CHAPTER 3

Algorithms and Problem Solving

Q. What is a computational problem? Discuss its main components.

Computational Problem

A **computational problem** is a task or challenge that can be solved using a step-by-step procedure. This procedure is known as algorithm. An **algorithm** is a finite set of instructions that can be executed by the computer to solve the problem.

Components of Computational Problem

A computational problem typically consists of three main components:

Input — It refers to the data given to the algorithm. For example, a list of numbers is given to the algorithm to be sorted.

Process — It refers to the steps or rules that are applied to the input to generate the output. For example, an algorithm applies the sorting process to the given list of numbers to sort it.

Output — It is the solution or result produced by the algorithm after processing the input. For example, the list of numbers in ascending order is the output.

Q. What are different classifications of computational problem?

Computational problems can be classified into different categories depending on the nature of the problem and how it needs to be solved.

The main categories are as follows:

1. Decision Problems

Decision problems are the problems where output is **"yes"** or **"no"**. An example of decision problem is as follows:

- **Problem:** Is a given number even?
- **Input:** 8
- **Output:** Yes

2. Search Problems

Search problems are the problems where the goal is to find a solution or item that satisfies certain conditions. An example of search problem is as follows:

- **Problem:** Find the first student in a list who scored more than 90%.
- **Input:** A list of student scores
- **Output:** The name of the first student with a score more than 90%

3. Optimization Problems

Optimization problems are the problems where the goal is to find the best solution from many possible solutions based on certain rules. An example of optimization problem is as follows:

- **Problem:** Find the shortest route to deliver packages to five locations.
- **Input:** List of locations and distances
- **Output:** The route with the shortest total distance

4. Counting Problems

Counting problems are the problems where the goal is to count how many ways certain conditions can be met. An example of counting problem is as follows:

- **Problem:** How many 3-letter passwords can be made using letters A, B and C?
- **Input:** Letters A, B, C
- **Output:** 27 (since $3^3 = 27$ possible combinations)

AAA	AAB	AAC	BAA	BAB	BAC	CAA	CAB	CAC
ABA	ABB	ABC	BBA	BBB	BBC	CBA	CBB	CBC
ACA	ACB	ACC	BCA	BCB	BCC	CCA	CCB	CCC

Figure: 3-letter passwords

Q. Briefly describe the types of problems based on how clearly they are defined. Give examples.

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

1. Well-defined Problems

A **well-defined problem** is a problem that has clear goals, inputs, processes and outputs. These problems are easy to understand because all aspects of the problem are known and unambiguous. They can be solved using a step-by-step method known as an algorithm. The well-defined problems are more suitable to be solved using a computer because the solution can be clearly programmed and automated.

Example

An example of a well-defined problem is as follows:

- **Problem:** Determine if a number is even.
- **Goal:** Check whether the given number is even.
- **Input:** A number such as 8
- **Process:** Divide the number by 2 and check if the remainder is 0
- **Output:** "Even" if the result is 0, otherwise "Odd"

This is a well-defined problem because all components such as goal, input, process and output are clearly defined.

2. Ill-defined Problems

An **ill-defined problem** is a problem that does not have a clear goal, input, process or expected output. These problems are usually open-ended and vague. It means that these problems have no single correct answer or they can be solved in different ways. They are difficult to solve using computational methods like algorithms.

Example

An example of an ill-defined problem is as follows:

- **Problem:** How can we reduce poverty in Pakistan?

- **Goal:** Reduce poverty which is broad and not clearly defined.
- **Input:** It is not clearly defined and may include economic data or government policies etc.
- **Process:** There is no single method to solve the problem. It may require the economic, social and political reforms which are not defined.
- **Output:** It is difficult to measure and varies depending on the criteria.

Q. Differentiate between well-defined and ill-defined problems.

The difference between well-defined and ill-defined problems is as follows:

Well-defined Problems	Ill-defined Problems
1. A well-defined problem has clear goals.	1. An ill-defined problem does not have clear goals.
2. It includes specific and clearly defined inputs.	2. It has vague, incomplete or varying inputs.
3. It follows a known step-by-step method to solve the problem.	3. It does not provide specific method to solve the problem.
4. It produces definite output.	4. It may produce multiple outputs.
5. It is suitable for computer that solves it using algorithms and programming.	5. It is difficult for computer to solve without additional human guidance.
6. An example includes checking if a number is odd or even.	6. An example includes reducing poverty in a country.

