UNIT 3

Algorithms and Problem Solving

Student Learning Outcomes

By the end of this chapter, students will be able to:

- Describe and categorize different types of computational problems.
- Explain the importance of algorithms in problem-solving.
- Apply the generate-and-test method to solve computational problems.
- Differentiate between solvable and unsolvable problems.
- Understand problem complexity and categorize problems into P, NP, NP-Hard, NP-Complete.
- Identify common computational problems like sorting and searching.
- Apply algorithm design techniques such as divide and conquer, greedy methods, and dynamic programming.
- Implement and compare algorithms such as Bubble Sort, Binary Search, BFS, and DFS.
- Evaluate algorithms in terms of efficiency and scalability.
- Develop algorithmic thinking to solve problems systematically.

Introduction

Understanding algorithms is essential not only for computer science but also for everyday problem-solving. We will start by learning what computational problems are and how to describe them clearly. Then, we will look at different types of algorithms and how they can help us solve various kinds of problems. We will also discuss how to measure the efficiency of algorithms to find the best solutions.

3.1 Understanding Computational Problems

A computational problem is a challenge that can be solved through a computational process, which involves using an algorithm, i.e., a set of step-by-step instructions that a computer can execute.

- **Input:** The data or information given to the algorithm at the beginning of the problem.
- **Process:** The steps or rules (i.e. the algorithm) that are applied to the input to generate the output.
- **Output:** The solution or result produced by the algorithm after processing the input.

3.1.1 Characterizing Computational Problems

To solve a problem computationally, we need to understand its characteristics. This involves identifying the inputs, the desired outputs, and the process needed to transform the inputs into outputs.

3.1.1.1 Classifying Computational Problems

Computational problems can be classified into different categories based on their characteristics and the methods required to solve them. Some common classifications include:

- **Decision Problems:** Problems where the output is a simple "yes" or "no".
- **Search Problems:** Problems where the task is to find a solution or an item that meets certain criteria.
- **Optimization Problems:** Problems where the goal is to find the best solution according to some criteria.
- **Counting Problems:** Problems where the objective is to count the number of ways certain conditions can be met.

3.1.1.2 Well-defined vs. ill-defined Problems

Problems can also be categorized based on how clearly they are defined:

- **Well-defined Problems:** These problems have clear goals, inputs, processes, and outputs. For instance, the problem of determining if a number is even, is a well-defined problem because it has a clear goal (determine if the number is even), clear input (a single integer), a clear process (check if the number is divisible by 2), and a clear output (even or odd) as shown in Figure 3.1.
- **Ill-defined Problems:** These problems lack clear definitions or may have ambiguous goals and requirements. For instance, consider a project aimed at "How to reduce poverty in Pakistan". This goal is vague and broad.

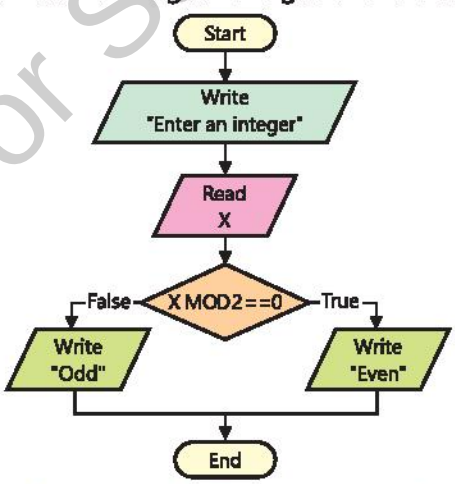

Figure 3.1: Finding an Even Number

3.2 Algorithms for Problem Solving

Algorithms are step-by-step procedures for solving problems, much like a recipe provides steps for cooking a dish. Understanding algorithms is essential because they provide the logic behind software operations, allowing us to solve complex problems, optimize performance, and ensure accuracy in various applications.

DO YOU KNOW?

The Google search engine uses a complex algorithm called PageRank to determine the relevance of web pages. This algorithm considers various factors, including the number of links to a page and the quality of those links, to rank pages in search results.

Tidbits

When learning about algorithms, try to relate them to real-life tasks you already know. This will help you understand how algorithms work and why they are important.

3.2.1 Generate-and-Test Method

This method works by generating potential solutions to a problem and then testing each one to determine if it meets the required conditions. The process continues until a satisfactory solution is found or all possible solutions have been exhausted.

