DrillBit

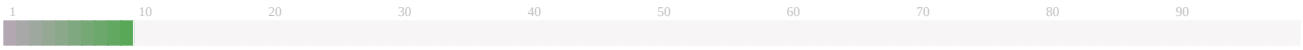
The Report is Generated by DrillBit Plagiarism Detection Software

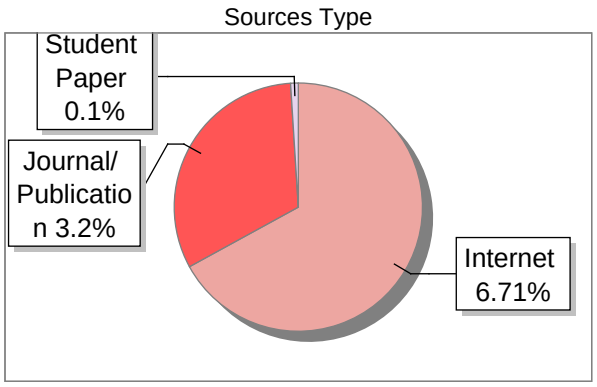
### *Submission Information*

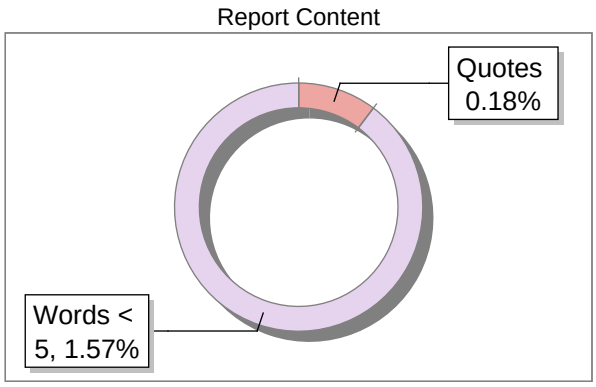
| | |
|---|---|
| Author Name | NIKHIL RAI |
| Title | DSA |
| Paper/Submission ID | 3026581 |
| Submitted by | librarian.adbu@gmail.com |
| Submission Date | 2025-01-24 15:20:43 |
| Total Pages, Total Words | 110, 12262 |
| Document type | Others |

### *Result Information*

Similarity **10 %**




### *Exclude Information*

| | |
|---|---|
| Quotes | Excluded |
| References/Bibliography | Excluded |
| Source: Excluded < 5 Words | Excluded |
| Excluded Source | **0 %** |
| Excluded Phrases | Not Excluded |

### *Database Selection*

| | |
|---|---|
| Language | English |
| Student Papers | Yes |
| Journals & publishers | Yes |
| Internet or Web | Yes |
| Institution Repository | Yes |

A Unique QR Code use to View/Download/Share Pdf File

DrillBit

**DrillBit Similarity Report**

| 10 | 47 | A |
|---|---|---|
| SIMILARITY % | MATCHED SOURCES | GRADE |

**A-Satisfactory (0-10%)**
**B-Upgrade (11-40%)**
**C-Poor (41-60%)**
**D-Unacceptable (61-100%)**

| LOCATION | MATCHED DOMAIN | % | SOURCE TYPE |
|---|---|---|---|
| 1 | www.geeksforgeeks.org | 1 | Internet Data |
| 2 | www.wileyindia.com | 1 | Publication |
| 3 | fastercapital.com | 1 | Internet Data |
| 4 | www.studysmarter.co.uk | <1 | Internet Data |
| 5 | egyankosh.ac.in | <1 | Publication |
| 6 | www.tutorchase.com | <1 | Internet Data |
| 7 | www.studysmarter.co.uk | <1 | Internet Data |
| 8 | medium.com | <1 | Internet Data |
| 9 | medium.com | <1 | Internet Data |
| 10 | apply.jaipur.manipal.edu | <1 | Publication |
| 11 | qdoc.tips | <1 | Internet Data |
| 12 | cse110textbook.s3-us-west-1.amazonaws.com | <1 | Publication |
| 13 | technodocbox.com | <1 | Internet Data |
| 14 | www.geeksforgeeks.org | <1 | Internet Data |

| | | | |
|---|---|---|---|
| 15 | docplayer.net | <1 | Internet Data |
| 16 | qdoc.tips | <1 | Internet Data |
| 17 | moam.info | <1 | Internet Data |
| 18 | documents.mx | <1 | Internet Data |
| 19 | www.dx.doi.org | <1 | Publication |
| 20 | www.msec.ac.in | <1 | Publication |
| 21 | www.tutorchase.com | <1 | Internet Data |
| 22 | iare.ac.in | <1 | Publication |
| 23 | www.crio.do | <1 | Internet Data |
| 24 | livrosdeamor.com.br | <1 | Internet Data |
| 25 | Scheduling Non-Preemptive Deferrable Loads by OBrien-2015 | <1 | Publication |
| 26 | Incremental computing with data structures by Morihata-2017 | <1 | Publication |
| 27 | pdfcookie.com | <1 | Internet Data |
| 28 | quizlet.com | <1 | Internet Data |
| 29 | Electrostatic lock in the transport cycle of the multidrug resistance by Vermaas-2018 | <1 | Publication |
| 30 | www.tutorchase.com | <1 | Internet Data |
| 31 | deptapp08.drexel.edu | <1 | Publication |
| 32 | moam.info | <1 | Internet Data |
| 33 | Resource Management Games for Distributed Network Localization by Chen-2017 | <1 | Publication |

| | | | |
|---|---|---|---|
| 34 | Submitted to U-Next Learning on 2025-01-24 11-36 3022004 | <1 | Student Paper |
| 35 | testbook.com | <1 | Internet Data |
| 36 | wcclas.net | <1 | Internet Data |
| 37 | Planar Graphs Have Bounded Queue-Number by Dujmovi-2020 | <1 | Publication |
| 38 | springeropen.com | <1 | Internet Data |
| 39 | www.academia.edu | <1 | Internet Data |
| 40 | A study on the dynamic characteristics of the Korean Yi-dynasty bell t by Su-1987 | <1 | Publication |
| 41 | CFSpro ray tracing for design and optimization of complex fenestration systems by Kostro-2016 | <1 | Publication |
| 42 | Control of chaos methods and applications in mechanics by Fradkov-2006 | <1 | Publication |
| 43 | instasize.com | <1 | Internet Data |
| 44 | moam.info | <1 | Internet Data |
| 45 | moam.info | <1 | Internet Data |
| 46 | revistaschilenas.uchile.cl | <1 | Internet Data |
| 47 | www-personal.acfr.usyd.edu.au | <1 | Publication |

# Module 1

# Unit 1: Basics of Algorithms

## Course Objectives

- To understand the fundamental principles of algorithms.
- To gain knowledge of problem-solving techniques and algorithmic classifications.
- To bridge theoretical concepts with practical applications.

## Course Outcomes

- Analyze and design algorithms for problem-solving.
- Classify algorithms based on different criteria.
- Apply algorithmic principles to real-world scenarios.

---

## 1.0 Introduction

Algorithms form the backbone of computer science and are pivotal in solving computational problems efficiently. They provide a systematic approach to problem-solving by outlining a sequence of steps to achieve a desired outcome.

By understanding the basic principles of algorithms, you can design solutions that are not only effective but also optimized for performance. This unit introduces the foundational concepts of algorithms, aiming to bridge theory with practical implementation.

In this unit, we will explore:

- Fundamental techniques for problem-solving.
- How algorithms are classified based on purpose, paradigm, and performance.

# 1.1 Fundamentals of Problem Solving

Problem-solving is one of the core skills in algorithm design, which involves a systematic approach to tackle computational challenges. The following steps outline this process in detail:

1. **Understanding the Problem**:
    - Clearly define the problem, ensuring the scope and requirements are well-understood.
    - Identify inputs, outputs, constraints, and assumptions to set a clear boundary for the solution.

2. **Detailed Explanation:** The first and most critical step is to understand the problem. If misinterpreted the problem can lead to inefficient or incorrect solutions. For example, while solving a sorting problem, it's essential to know whether the input is always a list of numbers or if it might include strings. Additionally, understanding constraints ensures that the solution remains efficient even for edge cases.

3. **Designing a Solution**:
    - Decompose the problem into manageable sub-problems or tasks.
    - Develop a strategy or algorithm to solve each sub-problem, aligning them to form the overall solution.

4. **Detailed Explanation:** Designing a solution involves creativity and logical thinking. For instance, in designing a search algorithm, one might compare the straightforward linear search to the efficient binary search for sorted data. Decomposing problems often reveals simpler methods or reusable patterns, like recursion or iteration.

5. **Developing an Algorithm**:
    - Translate the solution strategy into a precise, step-by-step process that can be implemented programmatically.
    - Consider both correctness and efficiency during the design phase.

6. **Detailed Explanation:** Algorithms are blueprints for solving problems. A well-designed algorithm not only solves the problem but also ensures optimal use of resources. For instance, a sorting algorithm like Merge Sort achieves better performance by using the divide-and-conquer paradigm.

7. **Implementation**:
    - Convert the algorithm into a functioning program using a programming language.
    - Adhere to best coding practices to ensure readability and maintainability.

8. **Testing and Optimization**:
    - Validate the algorithm with diverse test cases, which include edge cases and extreme inputs.
    - Optimize the algorithm for performance by reducing time and space complexity where possible.

9. **Detailed Explanation:** Testing ensures that the algorithm handles all inputs correctly. Optimization, on the other hand, makes the solution scalable. For instance, reducing time complexity from $O(n^2)$ to $O(n\log n)$ makes a remarkable difference for large datasets.

## Example: Finding the Largest Number in an Array

This example demonstrates a simple yet effective algorithm to identify the largest number in a given array. The steps ensure correctness and efficiency:

**Algorithm:**

1. Initialize a variable `maximum` with the 1st element from the array.
2. Traverse the array from the 2nd element onwards:
    - If the current element is greater than `maximum`, update `maximum` to this element.
3. Return or print the value of `maximum` .

