GIET
In Pursuit of Innovation

STUDY MATERIAL

On

ALGORITHIMS

(For 3RD Semester CSE)

SESSION(2026-27)

Prepared by :
STHITIPRAJNA BISWAL (CSE DEPT),
GIET (POLY),JAGATPUR,CUTTACK

Unit 1.Introduction to Algorithms

Introduction

An **algorithm** is a finite sequence of well-defined, step-by-step instructions used to solve a specific problem or accomplish a particular task. It forms the foundation of computer programming and software development. Before writing a program, programmers design an algorithm to ensure that the solution is correct, efficient, and easy to understand.

Algorithms are used in almost every area of computer science, including searching, sorting, networking, artificial intelligence, databases, operating systems, and web applications. A good algorithm helps solve problems quickly while using minimum memory and processing time.

Definition

Algorithm: An algorithm is a finite set of precise instructions that accepts input, processes it according to defined steps, produces the desired output, and terminates after a finite number of steps.

Characteristics of an Algorithm

A good algorithm has the following characteristics:

1. Input

An algorithm accepts zero or more input values required to solve a problem.

2. Output

It produces one or more outputs as the result of processing.

3. Definiteness

Every step of the algorithm must be clear, precise, and unambiguous.

4. Finiteness

The algorithm must end after a finite number of steps.

5. Effectiveness

Each instruction should be simple, practical, and executable within a reasonable amount of time.

Example of an Algorithm

Problem: Find the sum of two numbers.

Algorithm:

1. Start.
2. Read two numbers A and B.
3. Calculate `Sum = A + B`.
4. Display the value of Sum.
5. Stop.

Need for Algorithms

Algorithms are important because they:

- Provide a systematic method for solving problems.
- Help programmers write correct and efficient programs.
- Reduce execution time and memory usage.
- Make debugging and testing easier.
- Improve code readability and maintenance.
- Help compare different solutions to choose the most efficient one.

Types of Algorithms

1. **Brute Force Algorithm** – Tries every possible solution until the correct one is found.
2. **Divide and Conquer** – Divides a problem into smaller subproblems, solves them, and combines the results.
3. **Greedy Algorithm** – Makes the best choice at each step to reach an optimal solution.
4. **Dynamic Programming** – Solves complex problems by storing results of smaller subproblems.
5. **Backtracking** – Tries different solutions and returns to a previous step if a solution fails.
6. **Recursive Algorithm** – Solves a problem by calling itself with smaller inputs.

Time Complexity

Time complexity measures how the execution time of an algorithm increases with the size of the input.

Common time complexities:

- O(1) – Constant Time
- O(log n) – Logarithmic Time
- O(n) – Linear Time
- O(n log n) – Linearithmic Time
- $O(n^2)$ – Quadratic Time

Lower time complexity generally indicates better performance.

Space Complexity

Space complexity is the total amount of memory required by an algorithm during execution, including memory for variables, data structures, and recursion.

Advantages of Algorithms

- Easy to understand and implement.
- Helps identify logical errors before coding.
- Improves program efficiency.
- Saves development time.
- Simplifies maintenance and future modifications.

Applications of Algorithms

Algorithms are widely used in:

- Searching and Sorting
- Database Management Systems
- Operating Systems

- Computer Networks
- Artificial Intelligence and Machine Learning
- Data Compression
- Cryptography and Security
- GPS Navigation Systems
- Banking and E-commerce

Difference Between Algorithm and Program

Algorithm	Program
Step-by-step solution to a problem	Actual code written in a programming language
Language independent	Language dependent
Easier to modify	Requires coding knowledge to modify

Conclusion

Algorithms are the backbone of computer science and programming. They provide a clear and efficient method for solving problems before implementation in a programming language. Understanding algorithms helps students develop better programming skills and create efficient software solutions.

Important Exam Questions

1. Define an algorithm with an example.
2. Explain the characteristics of an algorithm.
3. Discuss the need and applications of algorithms.
4. Explain time complexity and space complexity.
5. Differentiate between an algorithm and a program.

Unit2: Algorithm Complexity

Introduction

Algorithm complexity is the measure of the resources required by an algorithm to solve a problem. The two main resources are **time** and **memory (space)**. Complexity analysis helps us compare different algorithms and choose the most efficient one.