The Generate-and-Test method is particularly useful in scenarios where:

- The problem space is small, making it feasible to generate and test all possible solutions.
- There is no clear strategy for finding a solution, and an exhaustive search is necessary.
- Heuristics or rules can be applied to reduce the number of generated solutions, making the process more efficient.

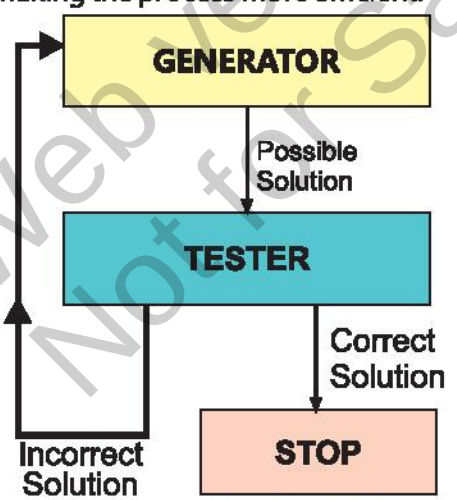

Figure 3.2: Flowchart of the Generate and Test Method

DO YOU KNOW?

The Generate-and-Test method is often used in AI applications, such as game playing and problem-solving, where the solution space is large, and the best approach is to try different possibilities until one works!

3.3 Problem Solvability and Complexity

Problem solvability and complexity helps us determine whether a problem can be solved using an algorithm and, if so, how efficiently it can be solved.

3.3.1 Solvable vs. Unsolvable Problems

In computer science, problems are classified as solvable or unsolvable based on whether there exists an algorithm that can provide a solution.

Solvable Problems: A problem is considered solvable if an algorithm can solve it within a finite amount of time. These problems have clearly defined inputs and outputs, and there is a step-by-step procedure to reach the solution.

Example: Calculating the greatest common divisor (GCD) of two integers is a solvable problem. The Euclidean algorithm provides a clear and finite method to determine the GCD, making it a classic example of a solvable problem.

Unsolvable Problems: On the other hand, a problem is unsolvable if no algorithm can be created that will provide a solution in all cases. These problems do not have a general procedure that can guarantee a solution for every possible input.

Example: The Halting Problem is a famous example of an unsolvable problem. It involves determining whether a given program will eventually halt (finish running) or continue to run forever. Alan Turing proved that no general algorithm can solve the Halting Problem for all possible program-input pairs, making it a fundamental example of an unsolvable problem.

Tidbits

When tackling complex problems, it's essential to first determine whether the problem is solvable. This saves time and resources by ensuring you are working on a problem that can be resolved using an algorithm.

3.3.2 Tractable vs. Intractable Problems

Once a problem is determined to be solvable, the next consideration is its computational complexity—how efficiently it can be solved. Problems are categorized as tractable or intractable based on the resources required (time and space) to solve them.

Tractable Problems: A problem is considered tractable if it can be solved in polynomial time, denoted as *P*. Polynomial time means that the time taken to solve the problem increases at a manageable rate (as a polynomial function) relative to the size of the input. Tractable problems are considered "efficiently solvable."

Example: Sorting a list of numbers using algorithms such as Merge Sort or Quick Sort is a tractable problem because these algorithms have a polynomial time complexity of $O(n \log n)$, where n is the number of elements in the list.

Intractable Problems: Intractable problems are those that require super-polynomial time to solve, often growing exponentially with the size of the input. These problems are impractical to solve for large inputs because the time required becomes unmanageable.

Example: The Traveling Salesman Problem (TSP), where the goal is to find the shortest possible route that visits a set of cities and returns to the origin, is an example of an intractable problem. The problem is NP-hard, meaning that as the number of cities increases, the number of possible routes grows factorially, making it infeasible to solve exactly for large instances.

3.3.3 Complexity Classes (P, NP, NP-hard, NP-complete)

Understanding the complexity of problems involves classifying them into different categories based on their solvability and the time required to solve them.

3.3.3.1 Class P

Class **P** refers to a category of problems that can be solved efficiently by a computer. In simpler terms, these are problems where a computer can find a solution quickly, even as the size of the problem grows:

problem grows.

Example: Let's consider a simple problem: sorting a list of numbers.

