AL-MUHIBBAH OPEN UNIVERSITY

DAI 812:

DATA STRUCTURES AND ALGORITHMS



Courseware

MASTER IN INFORMATION TECHNOLOGY (MIT)

COPYRIGHT PAGE

© 2026 Al-Muhibbah Open University, Nigeria

All rights reserved. No part of this publication may be reproduced in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior permission of the management.

Published by:

Al-Muhibbah Open University, Nigeria. 2026

Tel: +2348036826868

E-mail: info@almuhibbahooenuniversity.com

Website: www.aou.edu.ng

Printed by

Al-Muhibbah Open University Press Limited,

Bauchi State, Nigeria.

Tel: 08067567699

COURSE WRITERS/DEVELOPMENT TEAM

Subject Matter Experts
Dr. Muhammad Qaim Aliyu Sabo
Jamilu Waziri

Subject Matter Reviewers
Prof Edeghagba Eghosa Elijah

Language Reviewer
Jamilu Waziri

Instructional Designers/Graphics
Aliyu Ibrahim Usman MNLA, CLN

Hayatudeen Kobi

Editor

ODL Expert
Engr. Dr. Ibrahim Hassan
Abdulbasid Muhammad Maigari
Aliyu Ibrahim Usman MNLA, CLN

QUOTE

"Algorithms + Data Structures = Programs."

— Niklaus Wirth, Book Title (1976)

"I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships."

— Linus Torvalds, Creator of Linux

ACKNOWLEGEMENT



The invaluable contributions of Open Educational Resources that informed this courseware are acknowledged. Special acknowledgement is extended to Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (CLRS) for Introduction to Algorithms — the definitive reference for algorithm theory that informed the theoretical foundations of this course. Additional acknowledgement to Robert Sedgewick and Kevin Wayne (Princeton Algorithms course), Aditya Bhargava (Grokking Algorithms), and the LeetCode and HackerRank communities whose problem sets informed the practical programming exercises in this courseware.

COURSE INTRODUCTION



students with fundamental knowledge of data structures and algorithms for solving computational problems. Students will learn common linear and non-linear data structures and ways to implement them, as well as common sorting and searching algorithms.

The course also teaches students to analyse algorithm complexity and design efficient algorithms. All data structures and algorithms are implemented in Python, with focus on understanding the reasoning behind each structure's design rather than memorising implementations. Practical problem-solving using LeetCode and HackerRank-style exercises is integrated throughout.

AL-MUHIBBAH OPEN UNIVERSITY

COURSE DESCRIPTION



There are six modules. Module 1 covers linear data structures (arrays, linked lists, stacks, queues). Module 2 covers recursion and dynamic programming. Module 3 examines sorting algorithms. Module 4 covers searching algorithms. Module 5 addresses algorithm efficiency and complexity analysis. Module 6 covers non-linear data structures (trees, graphs, heaps, hash tables).

Course Aims

DAI 812 aims to equip students with a deep understanding of fundamental data structures and algorithms as tools for solving computational problems. Students will develop the ability to select and implement appropriate data structures for different problem domains, analyse algorithm efficiency using asymptotic notation, and design efficient algorithms using established paradigms — producing graduates capable of solving complex computational problems in professional software development environments.

Course Objectives

Upon completion of the course, you should be able to:

- Implement and manipulate linear data structures: arrays, linked lists, stacks, and queues
- Design and implement recursive algorithms with correct base cases and recursive cases
- Apply memoisation and dynamic programming to optimise recursive solutions

- Implement and compare sorting algorithms: bubble, selection, insertion, merge, quick, heap
- Implement searching algorithms: linear search, binary search, hash table search, BFS, DFS
- Analyse algorithm time and space complexity using Big-O, Omega, and Theta notation
- Apply the Master Theorem to solve recurrence relations for divide-and-conquer algorithms
- Implement and traverse binary trees and Binary Search Trees
- Implement heap data structure and heap sort
- Implement graph representations and traversal algorithms (BFS, DFS, Dijkstra)
- Select and justify the appropriate data structure and algorithm for a given computational problem

COURSE REQUIRMENT

You should note that although this course has no subject pre-requisite, you are expected to have:

- Satisfactory level of English proficiency
- Basic Computer Operations proficiency
- Online interaction proficiency
- Web 2.0 and social media interactive skills

LEARNING RESOURCES

Here are online resources that you need to bookmark if you want to have access to free open online resources.

S/NO	RESOURCE	LINK
1.	Coursera	https://www.coursera.org/
2.	EdX	https://www.edx.org/
3.	YouTube	https://www.youtube.com/
4.	Khan Academy	https://www.khanacademy.org/
5.	MIT Open Courseware	https://ocw.mit.edu/index.htm
6.	Open Yale Courses	https://oyc.yale.edu/

COURSE LEARNING OUTCOMES

At the end of this course, learners should be able to:



i. define ethical hacking and explain its purpose in modern cybersecurity
ii. distinguish between white hat, grey hat, and black hat hackers
iii. describe the five phases of a penetration test
iv. explain the role of authorisation, scope, and rules of engagement in professional testing

AL-MUHIBBAH OPEN UNIVERSITY

COURSE ACTIVITIES

Specifically, this course shall comprise the following activities:

- Studying courseware
- Listening to course audios
- Watching relevant course videos
- Field activities, industrial attachment or internship, laboratory or studio work
- (whichever is applicable)
- Course assignments (individual and group)
- Forum discussion participation
- Tutorials (optional)
- Semester examinations (Computer-Based Test and essay based).

AL-MUHIBBAH OPEN UNIVERSITY

COURSE SCHEDULE

This is a 15-week course. It requires a formal study time of 12 hours. We recommend you take an average of one to two hours for an extra personal study. You can also benefit from online discussions with your course tutor. Kindly see course activity table for more information and guide.

WEEKS	MODULE	STUDY UNIT	ACTIVITY
Week 1	STUDY MODULE 1	**Study Session 1:** Arrays and Linked Lists	1. Read Courseware for the Corresponding Study Unit. 2. View the Video(s) on this Study Unit 3. Listen to the Audio on this Study Unit 4. View any other Video/U-tube 5. View referred OER 6. View referred Animation 7. Read Chapter/page of Standard/relevant text. 8. Any additional study material 9. Any out of Class Activity
		Study Session 2: Stacks and Queues	1. Read Courseware for the Corresponding Study Unit. 2. View the Video(s) on this Study Unit 3. Listen to the Audio on this Study Unit 4. View any other Video/U-tube 5. View referred OER

			6. View referred Animation 7. Read Chapter/page of Standard/relevant text. 8. Any additional study material 9. Any out of Class Activity
		Study Session 3: Deque and Circular Structures	1. Read Courseware for the corresponding Study Unit. 2. View the Video(s) on this Study Unit 3. Listen to the Audio on this Study Unit 4. View any other Video/U-tube 5. View referred OER 6. View referred Animation 7. Read Chapter/page of Standard/relevant text. 8. Any additional study material 9. Any out of Class Activity
Week 2	STUDY MODULE 2	**Study Session 1:** Recursion — Principles and Applications	1. Read Courseware for the corresponding Study Unit. 2. View the Video(s) on this Study Unit 3. Listen to the Audio on this Study Unit 4. View any other Video/U-tube 5. View referred OER 6. View referred

			Animation 7. Read Chapter/page of Standard/relevant text. 8. Any additional study material 9. Any out of Class Activity
		Study Session 2: Memoisation and Dynamic Programming	1. Read Courseware for the corresponding Study Unit. 2. View the Video(s) on this Study Unit 3. Listen to the Audio on this Study Unit 4. View any other Video/U-tube 5. View referred OER 6. View referred Animation 7. Read Chapter/page of Standard/relevant text. 8. Any additional study material 9. Any out of Class Activity
		Study Session 3: Backtracking Algorithms	1. Read Courseware for the corresponding Study Unit. 2. View the Video(s) on this Study Unit 3. Listen to the Audio on this Study Unit 4. View any other Video/U-tube 5. View referred OER 6. View referred Animation 7. Read Chapter/page of

			Standard/relevant text. 8. Any additional study material 9. Any out of Class Activity
Week 3	STUDY MODULE 3	**Study Session 1:** Comparison-Based and Non-Comparison Sorting	1. Read Courseware for the corresponding Study Unit. 2. View the Video(s) on this Study Unit 3. Listen to the Audio on this Study Unit 4. View any other Video/U-tube 5. View referred OER 6. View referred Animation 7. Read Chapter/page of Standard/relevant text. 8. Any additional study material 9. Any out of Class Activity
Week 4		**Study Session 2:** Merge Sort and Quick Sort in Depth	1. Read Courseware for the corresponding Study Unit. 2. View the Video(s) on this Study Unit 3. Listen to the Audio on this Study Unit 4. View any other Video/U-tube 5. View referred OER 6. View referred Animation 7. Read Chapter/page of Standard/relevant text. 8. Any additional study

			material 9. Any out of Class Activity
Week 5		**Study Session 3:** Sorting in Practice and Library Algorithms	1. Read Courseware for the corresponding Study Unit. 2. View the Video(s) on this Study Unit 3. Listen to the Audio on this Study Unit 4. View any other Video/U-tube 5. View referred OER 6. View referred Animation 7. Read Chapter/page of Standard/relevant text. 8. Any additional study material 9. Any out of Class Activity
Week 6	STUDY MODULE 4	**Study Session 1:** Search Techniques — From Linear to Hashing	1. Read Courseware for the corresponding Study Unit. 2. View the Video(s) on this Study Unit 3. Listen to the Audio on this Study Unit 4. View any other Video/U-tube 5. View referred OER 6. View referred Animation 7. Read Chapter/page of Standard/relevant text. 8. Any additional study material 9. Any out of Class

			Activity
Week 7		**Study Session 2:** Graph Search — BFS and DFS	1. Read Courseware for the corresponding Study Unit. 2. View the Video(s) on this Study Unit 3. Listen to the Audio on this Study Unit 4. View any other Video/U-tube 5. View referred OER 6. View referred Animation 7. Read Chapter/page of Standard/relevant text. 8. Any additional study material 9. Any out of Class Activity
		Study Session 3: String Searching Algorithms	1. Read Courseware for the corresponding Study Unit. 2. View the Video(s) on this Study Unit 3. Listen to the Audio on this Study Unit 4. View any other Video/U-tube 5. View referred OER 6. View referred Animation 7. Read Chapter/page of Standard/relevant text. 8. Any additional study material 9. Any out of Class Activity

Week 8	STUDY MODULE 5	**Study Session 1:** Algorithm Analysis and Complexity	1. Read Courseware for the corresponding Study Unit. 2. View the Video(s) on this Study Unit 3. Listen to the Audio on this Study Unit 4. View any other Video/U-tube 5. View referred OER 6. View referred Animation 7. Read Chapter/page of Standard/relevant text. 8. Any additional study material 9. Any out of Class Activity
Week 9		**Study Session 2:** Space Complexity and Trade-offs	1. Read Courseware for the corresponding Study Unit. 2. View the Video(s) on this Study Unit 3. Listen to the Audio on this Study Unit 4. View any other Video/U-tube 5. View referred OER 6. View referred Animation 7. Read Chapter/page of Standard/relevant text. 8. Any additional study material 9. Any out of Class Activity

Week 10		**Study Session 3:** Algorithm Design Strategies	1. Read Courseware for the corresponding Study Unit. 2. View the Video(s) on this Study Unit 3. Listen to the Audio on this Study Unit 4. View any other Video/U-tube 5. View referred OER 6. View referred Animation 7. Read Chapter/page of Standard/relevant text. 8. Any additional study material 9. Any out of Class Activity
Week 11	STUDY MODULE 6	**Study Session 1:** Trees — Binary Trees, BSTs, and Heaps	1. Read Courseware for the corresponding Study Unit. 2. View the Video(s) on this Study Unit 3. Listen to the Audio on this Study Unit 4. View any other Video/U-tube 5. View referred OER 6. View referred Animation 7. Read Chapter/page of Standard/relevant text. 8. Any additional study material Any out of Class Activity

Week 12		**Study Session 2:** Graphs — Representation and Algorithms	1. Read Courseware for the corresponding Study Unit. 2. View the Video(s) on this Study Unit 3. Listen to the Audio on this Study Unit 4. View any other Video/U-tube 5. View referred OER 6. View referred Animation 7. Read Chapter/page of Standard/relevant text. 8. Any additional study material Any out of Class Activity
Week 13	TUTORIALS/REVISION/CONSOLIDATION		
Week 14 & 15	SEMESTER EXAMINATION		

GRADING CRITERIA

Grades will be based on the following outlined in the table below:

S/NO	ACTIVITIES	GRADE
1.	Individual Assignments	20
2.	Group Assignments	10
3.	Forum Discussion	5
4.	Quiz	5
5.	Final Examination	60
TOTAL		100

GRADING SCALE

Grading scale is as presented in the table below:

GRADE	SCORE	REMARK
A	70-100	PASS
B	60-69	PASS
C	50-59	PASS
F	0-49	FAIL

ASSESSMENTS



Generally, there are two types of assessment: formative assessment and summative assessment. With regards to your formative assessment, there are two basic forms of assessment in the course: in-text questions (ITQs) and self-assessment questions (SAQs). This manual is essentially filled with ITQs and SAQs. Feedbacks to the ITQs are placed immediately after the questions, SAQs answers are within the text/content.

Your summative assessment is your semester examination. This exam is in multiple choice as CBT / essay format; and it carries 60 percent of your total earning in the course. Schedule dates for submitting assignments and engaging in course activities are available on the Learning Management System (LMS).