What is Algorithm Complexity?

Algorithm complexity is a way of measuring how an algorithm's performance changes as the size of the input increases. It tells us how much time and memory an algorithm requires.

There are two main types:

1. Time Complexity
2. Space Complexity

Time Complexity

Time complexity measures the amount of time an algorithm takes to execute as the input size (**n**) increases.

Common Time Complexities

Notation	Name	Example
O(1)	Constant	Accessing an array element
O(log n)	Logarithmic	Binary Search
O(n)	Linear	Linear Search
O(n log n)	Linearithmic	Merge Sort
$O(n^2)$	Quadratic	Bubble Sort
$O(2^n)$	Exponential	Recursive Fibonacci
O(n!)	Factorial	Traveling Salesman (Brute Force)

Types of Time Complexity

1. Best Case

The minimum time required by an algorithm for a given input.

2. Average Case

The expected execution time over all possible inputs.

3. Worst Case

The maximum time required by an algorithm. This is the most commonly used measure.

Space Complexity

Space complexity is the total amount of memory required by an algorithm during execution.

It includes:

- Input memory

- Variables
- Temporary storage
- Recursive call stack

Asymptotic Notations

Big O Notation (O)

Represents the **upper bound (worst-case)** time complexity.

Big Omega (Ω)

Represents the **lower bound (best-case)** time complexity.

Big Theta (Θ)

Represents the **exact (average or tight)** bound of an algorithm.

Example

Linear Search

- Best Case: O(1)
- Average Case: O(n)
- Worst Case: O(n)

Binary Search

- Best Case: O(1)
- Average Case: O(log n)
- Worst Case: O(log n)

Importance of Complexity Analysis

- Compares algorithms.
- Helps select efficient solutions.
- Reduces execution time.
- Optimizes memory usage.
- Improves software performance.

Advantages

- Saves time and memory.
- Helps build efficient programs.

- Useful for large datasets.
- Improves system performance.

Applications

- Search engines
- Database management
- Artificial Intelligence
- Operating Systems
- Computer Networks
- Data Analysis
- Scientific Computing

Conclusion

Algorithm complexity is essential for evaluating the efficiency of algorithms. By analyzing time and space complexity, programmers can choose the best algorithm for solving problems efficiently.

Important Exam Questions

1. Define algorithm complexity.
2. Explain time complexity with examples.
3. Explain space complexity.
4. What are Big O, Big Omega, and Big Theta notations?
5. Differentiate between best, average, and worst case.
6. Compare the time complexity of Linear Search and Binary Search.

Unit3:Recursive Algorithms

Introduction

A **recursive algorithm** is an algorithm that solves a problem by calling itself repeatedly with smaller versions of the same problem until a simple condition, called the **base case**, is reached. Recursion is a powerful programming technique used to solve problems that can be divided into smaller subproblems.

Recursive algorithms are widely used in mathematics, searching, sorting, tree traversal, graph algorithms, and many other areas of computer science.

Definition

A **recursive algorithm** is an algorithm in which a function calls itself directly or indirectly until a stopping condition (base case) is satisfied.

Components of a Recursive Algorithm

A recursive algorithm consists of two essential parts:

1. Base Case

- The condition that stops the recursion.
- Prevents infinite recursion.
- Returns the final result.

2. Recursive Case

- The part where the function calls itself.
- Solves a smaller version of the original problem.

Working of Recursion

The recursive function repeatedly calls itself, reducing the problem size each time. When the base case is reached, the function stops calling itself, and the results return to previous function calls.

Example 1: Factorial of a Number

The factorial of a number **n** is:

- $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

Algorithm

1. Start.
2. Read number **n**.
3. If **n = 0** or **n = 1**, return 1.
4. Otherwise, return **n × Factorial(n − 1)**.
5. Stop.

Example

Factorial(5)

= 5 × Factorial(4)

= 5 × 4 × Factorial(3)

= 5 × 4 × 3 × Factorial(2)

= 5 × 4 × 3 × 2 × Factorial(1)

= 5 × 4 × 3 × 2 × 1 = **120**

Example 2: Fibonacci Series

The Fibonacci sequence is:

0, 1, 1, 2, 3, 5, 8, 13, ...