**Key Considerations:**

- **Edge Cases**: Handle scenarios such as an empty array by returning an error or default value.
- **Duplicates**: If duplicate maximum values exist, this algorithm still returns the correct result.

**Program (C++):**

```
# Finding the largest number in an array
#include <iostream>
#include <vector>

int main() {
    std::vector<int> arr = {3, 5, 7, 2, 8};

    if (arr.empty()) {
        std::cout << "Array is empty." << std::endl;
    }
    else {
```

```
        int max_value = arr[0];
        for (int num : arr) {
            if (num > max_value) {
                max_value = num;
            }
        }
        std::cout << "Largest number is: " << max_value << std::endl;
    }

    return 0;
}
```

---

# 1.2 Classification of Algorithms

Algorithms can be classified based on various criteria, each providing understanding of their structure, purpose, and application. Below are the primary classifications:

## 1.2.1 Based on Purpose

- **Sorting Algorithms**: Organize data in a specific order, such as ascending or descending. For example:
    - **Quick Sort**: Efficient for large datasets.
    - **Merge Sort**: Stable sorting algorithm with consistent performance.
- **Searching Algorithms**: Locate a specific element within a dataset. Examples include:
    - **Binary Search**: Ideal for sorted arrays with logarithmic complexity.
    - **Linear Search**: Simplistic but works for unsorted data.
- **Optimization Algorithms**: Solve problems by optimizing a specific criterion, such as cost or efficiency. Examples:
    - **Dynamic Programming for resource allocation**.
    - **Greedy Algorithms for route optimization**.

Detailed Explanation: Sorting algorithms are essential for organizing and processing data efficiently. Searching algorithms, instead, reduce the time needed to locate specific elements. Optimization algorithms focus on minimizing or maximizing a particular attribute, such as cost or speed.

## 1.2.2 Based on Design Paradigm

- **Divide and Conquer**:
    - Divide the problem into smaller, manageable parts, solve them recursively, and combine results.
    - Example: Merge Sort efficiently sorts arrays by dividing them into halves.

- **Dynamic Programming**:
    - Stores results of overlapping sub-problems to avoid redundant computations.
    - Example: Calculating Fibonacci numbers efficiently.

- **Greedy Algorithms**:
    - Focus on making locally optimal choices at each step to achieve a globally optimal solution.
    - Example: Dijkstra's Algorithm for shortest path problems.

## 1.2.3 Based on Performance

- **Time Complexity**: Measures how the running time of the algorithm grows with input size.
- **Space Complexity**: Assesses the memory required by the algorithm during execution.

By getting the insights of these classifications, you can select and design algorithms tailored to specific problems. This knowledge is especially useful in real-world applications, where trade-offs between time and space complexities often dictate algorithm choices.

### Example: Classification of Sorting Algorithms

The table below provides a quick comparison of common sorting algorithms based on their design paradigm and performance:

| Algorithm | Design Paradigm | Time Complexity |
|---|---|---|
| Bubble Sort | Iterative | O(n^2) |
| Quick Sort | Divide and Conquer | O(n log n) |
| Merge Sort | Divide and Conquer | O(n log n) |

**Significance:**

- **Bubble Sort**: Simple to implement but inefficient for large datasets.
- **Quick Sort**: Efficient and widely used, though performance depends on pivot selection.
- **Merge Sort**: Guarantees stable sorting, making it suitable for datasets where order preservation matters.

---

## 1.3 Unit Summary

In this unit, we introduced the basics of algorithms, covering fundamental problem-solving techniques and classifications. We discussed how to approach problem-solving systematically, the importance of algorithm design, and how algorithms are categorized by purpose, paradigm, and performance. With this foundation, you can now delve deeper into specific algorithms and their applications.

---

## Practice Problems

1. Write an algorithm to reverse an array.
2. Classify the following algorithms based on their paradigm:
    - Binary Search
    - Prim's Algorithm
    - Bellman-Ford Algorithm
3. Implement a C++ program to find the smallest number in an array.

# Quiz

1. What are the steps involved in problem-solving?

# Practical Questions

1. Implement Merge Sort in your preferred programming language.
2. Analyze the time complexity of Bubble Sort and suggest an optimization.

---

# Unit 2: Algorithm Analysis

## Course Objectives

- To understand the methodologies for analyzing algorithm efficiency.
- To learn asymptotic analysis techniques and their applications.
- To evaluate and compare iterative and recursive algorithms.

## Course Outcomes

- Apply asymptotic analysis for evaluating algorithm performance.
- Perform mathematical and empirical analysis of algorithms.
- Develop efficient algorithms using analytical insights.

---

## 2.0 Introduction

Algorithm analysis provides the tools to evaluate the efficiency and effectiveness of algorithms. It helps in understanding the resource requirements (time and space) and guides in selecting the most appropriate algorithm for a given problem.

This unit covers:

- Basics of algorithm analysis.
- Asymptotic analysis and its importance.
- Mathematical and empirical methods for evaluating algorithms.

---

# 2.1 Basics of Algorithm Analysis

Algorithm analysis focuses on determining the computational resources required to solve a problem. The key metrics used include:

## Time Complexity

- **Definition**: The amount of time an algorithm takes to complete as a function of input size.
- **Example**: In a linear search, the time complexity is O(n) because the number of comparisons grows linearly with the array size.

## Space Complexity

- **Definition**: The amount of memory an algorithm uses during its execution.
- **Example**: A recursive function requires additional stack space, contributing to its space complexity.

## Key Considerations

1. **Input Size**: Larger inputs typically require more time and space.
2. **Worst, Average, and Best Cases**:
    - Worst Case: Maximum time taken (e.g., searching for a missing element in a list).
    - Average Case: Expected time for random inputs.
    - Best Case: Minimum time taken (e.g., finding the first element in a list).

---

# 2.2 Asymptotic Analysis

Asymptotic analysis evaluates algorithm efficiency by analyzing its behavior for large input sizes. It abstracts away constants and lower-order terms, focusing on growth rates.

## Common Notations

1. **Big O (O)**:
    - Represents the upper bound of time complexity.
    - Example: Binary search has a time complexity of O(log n).

2. **Omega (Ω)**:
    - Represents the lower bound of time complexity.
    - Example: The best-case time complexity of linear search is Ω(1).

3. **Theta (Θ)**:
    - Represents the exact bound of time complexity.
    - Example: Merge Sort has a time complexity of Θ(n log n).

## Importance

- Helps compare algorithms independently of hardware or programming language.
- Provides insights into scalability and performance.

## Example: Time Complexity of Nested Loops

```
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        cout << i << " " << j << endl;
    }
}
```

- **Analysis**: Each loop runs n times, resulting in a time complexity of O(n^2).

---

# 2.3 Mathematical Analysis of Iterative and Recursive Algorithms

## Iterative Algorithms

- Use loops to repeat operations.
- **Example**: Finding the sum of all elements in an array.

**Analysis:**

1. Identify the number of loop iterations.
2. Determine the time complexity based on the operations within the loop.

**Example:**

```
sum = 0
for(i = 0; i < n; i++){
   sum += arr[i]
}
```

- **Time Complexity**: O(n).

## Recursive Algorithms

- Solve problems by breaking them into smaller sub-problems and solving them recursively.
- **Example**: Calculating Fibonacci numbers.

**Analysis:**

1. Formulate a recurrence relation.
2. Solve the recurrence to determine time complexity.

**Example:**

```
fibonacci(n){
   if (n <= 1):
      return n;
   return fibonacci(n-1) + fibonacci(n-2);
}
```

- **Recurrence Relation**: T(n) = T(n-1) + T(n-2) + O(1).
- **Time Complexity**: Exponential, O(2^n).

---

# 2.4 Empirical Analysis of Algorithms

Empirical analysis involves measuring algorithm performance through experimentation rather than theoretical evaluation.

## Steps:

1. **Implement the Algorithm**: Code the algorithm in a programming language.
2. **Set Up Test Cases**: Include edge cases, average cases, and worst cases.
3. **Measure Performance**: Use timing functions or profiling tools to evaluate execution time and memory usage.

## Advantages:

- Provides real-world performance insights.
- Helps identify implementation-specific bottlenecks.

### Example: Comparing Sorting Algorithms

Test sorting algorithms like Merge Sort, Bubble Sort, and Quick Sort on large datasets to measure execution time and memory consumption.

---

## 2.5 Unit Summary

In this unit, we discussed the importance of algorithm analysis, focusing on time and space complexity, asymptotic notations, and methods for analyzing iterative and recursive algorithms. Additionally, empirical analysis was introduced as a practical approach to evaluating algorithm performance.

---

## Practice Problems

1. Derive the time complexity of the following code snippet:

```
for (int i = 0; i < n; i++){
   for (int j = 0; j < i; j++){
       print(i, j)
   }
}
```

2. Analyze the space complexity of a recursive function for calculating factorial.
3. Compare the time complexity of Bubble Sort and Quick Sort.

## Quiz

1. Define time complexity and space complexity.
2. Explain the difference between worst-case and average-case time complexity.

# Practical Questions

1. Implement and analyze the performance of a recursive Fibonacci sequence generator.
2. Compare the execution time of Merge Sort and Insertion Sort for large datasets.

---

# Unit 3: Models of Computation

## Course Objectives

- To understand the theoretical models of computation.
- To explore the RAM model and Turing machines as foundational computational paradigms.
- To analyze the applicability of these models in algorithm design and complexity theory.

## Course Outcomes

- Explain the role of computational models in algorithm analysis.
- Differentiate between practical and theoretical models of computation.
- Apply Turing machines and RAM models to solve computational problems.

---

## 3.0 Introduction

Models of computation form the theoretical basis for understanding and analyzing algorithms. They abstract the capabilities and limitations of computational devices, offering a framework to study problems independent of specific hardware or software.

This unit introduces:

- The RAM model as a practical representation of computation.
- The Turing Machine as a theoretical framework for defining computability.