TABLE OF CONTENT

Copyright Page 2
Course Writers/Development Team 3
Quote 4
Acknowledgement 5
Course Introduction 6
Course Description 7
Course Requirement 9
Learning Resources 9
Course Learning Outcomes 10
Course Activities 11
Course Schedule 12
Grading Criteria 21
Grading Scale 21
Assessments 22
Table of Contents 23
Course Outline 23

MODULE 1: LINEAR DATA STRUCTURES

Unit 1: Arrays and Linked Lists

Unit 2: Stacks and Queues

Unit 3: Deque and Circular Structures

MODULE 2: RECURSION

Unit 1: Recursion — Principles and Applications

Unit 2: Memoisation and Dynamic Programming

Unit 3: Backtracking Algorithms

MODULE 3: SORTING ALGORITHMS

Unit 1: Comparison-Based and Non-Comparison Sorting

Unit 2: Merge Sort and Quick Sort in Depth

Unit 3: Sorting in Practice and Library Algorithms

MODULE 4: SEARCHING ALGORITHMS

Unit 1: Search Techniques — From Linear to Hashing

Unit 2: Graph Search — BFS and DFS

Unit 3: String Searching Algorithms

MODULE 5: ALGORITHM EFFICIENCY

Unit 1; Algorithm Analysis and Complexity

Unit 2: Space Complexity and Trade-offs

Unit 3: Algorithm Design Strategies

MODULE 6: NON-LINEAR DATA STRUCTURES

Unit 1: Trees — Binary Trees, BSTs, and Heaps

Unit 2: Graphs — Representation and Algorithms

Unit 3: Advanced Structures — Tries, AVL, and Hash Maps

AL-MUHIBBAH OPEN UNIVERSITY

Module 1

Study Unit 1: Arrays and Linked Lists

1.0 INTRODUCTION

A data structure is a way of organising and storing data in a computer so that it can be accessed and modified efficiently. The choice of data structure fundamentally determines the efficiency of algorithms: a problem solved in $O(n^2)$ time with a poor data structure may be solved in $O(n \log n)$ or $O(1)$ with a better choice. As John Hubbard notes, "the study of data structures is the study of algorithms." Together, data structures and algorithms are the foundational tools of all computer science.

This unit introduces linear data structures — structures in which elements are arranged sequentially, each element (except the first and last) having a unique predecessor and successor. We begin with

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
- 3.1 Introduction to Data Structures
- 3.2 Arrays — Structure and Operations
- 3.3 Linked Lists — Singly, Doubly, and Circular

4.0 Conclusion
5.0 Summary
6.0 Self-Assessment Questions (SAQs)
7.0 References/Further Reading

the two most fundamental linear structures: arrays (contiguous memory, index-based) and linked lists (pointer-based, dynamic memory).

2.0 Learning Outcomes

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

- Define a data structure and explain why data structure choice affects algorithm efficiency
- Describe the memory layout of arrays and explain index-based access
- Implement singly linked list operations: insert, delete, search, and traversal
- Compare arrays and linked lists in terms of time and space complexity
- Explain doubly and circular linked lists and their advantages

.

3.0 Main Content

3.1 Introduction to Data Structures

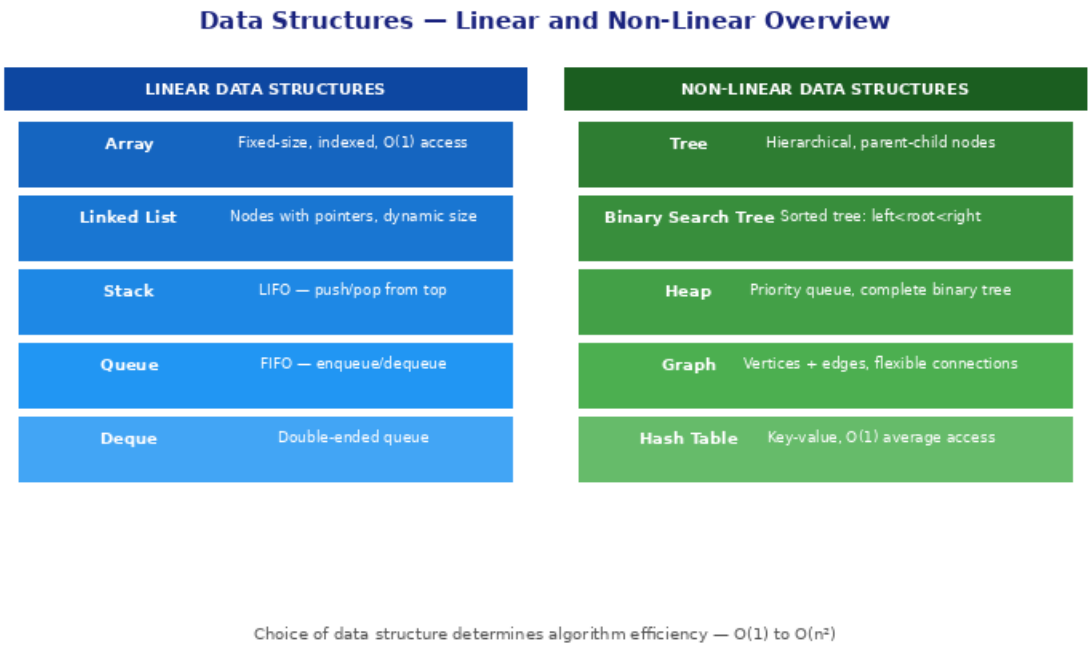


Figure 1.1: Data Structures Overview — Linear and Non-Linear Types, Characteristics, and Complexity

A data structure is a named location for storing and organising data. Every data structure has: a storage model (how data is laid out in memory), a set of operations (what can be done with the data), and a performance profile (the time and space complexity of each operation). Choosing the wrong data structure is like organising a library alphabetically by author when all queries are by topic — the data is correct but retrieval is inefficient.

Key Terms

Abstract Data Type (ADT): A logical description of a data structure defined by its operations and their behaviour, independent of implementation.

Time Complexity: A measure of how execution time grows with input size, expressed using Big-O notation.

Space Complexity: A measure of how memory usage grows with input size.

Linear Structure: A structure in which elements have a sequential ordering — each element has at most one predecessor and one successor.

Dynamic Structure: A structure whose size can grow or shrink during programme execution (linked lists, dynamic arrays).

3.2 Arrays — Structure and Operations

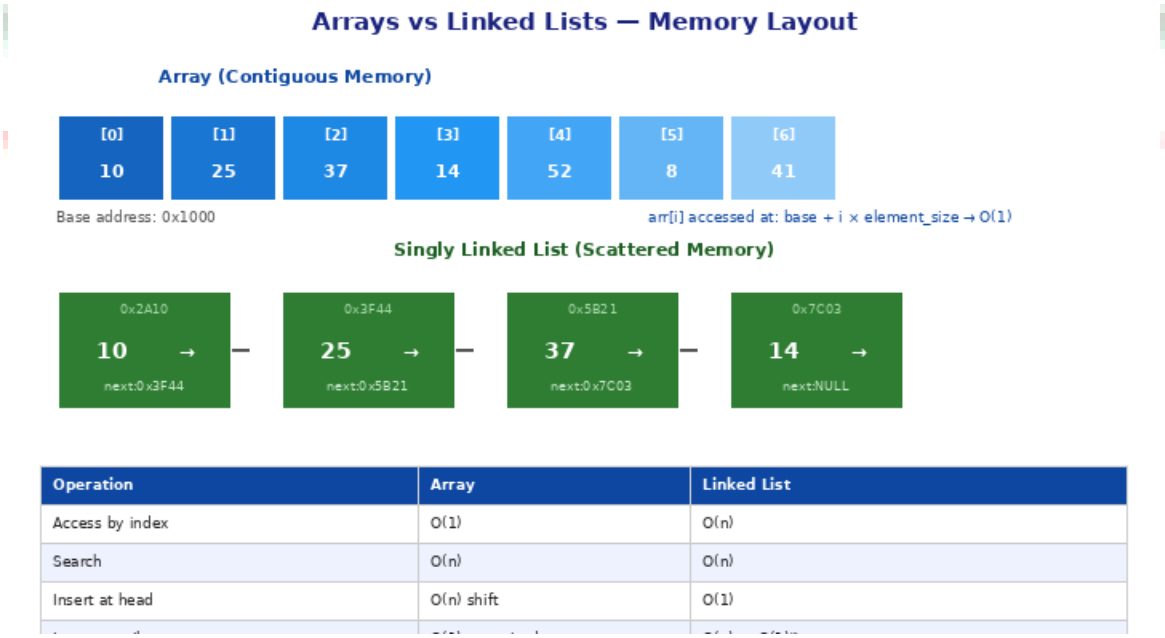


Figure 1.2: Array vs Linked List — Memory Layout, Access Pattern, and Complexity

Comparison

An array is a collection of elements of the same data type stored in contiguous (adjacent) memory locations. Each element is identified by an index (starting from 0 in most languages). The address of element at index i is computed as: base_address + i × element_size, giving O(1) direct access — the fundamental advantage of arrays.

Operations and Complexity

- Access by index: O(1) — computed directly from base address. The defining strength of arrays.
- Search (unsorted): O(n) linear scan. Search (sorted): O(log n) binary search.
- Insert at end: O(1) amortised (dynamic array may occasionally resize — O(n) — but averaged is O(1)).
- Insert at position i: O(n) — all elements from i onward must shift right.
- Delete at position i: O(n) — all elements from i+1 onward must shift left.

Static vs Dynamic Arrays

A static array has a fixed size declared at creation — if more space is needed, a new larger array must be allocated and elements copied. A dynamic array (Python list, Java ArrayList, C++ vector) automatically resizes when capacity is exceeded: typically doubling in size, then copying all elements. Although the occasional resize is O(n), the amortised cost over n insertions is O(1) per insertion — demonstrating how amortised analysis captures long-run efficiency better than worst-case alone.

```
# Python — Array/List operations
arr = [10, 25, 37, 14, 52]
```

```
# O(1) access
print(arr[2])          # 37

# O(n) insert at position
arr.insert(2, 99)      # [10, 25, 99, 37, 14, 52]

# O(n) delete
arr.pop(2)             # removes 99 → [10, 25, 37, 14, 52]

# 2D array (matrix)
matrix = [[1,2,3],[4,5,6],[7,8,9]]
print(matrix[1][2])    # 6 — row 1, col 2
```

3.3 Linked Lists — Singly, Doubly, and Circular

A linked list stores elements in nodes, where each node contains a data field and a pointer (reference) to the next node. Nodes are scattered across memory — not contiguous. The list is navigated by following pointers. This allows efficient insertion and deletion at any known position (O(1) given a reference to the predecessor) but sacrifices O(1) random access — reaching element i requires following i pointers from the head.

Singly Linked List Implementation

```
# Python — Singly Linked List
class Node:
```

```
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert_at_head(self, data):   # O(1)
        node = Node(data)
        node.next = self.head
        self.head = node

    def insert_at_tail(self, data):   # O(n)
        node = Node(data)
        if not self.head:
            self.head = node; return
        curr = self.head
        while curr.next:
            curr = curr.next
        curr.next = node

    def delete(self, data):           # O(n) search + O(1) delete
        if not self.head: return
```

```
        if self.head.data == data:
            self.head = self.head.next; return
        curr = self.head
        while curr.next and curr.next.data != data:
            curr = curr.next
        if curr.next:
            curr.next = curr.next.next

    def traverse(self):              # O(n)
        curr = self.head
        while curr:
            print(curr.data, end=" -> ")
            curr = curr.next
        print("NULL")
```

Doubly and Circular Linked Lists

A doubly linked list adds a prev pointer to each node, allowing traversal in both directions and O(1) deletion given only the node to delete (not needing to find the predecessor). A circular linked list connects the tail back to the head, enabling efficient circular traversal — useful for round-robin scheduling, circular buffers, and the Josephus problem. A doubly circular linked list combines both features and is the basis of Python's collections.deque implementation.

In-text Question



Given an array of n elements and a linked list of n elements, compare the time complexity of: (a) accessing the element at index k; (b) inserting an element at the beginning; (c) deleting the last element.

Intext Answer

(a) Accessing element at index k: Array — O(1), because address = base + k × size is computed in constant time. Linked list — O(k), because you must traverse k nodes from the head following pointers. (b) Inserting at the beginning: Array — O(n), because all n existing elements must shift right by one position to make room. Linked list — O(1), because you create a new node, set its next pointer to the current head, and update head to the new node — three pointer operations regardless of list length. (c) Deleting the last element: Array — O(1), simply decrement the size counter (or call pop()). Linked list — O(n) for a singly linked list, because you must traverse to find the second-to-last node to set its next pointer to NULL. O(1) for a doubly linked list if the tail pointer is maintained, since the tail's prev pointer gives direct access to the predecessor..

4.0 Study Unit Conclusion

In this unit you have studied the fundamental linear data structures: arrays (contiguous memory, O(1) index access, O(n) insertion/deletion) and linked lists (pointer-based, O(1) insertion at known position, O(n) access). The trade-offs between array and linked list structures — access speed vs insertion/deletion flexibility — are among the most fundamental considerations in algorithm design.

5.0 Summary

A data structure organises data for efficient access and modification. Arrays: contiguous memory, O(1) indexed access, O(n) insert/delete, static or dynamic sizing. Linked lists: node-pointer structure, O(1) insert at head, O(n) traversal to access, no index. Singly linked list: one direction; doubly: two pointers, O(1) delete given node; circular: tail→head, used for round-robin. Array strength: random access. Linked list strength: dynamic insertion/deletion without shifting.

6.0 Self-Assessment Questions (SAQs)

1. Define a data structure and explain why choosing the right data structure is critical for algorithm efficiency. Provide an example.
2. Implement a singly linked list in Python with operations: insert_at_head, insert_at_tail, delete_by_value, search, and print_list. Trace each operation on the list [10→20→30→40].
3. Compare arrays and linked lists for: (a) sequential traversal, (b) binary search, (c) implementing a stack, (d) implementing a cache with frequent insertions and deletions. Justify each choice.
4. What is a doubly linked list? Implement a doubly linked list insert_after(node, data) operation and explain why it is O(1) while singly linked list insert_after requires O(n) to find the predecessor.

7.0 References/ Further Reading

Cormen, T.H.; Leiserson, C.E.; Rivest, R.L. & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press.

Goodrich, M.T. & Tamassia, R. (2015). Data Structures and Algorithms in Python. Wiley.

Sedgewick, R. & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.

Bhargava, A.Y. (2016). Grokking Algorithms. Manning Publications.

SAQs Answers

SAQ Answer 1

A data structure organises data for efficient access and modification. Choosing the right structure determines algorithm efficiency. Example: binary search requires a sorted array (O(log n)), but impossible with linked list (O(n) access).

SAQ Answer 2

Operations:

- insert_at_head: O(1) - create node, point to head, update head
- insert_at_tail: O(n) - traverse to end, append
- delete_by_value: O(n) - search then delete
- search: O(n) - traverse until found
- print_list: O(n) - traverse and print

Trace for [10→20→30→40]: Each operation follows pointer navigation.