Q. What is an algorithm and how are algorithms used for problem solving? Also discuss "Generate and Test" algorithm.

Algorithm for Problem Solving

An **algorithm** is a step-by-step procedure used to solve a problem or perform a specific task. It is similar to a recipe in cooking where each instruction must be followed in a particular order to achieve the desired result. In computing, algorithms are used to process input data, perform calculations and produce output efficiently and accurately. The understanding of algorithms is important for the following reasons:

- They form the logic behind computer programs and software.
- They help to solve complex problems efficiently.
- They ensure accuracy and consistency in different applications.

Generate and Test Algorithm

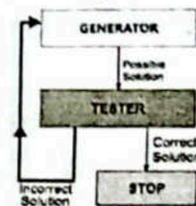
The **Generate and Test** algorithm is a basic and powerful method for solving the problems. It generates possible solutions and tests each solution to see if it works. The process continues until a satisfactory solution is found or all possible solutions have been tested. Different heuristics or rules can be applied to reduce the number of generated solutions. It makes the process more efficient.

This technique is based on two main steps as follows:

1. Generating the potential solutions of the problem.
2. Testing the solutions to see if they meet the required conditions.

The **Generate and Test** algorithm is useful in the scenarios where:

- The problem space is small and it is feasible to generate and test all possible solutions.
- There is no clear strategy for finding a solution and a search is necessary.



The Generate and Test algorithm is often used in **Artificial Intelligence (AI)** and game-based problem-solving. For example, it is used in the games like chess or puzzle-solving where the computer must try different moves or arrangements to find a winning solution.

Q. What is meant by problem solvability and complexity? Explain the solvable and unsolvable problems with examples.

Problem Solvability and Complexity

Problem solvability refers to whether a problem can be solved by an algorithm. A problem is solvable if a finite step-by-step procedure exists that leads to a solution. **Problem complexity** refers to how efficiently a problem can be solved. It considers the resources such as time and memory required by an algorithm to solve the problem. These concepts are important when designing the algorithms and evaluating their performance.

Solvable vs. Unsolvable Problems

The problems are generally classified into two broad categories named solvable and unsolvable. They are based on whether an algorithm exists that can solve them for all possible inputs.

Solvable Problems

A **solvable problem** is a problem that can be solved by an algorithm in a finite amount of time. These problems have:

- Clearly defined inputs and outputs
- A logical procedure (algorithm) to follow
- A guaranteed result after a certain number of steps

Example

Finding the greatest common divisor (GCD) of two integers is an example of a solvable problem. The Euclidean algorithm is a step-by-step method that efficiently finds the GCD and always completes in a finite number of steps.

Unsolvable Problems

An **unsolvable problem** is a problem for which no algorithm can be created that solves the problem for all possible cases or inputs. These problems do not have a general procedure that can guarantee a solution for every possible input. They often involve situations that are too complex or undefined for a computer to handle.

Example

The **Halting Problem** is a well-known example of unsolvable problem. It is used to determine whether a given computer program will eventually stop running or continue to run forever for a specific input. Alan Turing proved that there is no general algorithm that can correctly solve this problem for all inputs. Therefore, the Halting Problem is considered unsolvable.

Q. Differentiate between solvable and unsolvable problems.

The difference between solvable and unsolvable problems is as follows:

Solvable Problems	Unsolvable Problems
1. A solvable problem is a problem that can be solved by an algorithm in a finite amount of time.	1. An unsolvable problem is a problem for which no algorithm can be created that solves it for all possible inputs.
2. It can be completed in a limited number of steps.	2. It may require infinite steps or cannot be solved at all.
3. It always produces a definite and correct output.	3. It does not guarantee an output for every possible input.
4. These problems can be handled efficiently by computers using algorithms and programs.	4. These problems cannot be solved completely by any computer or program.
5. An example of a solvable problem is finding Greatest Common Divisor of two numbers using Euclid's algorithm.	5. An example of an unsolvable problem is the Halting Problem introduced by Alan Turing.

Q. Explain tractable and intractable problems with examples.

Tractable Problems

A **tractable problem** is a problem that can be solved in a reasonable amount of time even if the input size grows. These problems can be solved using algorithms that run in polynomial time. **Polynomial time** means that the time taken to solve the problem increases at a manageable rate as the input size grows. It makes it possible to solve tractable problems using a computer. Tractable problems are considered efficiently solvable.