Suppose you have the following list:

[4, 1, 3, 2, 5]

The goal is to arrange these numbers in ascending order:

[1, 2, 3, 4, 5]

The time required to sort the list grows at a manageable rate as the list size increases. For example, going from 5 numbers to 10 numbers will increase the time, but it remains within a reasonable limit.

3.3.3.2 Class NP

Class **NP** refers to a category of problems for which, if a solution is given, it can be checked quickly by a computer. These are problems where verifying a proposed solution is easy, but finding that solution might be difficult and time-consuming.

Example:

Consider a common example of solving a Sudoku puzzle. In Sudoku, you fill a 9 x 9 grid with numbers so that each row, column, and 3 x 3 sub grid contains all digits from 1 to 9 exactly once as shown in Figure 3.3.

9	1	3				5		
6		7					2	4
	5			8			7	
	7	9						
		2		9			4	3
					4		9	
	4				1	9		
7		6			9			5
		1			6	4		7

Figure 3.3: A simple Sudoku Puzzle

3.3.3.3 Class NP-Hard

NP-hard problems are a class of problems that are at least as difficult as the hardest problems in Non-deterministic Polynomial time (NP). Solving an NP-hard problem is challenging, and no efficient algorithm is known for finding a solution.

Example:

A well-known example of an NP-hard problem is the Traveling Salesman Problem (TSP), we discussed in Section 3.3.2.

3.3.3.4 NP- Complete

NP-Complete problems form a special subset of NP problems. They are both in NP and as hard as the hardest problems in NP. This means that these problems are particularly challenging, and if you can solve one NP-Complete problem efficiently, you can solve all NP-problems efficiently.

Example: A classic example of an NP-Complete problem is the Knapsack Problem.

In the Knapsack Problem, you have a knapsack with a maximum weight capacity and a set of items, each with a weight and a value. The goal is to determine the most valuable combination of items to put in the knapsack without exceeding its weight capacity.

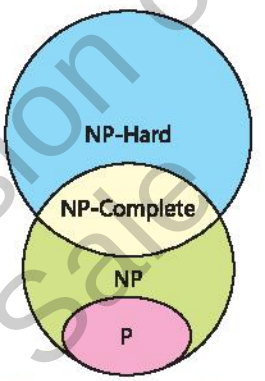

Figure 3.4 Venn diagram of the complexity classes P, NP, NP-hard, and NP-complete.

Figure 3.4 shows a Venn diagram illustrating the complexity classes P, NP, NP-hard, and NP-complete. It visually represents the relationships among these classes, highlighting how some problems can be solved efficiently, while others pose significant challenges in computational theory.

The question of whether P equals NP is one of the most important unsolved problems in computer science. It has significant implications for cryptography, algorithm design, and the overall understanding of computational complexity.

3.4 Algorithm Analysis

Algorithm analysis is the process of determining the computational complexity of algorithms, which includes their time and space complexity. This analysis helps predict the algorithm's performance and is crucial for selecting the best algorithm for a particular task.

3.4.1 Time Complexity

Time complexity is a measure of how the running time of an algorithm increases as the size of the input data grows. It helps us understand how efficiently an algorithm performs when dealing with larger amounts of data.

Example: Consider the task of sorting a list of numbers. If the list contains only a few numbers, the task might be quick. However, as the volume of numbers increases, the time required to sort them also increases. Time complexity allows us to predict how this runtime scale with the input size.

3.4.1.1 Big *O* Notation

Big *O* notation is a mathematical way to describe the time complexity of an algorithm. It provides an upper bound on the time an algorithm will take to complete as the input size grows. This notation helps in comparing the efficiency of different algorithms by giving a clear picture of their performance.

How Big *O* Notation Works

Big *O* notation uses symbols to describe how the runtime of an algorithm changes with the size of the input. Here are some common examples:

- ***O*(1)-Constant Time:** The runtime remains the same regardless of the input size.
- ***O*(n)-Linear Time:** The runtime grows linearly with the input size. For example, suppose there are *n* students in a college, each with a unique student ID. If we need to find a specific student by searching through the list of all *n* students, the time it takes depends on the number of students, hence it is linear.
- ***O*(n^2)-Quadratic Time:** The runtime increases with the square of the input size. For instance, consider a scenario where *n* students in a college are participating in a programming competition, and we want to compare the performance of each pair of students to determine the best team. To do this, we must compare each student with every other student. The number of comparisons required is the sum of the first *n* - 1 integers, which can be approximated as *n*(*n* -1), a quadratic growth.
- ***O*(*log n*)-Logarithmic Time:** The runtime grows logarithmically, meaning it increases very slowly relative to the input size. Imagine you are thinking a number between 1 and 100, and . I want to guess it. I can only ask yes / no questions like", Is it greater than 50? ", " Is it less than 25?", "is it equal to 37?".

