomially reducible to each other is not very surprising. For example, let us prove that the Hamiltonian circuit problem is polynomially reducible to the decision version of the travelling salesman problem. The latter can be
statedastheproblem todetermine whether there exists a Hamiltonian circuit in a given complete graph with positive integer weights whose lengths are not greater than a given positive integer \( m \). We can map a graph \( G \) of a given instance of the Hamiltonian circuit problem to an instance of the travelling salesman problem by assigning the weight to each edge in \( G \) and adding an edge of weight 2 between any pair of not adjacent vertices in \( G \). As the upper bound on the Hamiltonian circuit length, we take \( m = n \), where \( n \) is the number of vertices in \( G \) (and \( G' \)). Obviously, this transformation can be done in polynomial time.

Let \( G \) be a yes instance of the Hamiltonian circuit problem. Then \( G \) has a Hamiltonian circuit, and its image in \( G' \) will have length \( n \), making the image a yes instance of the decision travelling salesman problem. Conversely, if we have a Hamiltonian circuit of the length not larger than \( n \), then its length must be exactly \( l \) (why?) and hence the circuit must be made up of edges present in \( G \), making the inverse image of the yes instance of the decision travelling salesman problem be a yes instance of the Hamiltonian circuit problem. This completes the proof.

4.0 CONCLUSION

This unit shows how to distinguish between different computational problems. You have also been exposed to different decidable and undecidable problems.

5.0 SUMMARY

This unit teaches that

- Problems that can be solved in polynomial time are called tractable while those that cannot be solved in polynomial time are intractable. The class of decision problems that can be solved in polynomial time by (deterministic) algorithms are called polynomial problems and they are mostly with yes/no answer.
- Decision problems that cannot be solved at all by any algorithm are called undecidable problems.

6.0 TUTOR-MARKED ASSIGNMENT

A game of chess can be posed as the following decision problem: given a legal position of chess pieces and information about which side is to move, determine whether that side can win. Is this decision problem decidable?
7.0 REFERENCES/FURTHER READINGS


www.personal.kent.edu/wmuhma/algorithms

www.eslearning.algorithim.com
UNIT 5  PRACTICALEXERCISE II

CONTENTS

1.0  Introduction
2.0  Objectives
3.0  MainContent
   3.1  Exercise
   3.2  Solutions
4.0  Conclusion
5.0  Summary
6.0  Tutor-MarkedAssignment
7.0  References/FurtherReadings

1.0  INTRODUCTION

Thisunitpresentsanotherpracticalexerciseinorderforyoutofurtherunderstandhowtoanalyse efficiencies of algorithms.

2.0  OBJECTIVES

Attheendofthisunit,youshouldhavehadfurtherunderstandingintohowtodetermineefficiencyof algorithms.

3.0  MAINCONTENT

3.1  Exercise

Write the pseudocodes for the algorithm to find the average of the smallest and the largest of n integers.

3.2  Solution

MIN=A( I )
For I =1ton
   If MIN> A( I )
      TEMP = MIN
      MIN=A( I ) A( I )= TEMP
   endIf
endo
MAX= A( I )
For J = 2ton
   If MAX< A( I )
      TEMPl=MAX
      MAX= A( J )
     4n
A(J)= TEMP
endIf
endo
AVE = ( MIN+ MAX)/2
PRINT AVE
= 1*4n*4n*1

The basic operations are denoted by *. We can simply say that it is of O(n^2).

4.0 CONCLUSION

A clearer view of an estimation of the non-time growth of an algorithm has been presented.

5.0 SUMMARY

Basic operations have been clearly identified. Estimation of efficiency of algorithms is practically discussed.

6.0 TUTOR-MARKED ASSIGNMENT

Write the pseudocode of the algorithm to convert any binary number to decimal and estimate the worse-case runtime efficiency.

7.0 REFERENCES/FURTHER READINGS


www.personal.kent.edu/wmuhama/algorithms.

www.eslearning.algorithm.com
MODULE 3  SORTING AND SOME SPECIAL PROBLEMS

Unit 1  Introduction to Sorting and Divide-and-Conquer Algorithms
Unit 2  MergeSort
Unit 3  Quick Sort
Unit 4  BinarySearch
Unit 5  Selection Sort
Unit 6  Bubble Sort
Unit 7  Special Problems and Algorithms
Unit 8  Practical Exercise IV

UNIT 1  INTRODUCTION TO SORTING AND DIVIDE-AND-CONQUER ALGORITHMS

CONTENTS

1.0  Introduction
2.0  Objectives
3.0  Main Content
   3.1  Introduction to Sorting
   3.2  Fundamentals of Divide-and-Conquer Algorithms
   3.3  Practical Proof of Divided-and-Conquer Algorithms
4.0  Conclusion
5.0  Summary
6.0  Tutor-Marked Assignment
7.0  References/Further Readings

1.0  INTRODUCTION

This unit discusses the meaning of sorting. It also describes the concept of divide and conquer algorithms as problem-solving techniques.

2.0  OBJECTIVES

You are expected to be able to explain the following at the end of this unit:

meaning and significance of sorting
meaning and practical understanding of divide and conquer algorithms.
3.0 MAINCONTENT

3.1 Introduction to Sorting

Sorting is the process of arranging a set of items or objects in increasing or decreasing order.

The Significance of Sorting

For orderly analysis and presentation of items
For locating an item or items within a set
For finding duplicate values or nearest pair within a set
For finding the intersection or union of two or more sets
Sorting is also used as a part of many geometrical algorithms (e.g., convex hull, nearest pair of points in the plane).

3.1 Fundamental also of Divide-and-Conquer Algorithms

This is a method of designing algorithms that (informally) proceeds as follows:

Given an instance of the problem to be solved,

split this into several, smaller, sub-instances (of the same problem)
independently solve each of the sub-instances
and then combine the sub-instances solutions so as to yield a solution for the original instance.

