Module 1: Basic Algorithmic Analysis

Module Overview

In the study of computer science, you need to understand that the essence of algorithms, and also that of data structures, is very fundamental. Algorithms are not just important for you as programmers and computer science students, everyone who uses a computer wants it to run programs or tasks faster or to solve larger problems. This initial module, which discusses basic algorithmic analysis, provides you with an overview of algorithms, their place in modern computing systems, and how they are analysed. It also defines an algorithm and the asymptotes, and presents some examples together with some justification for you to choose one algorithm over another and considering them as part of a technology, Graphic User Interface (GUI), Object Oriented Systems (OOS), and networks. This module is therefore divided into two units which are:

Study Unit 1: Introduction to Algorithms, Concepts and Principles of Algorithms.
Study Unit 2: Analysis of Standard Complexity Classes, Time and Space Trade-off in Algorithms Analysis, and Asymptotic Analysis of Upper and Average Complexity.

Study Unit 1: Introduction to Algorithms, Concepts and Principles of Algorithms

Expected Duration: # week

Introduction

This section of the course on Algorithm is intended to provide you with an insight into the basic concepts of algorithms. A brief discussion shall be presented to you in the areas of history, definition, features and correctness of algorithms, examples and categories of algorithm, and specification of algorithms. We shall discuss herein the elementary aspects of the course, in line with the curriculum, and look at problem solving with algorithms, algorithms and program development, and attempt to end this section with implementation strategies for algorithms.

Learning Outcome for Study Unit 1

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

1.1 Define and use correctly the words in bold (SAQ 1.1)
1.2 List and explain the concepts, principles and features of algorithms (SAQ 1.2)
1.3 List and explain the use of algorithms in problem solving (SAQ 1.3)
1.4 Explain the role algorithms in program development (SAQ 1.4)
1.5 Mention and explain the implementation strategies for algorithm. (SAQ 1.5)

1.1. What is an Algorithm

In the perspective of history; the older form of the term "algorithm" first appeared in website's New World Dictionary as late as 1957, the older form is "Algorism". The term **algorithm** has an ancient meaning – The process of doing arithmetic using Arabic numerals. During the Middle Ages, abacists computed on the abacus and algorists computed by algorism. The term algorism subsequently changed to algorithm through many pseudo-etymological perversions (Knuth, 1997)

Algorithm can therefore be defined as an organised computational procedure that takes some values, or set of values, as input and produces some value, or set of values, as output (Cormen, et al, 2009).Thus, an algorithm is a sequence of computational steps that transform the input into the desired output. Alternatively, an algorithm could be seen as a tool for solving a well-specified computational problem.

Another significant attempt towards defining the notion of algorithm is from the point of view of mathematical set theory by (Knuth, 1997), which provides a definition of an algorithm by formally defining a computational method as a quadruple (Q, I, Ω, f) in which *Q* is a set containing subsets I and *Ω*, and *f* is a function from *Q* to itself. Furthermore *f* should leave *Ω* point wise fixed; that is *f(q)* should equal *q* for all elements *q* of *Ω*. The four quantities *Q, I, Ω, f* are intended to represent respectively the states of the computation, the input, the output, and the computational rule.

Understanding the basic elements of the above definitions will elucidate the features of algorithms that follow. Nevertheless, algorithms are known to concentrate on the high level design of data structures and methods for using them to solve problems. The subject is highly mathematical, but the mathematics can be compartmentalized and makes provision for emphasis to be placed on what rather than why. The assumed prerequisite is that scholars can take a description of an algorithm and relevant data structures, and use a programming tool to implement the algorithm.

1.2 Concepts, principles and features of algorithm

Having noted that an algorithm encompasses a finite set of rules that give a sequence of operations for solving a specific type of problem, we thus discuss the features and correctness of an algorithm. An algorithm should have at least five important features;

i. **Finiteness:** an algorithm must terminate after a finite number of steps. There are exceptions, of course, seen in computational methods and reactive processes, automatic teller machines, weather forecasts, and other Terminate and Stay Resident (TSR) programs.

ii. **Definiteness**: each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.

iii. **Input**: an algorithm has zero or more inputs: quantities that are given to it initially before the algorithm runs. These inputs are taken from specified sets of objects.

iv. **Output:** an algorithm has one or more outputs: quantities that have a specified relation to the inputs.

v. **Effectiveness:** this implies that the operations of an algorithm must all be sufficiently basic that they can in principle be done i.e. exactly and in a finite length of time by someone using pencil and paper (manually).

vi. **Correctness of Algorithms:** an algorithm is said to be correct it, for every input instance, it halts with the correct output. Then we can assert that a correct algorithm solves the given computational problem. An incorrect algorithm might not halt at all on some input instances, a common mistake made by programmers, or it might halt with an incorrect answer. Nevertheless, incorrect algorithms can sometimes be useful, provided we can control their error rate.

In-text Question: Define an algorithm in terms of input and output.

Solution: Algorithm can therefore be defined as an organised computational procedure that takes some values, or set of values, as input and produces some value, or set of values, as output.

1.2.1 Examples of Algorithms

a. Sorting Algorithms

Numerous computations and tasks become simple by properly sorting information in advance. Sorting can be with respect to a sequence of characters, say in alphabetical order, or a sequence of numbers into non decreasing order, etc.

The above problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools.

Even though we shall subsequently be discussing sorting in detail, here we formally define the problem: consider.

Input: A sequence of n numbers [a_1, a_2, ……, a_n]

Output: A permutation (reordering) [a_1^1, a_2^1,……a_n^1] of the input sequence such that: $a_1^1 \leq a_2^1 \leq \ldots\ldots \leq a_n^1$.