---

# 3.1 RAM Model

The Random Access Machine (RAM) model is a simplified abstraction of a real-world computer. It is commonly used for algorithm analysis as it closely mimics the behavior of modern computing systems.

## Features of the RAM Model

1. **Sequential Execution**: Instructions are executed one after another.
2. **Random Access Memory**: Any memory cell can be accessed in constant time.
3. **Instruction Set**: Supports basic operations such as addition, subtraction, comparison, and memory access.
4. **Unit Cost**: Assumes each operation, including memory access, takes one time unit.

## Importance of the RAM Model

- Facilitates the design and analysis of algorithms without concern for hardware specifics.
- Serves as the foundation for measuring time complexity in Big O notation.

## Example: Time Complexity in the RAM Model

Consider the problem of summing up n numbers of an array:

```
sum = 0;
for (int i = 0; i < n; i++){
   sum += arr[i];
}
```

- **Analysis**: The loop executes n iterations, and each iteration involves a memory access and addition. Thus, the time complexity is O(n).

---

# 3.2 Turing Machine

The Turing Machine is a theoretical construct introduced by Alan Turing to define the concept of computability. It is a impactful model that can simulate any algorithm.

## Components of a Turing Machine

1. **Tape**:
    - Infinite in length and divided into cells.
    - Each cell holds a symbol from a finite alphabet.
2. **Head**:
    - Reads and writes symbols on the tape.
    - Moves left or right based on the current state and input symbol.
3. **Finite State Control**:
    - Directs the machine's operations based on its current state and tape symbol.

## Working of a Turing Machine

1. **Initialization**: The input is written on the tape, and the head starts at a designated position.
2. **Transition**: The machine transitions between states based on the current state and tape symbol.
3. **Halting**: The machine stops when it reaches a designated halting state.

### Example: Turing Machine for Unary Addition

To add two unary numbers (e.g., 111 + 11 = 11111):

1. Scan to the rightmost 1 of the first number.
2. Replace the next blank cell with a 1.
3. Repeat until the second number is fully processed.

### Significance of Turing Machines

- Provides a formal definition of what it means for a function to be computable.
- Forms the basis for complexity classes such as P and NP.
- Demonstrates that some problems are undecidable (e.g., the Halting Problem).

---

## 3.3 Unit Summary

In this unit, we explored two fundamental models of computation:

- The **RAM Model**, which offers a practical framework for analyzing algorithm efficiency.
- The **Turing Machine**, which provides a theoretical foundation for understanding computability and complexity.

These models are essential for narrowing the gap between theoretical computer science and real-world algorithm design.

---

# Practice Problems

1. Write a RAM model algorithm to find the maximum value in an array and analyze its time complexity.
2. Design a Turing Machine to recognize strings of the form `0^n1^n` (equal number of 0s followed by 1s).

# Quiz

1. What is the significance of the RAM model in algorithm analysis?
2. Describe the main components of a Turing Machine.
3. Explain the Halting Problem and its implications.

# Practical Questions

1. Compare the RAM model and Turing Machine in terms of their applications and limitations.
2. Implement a C++ program to simulate a simple Turing Machine for unary addition.

---

# Module 2

# Unit 4: Abstract Data Types (ADTs)

## Course Objectives

- To understand the concept of Abstract Data Types (ADTs).
- To explore the implementation and applications of stacks, queues, and circular queues.
- To analyze the role of ADTs in algorithm development and problem-solving.

## Course Outcomes

- Explain the importance of ADTs in data organization.
- Implement and utilize stacks, queues, and circular queues in programming.
- Apply ADTs to solve real-world problems effectively.

---

## 4.0 Introduction

Abstract Data Types (ADTs) provide a theoretical framework for defining and organizing data structures. ADTs describe the operations that can be performed on data without specifying the implementation details, promoting modularity and reusability.

This unit focuses on three fundamental ADTs:

- **Stacks**: A linear data structure with Last-In-First-Out (LIFO) behavior.
- **Queues**: A linear data structure with First-In-First-Out (FIFO) behavior.

- **Circular Queues**: A variant of queues that optimizes memory usage by reusing previously allocated space.

---

# 4.1 Stacks

## Definition

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. Adding and removing elements from only one end, known as the "top" of the stack.

## Operations

1. **Push**: Adds an element to the top of the stack.
2. **Pop**: Removes the top element from the stack.
3. **Peek/Top**: Returns the top element without removing it.
4. **IsEmpty**: Makes sure if the stack is empty.
5. **IsFull**: Makes sure if the stack is full (in a fixed-size implementation).

## Applications

- Function call management (e.g., recursion).
- Expression evaluation and conversion (e.g., infix to postfix).
- Undo operations in text editors.

## Example: Stack Implementation in C++

```
#include <iostream>
#include <vector>
#include <string>

class Stack {
private:
    std::vector<int> stack; // Assuming the stack will hold integers

public:
    Stack() {}

    void push(int item) {
        stack.push_back(item);
    }

    std::string pop() {
        if (!is_empty()) {
            int item = stack.back();
            stack.pop_back();
            return std::to_string(item);
        } else {
            return "Stack is empty";
        }
    }

    std::string peek() {
        if (!is_empty()) {
            return std::to_string(stack.back());
        } else {
            return "Stack is empty";
        }
    }
```

```
    bool is_empty() {
        return stack.empty();
    }
};
```

---

# 4.2 Queues

## Definition

A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. Elements are added at the rear and removed from the front.

## Operations

1. **Enqueue**: Adds an element to the rear of the queue.
2. **Dequeue**: Removes an element from the front of the queue.
3. **Peek/Front**: Returns the front element without removing it.
4. **IsEmpty**: Makes sure if the queue is empty.
5. **IsFull**: Makes sure if the queue is full (in a fixed-size implementation).

## Applications

- Process scheduling in operating systems.
- Managing requests in web servers.
- Breadth-First Search (BFS) in graph traversal.

## Example: Queue Implementation in C++

```
#include <iostream>
#include <vector>
#include <string>

class Queue {
private:
    std::vector<int> queue; // Assuming the queue will hold integers

public:
    Queue() {}

    void enqueue(int item) {
        queue.push_back(item);
    }

    std::string dequeue() {
        if (!is_empty()) {
            int item = queue.front();
            queue.erase(queue.begin());
            return std::to_string(item);
        } else {
            return "Queue is empty";
        }
    }

    std::string peek() {
        if (!is_empty()) {
            return std::to_string(queue.front());
        } else {
            return "Queue is empty";
        }
    }
```

```
    bool is_empty() {
        return queue.empty();
    }
};
```

---

# 4.3 Circular Queues

## Definition

A circular queue is a linear data structure where the last position is connected to the first, forming a circle. This design eliminates the unused space issue in linear queues.

## Operations

1. **Enqueue**: Adds an element to the rear if space is available.
2. **Dequeue**: Removes an element from the front.
3. **Peek/Front**: Returns the front element without removing it.
4. **IsEmpty**: Makes sure if the queue is empty.
5. **IsFull**: Makes sure if the queue is full.

## Advantages of Circular Queues

- Efficient utilization of memory by reusing freed space.
- Prevents the "queue overflow" issue in a fixed-size queue.

## Example: Circular Queue Implementation in C++

```
#include <iostream>
#include <vector>

class CircularQueue {
private:
    int size;
    std::vector<int> queue;
    int front;
    int rear;

public:
    CircularQueue(int size) : size(size), front(-1), rear(-1) {
        queue.resize(size);
    }

    std::string enqueue(int item) {
        if ((rear + 1) % size == front) {
            return "Queue is full";
        } else if (is_empty()) {
            front = rear = 0;
        } else {
            rear = (rear + 1) % size;
        }
        queue[rear] = item;
        return "";
    }

    std::string dequeue() {
        if (is_empty()) {
            return "Queue is empty";
        }
        int temp = queue[front];
        if (front == rear) {
```

```
            front = rear = -1;
        } else {
            front = (front + 1) % size;
        }
        return std::to_string(temp);
    }

    bool is_empty() {
        return front == -1;
    }
};
```

---

# 4.4 Unit Summary

In this unit, we covered three essential Abstract Data Types:

- **Stacks**: A LIFO structure used in recursion, expression evaluation, and undo operations.
- **Queues**: A FIFO structure applied in scheduling, request handling, and BFS.
- **Circular Queues**: An optimized queue variant that efficiently utilizes memory.

By mastering these ADTs, you can design robust algorithms for a wide range of applications.

---

# Practice Problems

1. Implement a stack using a linked list and perform push, pop, and peek operations.
2. Write a program to simulate a queue using two stacks.
3. Create a circular queue for managing a print job queue.

# Quiz

1. What is the key difference between a queue and a stack?
2. Why is a circular queue more memory-efficient than a linear queue?

# Practical Questions

1. Analyze the time complexity of operations in a stack and queue.
2. Design a C++ program to evaluate a postfix expression using a stack.

---

# Unit 5: Implementations and Advanced Structures

## Course Objectives

- To explore advanced implementations of stacks and queues using each other.
- To understand the concepts of priority queues and heaps.
- To analyze the applications and significance of these data structures in real-world scenarios.

## Course Outcomes

- Implement stacks using queues and queues using stacks.
- Understand and apply priority queues and heaps in problem-solving.
- Evaluate the performance and use cases of advanced data structures.

---

## 5.0 Introduction

This unit dives deep into advanced implementations of basic data structures and explores specialized data structures like priority queues and heaps. These structures extend the functionality of traditional stacks and queues, offering new ways to solve complex computational problems efficiently.

---

# 5.1 Implementation of Stacks Using Queues

## Concept

Stacks and queues are fundamental data structures with distinct operational principles. Implementing a stack using one or more queues requires manipulating queue operations to emulate the Last-In-First-Out (LIFO) behavior of a stack.

## Methods

There are two primary approaches:

1. **Using Two Queues**:
    - Push operation is straightforward.
    - Pop operation involves transferring elements between queues to maintain the stack order.

2. **Using a Single Queue**:
    - Push involves adding elements and rotating the queue.
    - Pop is straightforward by removing from the front.