Every time I ask a question, I cut the range half this process (binary search) take logarithmic time.

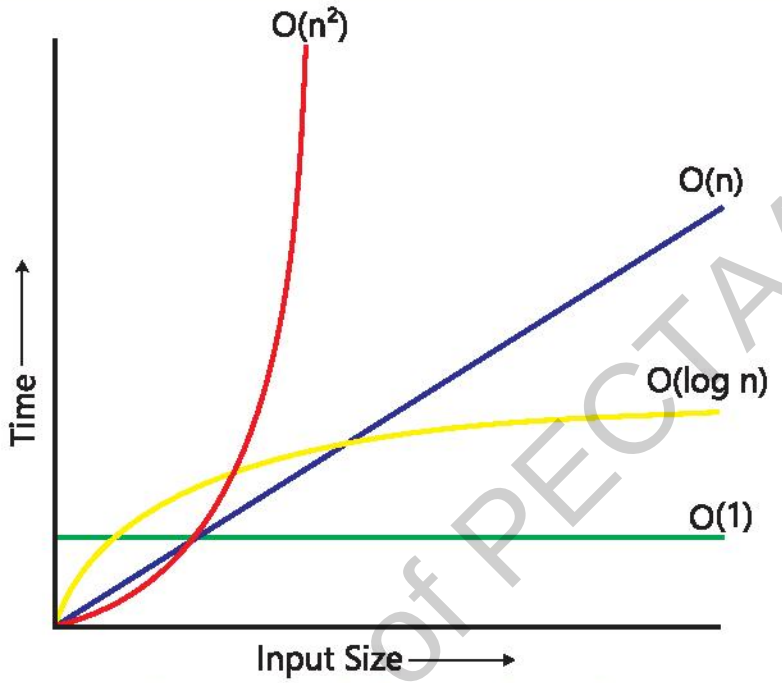

Figure 3.5: Growth of asymptotic notations

When comparing different time complexities, it's essential to understand how the time required for an algorithm grows as the size of the input *n* increases. Constant time, represented as *O*(1), remains unchanged regardless of the size of *n*. This means that no matter how large the input is, the time taken will be the same, as seen by the flat line in the graph.

3.4.2 Space Complexity

Space complexity measures how the amount of memory or space an algorithm uses changes as the size of the input data increases. It helps us understand how efficiently an algorithm uses memory when handling large datasets.

Example: if an algorithm needs to store a list of numbers, its space complexity tells us how much memory will be required as the volume of numbers increases.

3.5 Algorithm Design Techniques

Algorithm design is a critical aspect of problem-solving in computer science. It involves creating systematic methods to solve problems efficiently and effectively. There are several well-known algorithm design techniques that help in developing robust algorithms for a variety of computational problems.

3.5.1 Divide and Conquer

Divide and Conquer is a powerful algorithm design technique that works by breaking a large problem into smaller, more manageable parts. Each smaller part is solved

independently, and then their solutions are combined to solve the original problem, as shown in Figure 3.6. This approach is particularly effective for problems that can be divided into similar smaller problems, making it easier to find a solution step by step.

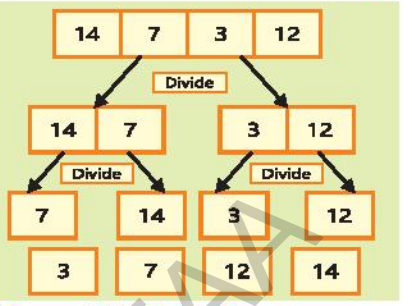

Figure 3.6: Merge Sort process

DO YOU KNOW?

Big O notation helps computer scientists understand the efficiency of an algorithm in the worst-case scenario, allowing them to predict how well it will perform as the size of the input data increases.

3.5.2 Greedy Algorithms

Greedy algorithms work by making a sequence of choices, each of which is locally optimal, with the hope that these choices will lead to a globally optimal solution. The greedy approach is often used when a problem has an optimal substructure, meaning that the optimal solution to the problem can be constructed from optimal solutions to its sub problems.