To illustrate, given the input sequence <31, 41, 59, 26, 41, 58> The sorting algorithm returns as the output the sequence - <26,31,41,41,58,59>.

Such an input sequence is called an instance of the sorting algorithm.

In general, an instance of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem. Sorting is a fundamental operation in computer science for the reason that many programs use it as an intermediate step. As a result, there exists a large number of sorting algorithms. However, the choice of sorting algorithm, for any given application, depends mostly on the following,

i. The number of items to be sorted.
ii. The extent to which the items are already somewhat sorted,
iii. The possible restrictions on the item values.
iv. The architecture of the computer, and the kind of storage devices to be used: main memory, disks, or even tapes.

Examples of sorting algorithm include insertion, bubble sort, merge sort etc.

b. Searching Algorithms:

Searching algorithms are very critical in the operation of applications of all vintages: small to large and the World Wide Web. They function by finding the existence of a target item within a collection.

Given a collection C of elements, there are three fundamental queries that could be asked.

i. Existence: Does C contain a target element t?
ii. Retrieval: Return the element in C that matches the target element t.
iii. Associative lookup: Return information associated in collection C with the target key element key element k. (Heineman, 2009).

Examples of searching algorithm include sequential search, binary search, hash-based search, etc.

c. Graph Algorithms:

Graphs are fundamental structures used in computer science to represent complex structured information. Inherently, a graph contains a set of elements, known as vertices, and relationships between pairs of these elements known as edges.

Types of graphs that occur commonly in algorithms include; undirected, directed, weighted, and hyper graphs.

d. Network flow Algorithms

As we see in Operations Research – Network problems, there are numerous problems that can be viewed as a network of vertices and edges, with a capacity associated with each edge over which commodities flow.

The application of this kind of algorithm is significant in solving problems of shortest path (minimum cost), transportation, transhipment, maximum flow (efficient capacity utilization), etc.

e. Computational Geometry Algorithms:

In Computer Science, the algorithms are useful for solving geometric problems. A computational geometry problem involves geometric objects, such as points, lines, and polygons. Precisely, a computational geometry problem is defined by;

(i) The type of input data to be processed

(ii) The computation to be performed, and

(iii) Whether the task is static or dynamic.

These classifications help identify the techniques that can improve efficiency across families of related problems.

In modern engineering and mathematics, computational geometry has applications in such diverse fields as computer graphics, robotics, VSLI design, Computer-aided design, molecular modelling, metallurgy, manufacturing, textile layout, forestry, and statistics. Input elements include, set of points, set of line segments, or the vertices of a polygon in counter clockwise order. The output is often a response to query about the objects, such as possible intersections, or new geometric objects etc.

In-text Question: Mention five examples of algorithms

Solution: Five examples of algorithms are; sorting, searching, graph, network flow and computational algorithms.

1.2.2 Categories of Algorithm

Algorithms can be categorized in terms of style, application, or complexity.

(a) **Style**

The Commonly used styles are divide-and-conquer, recursion, dynamic programming (bottom-up or memoization) and greedy strategy (do the best thing locally and hope for the best).

(b) **Application**

Common application categories include; mathematics, geometry, graphs, string matching, sorting, and searching.

(c) **Complexity Algorithms**

These algorithms are a larger category including any algorithm that must consider the best result among a large sample space of possibilities. Many complexity, or otherwise called combinational algorithms, are NP – complete. NP – complete problems are those that have shown not to be nonpolynomially solvable (i.e. no better method or more efficient method for solving them).

1.2.3 Specification of Algorithms

Algorithms can be specified in English, as a computer program, or even as a hardware design. The only requirement is that the specification must provide a precise description of the computational procedure to be followed. This provision emphasizes the need to pay attention to the principles of good programming style (to be discussed subsequently). Nevertheless, some of the implementations of the algorithms we will encounter will be in the same high level language as contained in the texts: while some appear in C–like language, students should be able to implement some of the algorithm using OOP tools, such as Java or C++.

1.3 Use of Algorithms in Problem Solving

This section answers the question of what problems that algorithms can solve. The Sorting algorithm discussed as an example earlier represents just a tiny fraction of problems algorithms can solve. There are several practical applications for which algorithm are known to be useful. These include and are not limited to the following;

i. The human Genome project – Towards identifying all the “100,000” genes in human DNA, storing this information in databases, and developing tools for data analysis.

Each of these steps requires sophisticated algorithms. (The details of these algorithms are beyond the scope of this course).

ii. The internet – very popular, enabling users to quickly access and retrieve large amounts of information. By using clever algorithms, sites on the internet are able to manage and manipulate large volumes of data. Typical among these include route finders (shortest path problem), search engines, etc.

iii. Electronic commerce – enabling "goods" and "services" to be negotiated and exchanged electronically, and significantly depends on the privacy of personal and sensitive information, such as credit card numbers, passwords etc. The core technologies used in electronic commerce include public–key cryptography and digital signatures, which are based on numerical algorithms and number theory.

iv. Manufacturing and other commercial enterprises – these require allocating relatively scarce resources in the most beneficial way, and desire to seek out the opportunity to minimize cost. (Optimization problems). Specifically;

- An oil company may wish to know where to place its wells in order to maximize its expected profit (maximum flow problems)

- A political candidate may want to determine where to spend "more" money buying campaign advertising in order to maximize chances of victory (Game Theory)

- An airline may wish to assign crews to flights in the least expensive way possible, making sure that each flight is covered and that extant regulations regarding crew scheduling are met. (Resource Allocation and Trans shipment Problems).