Recursive formula:

- F(0) = 0
- F(1) = 1
- F(n) = F(n−1) + F(n−2)

Advantages of Recursive Algorithms

- Easy to understand.
- Reduces code complexity.
- Suitable for tree and graph traversal.
- Useful in Divide and Conquer algorithms.
- Makes mathematical problems easier to implement.

Disadvantages of Recursive Algorithms

- Uses more memory due to function call stack.
- Slower than iteration for many problems.
- May cause stack overflow for very deep recursion.
- Sometimes difficult to debug.

Applications of Recursive Algorithms

- Factorial calculation
- Fibonacci series
- Binary Search
- Merge Sort
- Quick Sort
- Tower of Hanoi
- Tree Traversal (Preorder, Inorder, Postorder)

- Graph Traversal (DFS)
- Directory/File System Navigation

Recursive vs Iterative Algorithms

Recursive Algorithm	Iterative Algorithm
Function calls itself	Uses loops (for, while)
Uses call stack	Uses loop control variables
Easier for complex problems	Faster in many cases
More memory required	Less memory required

Time Complexity

The time complexity of recursive algorithms depends on the number of recursive calls.

Examples:

- Factorial: **O(n)**
- Fibonacci (simple recursion): **$O(2^n)$**
- Binary Search: **O(log n)**
- Merge Sort: **O(n log n)**

Space Complexity

Recursive algorithms require extra memory for the function call stack.

- Factorial: **O(n)**
- Binary Search: **O(log n)**

Conclusion

Recursive algorithms solve problems by breaking them into smaller subproblems. Every recursive algorithm must have a base case and a recursive case. Although recursion makes programs simpler and more elegant, it may use more memory and execution time than iterative methods.

Important Exam Questions

1. Define a recursive algorithm with an example.
2. Explain the components of recursion.
3. Write the recursive algorithm for factorial.
4. Explain recursion using the Fibonacci series.
5. List the advantages and disadvantages of recursive algorithms.

6. Differentiate between recursive and iterative algorithms.
7. Explain the applications of recursive algorithms.
8. Discuss the time and space complexity of recursive algorithms.

Unit4:Algorithm Paradigms

Introduction

An **Algorithm Paradigm** is a general method or strategy used to design and solve computational problems. Instead of creating a new solution for every problem, programmers use standard design techniques called paradigms. These paradigms help in developing efficient, organized, and reusable algorithms.

Different problems require different algorithm paradigms depending on their structure and complexity.

Definition

Algorithm Paradigm:
An algorithm paradigm is a general approach or technique used to design algorithms for solving a class of problems efficiently.

Need for Algorithm Paradigms

- Simplify complex problems.
- Improve efficiency and performance.
- Reduce execution time.
- Make algorithms easier to understand and maintain.
- Encourage code reusability.

Types of Algorithm Paradigms

1. Brute Force Paradigm

The brute force approach tries every possible solution until the correct one is found.

Characteristics

- Simple to implement.
- Guarantees a correct solution if one exists.
- Usually slower for large inputs.

Examples

- Linear Search
- Selection Sort
- Bubble Sort

Advantages

- Easy to understand.
- Works for almost all problems.

Disadvantages

- High time complexity.
- Inefficient for large datasets.

---

2. Divide and Conquer Paradigm

This technique divides a problem into smaller subproblems, solves them recursively, and combines their results.

Steps

1. Divide the problem.
2. Solve each subproblem.
3. Combine the solutions.

Examples

- Merge Sort
- Quick Sort
- Binary Search

Advantages

- Faster than brute force.
- Efficient for large datasets.

Disadvantages

- Recursive implementation may require additional memory.

---

3. Greedy Paradigm

The greedy approach makes the best possible choice at each step without reconsidering previous decisions.

Characteristics

- Chooses the locally optimal solution.
- Faster for many optimization problems.

Examples

- Dijkstra's Shortest Path Algorithm
- Prim's Algorithm
- Kruskal's Algorithm
- Huffman Coding

Advantages

- Simple and efficient.
- Low memory usage.

Disadvantages

- Does not always produce the globally optimal solution.

---

4. Dynamic Programming

Dynamic Programming (DP) solves problems by dividing them into overlapping subproblems and storing their solutions to avoid repeated computation.

