Unit 01

Q1. What is an Algorithm? Explain the characteristics of an algorithm in detail.

(10 Marks)

Ans:

Introduction

In computer science, an **algorithm** is the foundation of problem solving. Every software application, whether simple or complex, works on the basis of an algorithm. An algorithm provides a systematic and logical approach to solving a problem. It allows a programmer to convert a real-world problem into a sequence of instructions that a computer can understand and execute.

Algorithms are independent of programming languages and machines. The same algorithm can be implemented in different programming languages such as C, C++, Java, or Python. Therefore, designing a correct and efficient algorithm is one of the most important tasks in computer science.

Definition of Algorithm

An algorithm is defined as a **finite set of well-defined, unambiguous instructions** that takes some input, processes it step by step, and produces a desired output within a finite amount of time.

In simple words, an algorithm tells **what to do**, **how to do**, and **when to stop**.

Need for Algorithms

Algorithms are required because:

- They help in solving problems systematically.
- They reduce complexity by breaking problems into steps.
- They make programs easy to understand and debug.
- They improve efficiency and performance.

Characteristics of an Algorithm

For any procedure to be called an algorithm, it must satisfy the following characteristics:

---

1. Input

An algorithm must have **zero or more inputs**. Inputs are the data provided to the algorithm before it starts execution.
For example, in a sorting algorithm, the input is a list of numbers.

Without proper input, an algorithm cannot perform meaningful computation.

---

2. Output

An algorithm must produce **at least one output**. The output is the result obtained after processing the input.
For example, the output of a sorting algorithm is a sorted list.

The relationship between input and output defines the correctness of an algorithm.

---

3. Definiteness

Each step of the algorithm must be **clearly defined and unambiguous**. This means that every instruction should have only one meaning.

If steps are vague or confusing, different programmers may interpret them differently, leading to incorrect results.

---

4. Finiteness

An algorithm must terminate after a **finite number of steps**. It should not run indefinitely.

This property ensures that the algorithm reaches a conclusion and produces an output. Algorithms that do not terminate are not considered valid.

---

5. Effectiveness

All operations in an algorithm must be **basic and executable**. Each step should be simple enough to be performed in a finite amount of time using available computational resources.

An algorithm should not include impractical or impossible operations.

---

6. Correctness

An algorithm is said to be correct if it produces the **correct output for all valid inputs**. Correctness is one of the most important characteristics, as an efficient algorithm with wrong output is useless.

---

7. Generality

An algorithm should be capable of solving **all instances of a given problem**, not just a single case.
For example, a sorting algorithm should work for any size of input array.

Conclusion

Thus, an algorithm is a precise and structured method for solving problems. The characteristics discussed above ensure that the algorithm is reliable, efficient, and suitable for implementation in real-world applications.

Q2. Explain the analysis of algorithms and asymptotic complexity bounds. Discuss best, average, and worst-case analysis.

(10 Marks)

Ans:

Introduction

Algorithm analysis is an important area of computer science that deals with determining the **efficiency** of an algorithm. As the size of input data increases, the performance of an algorithm becomes critical. Algorithm analysis helps in understanding how much time and memory an algorithm requires to solve a problem.

Instead of measuring actual execution time, which depends on hardware and software, algorithm analysis provides a **machine-independent method** of comparison.

Analysis of Algorithms

Algorithm analysis mainly focuses on:

- **Time Complexity** – amount of time taken by an algorithm
- **Space Complexity** – amount of memory required by an algorithm

This analysis helps programmers select the best algorithm for a given problem.

Time Complexity

Time complexity is a function that represents the number of operations performed by an algorithm as a function of input size *n*.
It helps in predicting the growth rate of execution time.

Space Complexity

Space complexity refers to the amount of memory required by an algorithm during its execution, including:

- Input space
- Auxiliary space

Asymptotic Analysis

Asymptotic analysis studies the behavior of an algorithm as the input size becomes very large ($n \to \infty$). It ignores constant factors and lower-order terms.

Asymptotic Notations

1. Big-O Notation (O)

Represents the **upper bound** of time complexity. It describes the worst-case scenario.

2. Omega Notation (Ω)

Represents the **lower bound** of time complexity. It describes the best-case scenario.

3. Theta Notation (Θ)

Represents the **tight bound**, where best and worst cases are the same.