- An Internet service provider may wish to determine where to place additional resources to serve customers better. (Resource Allocation Problem).

1.4 Role of Algorithms in Program Development

Conventionally, initial attempts to solve problems start with the analysis of the problem and the required data. After the analysis phase, a detailed procedural solution can emerge. An algorithm, being a computable set of steps to achieve a designed step, requires precise definition of steps for solving problems.

Each step of an algorithm must be an unambiguous instruction to facilitate coding, which results in the actual program that can be executed by a computer. Importantly, knowledge of the factors that lead to improved performance of key algorithms is essential for the success of software applications. Needless to mention that implementation of most of the key algorithms can be found in literature and that, in practice, reinventing the wheel is usually not advisable. Nevertheless, the need to develop the skills of algorithms can never be overstated.

Operationally speaking, the steps of an algorithm must be ordered, termination of an algorithm is necessary whether or not the algorithm is successful; although there are exceptions, such as weather monitors, automatic teller machines, and other TSRs mentioned earlier. Notably, algorithms are chosen on the basis of efficiency, accuracy, and clarity.

In-text Question: Mention one way that algorithms assist in program development
Solution: Algorithms help to explicitly outline and define the steps involved in programming before it is done.

1.5 Implementation Strategies for Algorithms

There are various ranges of options for implementing algorithms to solve specific problems. For reasons of space and time, we "list" a few;

(i) **Understanding the problem -** problem identification and analysis is highly essential; in the first instance starting with the big picture (clear wholistic view), understanding the problem, identifying potential causes and courses of action, and digging into the details are vital measures.

(ii) **Experimentation** – developing test algorithms and experimenting with them to ascertain suitability to problems being addressed.

(iii) **Practice with real codes** – it is reasonable to ensure that algorithms are translatable to actual codes in target high-level languages to demonstrate effectiveness.

(iv) **Optimization** – as an essential strategy for improving performance of algorithms: determining the quantitative behaviour of algorithms is important, as the choice of an algorithm is dependent on the execution time, resource usage such as memory, adaptability (platform's independence), etc.

1.5 Summary of Study Unit 1

In this unit, you have learned the following;

1. An Algorithm is an organised computational procedure that takes some values, or set of values, as input and produces some value, or set of values, as output
2. An algorithm should have at least five important features which includes; finiteness, definiteness, input, output, effectiveness and correctness.
3. Examples of algorithms include, sorting, searching, graph, network and computational algorithms.
4. Algorithms can be categorized in terms of style, application, or complexity
5. Algorithms can be used in E-commerce, Internet, human Genome projects as well as manufacturing enterprises.
6. Algorithms help to explicitly outline and define the steps involved in programming before it is done
7. Implementation strategies for algorithms requires understanding of the problem, experimentation, practicing with codes, optimization among others.

1.6 Self-Assessment Questions (SAQs) for Study Unit 1

Now that you have completed this study session, you can check to see how well you have achieved its Learning Outcomes by answering the following questions.

SAQ 1.1 (Tests learning outcome1.1)

1. Define the term Algorithm

SAQ 1.2 (Tests learning outcome 1.2)

1. The five important features of an algorithm are?
2. List three examples of algorithms

3. Algorithms can be categorized in terms of ……, ……., or ……

SAQ 1.3 (Tests learning outcome1.3)

1. Mention one way algorithms can be applied in human Genome project.

SAQ 1.4 (Tests learning outcome1.4)

1. How can algorithms assist in program development?

SAQ 1.5 (Tests learning outcome1.5)

1. Which of the following is NOT an implementation strategy for algorithms?

   A. Experimentation B. Optimization C. Overhauling D. Problem identification

References and Further Reading

i. Cormen T. H., Leiserson, C. E., Rivest, R. L., and Stein, C., Introduction to Algorithms, Massachusetts Institute of Technology, Fourth Edition, 2022.
ii. Sedgewick, R. and Wayne, K., Algorithms, Addison Wesley, Fourth Edition, 2011.
iii. Weiss, M. A., Bhattachatjee, A. K., and Mukherjee, S., Data Structures and Algorithms in Java, Pearson Education Limited, Third Edition, 2012
iv. Knuth, D. E., The Art of Computer Programming, Addison Wesley, Volumes 1 to 4, Third Edition, 2011
v. Heineman, G. T., Pollice G., and Selkow, S., Algorithms in a Nutshell, O'Reilly Media Inc., Second Edition, 2015
vi. And Numerous Internet Resources.

Unit 2: Analysis of algorithms

Expected Duration: 1 week

Introduction

The process of **analysis** of algorithms is very important, as programming and program development can become complex and complicated tasks. Such a scenario, considering large projects, especially, require coordinated efforts of many people, storing and accessing large quantities of data efficiently, and solving complex computational problems.

Intuitively, quite a number of solutions can be designed to solve a particular problem. The set of solutions, each of which can often times be correct by producing the right output or result upon termination, is usually regarded as the set of feasible solutions, provide the basis from which the optimal solution (if it exists) can emerge through the analytical process of stepwise refinement.

In algorithms' design and analysis, the process of refinement considers the objective and/or quantitative properties of the algorithms in terms of efficiency, correctness, and sometimes semantic analysis of the target algorithm. In this unit, therefore, we discuss analysis of standard Complexity Classes, time and space trade-off in algorithms analysis, and asymptotic analysis of upper and average Complexity, together with how they can be categorised as best case, average case, and worst case analysis.