Example: A classic example of a greedy algorithm is the Coin Change problem. Suppose you have coins of different denominations and you want to make a specific amount with the fewest coins possible. The greedy algorithm would involve choosing the largest denomination coin that does not exceed the remaining amount, then subtracting that value and repeating the process until the desired amount is achieved.

Tidbits

Greedy algorithms are often faster and easier to implement than other techniques, but they don't always guarantee the optimal solution for every problem. Always analyze the problem to ensure that a greedy approach is appropriate.

3.5.3 Dynamic Programming

Dynamic Programming (DP) is an optimization technique used to solve problems by breaking them down into simpler subproblems and storing the results of these subproblems to avoid redundant calculations. DP is particularly useful for problems with overlapping subproblems and optimal substructure.

Example: The Fibonacci sequence is a well-known example where DP can be applied. Instead of recalculating Fibonacci numbers repeatedly, DP stores the results of each Fibonacci number as it is computed, allowing the algorithm to retrieve these values directly when needed, significantly reducing the number of calculations.

3.5.4 Backtracking

Backtracking is a method used in solving problems where you build up a solution step by step. If you find that a particular path doesn't lead to a solution, you simply go back and try a different path. It's like trying out different routes on a map and turning back if you find that you're going the wrong way. This method is often used for problems where you need to look at all possible options, like puzzles or problems that involve different combinations.

3.6 Commonly Used Algorithms

Algorithms are essential tools in computer science and are applied to a wide range of problems, from sorting data to searching for information in large datasets. Some algorithms are foundational, serving as building blocks for more complex operations. This section explores some of the most commonly used algorithms, including sorting, searching, and graph traversal algorithms.

3.6.1 Sorting Algorithms

Sorting algorithms are used to arrange data in a particular order, such as ascending or descending. Sorting is a fundamental operation that often serves as a prerequisite for other tasks like searching and data analysis.

3.6.1.1 Bubble Sort

Bubble Sort is one of the simplest sorting algorithms. It works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order. This process is repeated until the list is sorted.

Process:

- Start from the beginning of the list.
- Compare each pair of adjacent elements.
- Swap them if they are in the wrong order.
- Continue the process until no more swaps are needed.

Example: Consider the list [5,3,8,4,2]. Bubble Sort will first compare 5 and 3, swap them, then move to the next pair (5 and 8), and so on. After several passes through the list, the algorithm will sort the list as [2,3,4,5,8].

Complexity: The time complexity of Bubble Sort is $O(n^2)$, making it inefficient for large datasets. However, it is easy to understand and implement, making it useful for educational purposes and small datasets.

Tidbits

While Bubble Sort is easy to implement, consider using more efficient sorting algorithms like Quick Sort or Merge Sort for larger datasets to save time and resources.

3.6.1.2 Selection Sort

Selection Sort is another simple sorting algorithm. It works by selecting the smallest (or largest, depending on the desired order) element from the unsorted part of the list and swapping it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion of the list.

Process:

- Find the minimum element in the unsorted part of the list.
- Swap it with the first unsorted element.
- Move the boundary of the sorted and unsorted sections by one element.
- Repeat the process for the remaining elements.

Example: For the list [29,10,14,37,13], Selection Sort will first find the smallest element, 10, and swap it with 29. The list becomes [10,29,14,37,13]. The process continues until the list is fully sorted.

Complexity: The time complexity of Selection Sort is $O(n^2)$. Like Bubble Sort, it is not efficient for large datasets but is straightforward to implement.

3.6.2 Search Algorithms

Search algorithms are designed to find specific elements or a set of elements within a dataset. They are critical for tasks such as information retrieval, database queries, and decision-making processes.

3.6.2.1 Linear Search:

A linear search is a straightforward method for finding an item in a list. You check each item one by one until you find what you're looking for. Here's how it works,

1. **Start at the Beginning:** Look at the first item in the list.
2. **Check Each Item:** Compare the item you are looking for with the current item.
3. **Move to the Next:** If they don't match, move to the next item in the list.
4. **Repeat:** Continue this process until you find the item or reach the end of the list.

Example: Suppose you have a list of city names: [Karachi, Lahore, Islamabad, Faisalabad] And you want to find out if *Islamabad* is in the list.