SAQ Answer 3

(a) Sequential traversal: Both O(n) - linked list uses pointers, array uses indices

(b) Binary search: Array O(log n) - requires random access; linked list impossible

(c) Stack: Both O(1) - array append/pop, linked list head operations

(d) Cache with frequent insert/delete: Linked list O(1) - no shifting required

SAQ Answer 4

Doubly linked list has prev and next pointers. insert_after(node, data) is O(1) because given the node, you already have access to its next pointer and can update it directly. Singly linked list requires O(n) to find the predecessor because you must traverse from head.

Module 1

Study Unit 2: Stacks and Queues

1.0 INTRODUCTION

The stack and queue are restricted-access linear data structures that enforce specific rules about which element may be accessed next. This discipline — trading generality for predictability — makes them powerful tools for modelling real-world processes and implementing classical algorithms. Every function call in every programming language uses a call stack; every print job, message, and process scheduled by an operating system uses a queue.

This unit examines the stack (Last In, First Out — LIFO) and queue (First In, First Out — FIFO) abstract data types, their implementation using both arrays and linked lists, and their extensive

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
- 3.1 Stacks — LIFO Principle
- 3.2 Queues — FIFO Principle
- 3.3 Applications of Stacks and Queues

4.0 Conclusion
5.0 Summary
6.0 Self-Assessment Questions (SAQs)
7.0 References/Further Reading

applications in algorithm design and systems programming.

2.0 Learning Outcomes

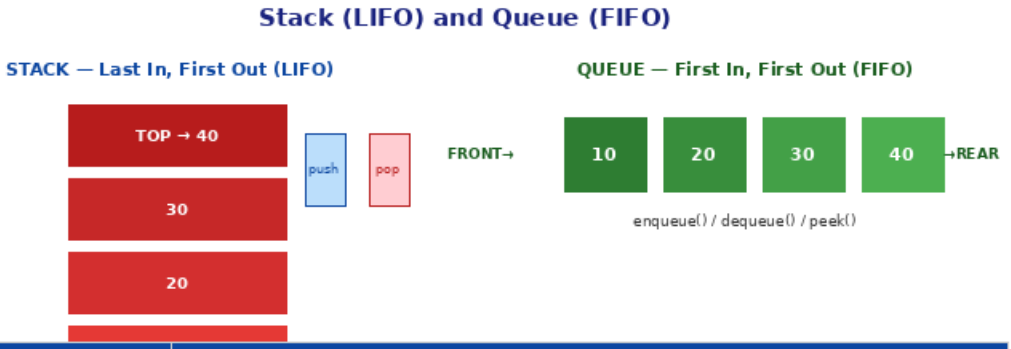
At the end of this unit, you should be able to:

- Explain the LIFO principle of stacks and implement push, pop, and peek
- Explain the FIFO principle of queues and implement enqueue, dequeue, and peek
- Implement stacks and queues using both arrays and linked lists
- Apply stacks to evaluate arithmetic expressions and implement recursive algorithms iteratively
- Describe the priority queue ADT and its implementation using a heap

.

3.0 Main Content

3.1 Stacks — LIFO Principle



Data Structure	Real-World Applications
Stack	Undo/redo, browser back-button, function call stack, expression evaluation, DFS
Queue	Print spooler, CPU scheduling, BFS, message queues (Kafka, RabbitMQ)
Deque	Sliding window algorithms, browser history, palindrome checker
Priority Queue	Dijkstra's shortest path, A* search, OS process scheduling, Huffman coding

Figure 1.3: Stack (LIFO) and Queue (FIFO) — Structure, Operations, and Real-World Applications

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle: the element inserted most recently is the first to be removed. Visualise a stack of plates — you can only add or remove from the top. The three fundamental stack operations: push (add element to top — O(1)), pop (remove and return top element — O(1)), and peek/top (return top element without removing — O(1)). An isEmpty() check is also standard.

```
# Python — Stack implementation using list
class Stack:
    def __init__(self):
```

```
        self._data = []

    def push(self, item):      # O(1) amortised
        self._data.append(item)

    def pop(self):             # O(1)
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self._data.pop()

    def peek(self):            # O(1)
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self._data[-1]

    def is_empty(self):
        return len(self._data) == 0

    def __len__(self):
        return len(self._data)

# Application: Balanced parentheses checker
def is_balanced(expr):
    s = Stack()
    matching = {')': '(', ']': '[', '}': '{'}
```

```
    for ch in expr:
        if ch in '([{': s.push(ch)
        elif ch in ')]}':
            if s.is_empty() or s.pop() != matching[ch]:
                return False
    return s.is_empty()

print(is_balanced("{[()]}"))  # True
print(is_balanced("{[(])}"))  # False
```

3.2 Queues — FIFO Principle

A queue is a linear data structure following the First In, First Out (FIFO) principle: elements are added at the rear (enqueue) and removed from the front (dequeue), exactly as customers queue at a bank. All four operations are O(1): enqueue (add to rear), dequeue (remove from front), peek/front (view front without removing), and isEmpty.

A naive array-based queue suffers from a wasted-space problem: as elements are dequeued from the front, indices are wasted and the "usable" array migrates rightward. A circular queue (ring buffer) solves this by treating the array as circular: (front + 1) % capacity and (rear + 1) % capacity wrap around. This achieves O(1) all operations with efficient space usage and is the standard queue implementation in operating systems and network buffers.

```
# Python — Queue using collections.deque (doubly-linked, O(1) both ends)
from collections import deque

class Queue:
    def __init__(self):
        self._data = deque()

    def enqueue(self, item):   # O(1) — add to rear
        self._data.append(item)

    def dequeue(self):         # O(1) — remove from front
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self._data.popleft()

    def peek(self):            # O(1)
        return self._data[0] if not self.is_empty() else None

    def is_empty(self):
        return len(self._data) == 0

# Application: BFS (Breadth-First Search)
from collections import deque

def bfs(graph, start):
```

```
    visited = set()
    q = deque([start])
    visited.add(start)
    order = []
    while q:
        node = q.popleft()
        order.append(node)
        for neighbour in graph[node]:
            if neighbour not in visited:
                visited.add(neighbour)
                q.append(neighbour)
    return order
```

3.3 Applications of Stacks and Queues

Stack Applications

- Function Call Stack: When function A calls function B, A's return address and local variables are pushed onto the call stack. When B returns, A's frame is popped and execution resumes. Stack overflow occurs when recursion is too deep — the stack runs out of memory.
- Expression Evaluation: Infix expressions (a + b) are converted to postfix (a b +) using a stack (Shunting Yard Algorithm). Postfix expressions are evaluated using a stack: push operands; when an operator is encountered, pop two operands, apply the operator, push the result.

- Undo/Redo: Text editors maintain an undo stack (each action pushed on edit) and redo stack (pushed when undo is called). O(1) undo and redo operations.
- Browser Back/Forward: Visited URLs pushed onto a "back" stack. Clicking back pops from back stack and pushes to forward stack.
- Depth-First Search (DFS): Graph traversal using an explicit stack (iterative) or the implicit call stack (recursive).

Queue Applications

- CPU and Process Scheduling: Ready processes wait in a queue; the CPU services them in FIFO order (or priority order for priority scheduling — requires a priority queue).
- Print Spooler: Print jobs queued and processed in the order received, preventing simultaneous access to the printer.
- Message Queues: Apache Kafka, RabbitMQ, and AWS SQS use queue semantics to decouple producers and consumers in distributed systems.
- Level-Order Tree Traversal (BFS): Process tree nodes level by level using a queue — enqueue the root, then for each dequeued node enqueue its children.

In-text Question



Explain how a stack is used to evaluate the postfix expression "3 4 2 × + 1 5 - ÷". Show the state of the stack after each step.

Intext Answer

Postfix (Reverse Polish Notation) evaluation reads tokens left-to-right: push operands; when an operator is encountered, pop two operands, compute, push result. Evaluating "3 4 2 × + 1 5 - ÷": Token "3": push 3. Stack: [3]. Token "4": push 4. Stack: [3, 4]. Token "2": push 2. Stack: [3, 4, 2]. Token "×": pop 2 and 4; compute 4×2=8; push 8. Stack: [3, 8]. Token "+": pop 8 and 3; compute 3+8=11; push 11. Stack: [11]. Token "1": push 1. Stack: [11, 1]. Token "5": push 5. Stack: [11, 1, 5]. Token "-": pop 5 and 1; compute 1-5=-4; push -4. Stack: [11, -4]. Token "÷": pop -4 and 11; compute 11÷(-4)=-2.75; push -2.75. Stack: [-2.75]. Result: -2.75. This is equivalent to the infix expression (3 + 4×2) ÷ (1-5) = (3+8) ÷ (-4) = 11 ÷ (-4) = -2.75..

4.0 Study Unit Conclusion

In this unit you have examined stacks (LIFO: push/pop/peek O(1)) and queues (FIFO: enqueue/dequeue/peek O(1)), their array and linked-list implementations, and their extensive applications — call stacks, expression evaluation, undo/redo, BFS, process scheduling, and message queues. The circular queue eliminates the wasted-space problem of naive array queues.

5.0 Summary

Stack (LIFO): push/pop/peek all O(1). Applications: call stack, balanced parentheses, expression evaluation (postfix), undo/redo, DFS, browser history. Queue (FIFO): enqueue/dequeue/peek all O(1). Circular queue solves wasted-space problem. Applications: BFS, CPU scheduling, print spooler, message queues. Python: list for stack (append/pop), collections.deque for queue (append/popleft). Priority queue: dequeue in priority order — implemented using a min-heap (O(log n) insert and extract-min).

6.0 Self-Assessment Questions (SAQs)

1. Implement a stack using a singly linked list (not a Python list). Demonstrate push and pop operations and explain why this implementation has O(1) push and pop.
2. Trace through the Shunting Yard algorithm to convert the infix expression "a + b × c - (d ÷ e)" to postfix notation, showing the stack state at each step.
3. Implement a queue using two stacks. Explain the amortised time complexity of enqueue and dequeue operations.
4. A hospital triage system processes patients in priority order (Critical > Serious > Stable). Design a priority queue solution and explain the data structure you would use to implement it efficiently.

7.0 References/ Further Reading

Cormen et al. (2022). Introduction to Algorithms (4th ed.). MIT Press.

Sedgewick, R. & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.

Goodrich, M.T. & Tamassia, R. (2015). Data Structures and Algorithms in Python. Wiley.

Python Documentation. (2023). collections.deque. https://docs.python.org/3/library/collections.html

SAQs Answers

SAQ Answer 1

python

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None


class Stack:
    def __init__(self):
        self.head = None
```

```
    def push(self, data):
        node = Node(data)
        node.next = self.head
        self.head = node  # O(1)

    def pop(self):
        if not self.head: return None
        data = self.head.data
        self.head = self.head.next  # O(1)
        return data
```

O(1) because operations only affect the head - no traversal needed.

SAQ Answer 2

Infix: a + b × c - (d ÷ e)

- Read a → output: a
- Read + → stack: +
- Read b → output: a b
- Read × → stack: + ×
- Read c → output: a b c
- Read - → pop × → output: a b c ×; pop + → output: a b c × +; push -

- Read (→ push (
- Read d → output: a b c × + d
- Read ÷ → push ÷
- Read e → output: a b c × + d e
- Read) → pop ÷ → output: a b c × + d e ÷

  Postfix: a b c × + d e ÷ -

SAQ Answer 3

python

```
class QueueWithStacks:
    def __init__(self):
        self.in_stack = []
        self.out_stack = []

    def enqueue(self, item):
        self.in_stack.append(item)  # O(1)

    def dequeue(self):
        if not self.out_stack:
```

```
        while self.in_stack:
            self.out_stack.append(self.in_stack.pop())
        return self.out_stack.pop()  # Amortised O(1)
```

Amortised O(1): each element moved once from in to out stack.

SAQ Answer 4

Use a **priority queue** implemented with a **min-heap** (or max-heap for highest priority). Each patient is stored with priority value: Critical=3, Serious=2, Stable=1. Insert O(log n), extract-highest O(log n). Python's heapq with negative priorities handles this efficiently.

Module 2

Study Unit 1: Recursion — Principles and Applications

1.0 INTRODUCTION

Recursion is a programming technique in which a function solves a problem by calling itself with a simpler version of the same problem. Far from being a curiosity, recursion is one of the most powerful tools in a programmer's repertoire: tree traversals, graph searches, divide-and-conquer algorithms (merge sort, binary search), and dynamic programming are all fundamentally recursive in nature.

This unit examines the principles of recursion — base case, recursive case, and the call stack — through classic recursive algorithms (factorial, Fibonacci, Tower of Hanoi, binary search). It then introduces memoisation and dynamic programming as techniques for transforming

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
- 3.1 Recursion Fundamentals
- 3.2 Recursive Algorithms — Classic Examples
- 3.3 Memoisation and Dynamic Programming

4.0 Conclusion
5.0 Summary
6.0 Self-Assessment Questions (SAQs)
7.0 References/Further Reading

exponentially slow naive recursion into polynomial-time algorithms.

2.0 Learning Outcomes

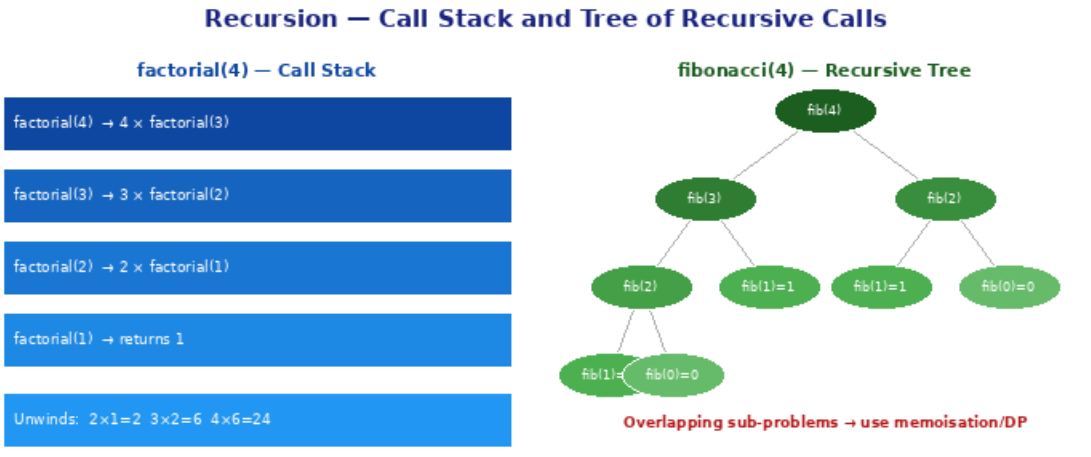
At the end of this unit, you should be able to:

- Explain the base case, recursive case, and call stack in recursive algorithms
- Trace recursive calls through the call stack for factorial and fibonacci
- Implement binary search recursively and analyse its time complexity
- Identify and resolve overlapping sub-problems using memoisation
- Implement bottom-up dynamic programming for the fibonacci and coin change problems

.

3.0 Main Content

3.1 Recursion Fundamentals



Rule	Description
Base Case	The condition that stops the recursion — without it, infinite recursion (stack overflow)
Recursive Case	The part that breaks the problem into smaller sub-problems
Progress	Each recursive call must move closer to the base case

Figure 2.1: Recursion — Call Stack for factorial (4) and Recursive Tree for fibonacci(4)

A recursive function must satisfy two conditions: (1) Base case — a condition under which the function returns a result directly without making a further recursive call. Without a base case (or with an incorrect one), the function recurses infinitely, eventually causing a stack overflow error. (2) Recursive case — the part of the function that calls itself with a smaller or simpler version of the problem, making progress toward the base case with each call.

Every recursive call creates a new stack frame on the call stack, storing the function's local variables, parameters, and return address. The call stack unwinds as base cases are reached and return values propagate back up. Deep recursion (e.g., n=10,000 for

factorial) can exhaust the call stack — Python's default recursion limit is 1,000.