The diagram below represents the divide and conquer technique:

```
            Problem of size n
                  \           /
                   \         /         
                    \     /           
                     \ /             
                      v               
                     solution to subproblem1
                      \.T(          )
                       \    \    \    
                        \   \   \   
                         \  \  \  
                          \ \ \ 
                           \v
                            solution to subproblem2

                           \             /           \            /
                            \\          /\          / \          / \\
                             v\        v \        v \        v  \  
                             solution to the original problem
```
Examples of algorithms that adopt the strategy of divide-and-conquer algorithms are mergesort, quicksort and binary search.

### 3.3 Practical Proof of Divide-and-Conquer Algorithms

Consider an algorithm, $\alpha$, say, which is known to solve all problem instances of size $n$ in at most $cn^2$ steps (where $c$ is some constant). We then discover an algorithm, $\beta$, say, which solves the same problem by:

1. Dividing an instance into $3$ sub-instances of size $n/2$.
2. Solving these $3$ sub-instances.
3. Combining the three sub-solutions taking $dn$ steps to do this.

Suppose our original algorithm, $\alpha$, is used to carry out step 2.

Let

$$T(\alpha)(n) = \text{Running time of } \alpha$$
$$T(\beta)(n) = \text{Running time of } \beta$$

Then,

$$T(\alpha)(n) = cn^2 \text{ (by definition of } \alpha)$$

But

$$T(\beta)(n) = 3T(\alpha)(n/2) + dn$$

$$= (3/4)(cn^2) + dn$$

So if

$$dn < (cn^2)/4 \text{ (i.e. } d < cn/4)$$

then

$\beta$ is faster than $\alpha$

In particular for all large enough $n$, $(n > 4d/c = \text{Constant})$, $\beta$ is faster than $\alpha$.

This realisation of $\beta$ improves upon $\alpha$ by just a constant factor. But if the problem size, $n$, is large enough then

$$n > 4d/c$$
$$n/2 > 4d/c$$
$$\ldots$$

$$n/2^i > 4d/c$$
which suggests that using $\beta$ instead of $\alpha$ for the “solves these” stage repeatedly until the sub-sub-sub...sub-instances are of size $n \leq (4d/c)$ will yield a still faster algorithm.

So consider the following new algorithm for instances of size $n$:

```
procedure gamma(n:problem size) is
    if $n \leq n^3$ then
        Solve problem using Algorithm $\alpha$;
    else
        Split into 3 sub-instances of size $n/2$; Use $\gamma$ to solve each sub-instance;
        Combine the 3 sub-solutions;
    end if;
end gamma;
```

Let $T(\gamma)(n)$ denote the running time of this algorithm.

If $n < n^0$

$$cn^2$$

Else

$$3T(\gamma)(n/2) + dn$$ otherwise

4.0 CONCLUSION

You should have understood sorting and what the divide-and-conquer techniques are all about.

5.0 SUMMARY

Sorting is the arrangement of items in a predetermined order. Divide-and-Conquer algorithms require dividing problems into sub-instances, solving these sub-instances and combining the solutions to form the original solution.

6.0 TUTOR-MARKED ASSIGNMENT

1. Write a pseudocode for a divide-and-conquer algorithm for finding a position of the largest element in an array of $n$ numbers.
2. Find out about the brute-force algorithm and compare it with the divide-and-conquer algorithm.
7.0 REFERENCES/FURTHERREADINGS


www.personal.kent.edu/wmuhamma/algorithm

www.eslearning.algorithm.com
UNIT2 MERGESORT

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 MainContent
   3.1 Understanding MergeSort
   3.2 TheMergeSortAlgorithm
   3.3 TheEfficiency of theMergeSortAlgorithm
4.0 Conclusion
5.0 Summary
6.0 Tutor-MarkedAssignment
7.0 References/FurtherReadings

1.0 INTRODUCTION

This unit describes mergesort sorting technique. It explains merge sort as an example of divide-and-conquer algorithm.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

   - give the practical meaning of merge sort
   - present the merge sort algorithm
   - state the performance of the merge sort algorithm.

3.0 MAINCONTENT

3.1 Understanding Mergesort

Mergesort is an example of an application that adopts the strategy of the divide-and-conquer technique. It sorts a given array \( E[0..n - 1] \) by dividing it into two halves \( E[0..[n/2] - 1] \) and \( A[[n/2].n - 1] \), sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one. This figure represents the mergesort technique.
3.2 The MergeSort Algorithm

ALGORITHM MergeSort(E[0..n−1])
    // Sorts array E[0..n−1] by recursive merge sort
    // Input: An array E[0..n−1] of orderable elements
    // Output: Array E[0..n−1] sorted in non-decreasing order if
    // n > 1
    copy E[0..[n/2]−1] to B[0..[n/2]−1]
    copy E[[n/2].n−1] to C[0..[n/2]−1]
    Mergesort (B[0..[n/2]−1])
    Mergesort (C[0..[n/2]−1])
    Merge (B, C, E)

The merging of two sorted arrays can be done as follows: Two pointers (array indices) are initialised to point to the first elements of the arrays being merged. Then the elements pointed to are compared and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is continued until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.
ALGORITHM Merge(B[0..p−1], C[0..q−1], E[0..p+q−1])
// Merge two sorted arrays into one sorted array
// Input: Arrays B[0..p−1] and C[0..q−1] both sorted
// Output: Sorted array A[0..p+1−1] of the elements of B and C
i ← 0; j ← 0; k ← 0
while i < p and j < q do if
   B[i] ≤ C[j]
      E[k] ← B[i]; i ← i + 1
   else A[k] ← C[j]; j ← j + 1
   k ← k + 1
   if i = p
      copy C[j..q−1] to E[k..p+q−1]
   else copy B[i..p−1] to E[k..p+q−1]

3.3 The Efficiency of the MergeSort Algorithm

The efficiency of mergesort is
C_worst(n) = n log n

The details of how this was arrived at will be studied in a future study in Algorithm Design.