Example

Sorting a list of numbers using algorithms such as **Merge Sort** or **Quick Sort** is an example of tractable problem. These algorithms have time complexity of **O(n log n)** where **n** is the number elements in the list. It means that these algorithms can handle large inputs efficiently. The list can be sorted in reasonable amount of time even if it contains a large number of elements.

Intractable Problems

An **intractable problem** is a problem that becomes extremely difficult and time-consuming to solve as the size of the input increases. These problems often require super-polynomial or exponential time. It means that these problems are impractical to solve for large inputs because the time required becomes unmanageable.

Example

The **Travelling Salesman Problem (TSP)** is an example of intractable problem. The task of this problem is to find the shortest possible route to visit a set of cities and return to the starting point. The number of possible routes grows extremely fast as the number of cities increases. It becomes impossible to solve it for a large number of cities.

Q. Describe different classes of problem complexity with examples.

Complexity Classes

The complexity of problems can be classified into different categories based on their solvability and the time required to solve them.

Different categories of complexity classes are as follows:

1. Class P

Class P refers to a category of problems that can be solved efficiently by a computer. It means that the computer can find a solution of these problems quickly even if the size of the problem grows.

Example

A simple example of class P problem is sorting a list of numbers. Suppose the given list of numbers is as follows:

[4, 1, 3, 2, 5]

The goal is to arrange these numbers in ascending order like [1, 2, 3, 4, 5]. The time required to sort the list grows at a manageable rate as the list size increases. For example, sorting a list of 10 numbers will require more time to sort but it remains within a reasonable limit.

2. Class NP

Class NP refers to a category of problems for which the given solution can be checked by a computer quickly. The proposed solution of these problems can be verified easily but finding that solution can be difficult and time-consuming.

Example

An example of class NP problem is solving a Sudoku puzzle that consists of 9 x 9 grid. It is pre-filled with some numbers. The user needs to fill the blank cells with the numbers using the following rules:

- No duplicate numbers in any row
- No duplicate numbers in any column
- No duplicate numbers in any 3×3 square

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

3. Class NP-Hard

Class NP-hard refers to a category 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 available to find a solution.

Example

The **Travelling Salesman Problem (TSP)** is an example of **Class NP-hard** problem. The task of this problem is to find the shortest possible route to visit a set of cities and return to the starting point. The number of possible routes grows extremely fast as the number of cities increases. It becomes impossible to solve it for a large number of cities.

4. NP- Complete

An **NP-Complete problem** is a problem that is both in NP and as hard as the hardest problems in NP. This means that if one NP-Complete problem can be solved efficiently then all NP problems can also be solved efficiently. NP-Complete problems form a subset of NP and are also considered NP-Hard. Solving an NP-Hard problem is considered computationally very difficult and no known algorithm can solve all NP-Hard problems efficiently. Many of these problems do not even have guaranteed solutions that can be verified in polynomial time.

Example

An example of NP-Complete problem is the Knapsack Problem. In this problem, there is a knapsack with a maximum weight capacity and a set of items 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.

Q. What is algorithm analysis? Describe time and space complexity with examples.

Algorithm Analysis

Algorithm analysis is the study of how efficiently an algorithm performs in terms of time and space. This analysis is used to predict the algorithm's performance. It is very important for selecting the best algorithm for a particular task.

Time Complexity

Time complexity measures how the runtime of an algorithm increases as the input size grows. It helps to understand the efficiency of an algorithm when it handles large amounts of data.

Example

Suppose there is a list of numbers to be sorted. The task may be quick if the list has only a few numbers. However, the time required to sort the numbers increases as the volume of numbers increases. Time complexity predicts how this runtime will grow as the size of the list becomes larger.

Space Complexity

Space complexity measures how much memory is used by an algorithm as the input size increases. It helps to understand how efficiently an algorithm uses memory when it deals with larger amounts of data.

Example

Suppose an algorithm needs to store a list of numbers. The space complexity tells the amount of memory required as the volume of numbers increases.

Q. Explain the concept of time complexity with the help of Big O notation and suitable examples.

Big O Notation

Big O notation is a mathematical way to describe the time or space complexity of an algorithm. It describes how the runtime or memory usage of an algorithm grows as the input size increases. This notation helps to compare the efficiency of different algorithms based on their scalability. **Scalability** refers to how well an algorithm handles increasing input sizes.