Learning Outcome for Study Unit 2

A the end of this unit, you are expected to do the following;

2.1 Define relevant keywords in bold (SAQ 2.1)

2.2. List and explain the various algorithms that can be analysed using RAM model (SAQ 2.2)

2.3. List and explain the approaches in Time and Space algorithms Analysis. (SAQ 2.3)

2.4. Mention and discuss the different approaches of asymptotic analysis of Upper and average Complexity (SAQ 2.4)

2.1 What is Algorithm's Analysis?

Analysing an algorithm is now generally accepted to imply predicting the resources that the algorithm requires. Therefore, analysis of algorithms points to considering resources such as memory, communication bandwidth, or energy consumption. In most cases, it often entails measuring computational time. When analysing several candidate algorithms for a given problem, the most efficient one can be identified. The result can show up with more than one viable candidate, however, the inferior algorithms can be ruled out in the process. One such process is seen in the use of the Random Access Machine (RAM) model for analysing algorithms. In this course, we treat the topic of standard complexity classes under empirical analysis of algorithms using the RAM model. Thus, the following section briefly discusses the use of the RAM model and illustrates same using an incremental algorithm, insertion sort algorithm, and a divide and conquer algorithm, such as merge sort. We shall be discussing the design of these algorithm and others that follow the divide and conquer technique in subsequent sections.

2.2 Empirical Analysis of Efficiency of Algorithm

In practical or empirical terms, models of implementations technology for algorithmic options are known to be useful for comparative analysis of algorithms. There can, of course, be models for the resources of a particular technology and their attendant costs. The RAM model is one viable option (empirical method) for comparative analysis of algorithms. Another is the mix machine model, which was nearly abandoned but there are indications of modernization (knuth, 1997). Mix will not be discussed here.

In the RAM model, instructions are executed, by assumption, one after another, with no provisions for concurrency. The RAM model contains instructions commonly found in real computers such as; arithmetic operations of addition, subtraction, multiplication, division, remainder, floor, ceiling; data movement operations of load, store, copy; and control operations of conditional and unconditional branch, subroutine call and return.

Each of such instructions takes a constant amount of time. For reasons of simplicity, the data types in the RAM model are integer and floating point. Also, an assumed limit is placed on the size of each word of data, so that the word size does not grow arbitrarily.

Continuing, the issue of exponentiation is regarded as having a constant time, even though in practice it may not be so especially considering operations such as x^y where x and y are real numbers.

Whereas RAM model analysis is useful for performance prediction on actual machines, analyzing a simple algorithm in the RAM model can still pose a challenge. Nevertheless, the need for a method that is simple to write and manipulate, which shows the important characteristic of an algorithm's resources requirements, even by suppressing tedious details, can never be over-emphasised.

2.2.1 Analysis of Insertion Sort Using RAM Model

Insertion sort is one of the important algorithms that we discussed in the preceding section of this course (Design of Algorithms).We recall that the time taken by the inserting-sort procedure depends or the input: Sorting a million numbers takes longer time than sorting a hundred; two input sequences of same size can sort at different times depending on how nearly sorted they are. Nevertheless, traditionally, the running time of a program is regarded as a function of the size of its input.

Thus, we distil out the two parameters of interest; "running time" and "input size". For some problems, the best measure of input size is the total number of bits needed to represent the input in ordinary binary notation. While in others such as a graph, the input size can be represented by a number such as the number of vertices and edges, in the graph. The running time is a function of the number of primitive operations or "steps" or "turns" executed.

In consonance with the RAM model, it is acceptable to assume that a constant time is required to execute each line of the INSERTION-SORT algorithm. One line may take a different amount of time than another line, but it can be assumed that each execution of the i^{th} line takes time C_i, where C_i is a constant

In our Insertion-sort procedure, j = 2,3,..., n; where n = A. length, let t_j denote the number of times the while loop test in line 5 is executed for that value of j.

When a "for" or "while" loop exists in the usual way (i.e. due to the test in the loop header), the test is executed one time more than the loop body.

Comments are not executable statements, and therefore, take no time.

INSERTION-SORT (A)	Cost	Times
1. for j = 2 to A.length	C_1	n
2. key = A [j]	C_2	n - 1
3. /* Insert A [j] into the sorted seq A[1…j-1] */	0	n - 1
4. i = j -1	C_4	n - 1
5. While i > o and A[I] > key	C_5	$\Sigma^{n}_{j=2}$ tj
6. A[i + 1] = A [i]	C_6	$\Sigma^{n}_{j=2}$ (tj -1)
7. i = i – 1	C_7	$\Sigma^{n}_{j=2}$ (tj-1)
8. A [i + 1] = key	C_8	n - 1

The running time of the algorithm is the sum of the running times for each statement executed: a statement that takes C_i steps to execute and executes n times will contribute C_i n to the total running time.

Thus, to compute the total time of insertion sort

T (n);

$$T(n) = C_1 n + C_2 (n-1) + C_4 (n-1) + C_5 \sum_{j=2}^{n} (t_j) + C_6 \sum_{j=2}^{n} (t_j - 1)$$

$$+ C_7 \sum_{j=2}^{n} (t_j - 1) + C_8 (n-1)$$

NB Bear in mind that the actual running times may differ for best – case; where input sequence is somewhat sorted, and worst –case; without any initial sorting. The worst – case running time of an algorithm gives an upper bound (known as the O-notation) on the running time for any input size. Determination of this value provides a guarantee that the algorithm will never take a time longer.