4.0 Conclusion

The meaning and algorithm of mergesort has been presented. The worse-case efficiency was also discussed.

5.0 Summary

Mergesort adopts the strategy of divide-and-conquer. It requires dividing the array into two halves and sorting them recursively, then merging the two smaller sorted arrays into a single one. Merge sort algorithm is divided into two: the merge sort (for splitting and sorting halves) and the merger (for merging two halves together).

6.0 Tutor-Marked Assignment

1. Apply mergesort to sort the list E, X, A, M, P, L in alphabetical order.
2. How stable is the divide-and-conquer algorithm?
7.0 REFERENCES/FURTHER READINGS


www.eslearning.algorithm.com


www.personal.kent.edu/wmuhamalgorithms.


UNIT 3 QUICKSORT

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
   3.1 Introduction to Quick Sort
   3.2 The Quick Sort Algorithm
   3.3 The Efficiency of the Quick Sort Algorithm
4.0 Conclusion
5.0 Summary
6.0 Tutor-Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

In this unit, you will be introduced to a new sorting technique called the Quicksort, which could be used in rearranging elements of any given array.

2.0 OBJECTIVES

By the end of this unit, you should be able to:

- Evaluate the procedures involved in a quicksort algorithm
- Transverse an array using the quicksort algorithm.

3.0 MAIN CONTENT

3.1 Introduction to Quick Sort

Quicksort is another important sorting algorithm that is based on the divide-and-conquer approach. Unlike merge sort, which divides its inputs' elements according to their position in the array, quicksort divides them according to their value. Specifically, it rearranges elements of a given array $A[0..n-1]$ to achieve its partition situation where all the elements before some position are smaller than or equal to $A[s]$ and all the elements after some position are greater than or equal to $A[s]$:

$$A[0]...a[s-1] \quad a[s] \quad a[s+1]...a[n-1]$$

All are $A[s]$ all are $A[s]$.

Obviously, after a partition has been achieved, $A[s]$ will be its final position in the sorted array, and we can continue sorting the two sub...
arrays of the elements preceding and following A[s]s independently (e.g. by the same method).

3.2 The Algorithm

ALGORITHM quicksort(A[1..r])
   // Sorts a subarray by quicksort
   // Input: A subarray A[1..r] of A[0..n−1], defined by its left and right indices
   // 1andr
   // Output: The subarray A[1..r] sorted in nondecreasing order if
   // i < r
   s ← Partition([Al..r]) // s is a split position
   Quicksort(A[1..s−1])
   Quicksort(A[s+ 1..r])

A partition of A[0..n−1] and more generally, of the subarray A[1..r] (0 ≤ r ≤ n−1) can be achieved by the following algorithm. First, we select an element with respect to whose value we are going to divide the subarray. Because of its guiding role, we call this element the pivot. There are several different strategies for selecting a pivot; we will return to this issue when we analyse the algorithm’s efficiency. For now, we use the simplest strategy of selecting the subarray’s first element: p = A[1].

Three situations may arise, depending on whether or not the scanning indices have crossed. If the scanning indices I and j havenot crossed, i.e., I < j, we simply exchange A[i] and A[j] and resume the scans by incrementing I and decrementing j respectively:

\[
\begin{align*}
\text{P} & \quad \text{all are p} \quad p \quad \ldots \quad p \quad \text{all are} \quad p \\
\text{I} & \quad \text{all are p} \quad p \quad \ldots \quad p \quad \text{all are} \quad p \\
\text{J} & \quad \text{all are p} \quad p \quad \ldots \quad p \quad \text{all are} \quad p
\end{align*}
\]

If the scanning indices have crossed over, i.e., I > j, we have partitioned the array after exchanging the pivot with A[j]:

\[
\begin{align*}
\text{P} & \quad \text{all are p} \quad p \quad \ldots \quad p \quad \text{all are} \quad p \\
\text{I} & \quad \text{all are p} \quad p \quad \ldots \quad p \quad \text{all are} \quad p \\
\text{J} & \quad \text{all are p} \quad p \quad \ldots \quad p \quad \text{all are} \quad p
\end{align*}
\]

Finally, if the scanning indices stop while pointing to the same element, i.e., I = j, the value they are pointing to must be equal to p (why?). Thus, we have partitioned the array:

\[
\begin{align*}
\text{P} & \quad \text{all are p} \quad = p \quad \text{all are} \quad p \\
\text{I} & \quad \text{all are p} \quad = p \quad \text{all are} \quad p \\
\text{J} & \quad \text{all are p} \quad = p \quad \text{all are} \quad p
\end{align*}
\]

We can combined the last case with the case of crossed-over indices(I > j) by exchanging the pivot with A[j] whenever I < j.
Here is a pseudocode implementing this partitioning procedure.

ALGORITHM Partition (A[l..r])
   // Partition a subarray by using its first element as a pivot
   // Input: A subarray A[l..r] of A[0..n–1], defined by its left
   // and right
   // indices l and r
   // Output: A partition of A[.], with the split position returned as
   // this function’s value
   p ← a[l]
   i ← l; j ← r + 1
   repeat
      repeat I ← I + 1 until A[I] ≥ p
      repeat j ← j – 1 until A[j] ≤ p
      swap(A[I], A[j])
      until I < j
      swap(A[I], A[j]) // uno last swap when I = j
      swap(A[I], A[j])
   return

3.3 The Efficiency of Quick Sort Algorithm

The efficiency of Quick sort is
C worst (n) = (n^2)
C best (n) = (n log n)
C avg (n) = 0

The details of how this was arrived at will be studied in the study in Algorithm Design.