## Example: Stack Using Two Queues

```
#include <queue>
#include <iostream>
#include <string>

class StackUsingQueues {
private:
    std::queue<int> q1;
    std::queue<int> q2;

public:
```

```
    StackUsingQueues() {}

    void push(int item) {
        q1.push(item);
    }

    std::string pop() {
        if (q1.empty()) {
            return "Stack is empty";
        }
        while (q1.size() > 1) {
            q2.push(q1.front());
            q1.pop();
        }
        int popped_item = q1.front();
        q1.pop();
        std::swap(q1, q2);
        return std::to_string(popped_item);
    }
};
```

---

# 5.2 Implementation of Queues Using Stacks

## Concept

Implementing a queue using stacks involves manipulating stack operations to mimic the First-In-First-Out (FIFO) behavior of a queue.

## Methods

1. **Using Two Stacks**:
   - Enqueue operation pushes elements to the first stack.
   - Dequeue operation transfers elements to the second stack to reverse the order.

2. **Using a Single Stack**:
   - Enqueue and dequeue operations involve recursive calls to achieve FIFO order.

## Example: Queue Using Two Stacks

```
#include <iostream>
#include <stack>
#include <string>

class QueueUsingStacks {
private:
    std::stack<int> stack1;
    std::stack<int> stack2;

public:
    void enqueue(int item) {
        stack1.push(item);
    }
```

```
    std::string dequeue() {
        if (stack1.empty() && stack2.empty()) {
            return "Queue is empty";
        }
        if (stack2.empty()) {
            while (!stack1.empty()) {
                stack2.push(stack1.top());
                stack1.pop();
            }
        }
        int front = stack2.top();
        stack2.pop();
        return std::to_string(front);
    }
};
```

---

# 5.3 Priority Queues and Heaps

## Priority Queues

A priority queue is a data structure where each element is associated with a priority. Elements with higher priorities are dequeued before those with lower priorities.

### Operations

1. **Insert**: Add an element with a priority.
2. **Delete**: Remove the element with the highest priority.

## Heaps

A heap is a specialized tree-based data structure that satisfies the heap property:

- **Max Heap**: Parent nodes are greater than or equal to their children.
- **Min Heap**: Parent nodes are less than or equal to their children.

### Applications

- Efficiently implement priority queues.
- Solve problems like finding the k-largest elements or merging sorted arrays.

## Example: Min Heap Implementation in C++

```
#include <iostream>
#include <queue>

int main() {
    std::priority_queue<int, std::vector<int>, std::greater<int>> heap;
    heap.push(10);
    heap.push(5);
    heap.push(20);
    std::cout << heap.top() << std::endl; // Output: 5
    return 0;
}
```

---

## 5.4 Unit Summary

In this unit, we explored:

- **Stacks Using Queues**: Implemented using single or multiple queues.
- **Queues Using Stacks**: Implemented using single or multiple stacks.
- **Priority Queues and Heaps**: Specialized data structures for handling prioritized data efficiently.

These advanced implementations and data structures are instrumental in solving complex computational problems across various domains.

---

## Practice Problems

1. Implement a stack using a single queue and perform push and pop operations.
2. Write a program to implement a priority queue using a max heap.
3. Compare the performance of a queue implemented using stacks versus a regular queue.

## Quiz

1. How does a heap differ from a binary search tree?
2. What is the key advantage of using two stacks to implement a queue?

## Practical Questions

1. Design a C++ program to find the k-largest elements in an array using a heap.
2. Analyze the time complexity of enqueue and dequeue operations in a queue implemented using two stacks.

# Unit 6: Linked Lists

## Course Objectives

- To understand the structure and types of linked lists.
- To learn how to perform basic operations such as search and update in linked lists.
- To analyze the applications of linked lists in computer science and programming.

## Course Outcomes

- Identify and implement different types of linked lists.
- Perform search and update operations on linked lists effectively.
- Apply linked list concepts to solve real-world problems.

---

## 6.0 Introduction

Linked lists are dynamic data structures that consist of nodes, where each node contains data and a reference to the next node. Unlike arrays, linked lists allow efficient insertion and deletion operations without the need to resize or reorganize memory. They are fundamental to understanding more advanced data structures such as stacks, queues, and graphs.

In this unit, we will explore:

- The different types of linked lists.
- How to perform search and update operations on linked lists.

---

## 6.1 Types of Linked Lists

Linked lists come in various forms, each suited for different scenarios based on their structure and functionality.

## 6.1.1 Singly Linked List

- **Structure**: Each node contains data and a pointer to the next node.
- **Advantages**: Simple to implement and efficient for operations like traversal and insertion.
- **Limitations**: Cannot traverse backward.

**Example**

```
#include <iostream>

class Node {
public:
    int data;
    Node* next;

    Node(int data) {
        this->data = data;
        this->next = nullptr;
    }
};

class SinglyLinkedList {
private:
    Node* head;

public:
    SinglyLinkedList() {
        this->head = nullptr;
    }

    void append(int data) {
        Node* newNode = new Node(data);
```

```
        if (!this->head) {
            this->head = newNode;
            return;
        }
        Node* current = this->head;
        while (current->next) {
            current = current->next;
        }
        current->next = newNode;
    }
};

int main() {
    SinglyLinkedList sll;
    sll.append(10);
    sll.append(20);
    return 0;
}
```

## 6.1.2 Doubly Linked List

- **Structure**: Each node contains data, a pointer to the next node, and a pointer to the previous node.
- **Advantages**: Can be traversed in both directions.
- **Limitations**: Requires more memory due to the additional pointer.

**Example**

```
#include <iostream>

class Node {
public:
    int data;
    Node* next;

    Node(int data) {
        this->data = data;
        this->next = nullptr;
    }
};

class SinglyLinkedList {
private:
    Node* head;

public:
    SinglyLinkedList() {
        this->head = nullptr;
    }

    void append(int data) {
        Node* newNode = new Node(data);
        if (!this->head) {
            this->head = newNode;
            return;
        }
        Node* current = this->head;
        while (current->next) {
            current = current->next;
        }
        current->next = newNode;
    }
};
```

```
class Node2 {
public:
    int data;
    Node2* next;
    Node2* prev;

    Node2(int data) {
        this->data = data;
        this->next = nullptr;
        this->prev = nullptr;
    }
};

class DoublyLinkedList {
private:
    Node2* head;

public:
    DoublyLinkedList() {
        this->head = nullptr;
    }

    void append(int data) {
        Node2* newNode = new Node2(data);
        if (!this->head) {
            this->head = newNode;
            return;
        }
        Node2* current = this->head;
        while (current->next) {
            current = current->next;
        }
        current->next = newNode;
        newNode->prev = current;
```

```
    }
};
```

## 6.1.3 Circular Linked List

- **Structure**: The last node points back to the first node, forming a circle.
- **Advantages**: Useful for applications requiring continuous traversal, such as buffering.
- **Limitations**: Complex implementation.

**Example**

```
#include <iostream>

class Node {
public:
    int data;
    Node* next;

    Node(int data) {
        this->data = data;
        this->next = nullptr;
    }
};

class CircularLinkedList {
private:
    Node* head;

public:
    CircularLinkedList() {
        this->head = nullptr;
    }
```

```
    void append(int data) {
        Node* new_node = new Node(data);
        if (!this->head) {
            this->head = new_node;
            new_node->next = this->head;
            return;
        }
        Node* current = this->head;
        while (current->next != this->head) {
            current = current->next;
        }
        current->next = new_node;
        new_node->next = this->head;
    }
};
```

---

# 6.2 Search and Update Operations

## Searching in a Linked List

To search for an element in a linked list, traverse the list and compare each node's data with the target value.

**Example**

```
bool search(Node* linked_list, int target) {
    Node* current = linked_list;
    while (current != nullptr) {
        if (current->data == target) {
            return true;
        }
        current = current->next;
    }
    return false;
}
```

## Updating a Node in a Linked List

To update a node, traverse the list until the desired node is found, and modify its data.

**Example**

```
Node* update(Node* linked_list, int target, int new_value) {
    Node* current = linked_list;
    while (current) {
        if (current->data == target) {
            current->data = new_value;
            return linked_list;
        }
        current = current->next;
    }
    return linked_list;
}
```

---

## 6.3 Unit Summary

In this unit, we covered:

- **Types of Linked Lists**: Singly, Doubly, and Circular Linked Lists.
- **Search and Update Operations**: Traversal-based techniques for modifying and retrieving data in linked lists.

Linked lists form the foundation for advanced data structures and are widely used in applications such as memory management and graph representations.

---

## Practice Problems

1. Implement a function to delete a node from a singly linked list.
2. Write a program to reverse a doubly linked list.
3. Create a circular linked list and implement a function to count its nodes.

## Quiz

1. What is the key difference between a singly and a doubly linked list?
2. In which scenarios are circular linked lists preferred?

## Practical Questions

1. Implement a linked list in your preferred programming language and demonstrate search and update operations.
2. Analyze the time complexity of searching in a doubly linked list versus a singly linked list.

---

# Unit 7: Trees

## Course Objectives

- To understand the fundamental concepts of tree data structures.
- To explore different types of trees and their applications.
- To learn traversal techniques and analyze balanced trees.

## Course Outcomes

- Implement and utilize tree structures in various programming contexts.
- Perform efficient operations using binary search trees, AVL trees, and other advanced trees.
- Apply tree traversal techniques to solve real-world problems.

---

## 7.0 Introduction

Trees are hierarchical data structures consisting of nodes connected by edges, with one node designated as the root. Unlike linear data structures, trees allow for efficient representation of hierarchical relationships. They are foundational in areas such as databases, compilers, and artificial intelligence.

In this unit, we will cover:

- Basic concepts and types of trees.
- Traversal techniques to navigate trees.
- Specialized trees such as AVL, Red-Black, and B-Trees.

---

# 7.1 Binary Trees

A binary tree is a tree where each node has at most two children, referred to as the left and right child.

## Properties:

- The height of a binary tree with n nodes is at least $\lceil \log_2(n+1) \rceil$.
- In a full binary tree, all levels except possibly the last are completely filled.