```
# Factorial — n! = n × (n-1)!
def factorial(n):
    if n <= 1:          # Base case
        return 1
    return n * factorial(n - 1)  # Recursive case

# Trace: factorial(4)
# → 4 × factorial(3)
#       → 3 × factorial(2)
#             → 2 × factorial(1)
#                   → 1  (base case)
# Unwind: 2×1=2, 3×2=6, 4×6=24

# Tower of Hanoi
def hanoi(n, source, target, auxiliary):
    if n == 1:
        print(f"Move disk 1 from {source} to {target}")
        return
    hanoi(n-1, source, auxiliary, target)
    print(f"Move disk {n} from {source} to {target}")
    hanoi(n-1, auxiliary, target, source)

# T(n) = 2^n - 1 moves — exponential time
```

```
hanoi(3, "A", "C", "B")
```

3.2 Recursive Algorithms — Classic Examples

Binary Search (Recursive)

```
# Binary Search — O(log n)
def binary_search(arr, target, low, high):
    if low > high:                # Base case: not found
        return -1
    mid = (low + high) // 2
    if arr[mid] == target:        # Base case: found
        return mid
    elif arr[mid] < target:
        return binary_search(arr, target, mid + 1, high)
    else:
        return binary_search(arr, target, low, mid - 1)

# Each recursive call halves the search space → O(log n)
# Recurrence: T(n) = T(n/2) + O(1) → T(n) = O(log n)
arr = [3, 9, 10, 14, 27, 38, 43, 52, 67, 82]
print(binary_search(arr, 38, 0, len(arr)-1))  # → 5
```

Merge Sort (Recursive)

```
# Merge Sort — O(n log n)
def merge_sort(arr):
    if len(arr) <= 1:          # Base case
        return arr
    mid = len(arr) // 2
    left  = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i]); i += 1
        else:
            result.append(right[j]); j += 1
    return result + left[i:] + right[j:]

# Recurrence: T(n) = 2T(n/2) + O(n) → O(n log n) by Master Theorem
```

3.3 Memoisation and Dynamic Programming

Naive recursive Fibonacci is exponentially slow: fib(n) has $O(2^n)$ time complexity because the same sub-problems are computed repeatedly (fib(3) is computed multiple times in fib(5)). Two related techniques solve this: Memoisation (top-down DP) caches computed results in a dictionary so each sub-problem is solved once. Tabulation (bottom-up DP) builds a table from the smallest sub-problems upward, filling in the solution iteratively.

```
# Naive Fibonacci — O(2^n)
def fib_naive(n):
    if n <= 1: return n
    return fib_naive(n-1) + fib_naive(n-2)

# Memoised Fibonacci — O(n)
def fib_memo(n, memo={}):
    if n in memo: return memo[n]   # Cache hit O(1)
    if n <= 1: return n
    memo[n] = fib_memo(n-1, memo) + fib_memo(n-2, memo)
    return memo[n]

# Bottom-up DP Fibonacci — O(n) time, O(n) space
def fib_dp(n):
    if n <= 1: return n
    dp = [0] * (n + 1)
    dp[1] = 1
```

```
    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]

# Optimised: O(1) space
def fib_opt(n):
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a

# Coin Change — DP
# Minimum coins to make amount using given denominations
def coin_change(coins, amount):
    dp = [float("inf")] * (amount + 1)
    dp[0] = 0
    for a in range(1, amount + 1):
        for coin in coins:
            if coin <= a:
                dp[a] = min(dp[a], dp[a - coin] + 1)
    return dp[amount] if dp[amount] != float("inf") else -1
```

In-text Question



Explain why naive recursive Fibonacci is $O(2^n)$. How does memoisation reduce this to O(n)?

Intext Answer

Naive recursive Fibonacci computes fib(n) = fib(n-1) + fib(n-2). Each call branches into two recursive calls, producing a binary tree of calls. The total number of nodes in this tree is $O(2^n)$ — for fib(50), approximately $2^{50} \approx 10^{15}$ function calls. The fundamental problem is overlapping sub-problems: fib(n-2) is computed once by the left branch of fib(n) and again by the right branch, and this duplication cascades exponentially down the tree. Memoisation solves this by storing the result of each fib(k) in a dictionary (cache) the first time it is computed. On any subsequent call to fib(k), the cached result is returned immediately in O(1) rather than recomputing. Since there are only n distinct sub-problems (fib(0), fib(1), ..., fib(n)), each computed once, the total time is O(n) — a dramatic improvement from exponential to linear. The space complexity is also O(n) for the cache and call stack. Bottom-up DP further reduces space to O(1) by only keeping track of the two most recent values.

4.0 Study Unit Conclusion

In this unit you have mastered recursion — the base case, recursive case, and call stack; recursive implementations of factorial, Tower of Hanoi, binary search, and merge sort; and the transformation of exponentially slow naive recursion into polynomial-time algorithms using memoisation (top-down DP) and tabulation (bottom-up DP).

5.0 Summary

Recursion: base case (stops recursion), recursive case (calls self with smaller input), call stack (stores frames — limited depth). Classic recursive algorithms: factorial (O(n)), binary search $T(n)=T(n/2)+O(1)\rightarrow O(\log n)$, merge sort $T(n)=2T(n/2)+O(n)\rightarrow O(n \log n)$, Tower of Hanoi $O(2^n)$. Naive Fibonacci: $O(2^n)$ due to overlapping sub-problems. Memoisation: cache results, O(n) time+space. Bottom-up DP: tabulate from base, O(n) time, O(1) space (optimised). Coin change: canonical DP problem — dp[a] = min(dp[a], dp[a-coin]+1).

6.0 Self-Assessment Questions (SAQs)

1. Trace the recursive calls for factorial(5), showing the call stack at each step and the values returned during the unwind phase.
2. Implement the Tower of Hanoi recursively for n=3 disks. Count the number of moves and prove that the minimum number of moves is 2^n-1.
3. Implement the Longest Common Subsequence (LCS) problem using both memoisation and bottom-up DP. What is the time and space complexity of each approach?
4. The coin change problem requires finding the minimum number of coins to make amount=11 with coins [1,5,6,9]. Trace through the DP table and show how the optimal solution is built from sub-problems.

7.0 References/ Further Reading

Cormen et al. (2022). Introduction to Algorithms (4th ed.). MIT Press.

Bhargava, A.Y. (2016). Grokking Algorithms. Manning Publications.

Skiena, S.S. (2020). The Algorithm Design Manual (3rd ed.). Springer.

LeetCode. (2023). Dynamic Programming Problems. https://leetcode.com/tag/dynamic-programming/

SAQs Answers

SAQ Answer 1

factorial(5) calls:

text

factorial(5) → 5 * factorial(4)

→ 4 * factorial(3)

→ 3 * factorial(2)

→ 2 * factorial(1)

→ 1 (base case)

Unwind: 2×1=2, 3×2=6, 4×6=24, 5×24=120

SAQ Answer 2

python

```
def hanoi(n, source, target, aux):
    if n == 1:
        print(f"Move 1 from {source} to {target}")
        return
    hanoi(n-1, source, aux, target)
    print(f"Move {n} from {source} to {target}")
    hanoi(n-1, aux, target, source)
```

For n=3: 7 moves. Proof: $T(n) = 2T(n-1)+1$, $T(1)=1 \rightarrow T(n)=2^n-1$.

SAQ Answer 3

python

```
def lcs_memo(s1, s2, i, j, memo):
    if i == 0 or j == 0: return 0
    if (i,j) in memo: return memo[(i,j)]
    if s1[i-1] == s2[j-1]:
        memo[(i,j)] = 1 + lcs_memo(s1,s2,i-1,j-1,memo)
    else:
```

```
    memo[(i,j)] = max(lcs_memo(s1,s2,i-1,j,memo),

                      lcs_memo(s1,s2,i,j-1,memo))

  return memo[(i,j)]



def lcs_dp(s1, s2):

  m, n = len(s1), len(s2)

  dp = [[0]*(n+1) for _ in range(m+1)]

  for i in range(1,m+1):

    for j in range(1,n+1):

      if s1[i-1] == s2[j-1]:

        dp[i][j] = 1 + dp[i-1][j-1]

      else:

        dp[i][j] = max(dp[i-1][j], dp[i][j-1])

  return dp[m][n]
```

Both O(m×n) time. Memo: O(m×n) space + recursion stack; DP: O(m×n) space (can optimise to O(min(m,n))).

SAQ Answer 4

DP table for amount=11, coins=[1,5,6,9]:

dp[0]=0, dp[1]=1, dp[2]=2, dp[3]=3, dp[4]=4, dp[5]=1, dp[6]=1, dp[7]=2, dp[8]=2, dp[9]=1, dp[10]=2 (5+5), dp[11]=2 (5+6).

Optimal: 5+6 = 2 coins.

AL-MUHIBBAH OPEN UNIVERSITY

Module 3

Study Unit 1: Comparison-Based and Non-Comparison Sorting

1.0 INTRODUCTION

Sorting — rearranging a collection of elements into a specified order — is one of the most fundamental and well-studied problems in computer science. Sorting is ubiquitous: database query optimisation, efficient search (binary search requires sorted data), data deduplication, and network routing all depend on sorted structures. Understanding sorting algorithms develops the analytical skills to reason about all algorithms: correctness, loop invariants, worst-case vs average-case behaviour, and the impact of input characteristics on performance.

This unit examines the full landscape of sorting algorithms from elementary $O(n^2)$ algorithms through efficient $O(n \log n)$ comparison-based sorts to linear-time non-

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
- 3.1 Elementary Sorting Algorithms
- 3.2 Efficient Sorting — Merge Sort and Quick Sort
- 3.3 Non-Comparison Sorting and the Sorting Lower Bound

4.0 Conclusion
5.0 Summary
6.0 Self-Assessment Questions (SAQs)
7.0 References/Further Reading

comparison sorts, establishing the theoretical lower bound for comparison-based sorting.

2.0 Learning Outcomes

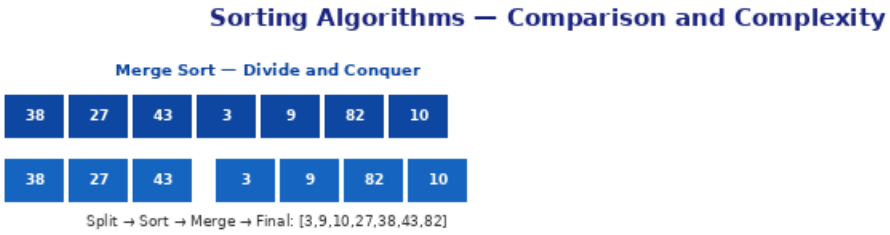
At the end of this unit, you should be able to:

- Implement and trace bubble sort, selection sort, and insertion sort
- Explain why o(n log n) is the theoretical lower bound for comparison-based sorting
- Implement merge sort and quick sort and analyse their time complexities
- Explain counting sort, radix sort, and bucket sort
- Select an appropriate sorting algorithm for a given application context

.

3.0 Main Content

3.1 Elementary Sorting Algorithms



Algorithm	Best	Average	Worst	Space	Stable?
Bubble Sort	O(n)	O(n²)	O(n²)	O(1)	Yes
Selection Sort	O(n²)	O(n²)	O(n²)	O(1)	No
Insertion Sort	O(n)	O(n²)	O(n²)	O(1)	Yes
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes
Quick Sort	O(n log n)	O(n log n)	O(n²)	O(log n)	No
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	No
Counting Sort	O(n+k)	O(n+k)	O(n+k)	O(k)	Yes
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)	Yes

Figure 3.1: Sorting Algorithms — Divide-and-Conquer (Merge Sort) and Complete Complexity Comparison Table

Bubble Sort

Bubble sort repeatedly scans the array, comparing adjacent pairs and swapping if out of order. The largest unsorted element "bubbles" to its correct position at the end after each pass. Requires n-1 passes for n elements. Worst case: $O(n^2)$ comparisons and swaps (reverse-sorted input). Best case: O(n) comparisons, O(1) swaps (already sorted, with early termination optimisation). Space: O(1) in-place. Bubble sort is rarely used in practice due to $O(n^2)$ performance, but it is instructive for understanding loop invariants.