4.0 CONCLUSION

This unit has presented another type of sorting technique called the quick sort technique which divides its important elements according to their position or value to the array.

5.0 SUMMARY

Quick sort is another important sorting algorithm that is based on the divide-and-conquer approach. It divides the input elements according to their respective values.
6.0 TUTOR-MARKED ASSIGNMENT

a. Design an algorithm to rearrange the elements of a given array of real numbers so that all its negative elements precede all its positive elements. Your algorithm should be both time and space-efficient.

b. Apply quicksort to sort the E, X, A, M, P, L, E

7.0 REFERENCES/FURTHER READINGS


UNIT 4  BINARY SEARCH

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
   3.1 Understanding Binary Search
   3.2 The Algorithm
   3.3 The Efficiency of Binary Search
4.0 Conclusion
5.0 Summary
6.0 Tutor-Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

This unit introduces the binary search algorithm as well as its efficiency and the procedures involved in traversing the elements of any given array. The binary search is an effective way of traversing arrays.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

   explain how the binary search operates
   explain how the binary search is used to traverse an array of elements
   explain the algorithm of the binary search
   explain the efficiency of the binary search.

3.0 MAIN CONTENT

3.1 Understanding the Binary Search

The Binary Search

The binary search is a remarkably efficient algorithm for searching in a sorted array. It works by comparing a search key K with the array’s middle element A[m]. If the elements match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if K < A[m] and for the second half if K > [m]:

...
K

Search here if 
K < A[m]

search here if 
K > A[m]

As an example, let us apply the binary search to searching for K = 70 in the array

314 27 3139 4255 7074 81 8593 98

The iterations of the algorithm are given in the following table:

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td></td>
<td>14</td>
<td>27</td>
<td>31</td>
<td>39</td>
<td>42</td>
<td>55</td>
<td>70</td>
<td>74</td>
<td>81</td>
<td>85</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interation</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>Interation</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>Interation</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r</td>
<td></td>
</tr>
</tbody>
</table>

3.2 The Algorithm

Although the binary search is clearly based on a recursive idea, it can be easily implemented as a non-recursive algorithm, too. Here is a pseudo code for this non-recursive version:

ALGORITHM BinarySearch(A[0..n–1],K)
// Implements non-recursive binary search
// Input: An array A[0..n–1] sorted in ascending order and
// a search key K
// Output: An index of the array’s element that is equal to K
// or –1 if there is no such element in A

← 0; r ← n–1
while l < r do
    m ← [(l + r)/2]
    if K = A[m] return m
    elseif K < A[m] r ← m–1
    elseif A[m] r ← m+ 1
    return –1

3.3 The Efficiencies of the Binary Search

The efficiencies of the binary search are
\[ C_{\text{worst}}(n) \sim \log n \]
\[ C_{\text{best}}(n) = 1 \]
\[ C_{\text{avg}}(n) = (\log_2 n) \]

The details of how this was arrived at will be studied in Algorithm Design.
4.0 CONCLUSION

This unit has presented another effective way of transversing arrays through the binary search algorithm which works by comparing a search key with the arrays’ middle element A[m].

5.0 SUMMARY

This unit has discussed the binary search. The binary search is based on recursive data while the implementation is based on non-recursive algorithm.

6.0 TUTOR-MARKED ASSIGNMENT

Write a program in any programming language to implement the binary search on any set of integers.

7.0 REFERENCES/FURTHER READINGS


UNIT5  SELECTIONSORT

CONTENTS

1.0  Introduction
2.0  Objectives
3.0  MainContent
   3.1  UnderstandingSelection
   3.2  TheAlgorithm
   3.3  Efficiencies of Selection Sort
4.0  Conclusion
5.0  Summary
7.0  Tutor-MarkedAssignment
7.0  References/FurtherReadings

1.0  INTRODUCTION

This unit introduces you to the selection sort algorithm used in scanning the all the given elements of an array to find its smallest element and substitute it within the first element.

2.0  OBJECTIVES

At the end of this unit, you should be able to:

- describe how selection sort operates
- explain how selection sort is used to transverse elements in an array describe an algorithm of the selection sort.

3.0  MAINCONTENT

3.1  Selection Sort

We start selection sort by scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list. Then we can start the list, starting with the second element, to find the smallest among the last n–1 elements and exchange it with the second element, putting the second smallest element in its final position. Generally, on the ith pass through the list, which we number from 0 to n–2, the algorithm searches for the smallest item among the last n–i elements and swaps it with Ai:

\[ A_0, A_1, \ldots, A_{i-1}, A_{\text{min}}, \ldots, A_{n-1} \]
In their final positions the last n-1 elements.

After n–1 passes, the list is sorted.

3.2 The Algorithm

Here is a pseudocode of this algorithm, which, for simplicity, assumes that the list is implemented as an array.

```
ALGORITHM Selection Sort(A[0..n–1])
   // The algorithm sorts a given array by selection sort
   // Input: An array A[0..n–1] of orderable elements
   // Output: Array A[0..n–1] sorted in ascending order
   i ← 0 to n–2 do
   min ← i
   for j ← i+1 to n–1 do
   swap A[i] and A[min]
```

As an example, the action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated in figure.

```
89  45  68  90  29  34  17
17  45  68  90  29  34  89
17  29  68  90  45  34  89
17  29  34  90  45  68  89
17  29  34  45  90  68  89
17  29  34  45  68  90  89
17  29  34  45  68  89  90
```

Figure: Selection sort’s operation on the list 89, 45, 68, 90, 29, 34, 17. Each line corresponds to one iteration of the algorithm, i.e., a pass through the list. The list elements to the left of the vertical bar are in order, and the elements in bold indicate that a smaller element was found. Elements to the right of the vertical bar are in their final positions and are not considered in this sort's subsequent iterations.