📌 **DIAGRAM TO DRAW:**
Graph showing **O(n), Ω(n), and Θ(n)** curves with input size on X-axis and time on Y-axis.

Case Analysis of Algorithms

Best-Case Analysis

Best case defines the **minimum time** required by an algorithm for the most favorable input.

Example:
Linear search when the element is found at the first position.

Best-case analysis is useful for understanding the optimal behavior of an algorithm.

Average-Case Analysis

Average case defines the **expected performance** of an algorithm over all possible inputs.

It considers probability distribution of inputs and provides a realistic performance measure.

Worst-Case Analysis

Worst case defines the **maximum time** required by an algorithm for the least favorable input.

Example:
Linear search when the element is not present in the list.

Worst-case analysis is most commonly used because it guarantees performance under all conditions.

Importance of Worst-Case Analysis

- Ensures reliability
- Helps in real-time systems
- Prevents performance degradation

Conclusion

Algorithm analysis and asymptotic complexity bounds provide a theoretical foundation for comparing algorithms. By understanding best, average, and worst-case behaviors, programmers can design efficient and scalable solutions.

✍️ **QUESTION 3 – HANDWRITING FORMAT**

Q3. Explain sorting techniques and analyze their performance.

(10 Marks)

Ans:

Introduction

Sorting is one of the most important operations in computer science. Sorting refers to the process of arranging data elements in a particular order such as ascending or descending. Efficient sorting improves the performance of searching, merging, and data processing operations. Many real-life applications such as database management systems, operating systems, and file handling systems make extensive use of sorting algorithms.

Need for Sorting

Sorting is required because:

- It makes searching faster
- It helps in organizing data efficiently
- It improves readability of data
- It is used in databases, scheduling, and data analysis

Common Sorting Techniques

---

1. Bubble Sort

Bubble sort is a simple comparison-based sorting algorithm. It works by repeatedly comparing adjacent elements and swapping them if they are in the wrong order. After each pass, the largest element moves to its correct position.

Algorithm Idea:
Compare → Swap → Repeat

Performance Analysis:

- Best Case: O(n) (already sorted list)
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$
- Space Complexity: O(1)

📌 **DIAGRAM TO DRAW:**
Array showing multiple passes with swapping of adjacent elements.

---

2. Selection Sort

Selection sort works by selecting the smallest element from the unsorted part of the array and placing it at the beginning. This process is repeated until the entire array is sorted.

Performance Analysis:

- Best Case: $O(n^2)$
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$
- Space Complexity: O(1)

Selection sort performs well for small datasets but is inefficient for large inputs.

---

3. Insertion Sort

Insertion sort works in the same way as sorting playing cards in hand. It picks one element at a time and inserts it into its correct position in the already sorted portion of the array.

Performance Analysis:

- Best Case: O(n)
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$
- Space Complexity: O(1)

Insertion sort is efficient for small and nearly sorted datasets.

---

4. Merge Sort

Merge sort is based on the divide and conquer technique. It divides the array into smaller sub-arrays, sorts them recursively, and then merges them back.

Performance Analysis:

- Best Case: O(n log n)
- Average Case: O(n log n)
- Worst Case: O(n log n)
- Space Complexity: O(n)

📌 **DIAGRAM TO DRAW:**
Divide-and-merge tree showing splitting and merging of arrays.

---

5. Quick Sort

Quick sort selects a pivot element and partitions the array into two parts such that elements smaller than the pivot are on the left and larger elements are on the right.

Performance Analysis:

- Best Case: O(n log n)
- Average Case: O(n log n)
- Worst Case: $O(n^2)$
- Space Complexity: O(log n)

📌 **DIAGRAM TO DRAW:**
Pivot selection and partitioning diagram.

Conclusion

Different sorting algorithms have different time and space requirements. The choice of sorting technique depends on the size of input, memory availability, and performance requirements.

✍️ **QUESTION 4 – HANDWRITING FORMAT**

Q4. Explain the concept of Time–Space Trade-Off with suitable examples.

(10 Marks)

Ans:

Introduction

Time–space trade-off is an important concept in algorithm design. It refers to the situation where an algorithm can be made faster by using more memory, or memory usage can be reduced by increasing execution time. Designing an efficient algorithm requires balancing both time and space constraints.