## Example:

```
    10
   /  \
  5    15
 / \   / \
3   7 12  18
```

---

# 7.2 Tree Traversals

Tree traversal refers to visiting all nodes in a tree in a specific order. Traversal methods include:

## 7.2.1 Inorder Traversal

- Visit the left subtree, the root, and then the right subtree.
- **Usage**: Retrieves nodes in sorted order for binary search trees.

**Example (C++):**

```
void inorder_traversal(Node* node) {
    if (node) {
        inorder_traversal(node->left);
        cout << node->data << endl;
        inorder_traversal(node->right);
    }
}
```

### 7.2.2 Preorder Traversal

- Visit the root, the left subtree, and then the right subtree.
- **Usage**: Used for creating tree copies.

### 7.2.3 Postorder Traversal

- Visit the left subtree, the right subtree, and then the root.
- **Usage**: Commonly used for deleting trees.

### 7.2.4 Level Order Traversal

- Visits nodes level by level.
- **Usage**: Suitable for breadth-first search (BFS) applications.

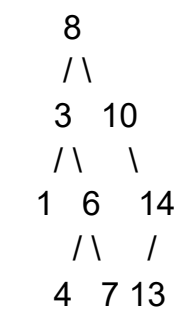
---

## 7.3 Binary Search Trees (BSTs)

Binary search trees ensure efficient searching, insertion, and deletion operations by maintaining an order property:

- All left descendants are less than the root.
- All right descendants are greater than the root.

### Example:



### Operations:

- **Search**: O(h), where h is the height of the tree.
- **Insertion and Deletion**: Similar to search but may require tree restructuring.

---

## 7.4 AVL Trees

AVL trees are self-balancing binary search trees where the height difference between the left and right subtrees of any node is at most one.

### Rotations:

- **Single Rotation**: Right or left rotation to restore balance.
- **Double Rotation**: Combination of left-right or right-left rotations.

### Example:

Insertion of a node may require:

- Right rotation for left-heavy imbalance.
- Left rotation for right-heavy imbalance.

---

# 7.5 Red-Black Trees

Red-Black trees are balanced binary search trees with the following properties:

1. Nodes are either red or black.
2. The root is always black.
3. No two consecutive red nodes exist.
4. Every path from the root to a null node has the same number of black nodes.

## Operations:

- Searching: O(logn)
- Insertion and Deletion: Maintain balance using rotations and color changes.

---

# 7.6 Splay Trees

Splay trees are self-adjusting binary search trees where recently accessed elements are moved to the root.

## Advantages:

- Frequently accessed nodes are quick to reach.
- Simple implementation.

## Example:

If node 20 is accessed frequently, it will be rotated to the root for faster future access.

---

# 7.7 B-Trees

B-Trees are self-balancing search trees optimized for systems that read and write large blocks of data.

### Properties:

1. Each node can have multiple keys and children.
2. Nodes are kept partially filled to maintain balance.

### Applications:

- Databases and file systems.

---

# 7.8 Unit Summary

In this unit, we explored:

- The structure and traversal of binary trees.
- Balanced trees such as AVL, Red-Black, and Splay trees.
- Advanced tree structures like B-Trees for efficient storage and retrieval.

---

# Practice Problems

1. Write a program to perform all four types of tree traversals on a binary tree.
2. Implement insertion and deletion operations in a binary search tree.
3. Create an AVL tree and demonstrate rotations for balancing.
4. Construct a Red-Black tree for a given set of keys.

# Quiz

1. What are the key differences between AVL trees and Red-Black trees?
2. In what scenarios are B-Trees preferred over other tree structures?

# Practical Questions

1. Design a tree structure for managing hierarchical data in a file system.
2. Compare the efficiency of Red-Black Trees and Splay Trees for dynamic data.

---

# Unit 8: Disjoint Sets

## Course Objectives

- To understand the concept and structure of disjoint sets.
- To learn the operations performed on disjoint sets and their efficiency.
- To explore the practical applications of disjoint sets in computer science.

## Course Outcomes

- Implement and utilize the union-find data structure.
- Analyze the efficiency of disjoint set operations.
- Apply disjoint sets to solve real-world problems in networks, graphs, and clustering.

---

## 8.0 Introduction

Disjoint sets are also known as union-find data structures, which are used to represent a collection of non-overlapping sets. They support two primary operations:

1. **Union**: Merges two sets into a single set.
2. **Find**: Determines a particular element belongs to which set.

Disjoint sets, widely used in algorithms for graph connectivity, network design, and clustering problems. In this unit, we will explore the structure, operations, and applications of disjoint sets.

---

# 8.1 Introduction to Disjoint Sets

Disjoint sets are a way of managing partitions of elements into groups where each element belongs to exactly one group.

## Key Concepts:

- **Set Representation**: Each set in Set Representation is represented by a unique identifier, often the root of a tree structure.
- **Parent Pointers**: Each element points to its parent, and the root points to itself.

## Example:

Set 1: {1, 2, 3} → Root: 1
Set 2: {4, 5} → Root: 4

## Visualization:

A disjoint set can be visualized as a forest of trees:

```
  1       4
 / \     / \
2   3   5   6
```

---

# 8.2 Union-Find Operations

The union-find data structure supports two fundamental operations:

## 8.2.1 Union Operation

Combines two disjoint sets into one.

- **Naive Approach**: Attach one tree as a subtree of another.
- **Optimized Approach**: Use techniques such as union by rank to keep the tree shallow.

**Example:**

Union of {1, 2, 3} and {4, 5} results in:

```
   1
  /|\
 2 3 4
       \
        5
```

## 8.2.2 Find Operation

Determines the representative (or root) of the set to which an element belongs.

- **Path Compression**: Flattens the tree structure during find operations to make future queries faster.

**Example (C++):**

```
int find(vector<int>& parent, int x) {
    if (parent[x] != x) {
        parent[x] = find(parent, parent[x]);  // Path compression
    }
    return parent[x];
}

void union_set(vector<int>& parent, vector<int>& rank, int x, int y) {
    int root_x = find(parent, x);
    int root_y = find(parent, y);

    if (root_x != root_y) {
```

```
        if (rank[root_x] > rank[root_y]) {
            parent[root_y] = root_x;
        } else if (rank[root_x] < rank[root_y]) {
            parent[root_x] = root_y;
        } else {
            parent[root_y] = root_x;
            rank[root_x]++;
        }
    }
}
```

---

# 8.3 Efficiency of Union-Find Operations

The efficiency of union-find operations depends on:

1. **Union by Rank**: Attaches smaller trees under larger ones.
2. **Path Compression**: Flattens the tree structure during find operations.

## Time Complexity:

- Each operation has an **amortized time complexity** of O(α(n)), where α(n) is the inverse Ackermann function, which grows very slowly.

## Performance Comparison:

| Operation | Without Optimization | With Optimization |
| --- | --- | --- |

| | | |
|---|---|---|
| Union | O(h) | O(α(n)) |
| Find | O(h) | O(α(n)) |

---

# 8.4 Practical Applications

Disjoint sets are broadly used in various fields of computer science:

## 8.4.1 Graph Connectivity

- **Kruskal's Algorithm**: Is used to find the minimum spanning tree of a graph.
- Detects cycles in undirected graphs.

## 8.4.2 Network Design

- Used in clustering networks to ensure efficient connections.

## 8.4.3 Image Processing

- Connected component labeling in binary images.

## Example: Kruskal's Algorithm

1. Sort all edges by weight.
2. Iterate through edges, adding them to the spanning tree if they don't form a cycle.
3. Use union-find to detect cycles efficiently.

---

# 8.5 Unit Summary

In this unit, we explored:

- The structure and operations of disjoint sets.
- Optimizations such as union by rank and path compression.
- Practical applications in graph algorithms, networks, and image processing.

---

# Practice Problems

1. Implement union and find operations with path compression.
2. Use disjoint sets to detect cycles in an undirected graph.
3. Implement Kruskal's Algorithm using the union-find structure.

# Quiz

1. What is the purpose of path compression in disjoint sets?
2. Explain the difference between naive union and union by rank.

# Practical Questions

1. Write a program to label connected components in a binary image using disjoint sets.
2. Analyze the impact of path compression on the efficiency of union-find operations.

---

# Module 3

## Unit 9: Sorting Algorithms

### Course Objectives

- To understand various sorting algorithms and their classifications.
- To analyze the performance of sorting techniques based on time and space complexity.
- To explore real-world applications of sorting algorithms in data processing and optimization.

## Course Outcomes

- Implement and compare different sorting algorithms.
- Analyze and optimize sorting operations based on algorithmic paradigms.
- Apply sorting techniques to solve practical problems in data management and processing.

---

# 9.0 Introduction

Sorting is a fundamental operation in computer science, used to arrange data in a specific order. Efficient sorting is crucial for optimizing search algorithms and enhancing data retrieval. This unit delves into various sorting techniques, categorized by algorithmic paradigms, and evaluates their performance and applications.

---

# 9.1 Brute Force Approach

Brute force sorting algorithms use straightforward methods to organize data but may lack efficiency for large datasets.

### 9.1.1 Sequential Search

- Searches for an element by iterating through each item in the list.
- **Complexity**: O(n) for unsorted data.

## 9.1.2 Bubble Sort

- Repeatedly swaps adjacent elements if they are in the wrong order.
- **Algorithm**:
    1. Compare adjacent elements.
    2. Swap if the first is greater than the second.
    3. Repeat for all elements until no swaps are needed.
- **Complexity**: $O(n^2)$.

**Example (C++):**

```
#include <iostream>
#include <vector>
using namespace std;

void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    for(int i=0;i < n; i++) {
        for(int j=0;j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}

int main() {
    vector<int> arr = {64, 23, 25, 57, 22, 61, 90};
    bubbleSort(arr);
    cout << "The Sorted array: ";
    for (int i = 0;i < arr.size(); i++) {
        cout << arr[i] << " ";
    }
```

```
    return 0;
}
```

### 9.1.3 Selection Sort

- Selects the smallest element from the unsorted portion and swaps it with the first unsorted element.
- **Complexity**: $O(n^2)$.