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. Some common examples are as follows:

1. O(1) Constant Time

An algorithm has **O(1) time complexity** when its runtime does not change with the size of the input. The **O(1)** means that an algorithm takes the same amount of time to run for any size of the input.

2. O(n) Linear Time

An algorithm has **O(n) time complexity** when its runtime increases directly in proportion to the size of the input. This means that if the input size doubles, the time required also approximately doubles.

Suppose there are **n** students in a college with a unique student ID. A specific student can be found by searching through the list of all **n** students. The time required to find a specific student will depend on the number of students.

3. O(n^2) Quadratic Time

An algorithm has **O(n^2) time complexity** when its runtime grows quadratically as the size of the input increases. The **O(n^2)** means that as the input size **n** increases, the runtime grows proportionally to the square of **n**. For example:

- If **n** doubles, the time increases by 4 times.
- If **n** triples, the time increases by 9 times and so on.

Suppose **n** students in a college are participating in a programming competition. The performance of each pair of students needs to be compared to determine the best team. Since each student must be compared with every other student exactly once, the number of required comparisons can be calculated as follows:

$$\frac{n(n-1)}{2}$$

4. O($\log^n$) Logarithmic Time

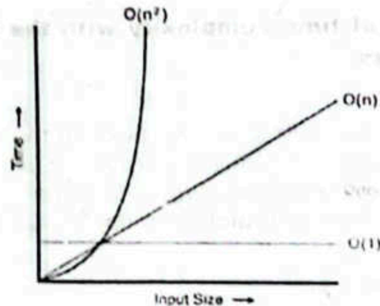
An algorithm has **O(log n) time complexity** when its runtime increases very slowly as the size of the input grows. Suppose a person is thinking a number between 1 and 100 and the other person is guessing the number. He can use the following questions:

- is it greater than 50?

affecting Runtime grows logarithmicly in O(log n)

- is it less than 25?
- is it less to 12?

Each question reduces the range and makes it easier to guess the number.



Q. Discuss the differences between time complexity and space complexity. How do they impact the choice of an algorithm for a specific problem?

Time complexity	Space complexity
1. Time complexity refers to how long an algorithm takes to complete as the input size increases.	1. Space complexity refers to how much memory an algorithm uses as the input size increases.
2. It measures number of steps taken by an algorithm to perform a task.	2. It measures amount of memory used by an algorithm to perform a task.
3. It focuses on the speed or runtime efficiency of an algorithm.	3. It focuses on the memory or storage efficiency of an algorithm.
4. It is useful when processing speed is critical such as in real-time systems.	4. It is useful when memory is limited such as in mobile devices.
5. It is expressed in terms of operations relative to input size such as $O(n)$, $O(n^2)$, $O(\log n)$.	5. It is expressed in terms of memory usage relative to input size such as $O(1)$, $O(n)$).

Impact on Algorithm Selection

Both time and space complexity are important when choosing an algorithm to solve a problem. The choice depends on the environment in which the algorithm will be applied. An algorithm with low space complexity is preferred if memory is limited such as in mobile devices. An algorithm with low time complexity is more appropriate if speed is crucial such as in gaming or real-time processing.

There are cases where improving time efficiency increases memory usage or saving memory slows down execution. This is known as the **time-space trade-off.** It refers to the practice of balancing between time and space by using more memory to reduce computation time or accepting slower execution in order to save memory. For example, a sorting algorithm called Merge Sort has time complexity $O(n \log n)$ and space complexity $O(n)$. It means it is faster but uses more memory.

Q. What is algorithm design technique? List the names of different algorithm design techniques.

Algorithm Design Techniques

Algorithm design is the process of developing step-by-step methods to solve computational problems efficiently. It involves selecting suitable techniques to ensure correctness, efficiency and simplicity.

Different algorithm design techniques are as follows:

1. Divide and Conquer
2. Greedy Algorithms
3. Dynamic Programming
4. Backtracking

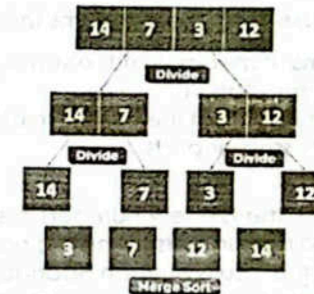
Q. What is Divide and Conquer algorithm design technique? Explain with an example.