In-text Question: Mention five instructions that can be found in a Ram Model

Solution: Five instructions found in a Ram model are addition, subtraction, multiplication, division, remainder

2.2.2 Analyzing Divide and Conquer Algorithms

When an algorithm contains a recursive call to itself, the running time can be described by a recurrence equation or simply recurrence, which describes the overall running time on a problem of size n in terms of the running time on smaller inputs.
Consider:

```
MERGE-SORT (A, P, r)

1.    if  p < r

2.        q = [(p + r)/2]  {use the floor or ceiling function for odd input sizes}

3.        MERGE-SORT (A, P, q)

4.        MERGE-SORT  (A, q + 1, r)

5         MERGE  (A, P, q, r)
```

A recurrence for the running time of a divide –and – conquer algorithm falls out from the three basic steps of the paradigm (i.e. divide, conquer, and combine).
As usual, let T(n) be the running time on a problem of size n. if the problem size is small enough, say $n \leq C$ for some constant C, the straight forward solution takes constant time, which is denoted as $\Theta(1)$ (i.e. $\Theta(1)$: big theta of 1).
Suppose that the division of the problem yields "a" sub problems, each of which is 1/b the size of the original (for merge-sort both "a" and "b" are 2, but there are some divide-and-conquer algorithms in which $a \neq b$).
It takes time T (n/b) to solve one sub problem of size n/b, and so it takes time aT (n/b) to solve "a" of them.

Where; a = no of sub problems (divisions);

b = no of elements in each sub problem

If we take D(n) time to divide the problem into sub problems and C(n) time to combine the solutions to the sub problems into the solution to the original problem, we get the recurrence.

$$T(n) = \begin{cases} \theta(1) & \text{If } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise.} \end{cases}$$

2.2.3 Analysis of Merge-Sort in the RAM Model

The pseudocode for merge-sort works correctly when the number of elements is not even* (Cormen, et al 2009). For simplicity, the original problem is assumed to be the power of 2. Thus, each divide step yields two subsequences of size exactly n/2.

For the worst-case running time of merge-sort on n numbers, where merge-sort on just one element takes a constant time, when there are n > 1 elements, the running time is as follows;

. Divide: This computes the middle of the sub array and takes a constant time.

Thus; $D(n) = \theta(1)$.

a. Conquer: This recursively solves two sub problems of size n/2 each and contributes time 2 T (n/2) to the running time.

b. Combine: The merge procedure on an n-element sub array takes time $\theta(n)$ and so $C(n) = \theta(n)$

c. When we add the functions D(n) and C(n) for the merge-sort analysis, we are adding a function that is $\theta(n)$ and a function that is $\theta(1)$. This sum is a linear function of n, that is $\theta(n)$. Adding this result to the 2 T(n/2) term from the conquer step gives the recurrence for the worst-case running time T(n) of merge-sort as:

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2\,T(n/2) + \theta(n) & \text{if } n > 1 \end{cases}$$

It has been proven, just as we saw in the time analysis of binary and sequential search, that merge-sort, with its $\theta(n \log n)$ running time, out performs insertion-sort, whose worst-case running time is $\theta(n^2)$, as n becomes increasingly large (i.e $n \rightarrow \infty$).

2.3 Time and Space Trade-off in Algorithm Analysis

There are fundamentally two approaches for analysis of algorithms. These are; the theoretical analysis – using proof of correctness and asymptotic representations (big – oh and related notations); and using empirical analysis – involving testing and measurement over a range of

instances (benchmark). In this category, from literature, we have the random access machine (RAM) model, which we have illustrated in the preceding section, there is also the obsolete MIX machine [Knuth, 1997], etc.

Irrespective of the approach and as stated earlier, considerations of efficiency can be; by a function of Input size, by measuring running time, or by worst-case, best-case, and average-case analysis.

2.3.1 Efficiency as a function of input size: Typically, more time and space are needed to run an algorithm on bigger inputs (e.g, more numbers, larger strings, larger graphs). To analyse efficiency as a function of n = size of input;

- Searching/sorting – n = number of items in list
- String processing – n = length of string(s)
- Matrix operations – n = dimension of matrix

  n x n matrix has n^2 elements.
- Graph processing - n_v = number of vertices and

  N_g = number of edges.

2.3.2 Measuring Running Time:

Measuring time efficiency of algorithms involves:

- Identifying the basic operation(s) contributing the most to running time,
- Characterizing the number of times it is performed as a function of input size.
- Basically, we can estimate running time T(n) by : $T(n) \approx Cop * C(n)$.

  Where:

  T(n) - running time as a function of n

  Cop - running time of a single basic operation, and

  C(n) - number of basic operations as a function of n.

In-text Question: The Divide and conquer algorithm works best when the number of elements is not even. TRUE or FALSE?

Solution: FALSE.

2.3.3 Worst-case, Best-case, and Average-case Analysis:

- Considering sequential search, for instance, which searches for a target element t in an array A of n elements, the number of turns to make will run from 0 to n – 1 [or can we say from 1 to n?].

We can state the case analysis as follows:

- Basic operations: The comparison in the loop
- Worst case: n comparisons
- Best case: 1 comparison
- Average case: (n + 1)/2 comparisons

  assuming each element equally likely to be searched.