---

## 9.2 Decrease-and-Conquer Approach

This approach solves problems by reducing their size at each step.

### 9.2.1 Insertion Sort

- Builds the sorted array one element at a time by inserting elements into their correct position.
- **Algorithm**:
    1. Start with the second element.
    2. Compare it with previous elements and insert it into the correct position.
- **Complexity**: $O(n^2)$ in the worst case.

### 9.2.2 Binary Search

- Efficiently searches for an element in a sorted list by dividing the search space into halves.
- **Complexity**: O(logn).

---

# 9.3 Divide-and-Conquer Approach

This approach splits the problem into smaller subproblems, solves them recursively, and combines the results.

## 9.3.1 Quick Sort

- Selects a pivot and partitions the array such that elements less than the pivot are on the left and greater on the right.
- **Complexity**:
  - Best Case: O(nlogn).
  - Worst Case: $O(n^2)$ (when pivot selection is poor).

## 9.3.2 Merge Sort

- Divides the array into halves, sorts each half, and merges them.
- **Algorithm**:
  1. Split the array into halves.
  2. Recursively sort each half.
  3. Merge the sorted halves.
- **Complexity**: O(nlogn).

**Example (C++):**

```
// Implementation of Merge Sort Algorithm in C++
#include <iostream>
#include <vector>
using namespace std;

void mergeSort(vector<int>& arr);
void merge(vector<int>& arr, vector<int>& left, vector<int>& right);
```

```
int main() {
    vector<int> arr = {100, 37, 10, 6, 45, 1, 16};

    cout << "Before Sorting: ";
    for (int num : arr)
        cout << num << " ";

    mergeSort(arr);

    cout << "\nAfter Sorting: ";
    for (int num : arr)
        cout << num << " ";

    return 0;
}

void mergeSort(vector<int>& arr) {
    int size = arr.size();
    if (size < 2)
        return;

    int mid = size / 2;
    vector<int> left(arr.begin(), arr.begin() + mid);
    vector<int> right(arr.begin() + mid, arr.end());

    mergeSort(left);
    mergeSort(right);
    merge(arr, left, right);
}

void merge(vector<int>& arr, vector<int>& left, vector<int>& right) {
    int lSize = left.size(), rSize = right.size();
    int i = 0, j = 0, k = 0;
```

```
    while (i < lSize && j < rSize) {
        if (left[i] <= right[j]) {
            arr[k++] = left[i++];
        } else {
            arr[k++] = right[j++];
        }
    }

    while (i < lSize) {
        arr[k++] = left[i++];
    }

    while (j < rSize) {
        arr[k++] = right[j++];
    }
}
```

---

# 9.4 Transform-and-Conquer Approach

Transforms data to simplify sorting.

## 9.4.1 Heap Sort

- Uses a binary heap to sort elements.
- **Algorithm**:

1. Build a max heap.
2. Extract the largest element (root) and rebuild the heap.

- **Complexity**: O(nlogn).

---

# 9.5 Linear Sorting Algorithms

Efficient algorithms for specific scenarios.

## 9.5.1 Counting Sort

- Counts occurrences of each element and arranges them in order.
- **Complexity**: O(n+k), where k is the range of input values.

## 9.5.2 Radix Sort

- Processes digits of numbers starting from the least significant to the most significant.
- **Complexity**: $O(d \cdot (n+k))$, where d is the number of digits.

## 9.5.3 Bucket Sort

- Divides the input into buckets, sorts each bucket, and concatenates the results.
- **Complexity**: O(n) for uniform distributions.

---

# 9.6 Unit Summary

In this unit, we explored:

- Various sorting algorithms and their classifications.
- Algorithmic paradigms like brute force, divide-and-conquer, and transform-and-conquer.

- Practical sorting algorithms like Quick Sort, Merge Sort, and Heap Sort.

---

## Practice Problems

1. Implement Bubble Sort, Quick Sort, and Merge Sort in C++.
2. Analyze the time complexity of Radix Sort for a dataset of large integers.
3. Write a program to sort a list of names using Bucket Sort.

## Quiz

1. What is the worst-case complexity of Quick Sort?
2. Which sorting algorithm is best suited for nearly sorted data?

## Practical Questions

1. Write a program to find the top 10 largest elements in an array using Heap Sort.
2. Compare the performance of Insertion Sort and Merge Sort for small and large datasets.

---

# Unit 10: Hashing Techniques

## Course Objectives

- To understand the principles and techniques of hashing.
- To analyze and implement hashing-based search operations.

- To explore strategies for collision resolution and their efficiency.

## Course Outcomes

- Design and evaluate hash functions for various applications.
- Analyze the performance of hashing-based search operations.
- Apply collision resolution techniques to optimize data storage and retrieval.

---

# 10.0 Introduction

Hashing is a technique used to map data to a fixed-size representation called a hash value. It is widely employed in data structures such as hash tables, enabling efficient storage and retrieval operations. By distributing data across a range of indices in a table, hashing ensures that operations like search, insert, and delete are performed efficiently.

This unit introduces the foundational concepts of hashing, explores different hash functions, and addresses challenges like collisions and their resolution. The focus is on understanding how hashing optimizes search operations and its applications in real-world scenarios.

---

# 10.1 Hash Functions

A hash function is a mathematical formula that converts input data into a fixed-size hash value. An ideal hash function distributes data uniformly across the hash table to minimize collisions.

## Properties of a Good Hash Function

- **Deterministic**: The same input always produces the same hash value.
- **Uniform Distribution**: Hash values are evenly distributed to minimize clustering.
- **Minimizes Collisions**: Reduces the likelihood of two inputs producing the same hash value.
- **Fast Computation**: Efficient to compute for real-time operations.

## Examples of Hash Functions

1. **Division Method**:
    - Formula: $h(k)=k \bmod m$
    - Here, k is the key, and m is the size of the hash table.
    - Example:
      Key: 15, Hash Table Size: 7
      h(15) = 15 % 7 = 1

2. **Multiplication Method**:
    - Formula: $h(k)=\lfloor m(k \times A \bmod 1) \rfloor$
    - A is a constant (e.g., A=0.618).

3. **Folding Method**:
    - Break the key into parts and combine them.
    - **Example**: Key = 123456, split into 123 and 456, then sum them.

4. **Mid-Square Method**:
    - Square the key and extract the middle digits.
    - **Example**:
      Key: 56
      Square: 3136
      Extract: Middle two digits = 13

# 10.2 Collisions in Hashing

Collisions occur when two keys produce the same hash value. Effective collision resolution techniques are crucial for maintaining the performance of hashing-based systems.

## Types of Collisions

1. **Primary Clustering**: Multiple keys map to the same or adjacent slots.
2. **Secondary Clustering**: Keys with the same hash value create clusters.
3. **Chaining**: Storing multiple elements in the same slot using linked lists.

## Collision Resolution Techniques

### 1. Open Addressing

- **Linear Probing**:
  - Incrementally check the next slot until an empty one is found.
  - Formula: $h'(k,i)=(h(k)+i) \bmod m$
- **Quadratic Probing**:
  - Check slots using a quadratic function.
  - Formula: $h'(k,i)=(h(k)+i^2) \bmod m$

- **Double Hashing**:

- Use a secondary hash function to compute step size.
- Formula: $h'(k,i)=(h_1(k)+i\times h_2(k)) \bmod m$

**2. Chaining**

- Store all elements with the same hash value in a linked list.

**3. Rehashing**

- When the load factor exceeds a threshold, resize the table and reapply the hash function.

## Example: Resolving Collisions

Given a hash table of size 7 and keys [10, 22, 31, 40, 59], resolve collisions using linear probing.

| Key | Hash Value (h(k)) | Placement |
|---|---|---|
| 10 | 10 % 7 = 3 | Slot 3 |
| 22 | 22 % 7 = 1 | Slot 1 |
| 31 | 31 % 7 = 3 | Slot 4 (next empty slot) |
| 40 | 40 % 7 = 5 | Slot 5 |
| 59 | 59 % 7 = 3 | Slot 6 (next empty slot) |

---

# 10.3 Analysis of Search Operations

The efficiency of search operations in hashing depends on factors such as table size, load factor, and collision resolution strategy.

## Key Metrics

- **Load Factor (α)**: Ratio of the number of elements to the size of the hash table.
    - Formula: α=n/m, where n = number of elements, m = table size.
    - Higher α increases the likelihood of collisions.

- **Search Time**:
    - **Best Case**: O(1) (direct access without collisions).
    - **Average Case**: Depends on the collision resolution strategy.
    - **Worst Case**: O(n) (all elements in the same slot).

## Practical Considerations

- **Trade-offs**: Smaller tables save memory but increase collisions, while larger tables minimize collisions at the cost of memory usage.
- **Applications**: Hashing is widely used in dictionaries, databases, and caching systems.

---

# 10.4 Unit Summary

In this unit, we explored hashing techniques, focusing on hash functions, collision resolution, and the analysis of search operations. Hashing is a foundational concept in computer science, enabling efficient data storage and retrieval. By mastering these techniques, you can design robust systems for real-world applications.

---

# Practice Problems

1. Design a hash function for a hash table of size 10 to store keys [11, 21, 31, 41].
2. Implement a C++ program to demonstrate linear probing for collision resolution.
3. Analyze the time complexity of double hashing.

# Quiz

1. What are the properties of a good hash function?
2. Explain the difference between chaining and open addressing.

# Practical Questions

1. Implement a hash table with chaining using linked lists in C++.
2. Compare and contrast linear probing and double hashing in terms of performance.

---

# Module 4

## Unit 11: Graph Algorithms

# Course Objectives

- To understand the fundamental concepts of graph representations and their applications.
- To learn traversal techniques and their role in exploring graph structures.
- To explore key algorithms for Minimum Spanning Trees, shortest paths, and dynamic programming applications in graphs.