Characteristics

- Uses memoization or tabulation.
- Improves efficiency.

Examples

- Fibonacci Series
- Knapsack Problem
- Longest Common Subsequence
- Matrix Chain Multiplication

Advantages

- Avoids repeated calculations.

- Reduces execution time.

Disadvantages

- Requires additional memory.

---

5. Backtracking

Backtracking builds a solution step by step. If a partial solution cannot lead to a valid answer, it goes back (backtracks) and tries another possibility.

Examples

- N-Queens Problem
- Sudoku Solver
- Maze Solving
- Graph Coloring

Advantages

- Finds all possible solutions.
- Suitable for constraint satisfaction problems.

Disadvantages

- Slow for large search spaces.

---

6. Recursive Paradigm

A recursive algorithm solves a problem by calling itself on smaller instances until a base case is reached.

Examples

- Factorial
- Fibonacci
- Tower of Hanoi
- Tree Traversals

Advantages

- Short and easy-to-read code.

- Suitable for hierarchical problems.

Disadvantages

- Uses extra memory due to the call stack.
- May be slower than iterative solutions.

Comparison of Algorithm Paradigms

Paradigm	Main Idea	Example
Brute Force	Try all possibilities	Linear Search
Divide and Conquer	Divide, solve, combine	Merge Sort
Greedy	Choose the best option at each step	Dijkstra's Algorithm
Dynamic Programming	Store solutions to subproblems	Knapsack Problem
Backtracking	Try, backtrack, and retry	N-Queens
Recursion	Function calls itself	Factorial

Applications

Algorithm paradigms are used in:

- Searching and sorting
- Artificial Intelligence
- Computer Networks
- Database Systems
- Operating Systems
- Cryptography
- Machine Learning
- Navigation Systems

Advantages of Using Algorithm Paradigms

- Improves problem-solving skills.
- Produces efficient algorithms.
- Saves development time.
- Makes code reusable and maintainable.
- Helps analyze time and space complexity.

Conclusion

Algorithm paradigms provide standard strategies for solving computational problems. Choosing the appropriate paradigm leads to efficient, reliable, and optimized algorithms. Understanding

these paradigms is essential for designing high-quality software and solving real-world computing problems.

Important Exam Questions

1. Define an algorithm paradigm.
2. Explain the need for algorithm paradigms.
3. Describe the Divide and Conquer paradigm with examples.
4. Explain the Greedy paradigm and its applications.
5. What is Dynamic Programming? Explain with examples.
6. Explain the Backtracking paradigm.
7. Differentiate between Greedy and Dynamic Programming.
8. Compare different algorithm paradigms with suitable examples

Unit4:Sorting Algorithms

Introduction

Sorting is the process of arranging data in a particular order, such as **ascending** or **descending**. Sorting makes searching, processing, and analyzing data faster and more efficient. It is one of the most important concepts in computer science and is widely used in databases, search engines, banking systems, and operating systems.

Definition

A **sorting algorithm** is a step-by-step procedure used to arrange data elements in a specified order.

Types of Sorting

1. **Ascending Order** – Smallest to largest (10, 20, 30, 40).
2. **Descending Order** – Largest to smallest (40, 30, 20, 10).

---

1. Bubble Sort

Definition

Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order. After each pass, the largest element moves to the end.

Algorithm

1. Start.
2. Compare adjacent elements.
3. Swap if the first element is greater than the second.
4. Repeat for all elements.
5. Continue until no swaps are needed.
6. Stop.

Advantages

- Simple to understand.
- Easy to implement.

Disadvantages

- Slow for large datasets.

Time Complexity

- Best Case: **O(n)**
- Average Case: **$O(n^2)$**
- Worst Case: **$O(n^2)$**

Space Complexity

- **O(1)**

---

2. Selection Sort

Definition

Selection Sort repeatedly selects the smallest element from the unsorted part and places it at the beginning.

Algorithm

1. Find the smallest element.
2. Swap it with the first unsorted element.
3. Repeat for the remaining elements.

Advantages

- Simple algorithm.
- Performs fewer swaps.

Disadvantages

- Slow for large datasets.

Time Complexity

- Best, Average, Worst: **$O(n^2)$**

Space Complexity

- **O(1)**

---

3. Insertion Sort

Definition

Insertion Sort inserts each element into its correct position in the already sorted part of the array.