Explanation of Time–Space Trade-Off

In many algorithms:

- Increasing memory usage reduces execution time
- Reducing memory usage increases execution time

Thus, time and space are inversely related in many cases.

Importance of Time–Space Trade-Off

Time–space trade-off is important because:

- Memory is limited
- Execution time is critical in real-time systems
- Performance optimization is required

Examples of Time–Space Trade-Off

---

1. Merge Sort

Merge sort uses additional memory to store temporary arrays during merging. This extra space helps achieve a faster time complexity of O(n log n).

---

2. Hashing

Hash tables use extra memory to store data but allow searching, insertion, and deletion operations in O(1) time.

---

3. Dynamic Programming

Dynamic programming stores intermediate results to avoid recomputation, thereby reducing execution time at the cost of additional memory.

---

4. Lookup Tables

Precomputed values are stored in memory to speed up calculations.

📌 **DIAGRAM TO DRAW:**
Graph showing trade-off between time and space.

Advantages

- Faster execution
- Improved system performance

Disadvantages

- Increased memory usage
- Not suitable for memory-constrained systems

Conclusion

Time–space trade-off helps in designing optimized algorithms. The choice of algorithm depends on available memory and time constraints.

Unit 01 part 2

Introduction

Many algorithms in computer science are recursive in nature, especially divide and conquer algorithms such as Merge Sort, Quick Sort, and Binary Search. To analyze such algorithms, recurrence relations are used. A recurrence relation expresses the running time of an algorithm in terms of the running time on smaller inputs.

—

Example 1: Substitution Method

Problem:

$$T(n) = T\left(\frac{n}{2}\right) + n$$

Solution:

Assume the solution is of the form:

$$T(n) = O(n)$$

Substituting into the recurrence:

$$T(n) = O\left(\frac{n}{2}\right) + n$$

$$T(n) = O(n)$$

Final Answer:

$$T(n) = O(n)$$

—

Example 2: Recursion Tree Method

Problem:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Solution:

At each level of the recursion tree, the total cost is:

$$n$$

The height of the recursion tree is:

$$\log n$$

Therefore, the total cost is:

$$T(n) = n\log n$$

Final Answer:

$$T(n) = O(n\log n)$$

—

Example 3: Master's Theorem

Problem:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Solution:

Comparing with the standard form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

We get:

$$a = 2, \quad b = 2, \quad f(n) = n$$

$$n^{\log_b a} = n^{\log_2 2} = n$$

Since:

$$f(n) = \Theta\left(n^{\log_b a}\right)$$

This corresponds to **Case 2 of Master's Theorem**.

Final Answer:

$$T(n) = \Theta(n\log n)$$

—

Example 4: Linear Recurrence

Problem:

$$T(n) = T(n-1) + 1$$

Solution:

Expanding the recurrence:

$$T(n) = T(n-2) + 2$$

$$T(n) = T(n-3) + 3$$

After n steps:

$$T(n) = T(1) + (n-1)$$

Final Answer:

$$T(n) = O(n)$$

—

Example 5: Logarithmic Recurrence

Problem:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Solution:

The input size is halved at each recursive call.

Number of recursive calls:

$$\log n$$

Each call takes constant time.

Final Answer:

$$T(n) = O(\log n)$$

—

Example 6: Master's Theorem (Case 1)

Problem:

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

Solution:

$$a = 4, \quad b = 2, \quad f(n) = n$$

$$n^{\log_b a} = n^{\log_2 4} = n^2$$

Since:

$$f(n) = O(n^{2-\epsilon})$$

This corresponds to **Case 1 of Master's Theorem**.

Final Answer:

$$T(n) = \Theta(n^2)$$

—

Example 7: Master's Theorem (Case 3)

Problem:

$$T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

Solution:

$$n^{\log_2 2} = n$$

Since:

$$f(n) = \Omega(n^{1+\epsilon})$$

This corresponds to **Case 3 of Master's Theorem**.

Final Answer:

$$T(n) = \Theta(n^2)$$

—

Unit 05

Q1. Explain Approximation Algorithms in detail. (10 Marks)

Introduction

In computer science, many real-world problems are computationally very complex and belong to the class of NP-Hard problems. For such problems, finding an exact optimal solution within reasonable time is not feasible, especially for large input sizes. In order to handle such situations efficiently, Approximation Algorithms are used.