# Course Outcomes

- Represent and manipulate graphs effectively in computational scenarios.
- Implement traversal algorithms for problem-solving.
- Apply graph algorithms to solve real-world problems such as network design and pathfinding.

---

# 11.0 Introduction

Graphs are a fundamental data structure that models relationships between objects. Used across diverse domains such as social networks, transportation systems, and communication networks, graph algorithms provide efficient ways to analyze and navigate these structures. This unit explores graph representations, traversal techniques, and core algorithms for spanning trees, shortest paths, and dynamic programming applications in graphs.

---

# 11.1 Graph Representations

Graphs can be represented in various ways, depending on their structure and intended use. The two most common representations are:

1. **Adjacency Matrix**:
   - A 2D array where each cell `(i, j)` is `1` if there is an edge from vertex `i` to vertex `j`, and `0` otherwise.
   - **Advantages**:
     - Fast lookup for edge existence: O(1).
     - Easy to implement for dense graphs.
   - **Disadvantages**:
     - High memory usage for sparse graphs.

2. **Adjacency List**:
   - Each vertex maintains a list of all its adjacent vertices.
   - **Advantages**:
     - Space-efficient for sparse graphs.
     - Easier traversal of neighbors.
   - **Disadvantages**:
     - Slower edge existence check: O(n).

## Example:

For a graph with vertices `{A, B, C}` and edges `{(A, B), (B, C)}`:

**Adjacency Matrix:**

| | **A** | **B** | **C** |
|---|---|---|---|
| A | 0 | 1 | 0 |
| B | 0 | 0 | 1 |
| C | 0 | 0 | 0 |

**Adjacency List:**

- A: [B]
- B: [C]
- C: []

---

# 11.2 Graph Traversal Techniques: BFS and DFS

Graph traversal algorithms explore nodes and edges systematically. The two primary techniques are:

## Breadth-First Search (BFS)

- **Approach**:
  - Explore all neighbors of the current vertex before moving to the next level.
  - Implemented using a queue.
- **Time Complexity**: O(V + E), where V is vertices and E is edges.
- **Applications**:
  - Shortest path in unweighted graphs.
  - Finding connected components.

## Depth-First Search (DFS)

- **Approach**:
  - Explore as far as possible along each branch before backtracking.
  - Implemented using a stack (explicit or recursion).
- **Time Complexity**: O(V + E).
- **Applications**:
  - Detecting cycles in graphs.
  - Topological sorting.

# 11.3 Minimum Spanning Trees (MST): Prim's and Kruskal's Algorithms

## Prim's Algorithm

- **Objective**: Build an MST by adding the smallest edge connecting a vertex in the tree to a vertex outside.
- **Approach**:
  - Use a priority queue to select the smallest edge.
- **Time Complexity**: O(E log V) with a priority queue.

## Kruskal's Algorithm

- **Objective**: Build an MST by adding edges in increasing order of weight, ensuring no cycles.
- **Approach**:
  - Use a union-find structure to detect cycles.
- **Time Complexity**: O(E log E).

---

# 11.4 Single Source Shortest Paths: Dijkstra's Algorithm

- **Objective**: Find the shortest path from a source vertex to all other vertices in a graph with non-negative edge weights.
- **Approach**:
  - Maintain a priority queue to process vertices with the smallest known distance.
- **Time Complexity**: O(V + E log V) with a priority queue.

### Example:

Given the graph:

- Vertices: `{A, B, C, D}`
- Edges: `{(A, B, 1), (B, C, 2), (A, C, 2), (C, D, 1)}`

Dijkstra's Algorithm calculates the shortest distances as:

- From `A`: `{A: 0, B: 1, C: 2, D: 3}`

---

# 11.5 Dynamic Programming in Graphs

Dynamic programming provides efficient solutions to complex graph problems by breaking them into smaller subproblems. Examples include:

1. **All-Pairs Shortest Path**:
   - Floyd-Warshall Algorithm: Computes shortest paths between all pairs of vertices in $O(V^3)$.

2. **Longest Path in a Directed Acyclic Graph (DAG)**:
   - Use topological sorting to calculate the longest path efficiently.

---

# 11.6 Unit Summary

This unit explored fundamental graph concepts, representations, and traversal techniques. We covered key algorithms such as Prim's and Kruskal's for MST, Dijkstra's for shortest paths, and dynamic programming applications like Floyd-Warshall. These concepts form the foundation for solving advanced computational problems in various domains.

---

# Practice Problems

1. Implement BFS and DFS for a given graph.
2. Compare Prim's and Kruskal's algorithms with an example graph.
3. Use Dijkstra's Algorithm to find the shortest path in a weighted graph.

# Quiz

1. What are the primary differences between adjacency matrix and adjacency list representations?
2. How does DFS detect cycles in a graph?

# Practical Questions

1. Implement Kruskal's Algorithm using the union-find technique.
2. Analyze the time complexity of BFS and DFS with an example graph.

---

# Unit 12: Algorithmic Design Techniques

## Course Objectives

- To understand key algorithmic design paradigms.
- To apply design techniques to solve complex computational problems.
- To analyze and compare algorithmic approaches.

## Course Outcomes

- Implement and optimize algorithms using different design techniques.
- Evaluate the efficiency of algorithms designed with various paradigms.
- Solve real-world problems using appropriate algorithmic strategies.

---

## 12.0 Introduction

Algorithmic design techniques form the core of problem-solving in computer science, offering structured methods to develop efficient algorithms. These techniques help in addressing a wide range of computational challenges by leveraging specific paradigms suited to the problem's nature.

In this unit, we explore:

- **Greedy Algorithms**: Optimizing solutions step-by-step.
- **Divide-and-Conquer**: Breaking down problems into manageable subproblems.
- **Dynamic Programming**: Solving complex problems by storing intermediate results to avoid redundancy.

By understanding these techniques, you can choose the most appropriate approach for a given problem and analyze its efficiency.

---

# 12.1 Greedy Algorithms

Greedy algorithms make locally optimal choices at each step with the hope of finding a global optimum. This approach is particularly effective for problems that exhibit the *greedy-choice property* and *optimal substructure*.

## Key Characteristics:

- **Greedy-Choice Property**: A global solution can be constructed by selecting local optima.
- **Optimal Substructure**: A problem's optimal solution can be composed of optimal solutions to its subproblems.

## Examples of Greedy Algorithms:

1. **Activity Selection Problem**:
    - Given a set of activities with start and end times, select the maximum number of activities that don't overlap.
    - **Algorithm**:
        1. Sort activities by their finish time.
        2. Select the first activity and iteratively choose the next non-overlapping activity.
    - **Complexity**: O(nlogn) (due to sorting).

2. **Huffman Coding**:
    - Constructs an optimal prefix code for data compression.
    - **Algorithm**:

1. Create a priority queue with all characters and their frequencies.
2. Extract the two smallest frequencies, combine them, and reinsert into the queue.
3. Repeat until the queue contains a single node.

- **Complexity**: O(nlogn).

---

# 12.2 Divide-and-Conquer

This paradigm involves dividing a problem into smaller subproblems, solving each recursively, and then combining their solutions.

## Key Steps:

1. **Divide**: Break the problem into smaller, independent subproblems.
2. **Conquer**: Solve each subproblem recursively.
3. **Combine**: Merge the solutions of the subproblems to form the overall solution.

## Examples of Divide-and-Conquer Algorithms:

1. **Merge Sort**:
   - Divides the array into halves, sorts each recursively, and merges the sorted halves.
   - **Complexity**: O(nlogn).

2. **Quick Sort**:
   - Selects a pivot, partitions the array around it, and sorts the partitions recursively.

    - **Complexity**: Average case O(nlogn); Worst case $O(n^2)$.

3. **Binary Search**:
    - Searches for an element in a sorted array by dividing the search range in half.
    - **Complexity**: O(logn).

### Advantages and Applications:

- Efficient for divide-and-conquer-friendly problems like sorting and searching.
- Often used in parallel computing for independent subproblem processing.

---

# 12.3 Dynamic Programming

Dynamic Programming (DP) solves problems by breaking them into overlapping subproblems and storing their solutions to avoid redundant computations. It is particularly useful for optimization problems.

### Key Characteristics:

- **Overlapping Subproblems**: Subproblems are solved multiple times in a naive approach.
- **Optimal Substructure**: The solution to a problem depends on the solutions to its subproblems.

### Steps to Design a DP Solution:

1. Identify if the problem has overlapping subproblems and optimal substructure.
2. Define the state variables to represent the problem.
3. Derive a recurrence relation to transition between states.
4. Implement the solution iteratively or recursively with memoization.

## Examples of Dynamic Programming:

1. **Fibonacci Numbers**:
   - **Recursive Approach**:

```
#include <iostream>
#include <unordered_map>
using namespace std;

long long fib(int n, unordered_map<int, long long>& memo) {
if (n <= 1) {
        return n;
    }
    if (memo.find(n) == memo.end()) {
        memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
    }
    return memo[n];
}

int main() {
    unordered_map<int, long long> memo;
    int n = 10; // Example input
    cout << "Fibonacci number " << n << " is: " << fib(n, memo)
<< endl;
    return 0;
}
```

   - **Complexity**: O(n).

2. **Knapsack Problem**:

- Maximizes the value of items in a knapsack without exceeding the weight limit.
- **Algorithm**:
  1. Define dp[i][w] as the maximum value achievable with the first i items and weight limit w.

  2. **Transition Relation:**
     $dp[i][w]=dp[i-1][w]$ if item i is excluded.
     $dp[i][w]=\max(dp[i-1][w],dp[i-1][w-wt[i]]+val[i])$
     if item i is included.

- **Complexity**: $O(n \cdot W)$, where n is the number of items and W is the weight limit.

---

# 12.4 Unit Summary

In this unit, we explored three fundamental algorithmic design paradigms:

- **Greedy Algorithms**: Effective for problems with the greedy-choice property and optimal substructure.
- **Divide-and-Conquer**: Ideal for problems that can be broken into independent subproblems.
- **Dynamic Programming**: Handles problems with overlapping subproblems and optimal substructure.