```
# Bubble Sort — O(n²) worst/average, O(n) best (optimised)
def bubble_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        swapped = False
        for j in range(n - 1 - i):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped:   # Early termination if already sorted
            break
    return arr

# Selection Sort — O(n²) all cases, O(n) swaps
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

```
# Insertion Sort — O(n²) worst, O(n) best; excellent for small/nearly-sorted
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
```

3.2 Efficient Sorting — Merge Sort and Quick Sort

Efficient comparison-based sorting algorithms achieve O(n log n) time — the theoretical minimum for comparison-based sorting (proven via the decision tree lower bound: any algorithm that sorts by comparisons must make at least $\log_2(n!) \approx n \log n$ comparisons in the worst case).

```
# Quick Sort — O(n log n) average, O(n²) worst
def quick_sort(arr, low=0, high=None):
    if high is None: high = len(arr) - 1
    if low < high:
        pi = partition(arr, low, high)
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)
```

```
def partition(arr, low, high):
    pivot = arr[high]   # Lomuto partition scheme
    i = low - 1
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

# Quick Sort analysis:
# Average: T(n) = 2T(n/2) + O(n) → O(n log n)
# Worst:   T(n) = T(n-1)  + O(n) → O(n²) — always picks min/max as pivot
# Fix: randomise pivot selection to make worst case very unlikely
# In-place: O(log n) stack space (recursion depth)

# Comparison: Merge Sort vs Quick Sort
# Merge Sort: stable, guaranteed O(n log n), O(n) extra space
# Quick Sort: in-place, O(n log n) average, cache-friendly, faster in practice
```

3.3 Non-Comparison Sorting and the Sorting Lower Bound

The Ω(n log n) lower bound applies only to comparison-based sorting. When additional information about the data is available (bounded integer range, fixed-length keys), non-comparison algorithms can sort in O(n) — linear time.

- Counting Sort: Counts occurrences of each value (requires integer keys in range [0, k]). Time: O(n+k). Space: O(k). Stable. Excellent for small integer ranges (e.g., sort exam scores 0-100).
- Radix Sort: Sorts integers digit by digit (least significant to most significant), using counting sort as a subroutine for each digit. Time: O(nk) where k is the number of digits. Effective for fixed-length integer keys or strings. Used in DNA sequence sorting and network packet processing.
- Bucket Sort: Distributes elements into buckets based on value range, sorts each bucket independently (using insertion sort), then concatenates. O(n) average for uniformly distributed real-valued data. Used for sorting floating-point numbers in [0,1).

```
# Counting Sort — O(n + k)
def counting_sort(arr, k):
    count = [0] * (k + 1)
    for val in arr:
        count[val] += 1
    # Prefix sum for stable sort
    for i in range(1, k + 1):
        count[i] += count[i - 1]
    output = [0] * len(arr)
```

```
    for val in reversed(arr):  # Reversed for stability
        output[count[val] - 1] = val
        count[val] -= 1
    return output

# When to use each:
# Insertion sort: n < 20 or nearly sorted
# Merge sort: stable sort required, or external sort
# Quick sort: general purpose in-memory, cache-friendly
# Heap sort: in-place O(n log n) guaranteed, no recursion
# Counting/Radix: integer keys with bounded range
# Python sorted()/list.sort() uses Timsort — O(n log n) worst, O(n) best
```

AL-MUHIBBAH OPEN UNIVERSITY

In-text Question



Explain why O(n log n) is the lower bound for comparison-based sorting using the decision tree argument.

Intext Answer

Any comparison-based sorting algorithm can be modelled as a binary decision tree where each internal node represents a comparison (is arr[i] ≤ arr[j]?) and each leaf represents a possible sorted permutation. For n elements, there are n! possible permutations — the tree must have at least n! leaves. A binary tree with L leaves has height at least ⌈log₂ L⌉. Therefore the minimum height (worst-case number of comparisons) is ⌈log₂(n!)⌉. Using Stirling's approximation: log₂(n!) = Σ log₂(k) from k=1 to n ≈ n log₂ n - n log₂ e ≈ Θ(n log n). This means any comparison-based sorting algorithm requires at least Ω(n log n) comparisons in the worst case — regardless of implementation cleverness. This is why merge sort and heap sort with O(n log n) worst-case performance are optimal among comparison-based algorithms. Non-comparison algorithms (counting sort, radix sort) bypass this lower bound by exploiting properties of the keys beyond pairwise comparisons.

4.0 Study Unit Conclusion

In this unit you have implemented and analysed the full spectrum of sorting algorithms: bubble sort, selection sort, insertion sort (all $O(n^2)$), merge sort (O(n log n) stable, O(n) space), quick sort (O(n log n) average, in-place), and linear-time non-comparison sorts (counting, radix, bucket). The Ω(n log n) decision tree lower bound establishes the theoretical minimum for comparison-based sorting.

5.0 Summary

Elementary sorts: Bubble $O(n^2)/O(n)$ best, Selection $O(n^2)$ always, Insertion $O(n^2)/O(n)$ best — use for small arrays. O(n log n) barrier: comparison-based sorting requires $\Omega(n \log n)$ by decision tree argument. Merge sort: stable, guaranteed O(n log n), O(n) space — use when stability required. Quick sort: in-place, O(n log n) average ($O(n^2)$ worst — randomise pivot), fastest in practice. Heap sort: in-place, O(n log n) guaranteed. Non-comparison: Counting sort O(n+k) for integers in [0,k]; Radix sort O(nk) for fixed-length keys; Bucket sort O(n) for uniform distribution. Python uses Timsort (adaptive merge+insertion sort).

6.0 Self-Assessment Questions (SAQs)

1. Trace bubble sort on [64, 34, 25, 12, 22, 11, 90], showing the array state after each pass. Count comparisons and swaps.
2. Explain the difference between stable and unstable sorting. Give an example where sort stability is critically important.
3. Implement quick sort with random pivot selection. Explain why randomisation reduces the probability of $O(n^2)$ worst-case behaviour.
4. Apply counting sort to the array [4,2,2,8,3,3,1] with k=8. Show the count array, cumulative count array, and output array at each step.

7.0 References/ Further Reading

Cormen et al. (2022). Introduction to Algorithms (4th ed.). MIT Press.

Sedgewick, R. & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.

Skiena, S.S. (2020). The Algorithm Design Manual (3rd ed.). Springer.

Princeton University. (2023). Coursera Algorithms Part I — Sorting. https://www.coursera.org/learn/algorithms-part1

SAQs Answers

SAQ Answer 1

Array: [64, 34, 25, 12, 22, 11, 90]

Pass 1: [34,25,12,22,11,64,90] - 6 comparisons, 5 swaps

Pass 2: [25,12,22,11,34,64,90] - 5 comparisons, 4 swaps

Pass 3: [12,22,11,25,34,64,90] - 4 comparisons, 3 swaps

Pass 4: [12,11,22,25,34,64,90] - 3 comparisons, 2 swaps

Pass 5: [11,12,22,25,34,64,90] - 2 comparisons, 1 swap

Total: 21 comparisons, 15 swaps

SAQ Answer 2

Stable sort preserves relative order of equal elements. Example: Sorting employees by department, then by name - stable sort keeps alphabetical order within each department. Merge sort is stable; quick sort is typically unstable.

SAQ Answer 3

python

```
import random

def quick_sort(arr, low=0, high=None):
    if high is None: high = len(arr)-1
    if low < high:
        pivot_idx = random.randint(low, high)
        arr[pivot_idx], arr[high] = arr[high], arr[pivot_idx]
        pi = partition(arr, low, high)
        quick_sort(arr, low, pi-1)
        quick_sort(arr, pi+1, high)
```

Randomisation makes worst-case (sorted input always choosing min/max pivot) extremely unlikely - probability of $O(n^2)$ is negligible.

SAQ Answer 4

Array: [4,2,2,8,3,3,1], k=8

Count: [0,1,2,2,1,0,0,0,1]

Cumulative: [0,1,3,5,6,6,6,6,7]

Output: [1,2,2,3,3,4,8]

Module 4

Study Unit 1: Search Techniques — From Linear to Hashing

1.0 INTRODUCTION

Searching — finding a specific element in a collection — is arguably the most frequently performed computational task in software systems. A web search queries billions of documents in milliseconds; a database finds a customer record among millions in sub-milliseconds; a navigation app finds the shortest path in a graph of millions of road segments in seconds. The efficiency of these operations depends entirely on the search algorithm and the data structure in which the data is stored.

This unit examines the spectrum of searching algorithms from naive linear search through binary search and hash tables to graph traversal algorithms (BFS and DFS) that underpin the navigation,

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
- 3.1 Linear and Binary Search
- 3.2 Hash Table Search
- 3.3 Graph-Based Search — BFS and DFS

4.0 Conclusion
5.0 Summary
6.0 Self-Assessment Questions (SAQs)
7.0 References/Further Reading

social networking, and recommendation systems that power modern digital life.

2.0 Learning Outcomes

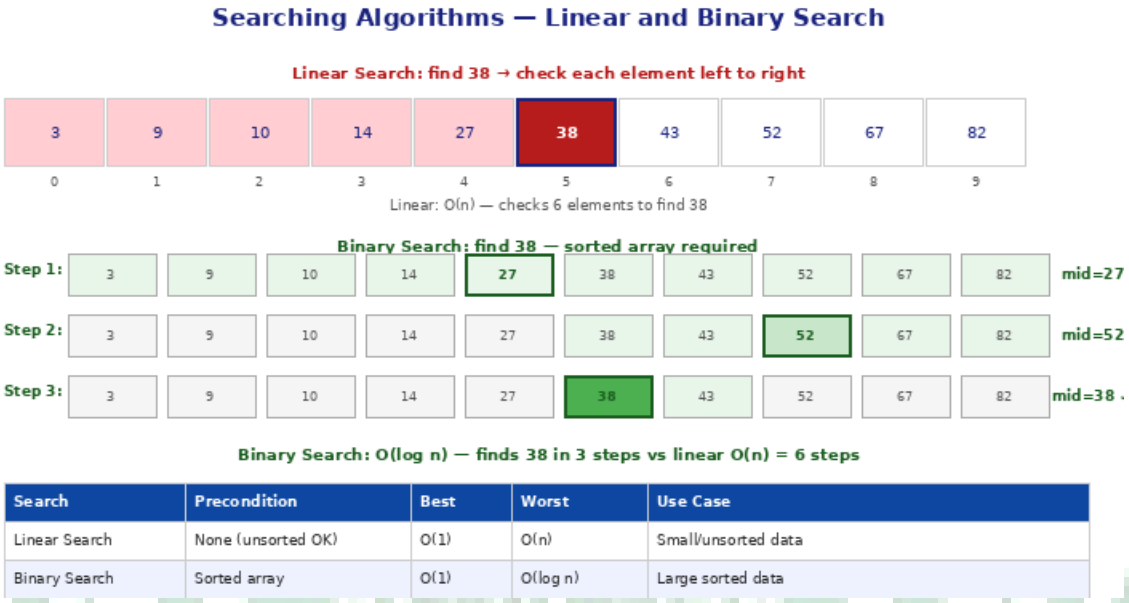
At the end of this unit, you should be able to:

- Implement linear search and binary search and analyse their time complexities
- Explain interpolation search and its advantage over binary search for uniformly distributed data
- Implement hash table search with chaining and open addressing collision resolution
- Implement bfs and dfs for graph traversal
- Apply dijkstra's algorithm to find shortest paths in weighted graphs

.

3.0 Main Content

3.1 Linear and Binary Search



Search	Precondition	Best	Worst	Use Case
Linear Search	None (unsorted OK)	O(1)	O(n)	Small/unsorted data
Binary Search	Sorted array	O(1)	O(log n)	Large sorted data

Figure 4.1: Linear Search vs Binary Search — Step-by-Step Visual Comparison and Algorithm Table

Linear search (sequential search) examines each element in sequence until the target is found or the collection is exhausted. Time: O(n) worst and average, O(1) best (target at first position). Space: O(1). Works on any collection regardless of order. Optimal for small collections and unsorted data.

Binary search exploits the sorted order of an array to eliminate half the remaining candidates at each step. It compares the target with the middle element: if equal, found; if target < middle, search the left half; if target > middle, search the right half. Time: O(log n) — for n=1,000,000, binary search requires at most 20 comparisons; linear search may require 1,000,000. Precondition: array must be sorted. Variants: lower_bound (first position ≥ target), upper_bound (first position > target), used extensively in interval and scheduling problems.

```
# Binary Search — O(log n)
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = low + (high - low) // 2  # Avoids integer overflow
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1

# Lower bound: first index where arr[i] >= target
def lower_bound(arr, target):
    low, high = 0, len(arr)
    while low < high:
        mid = (low + high) // 2
        if arr[mid] < target: low = mid + 1
        else: high = mid
    return low

# Interpolation Search — O(log log n) for uniform data
# Estimates position: pos = low + ((target-arr[low])/(arr[high]-arr[low])) × (high-
```

```
low)
# Better than binary for uniformly distributed sorted arrays
```

3.2 Hash Table Search

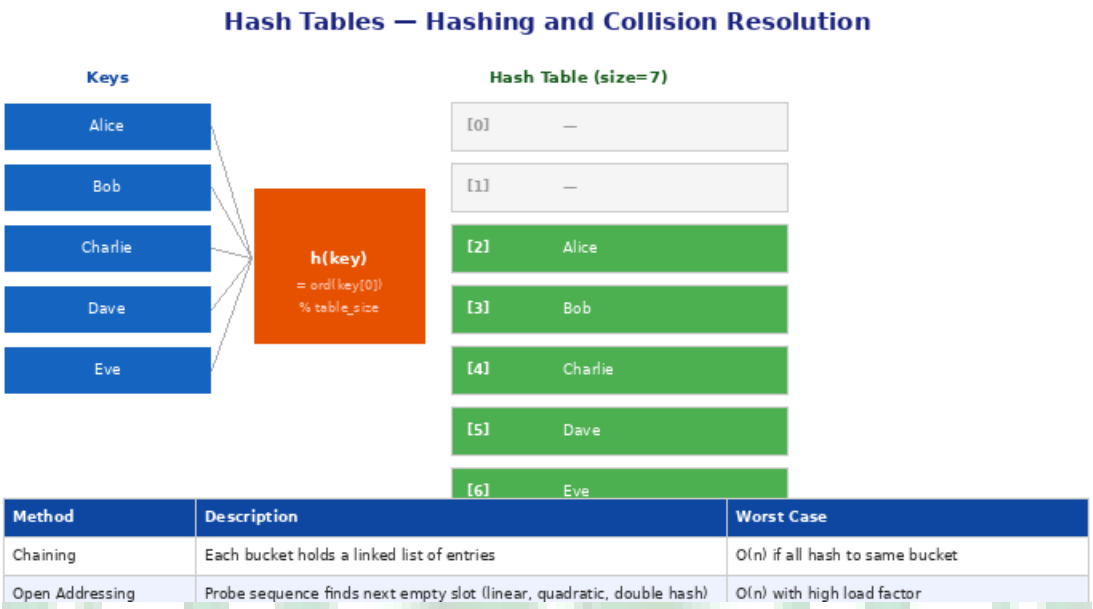


Figure 4.2: Hash Table — Hash Function, Key-to-Bucket Mapping, and Collision Resolution

A hash table achieves O(1) average-case search, insert, and delete by using a hash function to map keys directly to array indices. Given a key, hash(key) % table_size gives the storage index. Python's dict and set, Java's HashMap, and C++'s unordered_map all implement hash tables.