Divide and Conquer

Divide and Conquer is a powerful algorithm design technique. It divides a large problem into smaller and more manageable parts. Each smaller part is solved independently. All solutions are then combined to solve the original problem. This approach is very effective for the problems that can be divided into smaller problems. This makes it easier to find a solution step by step.

Example

Merge Sort uses the divide and conquer approach to sort the list. It first divides the list into smaller halves until each part has only one element. It then sorts these parts and merges the sorted parts back together into a single sorted list.



Q. What is Greedy algorithm design technique? Explain with an example.

Greedy Algorithms

A **greedy algorithm** is a problem-solving strategy that builds a solution by always choosing the best available option at each stage. Each choice is locally optimal with the hope that these choices will lead to a globally optimal solution. Greedy approach is often used when a problem has an optimal substructure. It means that if a problem can be divided into smaller parts and solved optimally, then combining those optimal parts gives the best overall solution.

Greedy algorithms are often faster and easier to implement than other techniques. However, they do not always guarantee the optimal solution for every problem. It is important to analyze the problem to ensure that a greedy approach is appropriate.

Example

A classic example of a greedy algorithm is the Coin Change problem. The goal of this problem is to use the fewest number of coins to make the specific amount.

The greedy algorithm works as follows:

1. Pick the largest coin that does not exceed remaining amount.
2. Subtracts that value from the remaining amount.
3. Repeat the process until the remaining amount is zero.

Suppose the desired amount is $27 and the available coins are $25, $10, $5, $1. The greedy algorithm will solve the problem as follows:

1. Pick the largest coin that does not exceed remaining amount: $25
2. Subtracts that value from the remaining amount: $27 – $25 = $2
3. Pick the largest coin that does not exceed remaining amount: $2
4. Subtracts that value from the remaining amount: $2 – $2 = 0
5. The problem is solved using two coins of $25 and $2.

Q. What is dynamic programming? Explain with example.

Dynamic Programming

Dynamic Programming is a method for solving complex problems by breaking them into smaller and simpler subproblems. It solves each subproblem once and stores the results to avoid repeating the calculations.

Dynamic programming is useful for the problems that have:

- **Overlapping subproblems:** It means that the same subproblems appear again and again during the computation.
- **Optimal substructure:** It means that the overall solution of the problem depends on the best solutions of its smaller parts.

Example

In the Fibonacci sequence, the first few numbers are: 0, 1, 1, 2, 3, 5, 8, 13. Each number is the sum of two preceding numbers. Dynamic programming can be applied to Fibonacci sequence. It stores the results of each Fibonacci number as it is computed instead of recalculating it repeatedly. It reduces number of calculations significantly.

Q. What is backtracking in algorithm design? Explain with example.

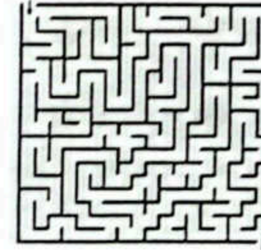
Backtracking

Backtracking is a problem-solving technique that builds a solution step by step. It tries a path but leaves it as soon as it determines that the path cannot lead to a valid solution. It then goes back to try a different path. This process continues until a solution is found or all possible options have been explored.

Example

Maze is an example where backtracking can be applied to solve the problem. The goal is to find a path from the start to the end.

1. Start from the initial position.
2. Try moving in one of the possible directions.
3. Move back if you hit a wall or it has already been visited.
4. Repeat the process until the end point is reached.



Q. What are the common computing algorithms? List some common computing algorithms.

Common Computing Algorithms

Different problems require different types of algorithmic techniques. Different programmers use different algorithms to solve the same problem. Many algorithms are considered fundamental and are commonly used in solving daily life problems. These algorithms are essential tools that help in performing a variety of tasks efficiently and effectively.

Some common computing algorithms are as follows:

1. Sorting Algorithms

Sorting algorithms are techniques used to arrange data in a particular order such as ascending or descending. In **ascending order,** the elements are arranged from the smallest to the largest such as [3, 7, 10, 15]. In **descending order,** the elements are arranged from the largest to the smallest such as [20, 12, 5, 1]. Sorting is a fundamental operation that is often required for other tasks such as searching, data analysis and organizing information efficiently.

Two common sorting algorithms include the following:

- Insertion sort algorithm
- Bubble sort algorithm

2. Searching Algorithms

Searching 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. Efficient search methods save time and improve program performance.

Two common searching algorithms include the following:

- Binary search algorithm
- Linear search algorithm