2.3.4 Order of Growth Classifications (Growth of Functions)

Some of the values of functions important for analysis of algorithms are summarized in the following table in order of magnitude:

N	**Log $_2$ n**	**N**	**n log$_2$ n**	**n ^{2}**	**n ^{3}**	**2 ^{n}**	**n!**
10	3.3	10^1	3.3×10^1	10^2	10^3	10^3	3.6×10^6
10^2	6.6	10^2	6.6×10^2	10^4	10^6	1.3×10^{30}	9.3×10^{157}
10^3	10	10^3	1.0×10^4	10^6	10^9		
10^4	13	10^4	1.3×10^5	10^8	10^{12}		
10^5	17	10^5	1.7×10^6	10^{10}	10^{15}		
10^6	20	10^6	2.0×10^7	10^{12}	10^{18}		

Constant: a program whose running time's order of growth is a constant and executes a fixed number of operations to finish its job; consequently its running time does not depend on n, the size of input.

Logarithm: a program with this running time is barely slower than a constant-time program. The classic example of a program whose running time is logarithmic in the problem size is binary search. The base of the logarithm is not relevant with respect to the order of growth (since all logarithms with a constant base are related by a constant factor), so we can use log n when referring to order of growth.

Linear: programs that spend a constant amount of time processing each piece of input data, or that are based on a single 'for loop' are quite common. The order of growth of such program is said to be linear: its running time is proportional to n.

Linearithmic: this is also seen texts, and used to describe programs whose running time for a problem of size n has order of growth of nlog n. Again the base of the logarithm is irrelevant with respect to the order of growth. The prototypical examples of linearithmic algorithms are merge-sort and quick-sort algorithms.

Quadratic: a typical program whose running time has order of growth of n^2 has two nested 'for loops', used for some calculation involving all pairs of n elements. The elementary sorting algorithms, such as Selection-Sort and Insertion-Sort are prototypes of the programs in this classification.

Cubic: a typical program whose running time has the order of growth of n^3 has three nested 'for loops', used for some calculation involving the triples of n elements. An example is a program to calculate three-sums.

Exponential: the term exponential is used to refer to algorithms whose order of growth is b^n for any constant $b > 1$, even though different values of b lead to vastly different running times. Exponential algorithms are extremely slow: you will never run one of them to completion for a large problem. Still, exponential algorithms play a critical role in the theory of algorithms: because there exists a large class of problems for which it seems that an exponential algorithm is the best choice.

The above stated classifications are most common, but certainly not the complete set. The order of growth of an algorithm's cost might be n^2 log n or $n^{3/2}$ or some similar function.

Indeed, the detailed analysis of algorithms can require the full gamut of mathematical tools that have been developed over the centuries.

It is extremely important to note that one of the primary reasons to study the order of growth of a program is to help design a faster algorithm to solve the same problem: by simply taking advantage of the implication of their running times on specified input size, n.

2.4 Asymptotic Analysis of Upper and Average Complexity

In our consideration (comparative analysis) of sequential search and binary search, we saw that, intuitively, we could compare their performances over an input size of one million records. However, we remarked that such simplistic comparisons will be difficult as the input size(s) grow(s), which makes generalization difficult.

In order to overcome the aforementioned constraints as input sizes grow, an analytical and theoretical approach called asymptotic efficiency of algorithms is most commonly used.

Asymptotic efficiency of algorithms makes it possible to simply consider input sizes large enough to make only the order of growth of the running time relevant. In this regard, emphasis is just on how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound (i.e. within limited time, as input size is unbounded). Usually, an algorithm that is asymptotically more efficient will be the best choice for all but for very small inputs. (Cormen, et al, 2009).

Asymptotic analysis of efficiency of algorithms therefore is a method of describing limiting behaviour by using asymptotic notations.

Asymptotic notations, also called Bachmann–Landau notation, describe limiting behaviours of functions as arguments of the functions grow or tends to infinity.

Actually, the asymptotes provide a way of comparing functions by ignoring constant factors and small input sizes.

Typically, the basic operation count can be approximated as $Cg(n)$, where $g(n)$ is the order of growth.

Comment

In this section, we formally define some of the asymptotic notations, as briefly as possible, owing to our limits. These are the big 0 (O - notation), the big omega (Ω - notations), the theta (Θ - notation), the little o (o - notation), and the little omega (ω - notation) respectively.

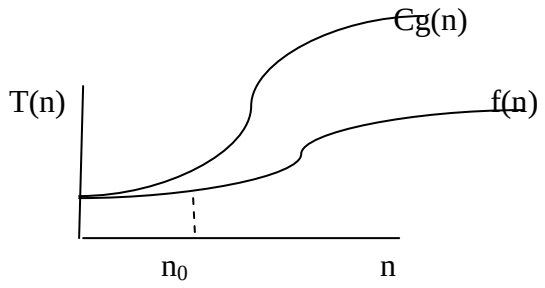
(1) The O - notation

The O – notation (pronounced big Oh) is used when there is only an asymptotic upper bound. It is one of the standard notations in computer science for characterizing the efficiency of an algorithm, in terms of running time or space requirement for a problem of size n written as O (expression), where expression is the expression in terms of n.

The O-notation is formally defined as follows; for a given function with value g(n),

$O(g(n)) = \{f(n)$: there exist positive constants C and n_0 such that $0 \leq f(n) \leq Cg(n)$ for all $n \geq n_0\}$

Illustrated graphically as:



- The O-notation is used to give an upper bound to the set of values of a function, to within a constant factor.
- Alternatively, we can view the O(g(n)) as follows;

  O(g(n)): functions $\leq$ C g(n)

In terms of time;

T(n) ε O(g(n)) if there are positive constants C and n_0 ə T(n) $\leq$ Cg(n) for all

$n \geq n_0$

- Different functions with the same growth rate may be represented using the same O–notation. It is commonly used to describe how closely a finite series approximates a given function, by omitting constant factors and lower order terms. The O-notation specifically describes worst-case scenario.