Collisions occur when two different keys hash to the same index. Two primary resolution strategies: Chaining stores colliding elements in a linked list at the bucket — worst case O(n) if many keys hash to the same bucket. Open addressing probes for the next available slot: linear probing (next slot), quadratic probing, or double hashing. Performance degrades as the load factor (α = n/m, number of elements/table size)

approaches 1 — tables are typically resized when $\alpha > 0.75$.

```
# Python dict is a hash table — O(1) average for all operations
phone_book = {}

# Insert: O(1) average
phone_book["Alice"] = "0812-345-6789"
phone_book["Bob"]   = "0903-456-7890"

# Search: O(1) average
if "Alice" in phone_book:
    print(phone_book["Alice"])  # "0812-345-6789"

# Delete: O(1) average
del phone_book["Bob"]

# Hash function example (for illustration)
def simple_hash(key, size):
    return sum(ord(c) for c in key) % size

# Separate chaining hash table
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.buckets = [[] for _ in range(size)]
```

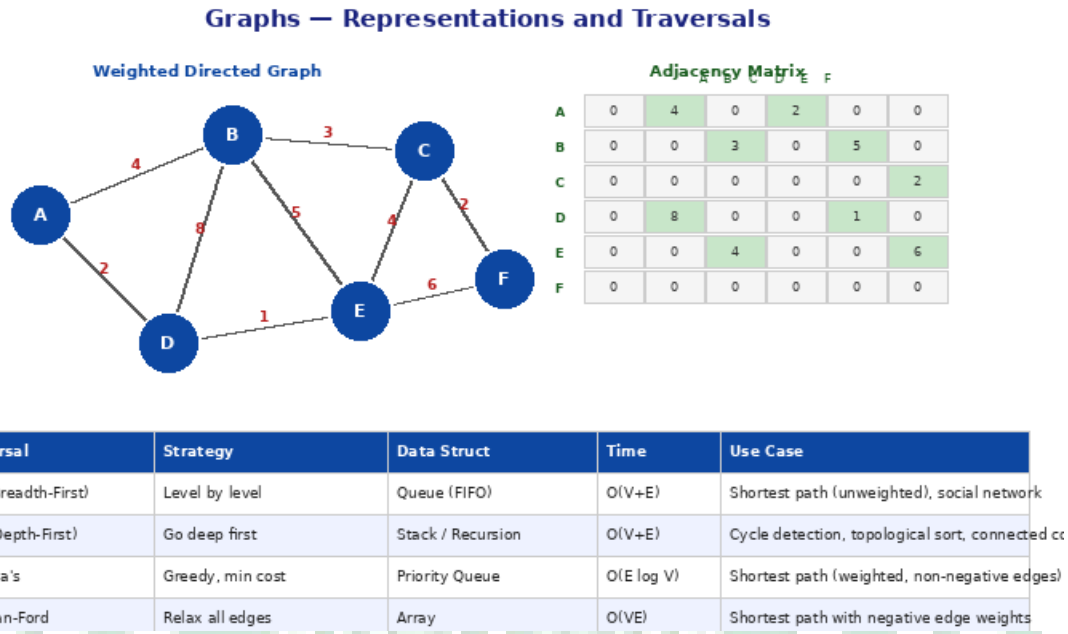
```
def _hash(self, key):
    return hash(key) % self.size

def insert(self, key, value):      # O(1) average
    idx = self._hash(key)
    for i, (k, v) in enumerate(self.buckets[idx]):
        if k == key:
            self.buckets[idx][i] = (key, value); return
    self.buckets[idx].append((key, value))

def search(self, key):             # O(1) average
    idx = self._hash(key)
    for k, v in self.buckets[idx]:
        if k == key: return v
    return None
```

AL-MUHIBBAH OPEN UNIVERSITY

3.3 Graph-Based Search — BFS and DFS



Traversal	Strategy	Data Struct	Time	Use Case
BFS (Breadth-First)	Level by level	Queue (FIFO)	O(V+E)	Shortest path (unweighted), social network
DFS (Depth-First)	Go deep first	Stack / Recursion	O(V+E)	Cycle detection, topological sort, connected cc
Dijkstra's	Greedy, min cost	Priority Queue	O(E log V)	Shortest path (weighted, non-negative edges)
Bellman-Ford	Relax all edges	Array	O(VE)	Shortest path with negative edge weights

Figure 4.3: Graph — Weighted Directed Graph, Adjacency Matrix, and BFS/DFS/Dijkstra Comparison

Graphs model networks of relationships — road networks, social networks, dependency graphs, web link structures, and communication networks. Graph search algorithms explore vertices and edges to answer questions about connectivity, reachability, and optimal paths.

```
# BFS — O(V+E), finds shortest path (unweighted)
from collections import deque

def bfs_shortest_path(graph, start, target):
    if start == target: return [start]
    visited = {start}
    queue = deque([(start, [start])])
```

```
    while queue:
        node, path = queue.popleft()
        for neighbour in graph[node]:
            if neighbour not in visited:
                new_path = path + [neighbour]
                if neighbour == target: return new_path
                visited.add(neighbour)
                queue.append((neighbour, new_path))
    return None  # No path found

# DFS — O(V+E), used for topological sort, cycle detection, SCC
def dfs(graph, start, visited=None):
    if visited is None: visited = set()
    visited.add(start)
    print(start, end=" ")
    for neighbour in graph[start]:
        if neighbour not in visited:
            dfs(graph, neighbour, visited)

# Dijkstra — O((V+E) log V) with priority queue
import heapq

def dijkstra(graph, start):
    dist = {node: float("inf") for node in graph}
```

```
    dist[start] = 0
    heap = [(0, start)]       # (distance, node)
    while heap:
        d, u = heapq.heappop(heap)
        if d > dist[u]: continue
        for v, weight in graph[u]:
            if dist[u] + weight < dist[v]:
                dist[v] = dist[u] + weight
                heapq.heappush(heap, (dist[v], v))
    return dist
```

AL-MUHIBBAH OPEN UNIVERSITY

In-text Question



Explain the difference between BFS and DFS for graph traversal. For each algorithm, give one problem it solves optimally and explain why the other algorithm would be less suitable.

Intext Answer

BFS (Breadth-First Search) explores a graph level by level using a queue — it visits all vertices at distance 1, then distance 2, and so on. DFS (Depth-First Search) explores as deep as possible along each branch before backtracking, using a stack (explicit or recursive). BFS optimal problem — Shortest path in an unweighted graph: BFS visits vertices in order of their distance from the source. The first time BFS reaches the target, it has used the minimum number of edges. DFS would not be suitable because DFS may find a path immediately but it is not guaranteed to be the shortest — DFS might traverse a longer path before discovering a shorter one. DFS optimal problem — Detecting cycles and topological sorting: DFS naturally tracks which vertices are currently "in progress" (on the current path) using a "visited" and "in-stack" mechanism. A back edge (edge to a currently in-stack vertex) indicates a cycle. DFS produces a topological ordering by recording finish times (vertices that finish later appear earlier in topological order). BFS would be less natural for cycle detection and does not directly produce topological order.

4.0 Study Unit Conclusion

In this unit you have examined the full spectrum of searching algorithms: linear search O(n), binary search O(log n) on sorted arrays, hash table search O(1) average using chaining or open addressing, BFS O(V+E) for unweighted shortest paths, DFS O(V+E) for cycle detection and topological sort, and Dijkstra's algorithm O((V+E) log V) for weighted shortest paths.

5.0 Summary

Linear search: O(n) — no precondition. Binary search: O(log n) — requires sorted array; lower_bound/upper_bound variants. Interpolation search: O(log log n) for uniform data. Hash tables: O(1) average insert/search/delete; collision resolution via chaining (linked lists) or open addressing (probing); resize at $\alpha > 0.75$. Python dict is a hash table. BFS: queue-based, level-order, O(V+E), finds shortest path in unweighted graphs. DFS: stack/recursive, depth-first, O(V+E), cycle detection, topological sort, connected components. Dijkstra: priority queue (min-heap), O((V+E) log V), shortest path with non-negative edge weights.

6.0 Self-Assessment Questions (SAQs)

1. Implement binary search iteratively and recursively. Compare their space complexity and explain when you would prefer each.
2. Implement a hash table using open addressing with linear probing. Demonstrate insert, search, and delete on keys that cause collisions.
3. Given the graph: A-B(4), A-D(2), B-C(3), D-E(1), E-C(4), B-E(5), C-F(2), apply Dijkstra's algorithm from source A. Show the priority queue state and distance table at each step.

4. Explain topological sorting. Implement DFS-based topological sort for a directed acyclic graph representing course prerequisites at a university.

7.0 References/ Further Reading

Cormen et al. (2022). Introduction to Algorithms (4th ed.). MIT Press.

Skiena, S.S. (2020). The Algorithm Design Manual (3rd ed.). Springer.

Sedgewick, R. & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.

Dijkstra, E.W. (1959). A Note on Two Problems in Connexion with Graphs. Numerische Mathematik, 1(1), 269-271.

SAQs Answers

Study Unit 1: Search Techniques - From Linear to Hashing

SAQ Answer 1

Iterative binary search: O(1) space, no recursion overhead
Recursive binary search: O(log n) space (call stack), simpler code
Prefer iterative for large arrays (avoid stack overflow), recursive for clarity in small/medium cases.

SAQ Answer 2

python

```
class HashTableOA:

  def __init__(self, size=10):
```

```
    self.size = size
    self.keys = [None] * size
    self.values = [None] * size
    self.DELETED = object()


def hash(self, key):
    return hash(key) % self.size


def insert(self, key, value):
    idx = self.hash(key)
    while self.keys[idx] not in (None, self.DELETED):
        if self.keys[idx] == key:
            break
        idx = (idx + 1) % self.size
    self.keys[idx] = key
    self.values[idx] = value
```

```
def search(self, key):
    idx = self.hash(key)
    while self.keys[idx] is not None:
        if self.keys[idx] == key:
            return self.values[idx]
        idx = (idx + 1) % self.size
    return None
```

SAQ Answer 3

Graph: A-B(4), A-D(2), B-C(3), D-E(1), E-C(4), B-E(5), C-F(2)

Dijkstra from A:

Initialize: dist[A]=0, others=∞; heap=[(0,A)]

Pop A: update B=4, D=2 → heap=[(2,D),(4,B)]

Pop D: update E=3 → heap=[(3,E),(4,B)]

Pop E: update C=7 → heap=[(4,B),(7,C)]

Pop B: update C=6, F=11 → heap=[(6,C),(7,C),(11,F)]

Pop C: update F=8 → heap=[(8,F)]

Final distances: A=0, B=4, C=6, D=2, E=3, F=8

SAQ Answer 4

Topological sort: ordering vertices such that for every edge u→v, u appears before v.

python

```
def topological_sort(graph):
    visited = set()
    result = []

    def dfs(node):
        visited.add(node)
        for neighbour in graph.get(node, []):
            if neighbour not in visited:
                dfs(neighbour)
        result.append(node)

    for node in graph:
        if node not in visited:
            dfs(node)
    return result[::-1]  # Reverse for topological order
```

Module 5

Study Unit 1: Algorithm Analysis and Complexity

1.0 INTRODUCTION

Algorithm efficiency is the formal study of how resource consumption — time and memory — scales with input size. An algorithm that works correctly but requires 10^{12} operations for input $n=10^6$ is practically useless regardless of hardware. Conversely, an algorithm requiring only n log n operations for the same input size is fast enough to process the entire internet. Asymptotic analysis — expressing algorithmic complexity using Big-O, Omega, and Theta notation — provides a hardware-independent framework for comparing algorithms.

This unit establishes the formal machinery of complexity analysis: Big-O notation, the rules for computing algorithm complexity from code, recurrence relations and the

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
- 3.1 Big-O Notation and Asymptotic Analysis
- 3.2 Recurrence Relations and the Master Theorem
- 3.3 Algorithm Design Paradigms

4.0 Conclusion
5.0 Summary
6.0 Self-Assessment Questions (SAQs)
7.0 References/Further Reading

Master Theorem for divide-and-conquer algorithms, space complexity, and the major algorithm design paradigms (brute force, divide and conquer, dynamic programming, greedy). testing engagement. It establishes the conceptual and professional.

2.0 Learning Outcomes

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

- Explain and apply Big-O, Ω (Omega), and Θ (Theta) notation
- Compute the time complexity of algorithms by counting operations
- Solve recurrence relations using substitution and the Master Theorem
- Explain the P vs NP distinction and NP-hard problems
- Apply greedy, divide-and-conquer, and dynamic programming design paradigms.