3.3 The Efficiencies

Thus, selection sort is \( (n^2) \) algorithm on all inputs. Note, however, that the number of key swaps is only \( (n) \) or, more precisely, \( n–1 \) (one for each repetition of the loop). This property distinguishes selection sort positively from many other sorting algorithms.
4.0 CONCLUSION

By now, you should be able to sort an array of elements in a desired order, using the selection sort algorithm.

5.0 SUMMARY

This unit further explains selection sort, its algorithm, and its efficiencies. The number of key swaps is only \( n \) or, more precisely, \( n-1 \), which distinguishes it from other sorting algorithms.

6.0 TUTOR-MARKED ASSIGNMENT

a. Illustrate selection sort using any set of sample data
b. Write a program in any programming language to implement a selection sort on any set of integers.

7.0 REFERENCES/FURTHER READINGS


UNIT6   BUBBLESORT

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 MainContent
   3.1 Understanding Bubble Sort
   3.2 TheAlgorithm
   3.3 Efficiencies of BubbleSort
4.0 Conclusion
5.0 Summary
6.0 Tutor-MarkedAssignment
7.0 References/FurtherReadings

1.0 INTRODUCTION

The bubble sort is also another effective way in the variety of sort techniques available in programming. It will be introduced in this unit as well as its algorithm.

2.0 OBJECTIVES

At the end of this unit you should be able to:

- explain how bubblesort operates
- explain how bubblesort is used to transverse array of homogenous elements
- describe the algorithm of bubblesort
- explain the efficiency of the bubblesort algorithm.

3.0 MAINCONTENT

3.1 Understanding Bubble Sort

Another brute-force application to the sorting problem is to compare adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, we end up “bubbling up” the largest element to the last position on the list. Then, next pass bubbles up the second largest element, and so on, until, after n–1 passes, the list is sorted. Pass I (0 ≤ I ≤ n–2) of bubblesort can be represented by the following diagram: A0,

... Aj ↔ Aj+1, ... , A_{n-i-1}A_{n-i} ... A_{n-1}

...
The action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated as an example.

Bubble Sort
89 \leftrightarrow 45 \ 68 \ 90 \ 29 \ 34 \ 17
45 \ 89 \leftrightarrow 68 \ 90 \ 29 \ 34 \ 17
45 \ 68 \ 89 \leftrightarrow 90 \leftrightarrow 29 \ 34 \ 17
45 \ 68 \ 89 \ 29 \ 90 \leftrightarrow 84 \ 17
45 \ 68 \ 89 \ 29 \ 34 \ 90 \leftrightarrow 17
45 \ 68 \ 89 \ 29 \ 34 \ 17 \ |90

45 \leftrightarrow 68 \leftrightarrow 89 \leftrightarrow 29 \ 34 \ 17 \ |90
45 \ 68 \ 29 \ 89 \leftrightarrow 84 \ 17 \ |90
45 \ 68 \ 29 \ 34 \ 89 \leftrightarrow 17 \ |90
45 \ 68 \ 29 \ 34 \ 17 \ 89 \ 90 \ etc

The first two passes of bubblesort on the list 89, 45, 68, 90, 29, 34, 17. Notice that a new line is shown after a swap of two elements is done. The element to the right of the vertical bar are in their final positions and are not considered in subsequent iterations of the algorithm.

3.2 The Algorithm

Here is a pseudocode of this algorithm.

ALGORITHM bubblesort(A[0..n−1])

    // The algorithm sorts array A[0..n−1] by bubblesort
    // Input: An array A[0..n−1] of orderable elements
    // Output: Array A[0..n−1] sorted in ascending order for
    //         I ← 0 to n−2 do
    //           for j ← 0 to n−2−I do

3.3 The Efficiency

It is also in \( \Theta(n^2) \) in the worst and average cases. In fact, even among elementary sorting methods, bubble sort is an inferior choice, and if it were not for its catchy name, you would probably never hear of it.

4.0 CONCLUSION

By now you should be able to use the bubblesort, having discovered that it is a fast and easy way to transverse an array of any given set of elements.
5.0 SUMMARY

In the unit you have just concluded, the following were examined:

- The bubble sort algorithm.
- Its way of sorting elements of any given array.
- Bubblesort effectiveness.

6.0 TUTOR-MARKED ASSIGNMENT

a. Write a program in any programming language to implement the bubble sort algorithm on any set of integers.
b. Determine the better one between Quick sort and Bubble sort.

7.0 REFERENCES/FURTHER READINGS


UNIT 7  SPECIAL PROBLEMS AND ALGORITHMS

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
  3.1 Hill Climbing Technique
  3.2 Knight-tour Problem
  3.3 N-Queen Problems
  3.4 Game Trees
  3.5 Subset Sum
  3.5 Text Segmentation - Longest Increasing Subsequence
4.0 Conclusion
5.0 Summary
6.0 Tutor-Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

This unit shows some special problems and algorithms. Hill climbing can be used to solve problems that have many solutions, but where some solutions are better than others.

2.0 OBJECTIVES

At the end of this unit you should be able to:

- operate the hill climbing technique
- show how hill climbing is used to solve problems
- resolve the Knight's tour problem
- describe and resolve ann*ntour problem
- knight-tour Problem
- understand N-Queen Problems
- describe Game Trees
- understand Subset Sum

3.0 MAIN CONTENT

3.1 Hill Climbing Technique

Hill climbing is an optimisation technique which belongs to the family of local search. It is relatively simple to implement, making it a popular
first choice. Although more advanced algorithms may give better results, there are situations where hill climbing works well.