Examples:

(a) Using O-notation to simplify functions:

Consider: $f(x) = 6x^4 - 2x^3 + 5$

- Recall that the asymptotic notation ignores small input values and constant factors. Of the terms in f(x) above, the one with the highest growth rate is the one with the largest exponent as a function of x, namely $6x^4$, which is the product of 6 and x^4. The first factor does not depend on x. This leaves us with the simplified form as x^4.
- Thus, $f(x) = O(x^4)$ (i.e. f(x) is big O of (x^4))
- Applying formal definition of the O-notation, the statement that

  $f(x) = O(x^4)$ is equivalent to its expansion;

  $/f(x)/ \leq m /g(x)/$ for some suitable choices of x_0 and M for all $x > x_0$

  [The burden of proof and application is on students].

(b) Suppose an algorithm is being developed to operate on a set of n-elements,

where it is required to determine time T(n) for the algorithm to run in terms of

the number of elements in the input size.

- The algorithm works by first calling a subroutine to sort elements in the set and then performs its own operations.
- The sort procedure has a known time complexity of $O(n^2)$, and after the subroutine runs, the algorithm must take an additional $55n^3 + 2n + 10$ time before it terminates.
  - Thus the overall time complexity of the algorithms is $T(n) = O(n^2) + 55n^3 + 2n + 10$. Using the O-notation, simplify T(n).

In-text Question:

The type of algorithmic growth with order of growth as b^n for any constant $b > 1$ is known as?

Solution: Exponential Algorithm

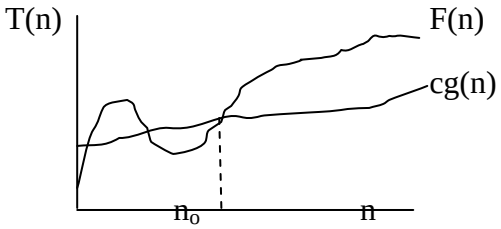
2. The Ω – notation

The Ω – notation (pronounced "big-omega of g of n") provides an asymptotic lower bound, just as the O-notation provides an asymptotic upper bound on a function.

For a given function g(n), the Ω – notation is usually denoted at Ω(g(n)), and formally defined as:

$\Omega(g(n)) = \{ f(n)$: There exist positive constants c and n_o such that $o \leq cg(n) \leq f(n)$ $n \geq n_o\}$

- This can be illustrated graphically as follows;



- From the above figure, for all values n at or to the right of n_o (some threshold) the value of f(n) is on or above cg(n).

(3) The Θ-notation

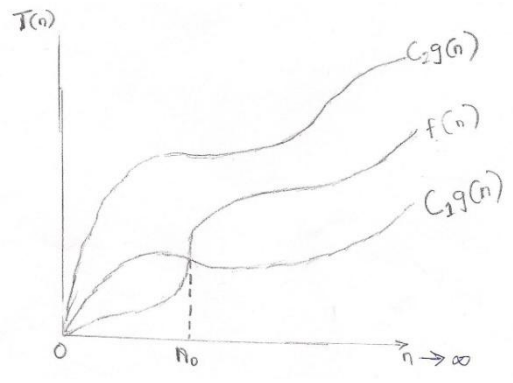
For a given function g(n), we denote by Θ(g(n)) the set of functions:

$\Theta(g(n)) = \{f(n)$: there exist positive constants C_1, C_2, and n_o such that

$\theta \leq C_1 g(n) \leq f(n) \leq C_2 g(n)$ for all $n \geq n_o\}$

- This implies that the function f(n) belongs to the set Θ(g(n)) if there exist positive constants C_1 and C_2 such that it can be "sandwiched" between $C_1g(n)$ and $C_2g(n)$, for sufficiently large n.

- Θ-notation is illustrated graphically:



The Θ(f(n)) or Θ(g(n)) is the set of all functions with the same order of growth as f(n) or g(n), as the case may be.

Note: T(n) = Θ(f(n)) is equivalent to stating that T(n) = Ω(f(n)) but not the converse.

If T(n) = Ω(f(n)) and T(n) = O(f(n)).

Then T(n) = Θ(f(n)).

{Onus of proof and example on students}.

The Θ-notation allows us to immediately determine the algorithm that can run faster, all things being equal.

Generally, asymptotic notations are the machine–independent (i.e. actual machine, compiler, and programming language) notation for the running times, and perhaps other efficiency parameters, of an algorithm.

We take a quick look at the remainder; the o-notation and the ω-notation.

(4) The o -notation

- The little o-notation is relevant as the asymptotic upper bound provided by O-notation may or may not be asymptotically tight. The bound

  $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not. [Cormen, et al, 2009]

- The o-notation (little – oh of g of n) → o(g (n))

  is formally defined as;

  O (g(n) = {f(n): for any positive constant c > o, there exists a constant n_o > o such that $0 \leq f(n) < cg(n)$ for all $n \geq n_o$}

For example, $2n = o(n^2)$, but $2n^2 \neq o(n^2)$

Observe that the definition of O-notation and o-notation are similar. The main difference is that in $f(n) = O(g(n))$, the bound $o \leq f(n) \leq cg(n)$ holds for some constant $c > o$, but in the case of o-notation;

$f(n) = o(g(n))$, the bound $0 \leq f(n) < cg(n)$ holds for all constants $c > o$.

Intuitively, in o-notation, the function f(n) becomes insignificant relative to g(n) as n approaches infinity; that is,

$$\lim_{n \to \infty} f(n)/g(n) = 0$$

This is sometimes used as the definition for o-notation.