Understanding these techniques equips you with the tools to tackle a wide range of computational challenges effectively.

---

# Practice Problems

1. Solve the Activity Selection Problem using a greedy approach.
2. Implement Merge Sort and analyze its complexity.
3. Solve the 0/1 Knapsack Problem using dynamic programming.

# Quiz

1. What are the key characteristics of a problem suited for the greedy approach?
2. Explain the difference between Divide-and-Conquer and Dynamic Programming with examples.
3. Write a recurrence relation for the Fibonacci sequence.

# Practical Questions

1. Implement Huffman Coding for data compression.
2. Compare and contrast the performance of Quick Sort and Merge Sort.
3. Solve the Longest Common Subsequence problem using dynamic programming.

---

# Module 5

# Unit 13: Tractability and Computability

## Course Objectives

- To understand the principles of computability and tractability in algorithms.
- To analyze and differentiate between computational complexity classes.
- To explore the significance of P, NP, NP-complete, and NP-hard problems in computer science.

## Course Outcomes

- Understand the limitations of algorithmic computation.
- Classify problems based on their computational complexity.
- Apply the concepts of computational classes to assess algorithm feasibility.

---

## 13.0 Introduction

Tractability and computability are foundational concepts in theoretical computer science that define the boundaries of what can be solved efficiently by algorithms. Tractability pertains to whether a problem can be solved within a reasonable amount of time, while computability examines whether a problem can be solved at all, regardless of time constraints.

This unit focuses on understanding these concepts, classifying problems into computational complexity classes, and exploring the implications of these classifications on real-world problem-solving.

# 13.1 Computability of Algorithms

Computability determines whether a problem can be solved algorithmically. It is a fundamental concept rooted in the work of Alan Turing, who introduced the Turing Machine model as a mathematical abstraction for computation.

## Key Concepts

- **Decidable Problems**:
  - Problems for which an algorithm can always provide a correct yes/no answer within a finite amount of time.
  - Example: Determining whether a number is prime.

- **Undecidable Problems**:
  - Problems for which no algorithm can guarantee a solution for all possible inputs.
  - Example: The Halting Problem, which asks whether a given program will halt or run indefinitely.

## Example: The Halting Problem

The Halting Problem is a classical example of an undecidable problem. Given a program and its input, it is impossible to construct an algorithm that determines whether the program will terminate.

**Implications:**

- Highlights the limitations of computational systems.
- Emphasizes the need for theoretical analysis when designing algorithms.

---

# 13.2 Computability Classes: P, NP, NP-complete, NP-hard

The classification of problems into computational classes helps determine the feasibility of solving them within practical constraints.

## 13.2.1 Class P (Polynomial Time)

- Problems that can be solved in polynomial time by a deterministic algorithm.
- Example: Sorting an array (O(n log n)).
- **Significance**: Problems in P are considered efficiently solvable.

## 13.2.2 Class NP (Nondeterministic Polynomial Time)

- Problems for which a solution can be verified in polynomial time, even if finding the solution might not be efficient.
- Example: The Traveling Salesman Problem (TSP).
- **Significance**: NP includes problems with solutions that are hard to compute but easy to verify.

## 13.2.3 NP-complete Problems

- Subset of NP problems that are as hard as any problem in NP.
- If a polynomial-time solution exists for one NP-complete problem, all NP problems can be solved in polynomial time.
- Example: The Satisfiability Problem (SAT).
- **Significance**: Used to identify problems with practical computational challenges.

### 13.2.4 NP-hard Problems

- Problems that are at least as hard as the hardest problems in NP but may not be in NP themselves.
- Example: The Halting Problem.
- **Significance**: These problems are used to explore the boundaries of computational feasibility.

### Diagram: Relationship Between P, NP, NP-complete, and NP-hard

A Venn diagram depicting the relationship between these classes can clarify their distinctions and overlaps.

---

# 13.3 Unit Summary

In this unit, we explored the concepts of tractability and computability, focusing on the limitations of algorithms and their classification into computational complexity classes. By understanding P, NP, NP-complete, and NP-hard problems, you can assess the feasibility of algorithmic solutions and their implications for practical applications.

---

# Practice Problems

1. Define the difference between P and NP classes.
2. Provide an example of an undecidable problem and explain why it is undecidable.
3. Explain the significance of NP-complete problems in algorithm design.

# Quiz

1. What is the Halting Problem, and why is it significant in computability theory?

# Practical Questions

1. Analyze whether the Traveling Salesman Problem belongs to the class P, NP, NP-complete, or NP-hard.
2. Implement a solution for a problem in class P and measure its time complexity.

---

# Unit 14: Advanced Algorithmic Techniques

## Course Objectives

- To understand advanced techniques for solving complex computational problems.
- To learn the principles of backtracking, branch-and-bound, approximation, and randomized algorithms.
- To explore the practical applications of these techniques in real-world problem-solving.

## Course Outcomes

- Develop and implement algorithms using advanced methodologies.
- Analyze the efficiency and practicality of various algorithmic techniques.
- Apply appropriate techniques to optimize solutions for complex problems.

---

## 14.0 Introduction

As computational problems become increasingly complex, advanced algorithmic techniques provide tools to tackle these challenges effectively. This unit delves into four significant techniques:

- **Backtracking**: A recursive strategy for solving problems by exploring all possible solutions.
- **Branch-and-bound**: A systematic approach to solving optimization problems.

- **Approximation Algorithms**: Strategies for finding near-optimal solutions when exact solutions are computationally infeasible.
- **Randomized Algorithms**: Algorithms that utilize randomization to simplify problem-solving or improve performance.

---

# 14.1 Basics of Backtracking

Backtracking is a recursive technique used for solving combinatorial problems. It involves building a solution incrementally, abandoning a solution as soon as it is determined that it cannot lead to a feasible solution.

## Key Concepts

- **Recursive Exploration**:
  - Explore each possible solution recursively.
  - Abandon paths that do not satisfy constraints (pruning).

- **Applications**:
  - N-Queens Problem
  - Sudoku Solver
  - Subset Sum Problem

## Example: N-Queens Problem

Place N queens on an N×N chessboard such that no two queens threaten each other.

**Algorithm:**

1. Start in the leftmost column.
2. If all queens are placed, return success.
3. Try placing the queen in each row of the current column.
4. If placement is safe, recursively place the queen in the next column.

5. If placing leads to no solution, backtrack and try the next row.

**C++Code:**

```
#include <iostream>
#include <vector>
using namespace std;

bool isSafe(vector<vector<int>>& board, int row, int col) {
    for (int i = 0; i < col; i++) {
        if (board[row][i] == 1)
            return false;
    }
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 1)
            return false;
    }
    for (int i = row, j = col; i < board.size() && j >= 0; i++, j--) {
        if (board[i][j] == 1)
            return false;
    }
    return true;
}

bool solveNQueens(vector<vector<int>>& board, int col) {
    if (col >= board.size())
        return true;

    for (int i = 0; i < board.size(); i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
            if (solveNQueens(board, col + 1))
                return true;
            board[i][col] = 0;
        }
```

```
    }
    return false;
}

int main() {
    int n = 8; // Example size for the chessboard
    vector<vector<int>> board(n, vector<int>(n, 0));

    if (solveNQueens(board, 0)) {
        cout << "Solution:\n";
        for (auto& row : board) {
            for (int cell : row) {
                cout << (cell ? "Q " : ". ");
            }
            cout << endl;
        }
    } else {
        cout << "No solution exists." << endl;
    }

    return 0;
}
```

---

# 14.2 Branch-and-Bound Methodology

Branch-and-bound is a systematic technique for solving optimization problems. It involves dividing the problem into smaller subproblems and using bounds to eliminate suboptimal solutions.

## Key Concepts

- **Branching**:
    - Divide the problem into smaller subproblems.

- **Bounding**:
  - Calculate bounds for subproblems to eliminate infeasible paths.

### Applications

- Knapsack Problem
- Traveling Salesman Problem (TSP)

### Example: Knapsack Problem

Find the maximum value that can be obtained by selecting items within a weight limit.

---

## 14.3 Approximation Algorithms

Approximation algorithms are used for NP-hard problems where finding an exact solution is computationally infeasible. These algorithms provide solutions that are close to the optimal.

### Key Concepts

- **Approximation Ratio**: Measures the quality of the approximation.
- **Greedy Approaches**: Commonly used in approximation algorithms.

### Applications

- Vertex Cover
- Traveling Salesman Problem (TSP)
- Set Cover Problem

### Example: Vertex Cover

Find a minimum subset of vertices such that every edge in the graph is incident to at least one vertex.

# 14.4 Randomized Algorithms

Randomized algorithms incorporate random choices in their logic, often simplifying problem-solving or improving efficiency.

## Key Concepts

- **Types**:
  - Las Vegas Algorithms: Always produce a correct result, with running time varying.
  - Monte Carlo Algorithms: May produce incorrect results with a small probability.

- **Applications**:
  - Quick Sort (randomized pivot selection).
  - Randomized Graph Traversal.

## Example: Randomized Quick Sort

1. Choose a pivot element randomly.
2. Partition the array around the pivot.
3. Recursively sort the partitions.

## 14.5 Unit Summary

In this unit, we explored advanced algorithmic techniques including backtracking, branch-and-bound, approximation algorithms, and randomized algorithms. These methodologies enable solving complex problems more effectively and efficiently. By understanding their principles and applications, you can select the most suitable approach for a given problem.

---

# Practice Problems

1. Implement the N-Queens problem using backtracking.
2. Solve the 0/1 Knapsack Problem using branch-and-bound.
3. Write a randomized algorithm for finding the median of an array.

# Quiz

1. What is the difference between Las Vegas and Monte Carlo algorithms?
2. Define the term "approximation ratio" and explain its significance.

# Practical Questions

1. Apply branch-and-bound to solve the Traveling Salesman Problem for a small graph.
2. Implement and compare the efficiency of a randomized and a deterministic Quick Sort algorithm.

---