Hill climbing can be used to solve problems that have many solutions but where some solutions are better than others. The algorithm is started with a (bad) solution to the problem, and sequentially makes small changes to the solution, each time improving it a little bit. At some point the algorithm arrives at a point where it cannot see any improvement anymore, at which point the algorithm terminates. Ideally, at that point a
solution is found that is closer to the optimal, but it is not guaranteed that the hill climbing will ever come close to the optimal solution.

An example of a problem that can be solved with hill climbing is the Travelling Salesman Problem. It is easy to find a solution that will visit all the cities, but the solution will probably be very bad compared to the optimal solution. The algorithm starts with such a solution and makes small improvements to it, such as switching the order in which two cities are visited. Eventually, a much better route is obtained.

Hill climbing is used widely in artificial intelligence fields, for reaching a goal state from a starting node. The choice of next node and starting node can be varied according to the related algorithms.

Hill climbing terminates when there are no successors of the current state which are better than the current state itself. This is often a problem. For example, consider the following route selection:

These problems are essentially the result of local maxima in the search space. Points which are better than any surrounding state, but which are not the solution. There are some ways in which we can get around this (to some extent) by tweaking or extending the algorithm. We could use a limited amount of backtracking, so that we record alternative reasonable looking paths which weren't taken and go back to them. Or we could weaken the restriction that the next state has to be better by looking ahead a bit in the search; maybe the next but one states should be better than the current one. None of these solutions is perfect, and in general, hill climbing is only good for a limited class of problems where we have an evaluation function that fairly accurately predicts the actual distance to a solution.

This can be described as follows:

1. Start with \textit{current-state} = initial-state.
2. Until \textit{current-state} = goal-state OR there is no change in \textit{current-state} do:
   a. Get the successors of the current state and use the evaluation function to assign a score to each successor.
   b. If one of the successors has a better score than the current state, then set the new current state to the successor with the best score.

If one of the successors has a better score than the current state, then set the new current state to the successor with the best score.
Note that the algorithm does not attempt to exhaustively try every node and path, so non-deterministic behavior is maintained—just the current node. If there are loops or if the search space then using hill climbing, you shouldn't encounter them; you can't keep going up and still get back to where you were before.

### 3.2 The Knight's Tour Problem

A knight's tour of a chessboard (or any other grid) is a sequence of moves by a knight chess piece (which may only make moves which simultaneously shift one square along one axis and two along the other) such that each square of the board is visited exactly once. It is therefore a Hamiltonian path on the graphs consisting of vertices corresponding to the chessboard squares and edges corresponding to legal knight moves. If the final position of such a tour is a knight's move away from the initial position of the knight, the tour is called a re-entrant or closed, and is therefore a Hamiltonian circuit. The figures below show six knight's tours on an $8 \times 8$ chessboard, all but the first of which are re-entrant. The final tour has the additional property that it is a semi-magic square with row and column sums of 260 and main diagonal sums of 348 and 168.

![Six Knight's Tours on an $8 \times 8$ Chessboard](image)

The $m \times n$ knight's tour graph is a graph with $m \times n$ vertices in which each vertex represents a square in an $m \times n$ chessboard, and each edge corresponds to a legal move by a knight. The $m \times n$ knight's tour graph implemented as `KnightTourGraph[m, n]` in the Mathematica package.

The number of edges in the $n \times n$ knight's tour graph is $4(n - 2)(n - 1)$ (8 triangular numbers), so for $n = 1, 2, \ldots$, the first few values are 0, 0, 8, 24, 48, 80, 120, ...

The numbers of (undirected) closed knight's tours on a $(2, n) \times (2, n)$ chessboard for $n = 1, 2, \ldots$, are 0, 0, 9862, 13267364410532, ... There are no closed tours for $m \times m$ boards with $m$ odd. The number of cycles covering the directed knight's graph for an $8 \times 8$ chessboard was computed by Löbbing and Wegener (1996) as 8121130233753702400. They also computed the number of undirected tours, obtaining an
incorrect answer of 33439123484294 (which is not divisible by 4 as it must be), and so we are redoing the calculation. The apparently correct
value of 13267364410532 appears in Wegener's subsequent book (Wegener2000), and also agrees with unpublished calculations of B.D. McKay.

The number of possible tours on a $4 \times k$ board for $k = 3, 4, \ldots$ are 8, 0, 82, 744, 6378, 31088, 189688, 1213112, ..

The following additional results are given by Kraitchik (1942, pp. 264-265). There are 14 tours on the $3 \times 7$ rectangle, two of which are symmetrical. There are 376 tours on the $3 \times 8$ rectangle, none of which is closed. There are 16 symmetric tours on the $3 \times 9$ rectangle and 8 closed tours on the $3 \times 10$ rectangle. There are 58 symmetric tours on the $3 \times 11$ rectangle and 28 closed tours on the $3 \times 12$ rectangle. There are five doubly symmetric tours on the $6 \times 6$ square. There are 1728 tours on the $5 \times 5$ square, 8 of which are symmetric. The longest "uncrossed" knight's tour on an $n \times n$ board for $n = 3, 4, \ldots$ are 2, 5, 10, 17, 24, 35, ..

Backtracking algorithms (in which the knight is allowed to move as far as possible until it comes to a blind alley, at which point it backs up to a given position) can be used to find knight's tours, but such methods can be very slow. A backtracking algorithm tries to construct a solution to a computational problem incrementally, one small piece at a time. Whenever the algorithm needs to decide between multiple alternatives to the next component of the solution, it recursively evaluates every alternative and then chooses the best one.