1. Start with Karachi. Since Karachi isn't Islamabad, move to the next city.
2. Next is Lahore. Lahore is not Islamabad, so move to the next city.
3. Now you have Islamabad. This is the city you're looking for!

In this case, you've found Islamabad in the list. If Islamabad weren't in the list, you would check all the cities one by one and then conclude that it's not there. This method is called a linear search because you check each item in a straight line, from start to finish.

3.6.2.2 Binary Search

Binary Search is an efficient algorithm for finding an item in a sorted list. It works by repeatedly dividing the search interval in half and discarding the half where the item cannot be, until the item is found or the interval is empty.

Process:

- Start with the middle element of the sorted list.
- If the middle element is the target, return its position.
- If the target is smaller than the middle element, repeat the search on the left half.
- If the target is larger, repeat the search on the right half.

Example: Suppose you have a sorted list [1,3,5,7,9,11,13] and you are searching for a number.

- Binary Search will start at the middle element (7) and find the target immediately.

Complexity: The time complexity of Binary Search is $O(\log n)$, making it much faster than linear search algorithms, especially for large datasets. Figure 3.6 illustrates the binary search process, showing how the search interval is halved at each step, making the search more efficient.

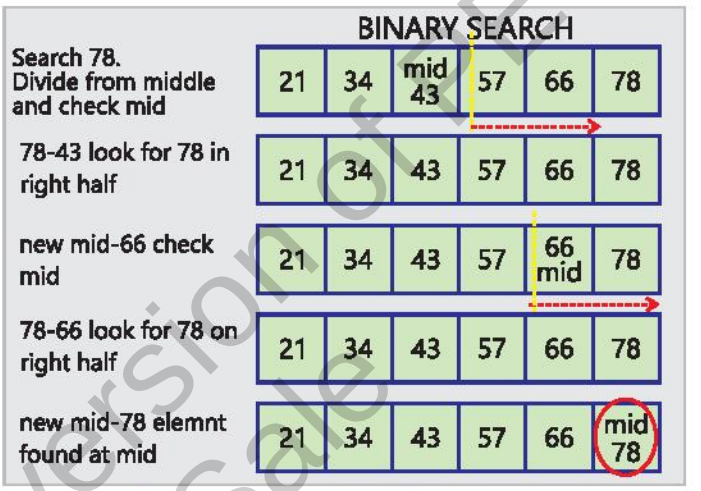

Figure 3.6: Binary Search Process

Binary Search is only effective on sorted lists. If your data isn't sorted, consider using a sorting algorithm like Merge Sort before applying Binary Search!

3.6.3 Graph Algorithms

Graph algorithms are used to explore and analyze graphs, which are data structures made up of nodes (vertices) connected by edges. These algorithms are essential for network analysis, route planning, and social network analysis.

3.6.3.1 Breadth-First Search (BFS)

Breadth-First Search (BFS) is a graph traversal algorithm that explores all the nodes of a graph level by level, starting from a given node (often called the root). It uses a queue to keep track of the nodes that need to be explored.

Process:

- Start from the root node and enqueue it.
- Dequeue a node, process it, and enqueue all its unvisited neighbors.
- Repeat the process until the queue is empty.

Example: In a social network graph, where each node represents a person and edges represent friendships, BFS can be used to find the shortest path between two people (e.g., finding the degree of separation between two users).

Complexity: The time complexity of BFS is $O(V + E)$, where V is the number of vertices and E is the number of edges. This makes it efficient for exploring large graphs.

3.6.3.2 Depth-First Search (DFS)

Depth-First Search (DFS) is another graph traversal algorithm that explores as far down a branch as possible before backtracking to explore other branches. It uses a stack to manage the nodes to be explored.

Process:

- Start from the root node and push it onto the stack.
- Pop a node, process it, and push all its unvisited neighbors onto the stack.
- Repeat the process until the stack is empty.