(5) The ω -notation

- The ω-notation (little – omega notation) is used to denote a lower bound that is not asymptotically tight. Just as o-notation is to O-notation, ω-notation is to Ω-notation.

- We formally define ω-notation as;

  $\omega(g(n)) = \{ f(n)$: for any positive constant $c > o$, there exists a constant $n_o > o$, such that $0 \leq cg(n) < f(n)$, for all $n \geq n\ o\}$

  Alternative definition is given by;

  $f(n) \in \omega(g(n))$ iff $g(n) \in o(f(n))$

- For example $n^2/2 = \omega(n)$, but $n^2/2 \neq \omega(n^2)$.

- The relation $f(n) = \omega(g(n))$ implies that

$$\lim_{n \to \infty} f(n)/g(n) = \infty$$

The above means that f(n) becomes increasingly larger than g(n) as n becomes larger and tends to infinitely.

In-text Question:

Under the o-notation, the expression $2n = o(n^2)$, but $2n^2 = o(n^2)$. TRUE or FALSE

Solution: FALSE

2.5 Summary of Study Unit 2

In this unit, you have learned the following;

1. The meaning of algorithm analysis
2. Some of the features of Ram model instruction include; data type, exponentiation, mathematical operations, single instruction at a time, constant time for a single instruction, etc.
3. Algorithm analysis using Insertion sort, Merge sort algorithms
4. Some of the factors that determine the efficiency of an algorithm include; Input size and running time.
5. Asymptotic efficiency of algorithms makes it possible to simply consider input sizes large enough to make only the order of growth of the running time relevant.
6. The O– notation (pronounced big Oh) is one of the standard notations in computer science for characterizing the efficiency of an algorithm, in terms of running time or space requirement for a problem of size n written as O(expression) and it is used when there is only an asymptotic upper bound. Similarly, the Ω is used when there is an asymptotic lower bound.

2.6 Self-Assessment Questions (SAQs) for Study Unit 2

SAQ 2.1 (Tests learning outcome2.1)

1. Define the term Algorithm analysis

SAQ 2.2 (Tests learning outcome 2.2)

1. Mention two algorithms that can be analysed using the Ram Model.
2. A type of algorithm analysis procedure which computes the middle of the sub array and takes a constant time is known as?

SAQ 2.3 (Tests learning outcome2.3)

1. The two fundamental approaches for analysis of algorithms are ……and…….

SAQ 2.4 (Tests learning outcome2.4)

1. The type of algorithm analysis that makes it possible to consider input sizes large enough to make only the order of growth of the running time relevant is called?

Notes to the Self- Assessment Questions (SAQs) for Study Unit 1

References and Further Reading

i. Cormen T. H., Leiserson, C. E., Rivest, R. L., and Stein, C., Introduction to Algorithms, Massachusetts Institute of Technology, Fourth Edition, 2022.
ii. Sedgewick, R. and Wayne, K., Algorithms, Addison Wesley, Fourth Edition, 2011.
iii. Weiss, M. A., Bhattachatjee, A. K., and Mukherjee, S., Data Structures and Algorithms in Java, Pearson Education Limited, Third Edition, 2012
iv. Knuth, D. E., The Art of Computer Programming, Addison Wesley, Volumes 1 to 4, Third Edition, 2011
v. Heineman, G. T., Pollice G., and Selkow, S., Algorithms in a Nutshell, O'Reilly Media Inc., Second Edition, 2015
vi. And Numerous Internet Resources.

SAQ 1.1 (Tests learning outcome1.1)

1. An algorithm should have at least five important features which includes; finiteness, definiteness, input, output, effectiveness and correctness

SAQ1.2 (Tests learning outcome 1.2)

1. The five important features of an algorithm are, finiteness, definiteness, input, output, effectiveness and correctness.

2. Three examples of algorithms are ; sorting, searching, graph algorithms.

3. Algorithms can be categorized in terms of style, application, or complexity.

SAQ 1.3 (Tests learning outcome1.3)

1. One way of applying algorithms to human genome is by sequential storing and retrieval gene information in databases for analysis.

SAQ 1.4 (Tests learning outcome1.4)

1. Algorithms helps to explicitly outline and define the steps involved in programming before it is done

Notes to the Self- Assessment Questions (SAQs) for Study Unit 2

SAQ 2.1 (Tests learning outcome 2.1)

1. Algorithm analysis points to considering resources such as memory, communication bandwidth, or energy consumption and measuring computational time.

SAQ 2.2 (Tests learning outcome2.2)

1. Insertion sort and Divide-and-conquer algorithms

2. Divide and rule algorithm

SAQ 2.3 (Tests learning outcome2.3)

1. Two fundamental approaches for analysis of algorithms are Theoretical and Empirical approaches.

SAQ 2.4 (Tests learning outcome2.4)

1. Asymptotic efficiency

References and Further Reading

i. Cormen T. H., Leiserson, C. E., Rivest, R. L., and Stein, C., Introduction to Algorithms, Massachusetts Institute of Technology, Fourth Edition, 2022.
ii. Sedgewick, R. and Wayne, K., Algorithms, Addison Wesley, Fourth Edition, 2011.
iii. Weiss, M. A., Bhattachatjee, A. K., and Mukherjee, S., Data Structures and Algorithms in Java, Pearson Education Limited, Third Edition, 2012
iv. Knuth, D. E., The Art of Computer Programming, Addison Wesley, Volumes 1 to 4, Third Edition, 2011

v. Heineman, G. T., Pollice G., and Selkow, S., Algorithms in a Nutshell, O'Reilly Media Inc., Second Edition, 2015
vi. And Numerous Internet Resources.