Warnsdorff (1823) proposed an algorithm that finds a path without any backtracking by computing ratings for "successor" steps at each position. Here, successors of a position are those squares that have not yet been visited and can be reached by a single move from the given position. The rating is highest for the successor whose number of successors is least. In this way, a square ending in a blind alley is visited first and therefore prevented from being isolated. (Roth). The time needed for this algorithm grows roughly linearly with the number of squares of the chessboard, but unfortunately computer implementation shows that this algorithm runs into blind alleys for chessboards bigger than $76 \times 76$, despite the fact that it works well on smaller boards (Roth).
Recently, Conrad et al. (1994) discovered another lineartime algorithm and proved that it solves the Hamiltonian path problem for all \( n \geq 5 \). The Conrad et al. algorithm works by decomposing the chessboard into smaller chessboards (not necessarily square) for which explicit solutions are known. This algorithm is rather complicated because it has to deal...
with many special cases, but has been implemented in Mathematica
by A. Roth. Example tours are illustrated above for \( n \times n \) boards with \( n = 5 \) to 8.

### 3.3 \( n^n \) Queen’s Problem

In chess, a queen can move as far as she pleases, horizontally, vertically, or diagonally. A chessboard has 8 rows and 8 columns. The standard 8 by 8 Queen’s problem describes how to place 8 queens on an ordinary chessboard so that none of them can hit any other in one move. This is an amusing puzzle and chess players and researchers have been finding best solutions to this problem.

The problem is to place \( n \) queens on an \( n \times n \) chessboard, so that no two queens are attacking each other. For readers not familiar with the rules of chess, this means that no two queens are in the same row, the same column, or the same diagonal.

![Chessboard with queens](image)

**Figure 3** Gauss’s first solution to the 8 queens problem, represented by the array [5, 7, 1, 4, 2, 8, 6, 3]

An obvious modification of the 8 by 8 problem is to consider an \( N \) by \( N \) "chessboard" and ask if one can place \( N \) queens on such a board. It is easy to see that this is impossible if \( N \) is 2 or 3, and it's reasonably straightforward to find solutions when \( N \) is 4, 5, 6, or 7. The problem begins to become difficult when there is no solution precisely when \( N = 8 \). The fact that this number coincidentally equals the dimension of an ordinary chessboard has probably contributed to the popularity of the problem.

### 3.4 Game Trees
Consider the following simple two-player game played on an $n \times n$ square grid with a border of squares. Each player has $n$ tokens that they move across the board from one side to the other.

A state of the game consists of the locations of all the pieces and the identity of the current player. These states can be connected into a game tree, which has an edge from state $x$ to state $y$ if and only if the current player in state $x$ can legally move to state $y$. The root of the game tree is the initial position of the game, and every path from the root to a leaf is a complete game.

To navigate through this game tree, we recursively define a game state to be good or bad as follows:

- A game state is good if either the current player has already won, or if the current player can move to a bad state for the opposing player.
- A game state is bad if either the current player has already lost, or if every available move leads to a good state for the opposing player.

Equivalently, a non-leaf node in the game tree is good if it has at least one bad child, and a non-leaf node is bad if all its children are good. By induction, any player that finds the game in a good state on their turn can win the game, even if their opponent plays perfectly; on the other hand, starting from a bad state, a player can win only if their opponent makes a mistake. This recursive definition immediately suggests the following recursive backtracking algorithm to determine whether a given game state is good or bad. At its core, this algorithm is just a depth-first search of the game tree; equivalently, the game tree is the recursion tree of the algorithm! A simple modification of this backtracking algorithm finds a good move (or even all possible good moves) if the input is a good game state.
All game-playing programs are ultimately based on this simple backtracking strategy. However, since most games have an enormous number of states, it is not possible to traverse the entire game tree in practice. Instead, game programs employ other heuristics to prune the game tree, by ignoring states that are obviously (or “obviously”) good or bad, or at least better or worse than other states, and/or by cutting off the tree at a certain depth (or ply) and using a more efficient heuristic to evaluate the leaves.

3.5 Subset Sum
Let’s consider a more complicated problem, called SubsetSum: Given a set \( X \) of positive integers and target integer \( T \), is there a subset of elements in \( X \) that add up to \( T \)? Notice that there can be more than one such subset. For example, if \( X = (8, 6, 7, 5, 3, 10, 9) \) and \( T = 15 \), the answer is True, because the subsets \((8, 7)\) and \((7, 5, 3)\) and \((6, 9)\) and \((5, 10)\) all sum to 15. On the other hand, if \( X = (11, 6, 5, 1, 7, 13, 12) \) and \( T = 15 \), the answer is False.

There are two trivial cases. If the target value \( T \) is zero, then we can immediately return True, because the empty set is a subset of every set \( X \), and the elements of the empty set add up to zero.6 On the other hand, if \( T < 0 \), or if \( T \neq 0 \) but the set \( X \) is empty, then we can immediately return False. For the general case, consider an arbitrary element \( x \in X \). (We’ve already handled the case where \( X \) is empty.) There is a subset of \( X \) that sums to \( T \) if and only if one of the following statements is true:

- There is a subset of \( X \) that includes \( x \) and whose sum is \( T \).
- There is a subset of \( X \) that excludes \( x \) and whose sum is \( T \).

In the first case, there must be a subset of \( X \setminus \{x\} \) that sums to \( T - x \); in the second case, there must be a subset of \( X \cap x \), that sums to \( T \). So we can solve \( \text{SubsetSum}(X, T) \) by reducing it to two simpler instances:
SubsetSum($X \backslash \{x\}, T-x$) and SubsetSum($X \backslash \{x\}, T$). The resulting recursive algorithm is shown below.

![Algorithm Diagram]

3.6 Text Segmentation – Longest Increasing Subsequence
Suppose you are given a string of letters representing text in some foreign language, but without any spaces or punctuation, and you want to break this string into its individual constituent words.

For any sequence $S$, a subsequence of $S$ is another sequence obtained from $S$ by deleting zero or more elements, without changing the order of the remaining elements; the elements of the subsequence need not be contiguous in $S$. For example, when you drive down a major street in any city, you drive through a sequence of intersections with traffic lights, but you only have to stop at a subsequence of those intersections, where the traffic lights are red. If you’re very lucky, you never stop at all: the empty sequence is a subsequence of $S$. On the other hand, if you’re very unlucky, you may have to stop at every intersection: $S$ is a subsequence of itself.