3.0 Main Content

3.1 Big-O Notation and Asymptotic Analysis

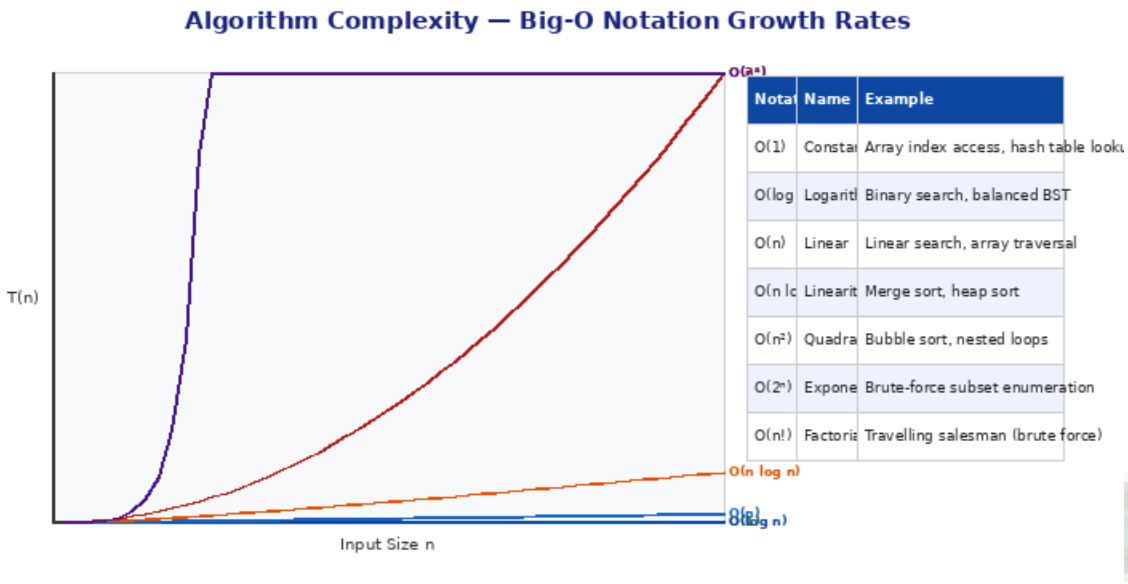


Figure 5.1: Big-O Complexity Growth Rates — O(1) through O(2ⁿ) Plotted Against Input Size n

Big-O notation describes an upper bound on the growth rate of an algorithm's resource consumption as input size n grows toward infinity. f(n) = O(g(n)) means there exist constants $c > 0$ and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$ — the algorithm's cost grows no faster than g(n) up to a constant factor.

Omega (Ω) notation provides a lower bound: f(n) = Ω(g(n)) means f grows at least as fast as g. Theta (Θ) notation provides a tight bound: f(n) = Θ(g(n)) means f grows at the same rate as g (both upper and lower bounds match). In practice, "Big-O" is commonly used loosely to mean the tight asymptotic bound.

Rules for Computing Complexity

- Constants are dropped: $3n + 7 = O(n)$. Constant factors do not affect asymptotic complexity.
- Lower-order terms are dropped: $n^2 + 3n + 10 = O(n^2)$. For large n, the dominant term overwhelms all others.
- Sequential operations (not nested): $O(f) + O(g) = O(\max(f, g))$. Two sequential O(n) loops is still O(n).
- Nested operations: $O(f) \times O(g) = O(f \times g)$. A loop running n times inside a loop running n times is $O(n^2)$.

```
# Complexity from code examples

# O(1) — no iteration, fixed operations
def get_first(arr): return arr[0]

# O(n) — single loop over all elements
def find_max(arr):
    max_val = arr[0]
    for x in arr:           # n iterations
        if x > max_val: max_val = x
    return max_val

# O(n²) — nested loops, each running n times
def has_duplicate(arr):
```

```
    for i in range(len(arr)):          # n iterations
        for j in range(i+1, len(arr)): # up to n-1 iterations
            if arr[i] == arr[j]: return True
    return False

# O(n log n) — outer loop n, inner halves (binary search inside loop)
# O(log n) — problem halved each iteration (binary search)
# O(2^n) — recursive branching with no memoisation (naive Fibonacci)
```

3.2 Recurrence Relations and the Master Theorem

Recursive algorithms express their running time as a recurrence relation — a function defined in terms of itself for smaller inputs. Solving recurrences yields the closed-form time complexity.

The Master Theorem provides a general solution for recurrences of the form $T(n) = aT(n/b) + f(n)$, where $a \geq 1$ recursive calls each on a sub-problem of size n/b, plus $f(n)$ work at the current level:

- Case 1: If $f(n) = O(n^{(\log_b a - \varepsilon)})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$. (Sub-problems dominate — recursive work dominates.)
- Case 2: If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \times \log n)$. (Balanced — each level does similar work.)
- Case 3: If $f(n) = \Omega(n^{(\log_b a + \varepsilon)})$, then $T(n) = \Theta(f(n))$. (Non-recursive work dominates.)

Algorithm	Recurrence	Master Theorem	Result
Binary Search	$T(n) = T(n/2) + O(1)$	$a=1, b=2, f=O(1)$ — Case 2	$O(\log n)$
Merge Sort	$T(n) = 2T(n/2) + O(n)$	$a=2, b=2, f=O(n)$ — Case 2	$O(n \log n)$
Quick Sort (avg)	$T(n) = 2T(n/2) + O(n)$	Same as merge sort	$O(n \log n)$
Strassen Matrix	$T(n) = 7T(n/2) + O(n^2)$	$a=7, b=2, \log_2 7 \approx 2.81$ — Case 1	$O(n^{2.81})$
Naive Fibonacci	$T(n) = 2T(n-1) + O(1)$	Not master — $T(n) = O(2^n)$	$O(2^n)$

3.3 Algorithm Design Paradigms

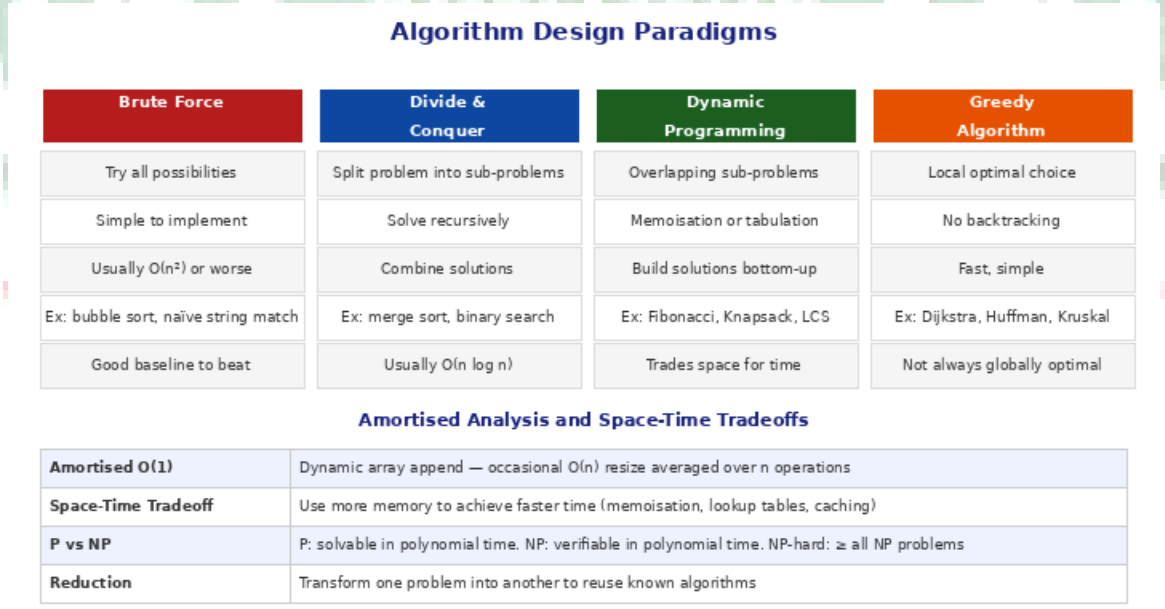


Figure 5.2: Algorithm Design Paradigms — Brute Force, Divide & Conquer, Dynamic Programming, and Greedy

The four major algorithm design paradigms represent strategies for decomposing and solving computational problems. Expert algorithm designers recognise which paradigm applies to a given problem — often the key insight that transforms an infeasible O(n!) solution into a practical O(n log n) one.

Greedy Algorithms

A greedy algorithm makes the locally optimal choice at each step, hoping this leads to a global optimum. Greedy algorithms are simple, fast, and correct for problems with the greedy choice property (a global optimum can be arrived at by making locally optimal choices) and optimal substructure (an optimal solution contains optimal solutions to sub-problems). Classic examples: Huffman coding (greedy builds optimal prefix code), Activity Selection Problem (select maximum non-overlapping activities), Kruskal's MST (always add the cheapest edge that doesn't form a cycle), Dijkstra's shortest path. When greedy fails: 0/1 Knapsack (requires dynamic programming), Travelling Salesman (NP-hard).

P vs NP

P (Polynomial time) is the class of decision problems solvable in polynomial time $O(n^k)$. NP (Non-deterministic Polynomial) is the class of decision problems whose solutions can be verified in polynomial time. $P \subseteq NP$. The P vs NP question — whether every problem whose solution can be verified quickly can also be solved quickly — is the most famous unsolved problem in computer science. NP-hard problems are at least as hard as the hardest NP problems; NP-complete problems are both NP-hard and in NP. Examples of NP-complete problems: Travelling Salesman Decision, 3-SAT, Graph Colouring, Subset Sum, Clique. For NP-hard problems in practice: use approximation algorithms, heuristics, or dynamic programming for small instances.

In-text Question



Apply the Master Theorem to determine the time complexity of merge sort T(n) = 2T(n/2) + n. Show your working.

Intext Answer

Merge sort satisfies the recurrence T(n) = 2T(n/2) + n. Identifying the parameters: a = 2 (two recursive calls), b = 2 (each call on a sub-problem of size n/2), f(n) = n = O(n¹) (the merge step). Computing the critical exponent: log_b(a) = log₂(2) = 1. Comparing f(n) = n = Θ(n¹) with n^(log_b a) = n¹: f(n) = Θ(n^(log_b a)), so we are in Case 2 of the Master Theorem. Case 2 conclusion: T(n) = Θ(n^(log_b a) × log n) = Θ(n¹ × log n) = Θ(n log n). Interpretation: at each level of the recursion tree, there are 2^i sub-problems each of size n/2^i, costing n/2^i × 2^i = n work. There are log₂ n levels. Total work = n × log₂ n = O(n log n). This also confirms by the recurrence that merge sort's O(n log n) performance holds for all inputs (unlike quick sort which degrades to O(n²) in the worst case), making merge sort the preferred algorithm when worst-case performance guarantees are required..

4.0 Study Unit Conclusion

In this unit you have mastered asymptotic analysis (Big-O, Omega, Theta notation), rules for computing complexity from code, recurrence relations and the Master Theorem for divide-and-conquer algorithms, and the four major algorithm design paradigms (brute force, divide and conquer, dynamic programming, greedy). The P vs NP problem establishes the boundary between tractable and intractable computational problems.

5.0 Summary

Big-O: upper bound — constants and lower terms dropped. Omega (Ω): lower bound. Theta (Θ): tight bound. Complexity ladder: O(1) < O(log n) < O(n) < O(n log n) < O(n^2) < O(2^n) < O(n!). Sequential code: add complexities; nested loops: multiply. Master Theorem: T(n)=aT(n/b)+f(n) — compare f(n) with n^(log_b a) → 3 cases. Merge sort: T(n)=2T(n/2)+n → Case 2 → O(n log n). Greedy: locally optimal choices, correct when greedy choice property holds. DP: overlapping sub-problems + optimal substructure. P vs NP: P solvable in poly time, NP verifiable in poly time — NP-complete problems (TSP, 3-SAT) require approximation or heuristics in practice.

6.0 Self-Assessment Questions (SAQs)

1. Determine the time complexity of the following code fragments and justify your answer: (a) two sequential loops each O(n); (b) a loop from 1 to n^2 with O(1) body; (c) a loop where i doubles each iteration (i=1,2,4,8,...,n).
2. Apply the Master Theorem to solve: (a) T(n) = 4T(n/2) + n; (b) T(n) = 2T(n/2) + n^2; (c) T(n) = 3T(n/3) + n log n.
3. Explain the greedy choice property and optimal substructure. For each, give an example of a problem that has both properties (and is therefore solvable greedily) and an example that has optimal substructure but not the greedy choice property (requiring DP).
4. What is P vs NP? Explain why the Travelling Salesman Problem is NP-hard and describe how you would approach solving it for a Nigerian logistics company with 50 delivery stops.

7.0 References/ Further Reading

Cormen et al. (2022). Introduction to Algorithms (4th ed.). MIT Press.

Sipser, M. (2013). Introduction to the Theory of Computation (3rd ed.). Cengage.

Skiena, S.S. (2020). The Algorithm Design Manual (3rd ed.). Springer.

Arora, S. & Barak, B. (2009). Computational Complexity: A Modern Approach. Cambridge.

SAQs Answers

SAQ Answer 1

(a) Two sequential O(n) loops → O(n) + O(n) = O(n)

(b) Loop from 1 to n^2 → $O(n^2)$

(c) i doubles: i=1,2,4,8,...,2^k=n → $k=\log_2 n$ iterations → O(log n)

SAQ Answer 2

(a) $T(n) = 4T(n/2) + n$: a=4,b=2, $\log_2 4=2$, $f(n)=O(n^1) < n^2$ → Case 1 → $T(n)=\Theta(n^2)$

(b) $T(n) = 2T(n/2) + n^2$: a=2,b=2, $\log_2 2=1$, $f(n)=\Omega(n^{(1+\varepsilon)})$ → Case 3 → $T(n)=\Theta(n^2)$

(c) $T(n) = 3T(n/3) + n \log n$: a=3,b=3, $\log_3 3=1$, $f(n)=n \log n > n^1$ but not polynomially larger → Master Theorem not directly applicable → $T(n)=\Theta(n \log^2 n)$

SAQ Answer 3

Greedy choice property: local optimal choices lead to global optimum.

Optimal substructure: optimal solution contains optimal sub-solutions.

Has both (greedy works): Activity Selection - choose earliest finishing activity.

Has optimal substructure but not greedy (needs DP): 0/1 Knapsack - optimal solution

contains optimal choice for remaining capacity, but greedy (take highest value/weight ratio) fails.

SAQ Answer 4

P vs NP: P = problems solvable in polynomial time; NP = problems verifiable in polynomial time. TSP is NP-hard because it's at least as hard as NP problems. For 50 stops: use approximation algorithms (nearest neighbour, Christofides), metaheuristics (genetic algorithms, simulated annealing), or dynamic programming for exact solution ($O(n^2 2^n)$ - 50! is impossible but 2^{50} possible).

Module 6

Study Unit 1: Trees — Binary Trees, BSTs, and Heaps

1.0 INTRODUCTION

Non-linear data structures represent data in which elements have a hierarchical or network relationship — not a simple sequence. Trees model organisational hierarchies, file systems, parsed language, and decision processes. Heaps power priority queues and the fastest sorting algorithms. Graphs model the relationships that underpin road networks, social media connections, dependency resolution, and the internet itself.

This unit examines trees in depth — binary tree structure and traversals, Binary Search Trees (BSTs) and their search efficiency, and heaps as array-based priority structures. The subsequent units extend to graphs and advanced specialised structures (AVL trees, tries, advanced hash maps).