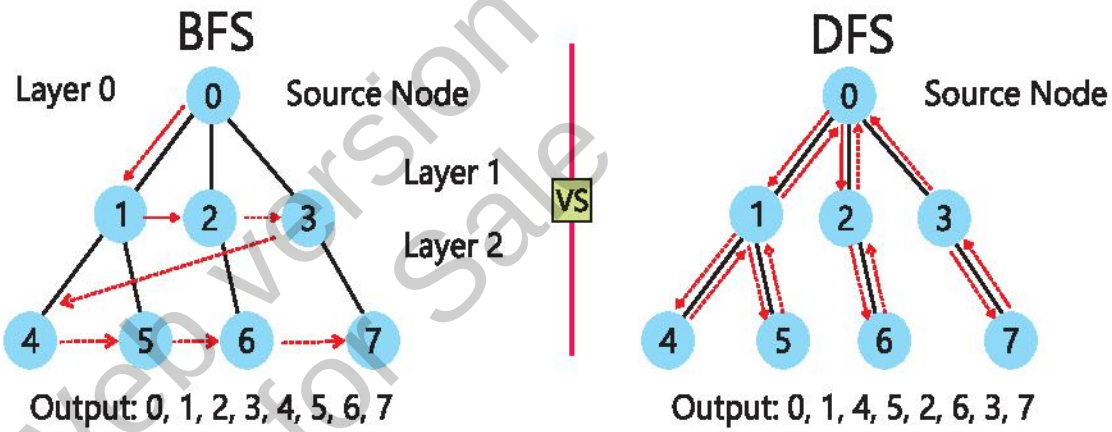

Figure 3.7: Comparison of BFS and DFS

Example: DFS can be used in solving puzzles like mazes, where the algorithm explores one possible path to the end, and if it hits a dead end, it backtracks and tries another path.

Complexity: The time complexity of DFS is O (V+E), similar to BFS. However, DFS is more memory-efficient for deep graphs, while BFS is more suited for shallow graphs.

EXERCISE

Multiple Choice Questions

1. The characteristic of a well-defined problem is:
   a) Ambiguous goals and unclear requirements
   b) Vague processes and inputs
   c) Clear goals, inputs, processes, and outputs
   d) Undefined solutions
2. Complexity class representing problems solvable efficiently by a deterministic algorithm:
   a) NP b) NP-hard
   c) NP-complete d) P
3. The statement that applies to unsolvable problems:
   a) They can be solved in polynomial time
   b) They cannot be solved by any algorithm
   c) They are always in NP class
   d) They require exponential time to solve
4. The meaning of NP in computational complexity is:
   a) Non-deterministic Polynomial time
   b) Negative Polynomial time
   c) Non-trivial Polynomial time
   d) Numerical Polynomial time
5. Search algorithm more efficient for large datasets:
   a) Bubble Sort b) Merge Sort
   c) Selection Sort d) Quick Sort
6. A scenario where Dynamic Programming proves most useful:
   a) Problems without overlapping subproblems
   b) Problems solved by making local choices
   c) Problems with overlapping subproblems and optimal substructure
   d) Problems divided into independent subproblems
7. An algorithm that sorts data by stepping through the list and swapping adjacent elements if needed is:
   a) Selection Sort b) Quick Sort
   c) Bubble Sort d) Merge Sort
8. Time complexity of Depth-First Search (DFS) in a graph is:
   a) O(n log n) b) O(V) c) O(V + E) d) O(n)

9. Best description of time complexity:
   a) Amount of memory an algorithm needs
   b) Time taken as a function of input size
   c) Efficiency as input size grows
   d) Upper bound of space requirements
10. An algorithm with a time complexity of O(n log n):
   a) Bubble Sort b) Binary Search c) Merge Sort d) Insertion Sort

Short Questions

1. Differentiate between well-defined and ill-defined problems within the realm of computational problem-solving.
2. Outline the main steps involved in the Generate-and-Test method.
3. Compare tractable and intractable problems in the context of computational complexity.
4. Summarize the key idea behind Greedy Algorithms.
5. Discuss the advantages of using Dynamic Programming.
6. Compare the advantages of Breadth-First Search (BFS) with Depth-First Search (DFS) in graph traversal.
7. Explain the importance of breaking down a problem into smaller components in algorithmic thinking.
8. Identify the key factors used to evaluate the performance of an algorithm.

Long Questions

1. Provide a detailed explanation of why the Halting Problem is considered unsolvable and its implications in computer science.
2. Discuss the characteristics of search problems and compare the efficiency of Linear Search and Binary Search algorithm.
3. Discuss the nature of optimization problems and provide examples of their applications in real-world scenarios.
4. Explain the process and time complexity of the Bubble Sort algorithm. Compare it with another sorting algorithm of your choice in terms of efficiency.
5. Discuss the differences between time complexity and space complexity. How do they impact the choice of an algorithm for a specific problem?