Approximation algorithms provide solutions that are close to the optimal solution and can be computed in polynomial time. These algorithms are widely used in practical applications where a near-optimal solution is sufficient.

Definition

An Approximation Algorithm is an algorithm that finds a solution whose value is guaranteed to be within a certain bound of the optimal solution for an optimization problem.

Need for Approximation Algorithms

Approximation algorithms are required due to the following reasons:

- Many optimization problems are NP-Hard.
- Exact algorithms have exponential time complexity.
- Large input sizes make exact computation impractical.
- Real-time applications require fast solutions.
- Slight loss of accuracy is acceptable in real-world systems.

Approximation Ratio

The performance of an approximation algorithm is measured using the approximation ratio.

$$\text{Approximation Ratio} = \frac{\text{Cost of Approximate Solution}}{\text{Cost of Optimal Solution}}$$

For minimization problems, the approximation ratio is always greater than or equal to 1, whereas for maximization problems, it is less than or equal to 1.

Types of Approximation Algorithms

1. Greedy Approximation Algorithms
2. Polynomial Time Approximation Scheme (PTAS)
3. Fully Polynomial Time Approximation Scheme (FPTAS)

Example: Vertex Cover Problem

In the Vertex Cover problem, the objective is to find the minimum number of vertices such that every edge is incident to at least one selected vertex. A simple approximation algorithm selects both endpoints of an edge and removes all associated edges. This algorithm guarantees a 2-approximation of the optimal solution.

Applications

Approximation algorithms are used in network design, scheduling problems, routing, cloud computing, and the Traveling Salesman Problem.

Diagram

(Draw a diagram showing comparison between optimal and approximate solution)

Conclusion

Approximation algorithms play a significant role in solving complex problems efficiently by trading optimality for performance.

Q2. Explain Randomized Algorithms in detail.
(10 Marks)

Introduction

Randomized Algorithms are algorithms that make random choices during execution to improve performance or simplify implementation. These algorithms are particularly useful when deterministic algorithms are slow or difficult to design.

Definition

A Randomized Algorithm is an algorithm that uses random numbers as part of its logic to produce output.

Need for Randomized Algorithms

- To avoid worst-case scenarios
- To simplify algorithm design
- To improve average-case performance
- To handle large datasets efficiently

Types of Randomized Algorithms

Las Vegas Algorithms

Las Vegas algorithms always produce the correct result, but their running time is random. An example is Randomized Quick Sort.

Monte Carlo Algorithms

Monte Carlo algorithms run in fixed time but may produce incorrect results with a small probability. An example is probabilistic primality testing.

Performance Analysis

Randomized algorithms are analyzed using expected time complexity and probability of correctness.

Advantages

- Faster average performance
- Simple implementation
- Avoids adversarial inputs

Disadvantages

- Non-deterministic behavior
- Difficult to debug

Applications

Randomized algorithms are used in cryptography, machine learning, load balancing, and hashing.

Diagram

(Draw a flowchart showing random decision paths)

Conclusion

Randomized algorithms provide efficient and practical solutions by incorporating randomness into computation.

Q3. Explain Distributed Hash Table (DHT) in detail. (10 Marks)

Introduction

With the rapid growth of distributed systems and peer-to-peer networks, centralized data storage systems face issues such as scalability and single point of failure. Distributed Hash Tables address these challenges by distributing data across multiple nodes.

Definition

A Distributed Hash Table (DHT) is a decentralized data structure that stores key-value pairs across a distributed network of nodes.

Working Principle

- Each node is assigned a unique identifier.
- Keys are mapped to nodes using a hash function.
- Each node stores a portion of the data.
- Lookup operations are performed efficiently.

Components of DHT

1. Nodes
2. Hash Function
3. Routing Table
4. Key-Value Pairs

Popular DHT Algorithms

- Chord
- Pastry

- Kademlia
- CAN

Advantages

- Scalability
- Fault tolerance
- No single point of failure

Disadvantages

- Complex implementation
- Network overhead
- Security issues

Applications

DHTs are used in peer-to-peer networks, file sharing systems, blockchain, and distributed databases.

Diagram

(Draw a circular Chord DHT ring diagram)

Conclusion

Distributed Hash Tables provide an efficient and scalable solution for data management in distributed systems.