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
    3.1 What is Ethical Hacking?
    3.2 Hacker Categories and Mindset
    3.3 The Penetration Testing Engagement
4.0 Conclusion
5.0 Summary
6.0 Self-Assessment Questions (SAQs)
7.0 References/Further Reading

2.0 Learning Outcomes

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

- Describe binary tree structure and implement the three DFS traversals
- Implement Binary Search Tree insert, search, and delete operations
- Explain BST search complexity and the impact of tree balance
- Implement a max-heap and explain heapify, insert, and extract-max
- Explain the relationship between heap sort and the max-heap structure

.

3.0 Main Content

3.1 Binary Trees and Traversals

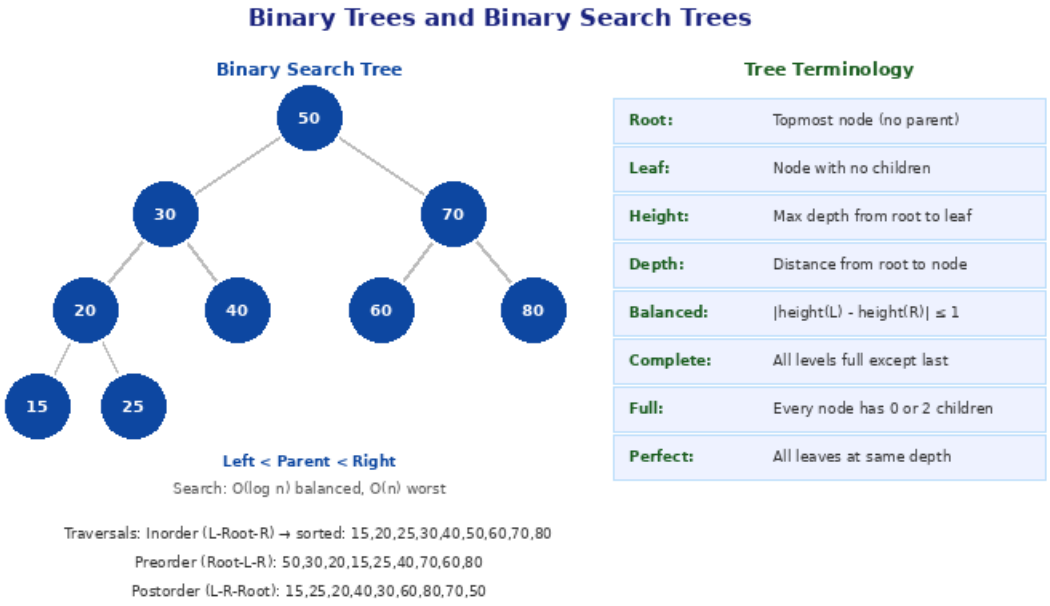


Figure 6.1: Binary Search Tree — Structure, Terminology, and Three DFS Traversal Orders

A binary tree is a hierarchical structure in which each node has at most two children, designated left and right. Key terminology: root (no parent), leaf (no children), height (longest path root-to-leaf), depth (distance from root), and balanced (height difference between left and right subtrees ≤ 1 at every node).

Tree traversal visits every node exactly once. Three DFS traversal orders for a binary tree with root R, left subtree L, right subtree R_sub: Inorder (L → Root → R_sub): for a BST, inorder traversal produces elements in sorted order — invaluable for sorted iteration. Preorder (Root → L → R_sub): used to copy or serialise a tree; the root is always visited first. Postorder (L → R_sub → Root): used to delete a tree (children

before parent) or evaluate expression trees (operands before operator). Level-order (BFS): visits nodes level by level using a queue.

```
# Binary Tree Node
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left  = None
        self.right = None

# Three DFS traversals
def inorder(root):    # L-Root-R: sorted output for BST
    if root:
        inorder(root.left)
        print(root.val, end=" ")
        inorder(root.right)

def preorder(root):   # Root-L-R: serialisation, copy
    if root:
        print(root.val, end=" ")
        preorder(root.left)
        preorder(root.right)

def postorder(root):  # L-R-Root: deletion, expression trees
    if root:
```

```
        postorder(root.left)
        postorder(root.right)
        print(root.val, end=" ")

# Level-order (BFS)
from collections import deque
def level_order(root):
    if not root: return
    q = deque([root])
    while q:
        node = q.popleft()
        print(node.val, end=" ")
        if node.left:  q.append(node.left)
        if node.right: q.append(node.right)
```

3.2 Binary Search Trees

A Binary Search Tree (BST) is a binary tree with the BST property: for every node N, all values in N's left subtree are less than N's value, and all values in N's right subtree are greater. This property enables O(log n) average-case search, insert, and delete — making BSTs the basis for ordered map and set implementations.

```
# BST Insert and Search
class BST:
    def __init__(self):
```

```
        self.root = None

    def insert(self, val):          # O(h) — h = tree height
        self.root = self._insert(self.root, val)

    def _insert(self, node, val):
        if not node: return TreeNode(val)
        if val < node.val:
            node.left  = self._insert(node.left,  val)
        elif val > node.val:
            node.right = self._insert(node.right, val)
        return node                 # Duplicate: ignore

    def search(self, val):          # O(h)
        return self._search(self.root, val)

    def _search(self, node, val):
        if not node or node.val == val: return node
        if val < node.val: return self._search(node.left,  val)
        return self._search(node.right, val)
    def inorder_sorted(self):       # O(n) — returns sorted list
        result = []
        self._inorder(self.root, result)
        return result
```

```
    def _inorder(self, node, res):
        if node:
            self._inorder(node.left, res)
            res.append(node.val)
            self._inorder(node.right, res)
# BST height = O(log n) balanced, O(n) degenerate (sorted input)
# AVL and Red-Black trees maintain O(log n) height through rotations
```

3.3 Heaps and Priority Queues

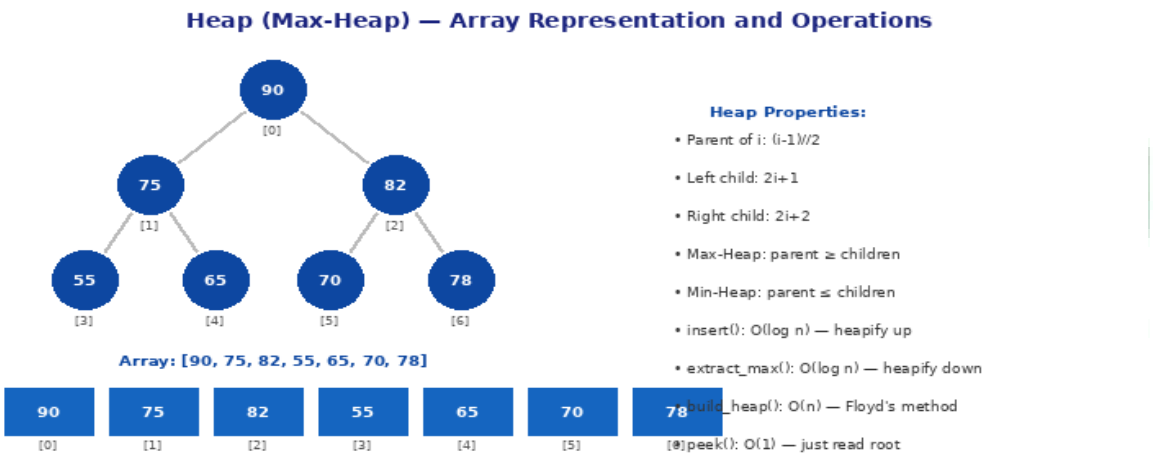


Figure 6.2: Max-Heap — Array Representation, Parent-Child Index Formulas, and Operation Complexities

A heap is a complete binary tree (all levels fully filled except possibly the last, which is filled left-to-right) satisfying the heap property. A max-heap: parent ≥ both children (root is maximum). A min-heap: parent ≤ both children (root is minimum). A heap is efficiently stored as an array: root at index 0; left child of node i at 2i+1; right child at 2i+2; parent of node i at (i-1)//2.

```
# Max-Heap implementation
class MaxHeap:
    def __init__(self):
        self.heap = []

    def insert(self, val):          # O(log n)
        self.heap.append(val)
        self._heapify_up(len(self.heap) - 1)
    def _heapify_up(self, i):       # Bubble up to restore heap property
        parent = (i - 1) // 2
        if i > 0 and self.heap[i] > self.heap[parent]:
            self.heap[i], self.heap[parent] = self.heap[parent], self.heap[i]
            self._heapify_up(parent)

    def extract_max(self):          # O(log n)
        if not self.heap: raise IndexError("empty heap")
        max_val = self.heap[0]
        self.heap[0] = self.heap[-1]
        self.heap.pop()
        self._heapify_down(0)
        return max_val
    def _heapify_down(self, i):     # Sink down to restore heap property
        n = len(self.heap)
        largest = i
```

```
        left, right = 2*i+1, 2*i+2
        if left  < n and self.heap[left]  > self.heap[largest]: largest = left
        if right < n and self.heap[right] > self.heap[largest]: largest = right
        if largest != i:
            self.heap[i], self.heap[largest] = self.heap[largest], self.heap[i]
            self._heapify_down(largest)
    def peek(self): return self.heap[0] if self.heap else None
# Python heapq module implements min-heap
import heapq
min_heap = []
heapq.heappush(min_heap, 10)
heapq.heappush(min_heap, 3)
heapq.heappush(min_heap, 7)
print(heapq.heappop(min_heap))  # 3 — minimum

# Heap Sort — O(n log n), in-place
# 1. Build max-heap from array: O(n) — Floyd's algorithm
# 2. Repeatedly extract max and place at end: n × O(log n) = O(n log n)
```

In-text Question



Explain the difference between BST search complexity O(log n) and O(n). What causes degenerate BST behaviour and how do AVL trees address it?

Intext Answer

In a balanced BST, the height is O(log n) — the left and right subtrees at every node have approximately equal depth. Search follows the BST property at each node, eliminating half the remaining candidates at each step — analogous to binary search. For n=1,000,000, a balanced BST requires at most $\log_2(1{,}000{,}000) \approx 20$ comparisons. Degenerate behaviour occurs when elements are inserted in sorted (or reverse-sorted) order: insert 1, 2, 3, 4, 5 into a BST and the result is a right-skewed linked list — each node has only a right child. The height becomes O(n), and search degrades to O(n) — equivalent to linear search. AVL trees (Adelson-Velsky and Landis, 1962) maintain the balance invariant: $|\text{height(left subtree)} - \text{height(right subtree)}| \leq 1$ at every node after every insert and delete. When this invariant is violated, AVL trees perform rotations (single or double) to restore balance in O(log n) time. This guarantees O(log n) search, insert, and delete for all inputs. Red-Black trees (used in Java TreeMap, C++ std::map) provide a similar guarantee with slightly less strict balancing but fewer rotations in practice.

4.0 Study Unit Conclusion

In this unit you have examined binary trees (structure, four traversal orders), Binary Search Trees (BST property, insert/search/delete, O(log n) balanced vs O(n) degenerate), AVL tree self-balancing via rotations, and heaps (max-heap array representation, heapify up/down, extract-max, heap sort). The priority queue ADT, backed by a min-heap, is among the most practically useful data structures in algorithm design.

5.0 Summary

Binary tree: each node ≤ 2 children. Traversals: inorder (L-Root-R, sorted for BST), preorder (Root-L-R, serialise), postorder (L-R-Root, delete), level-order (BFS). BST property: left < root < right. Search/insert/delete: O(log n) balanced, O(n) degenerate (sorted input). AVL trees maintain balance via rotations — O(log n) guaranteed. Max-heap: complete binary tree, parent ≥ children, stored in array: parent at (i-1)//2, children at 2i+1 and 2i+2. insert: heapify-up O(log n). extract-max: heapify-down O(log n). build-heap: O(n) Floyd's. Heap sort: build-heap + n extractions = O(n log n), in-place. Python heapq: min-heap.

6.0 Self-Assessment Questions (SAQs)

1. Build a BST by inserting values [50,30,70,20,40,60,80] in that order. Draw the resulting tree and perform inorder, preorder, and postorder traversals, showing the output sequence for each.
2. What is the worst-case input sequence for a BST? Demonstrate by inserting [1,2,3,4,5] and showing the resulting degenerate tree structure. Explain how AVL trees prevent this.

3. Implement heap sort step by step on the array [4,10,3,5,1]: (a) build a max-heap, showing the array state; (b) repeatedly extract max and place at the sorted end.
4. Compare BST and hash table for implementing a phone book (insert name+number, search by name, delete, list all names alphabetically). Which would you choose and why?

7.0 References/ Further Reading

Cormen et al. (2022). Introduction to Algorithms (4th ed.). MIT Press.

Sedgewick, R. & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.

Adelson-Velsky, G.M. & Landis, E.M. (1962). An Algorithm for the Organisation of Information. Doklady Akademii Nauk SSSR.

Goodrich, M.T. & Tamassia, R. (2015). Data Structures and Algorithms in Python. Wiley.

SAQs Answers

SAQ Answer 1

BST: 50(root) → left:30→20,40; right:70→60,80

Inorder: 20,30,40,50,60,70,80 (sorted)

Preorder: 50,30,20,40,70,60,80

Postorder: 20,40,30,60,80,70,50

SAQ Answer 2

Worst-case: sorted input [1,2,3,4,5] creates right-skewed tree (each node has only

right child) → height=n → O(n) search. AVL trees maintain balance through rotations after each insert, keeping height O(log n).

SAQ Answer 3

Array: [4,10,3,5,1]

Build max-heap: [10,5,3,4,1]

Extract max: [10,5,4,3,1] → [5,4,3,1,10]

Extract max: [5,4,3,1,10] → [4,3,1,5,10]

Extract max: [4,3,1,5,10] → [3,1,4,5,10]

Extract max: [3,1,4,5,10] → [1,3,4,5,10]

Sorted: [1,3,4,5,10]

SAQ Answer 4

For phone book:

- **BST**: O(log n) search, O(log n) insert/delete, O(n) alphabetical traversal. Good if alphabetical listing is frequent.
- **Hash Table**: O(1) average search/insert/delete, but O(n) for alphabetical listing (requires sorting).

  Choose **BST** if alphabetical listing is important; choose **Hash Table** if only exact-name lookups are needed. Python's dict for hash table, sortedcontainers for ordered map.