As another example, the strings BENT, ACKACK, SQUARING, and SUBSEQUENT are all subsequences of the string SUBSEQUENCEBACKTRACKING, as are the empty string and the entire string SUBSEQUENCEBACKTRACKING, but the strings QUEUE and EQUUS and TALLYHO are not. A subsequence whose elements are contiguous in the original sequence is called a substring; for example, MASHER and LAUGHTER are both subsequences of MANSLAUGHTER, but only LAUGHTER is a substring.

Now suppose we are given a sequence of integers, and we need to find the longest subsequence whose elements are in increasing order. More concretely, the input is an integer array $A[1..n]$, and we need to compute the longest possible sequence of indices $1 \leq i_1 < i_2 < \ldots < i_n$.
n such that $A[ik] < A[ik+1]$ for all $k$. One natural approach to building this *longest increasing subsequence* is to decide, for each index $j$ in order from 1 to $n$, whether or not to include $A[j]$ in the subsequence.

```
LIBIGGER(prev,A[1..n]):
  if n = 0
    return 0
  else if $A[1] \leq prev$
    return LIBIGGER(prev,A[2..n])
  else
    skip ← LIBIGGER(prev,A[2..n])
    take ← LIBIGGER(A[1],A[2..n]) + 1
  return max(skip, take)
```

### 4.0 CONCLUSION

In this unit, you will observe that the hill climbing can be used to solve problems that have many solutions but where some solutions are better than others. An example of a problem that can be solved with hill climbing is the travelling salesman problem. It is also used widely in artificial intelligence fields for reaching a goal state from a starting node.

### 5.0 SUMMARY

This unit has addressed the following:

- Hill climbing technique and algorithm
- How hill climbing is used in solving problems
- The Knight’s tour problems.
- The Queen’s problem
- Games trees
6.0 TUTOR-MARKED ASSIGNMENT

a. Briefly describe the climbing technique and algorithm.
b. Briefly explain how the climbing technique is used in solving problems.
c. Compare the Knight’s tour problem and the Queen’s problem you have learnt about in this unit.
d. State two application areas of game tree algorithm.

7.0 REFERENCES/FURTHER READINGS


Euler, L. "Solution d'une question curieuse qui repose sur un soumis a aucune analyse." Mémoires de l'Académie Royale des Sciences et Belles Lettres de Berlin, Année 1759-15, 310-337, 1766.

Friedel, F. "The Knight's Tour."


Jelliss, G. "Knight's Tour Notes."


Pegg, E. Jr. "Leapers (Chess Knights and the Like)"


Rose, C. "The Distribution of the Knight."


Roth, A. "The Problem of the Knight: A Fast and Simple Algorithm."

http://library.wolfram.com/information/MathSource/909/.
Ruskey, F."Information onthe Knight's Tour Problem."  
http://www.theory.csc.uvic.ca/~cos/inf/misc/Knight.html.


Sloane, N. J.A. Sequences A001230,A003192/M1369,  
A006075/M3224, A033996,and A079137 in "TheOn-Line Encyclopediaof Integer Sequences."


Thomasson,D."The Knight's Tour."  

vander Linde, A.(1874).  

Vandermonde,A.-T."Remarquessur les Problèmes de Situation."  

Velucchi, M. "Knight's Tour: The Ultimate Knight's Tour Page of Links."  


UNIT 8 PRACTICAL EXERCISE III

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
   3.1 Problem I
   3.2 Problem II
4.0 Conclusion
5.0 Summary
6.0 Tutor-Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

In this unit, you will be exposed to the practical applications of all the sorting algorithms you learned in the previous units of this course.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

   explain how bubble sort is implemented in C programming language
   show how quick sort is implemented in C programming language.

3.0 MAIN CONTENT

3.1 Problem I

Write the C codes for implementing bubble sort

Solution

```c
void bubblesort(int numbers[], int array_size)
{
    int i, j, temp;

    for (i = (array_size - 1); i > 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (numbers[j-1] > numbers[j])
            {
                temp = numbers[j-1];
                numbers[j-1] = numbers[j];
                numbers[j] = temp;
            }
        }
    }
}
```
numbers[j] = temp;
}
}

3.2 Problem II

Write the C codes for implementing quicksort

Solution

void quicksort(int numbers[], int array_size)
{
    q_sort(numbers, 0, array_size - 1);
}

void q_sort(int numbers[], int left, int right)
{
    int pivot, l_hold, r_hold;

    l_hold = left;
    r_hold = right;
    pivot = numbers[left];
    while (left < right)
    {
        while ((numbers[right] >= pivot) && (left < right))
            right--;
        if (left == right)
        {
            numbers[left] = numbers[right];
            left++;
        }
        while ((numbers[left] <= pivot) && (left < right))
            left++;
        if (left == right)
        {
            numbers[right] = numbers[left];
            right--;
        }
    }
    numbers[left] = pivot;
    pivot = left;
    left = l_hold;
    right = r_hold;
    if (left < pivot)
4.0 CONCLUSION

This unit has showed you how to implement bubble sort using the C codes. Also, it showed you how to execute the quicksort using the C codes as well.

5.0 SUMMARY

By now you should have learnt how:

- To implement the bubble and the quick sort using the C codes.
- To write programs using the C programming language.

6.0 TUTOR-MARKED ASSIGNMENT

1. Study the codes above and run them with sampled data on a C compiler.
2. Write the codes for implementing any sorting algorithm in any programming language that you have learnt.
3. Apply game trees algorithm to solve a practical problem.

7.0 REFERENCES/FURTHER READINGS