Algorithm

1. Assume the first element is sorted.
2. Pick the next element.
3. Insert it into its correct position.
4. Repeat until all elements are sorted.

Advantages

- Efficient for small datasets.
- Stable sorting algorithm.

Disadvantages

- Slow for large datasets.

Time Complexity

- Best: **O(n)**
- Average: **$O(n^2)$**
- Worst: **$O(n^2)$**

Space Complexity

- **O(1)**

---

4. Merge Sort

Definition

Merge Sort uses the **Divide and Conquer** technique. It divides the array into halves, sorts each half, and merges them.

Advantages

- Very efficient.
- Stable sorting algorithm.

Disadvantages

- Requires extra memory.

Time Complexity

- Best, Average, Worst: **O(n log n)**

Space Complexity

- **O(n)**

---

5. Quick Sort

Definition

Quick Sort selects a pivot element and partitions the array into two parts.

Advantages

- Very fast in practice.
- Efficient for large datasets.

Disadvantages

- Worst case occurs when pivot selection is poor.

Time Complexity

- Best: **O(n log n)**
- Average: **O(n log n)**
- Worst: **O(n^2)**

Space Complexity

- **O(log n)** (average recursion stack)

Comparison of Sorting Algorithms

Algorithm	Best	Average	Worst	Space
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Quick Sort	O(n log n)	O(n log n)	O(n^2)	O(log n)

Applications of Sorting

- Database management
- Search engines
- Banking systems
- Student result processing
- E-commerce product listing
- Operating systems
- Data analysis

Advantages of Sorting

- Makes searching faster.
- Organizes data efficiently.
- Improves processing speed.
- Simplifies data analysis.

Conclusion

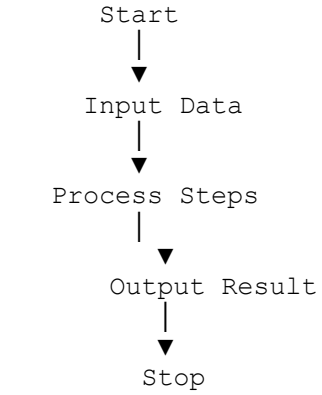
Sorting algorithms arrange data in a specific order to improve efficiency. Simple algorithms like Bubble, Selection, and Insertion Sort are suitable for small datasets, while Merge Sort and Quick Sort are preferred for large datasets due to their better performance.

Important Exam Questions

1. Define sorting and explain its types.
2. Explain Bubble Sort with its algorithm and complexity.
3. Explain Selection Sort with advantages and disadvantages.
4. Explain Insertion Sort with an example.
5. Describe Merge Sort using the Divide and Conquer technique.
6. Explain Quick Sort and its working.
7. Compare Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, and Quick Sort.
8. Explain the applications and advantages of sorting algorithms.

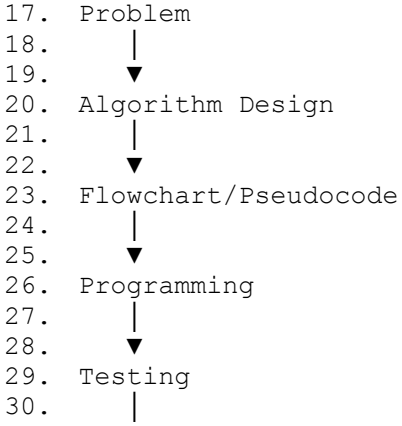
Unit6:Graphs

1. 1. Structure of an Algorithm



---

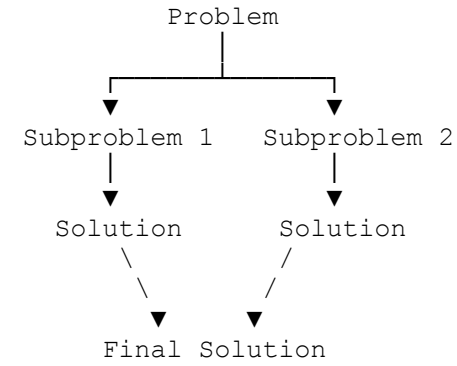
16. 2. Algorithm Design Process



Final Output

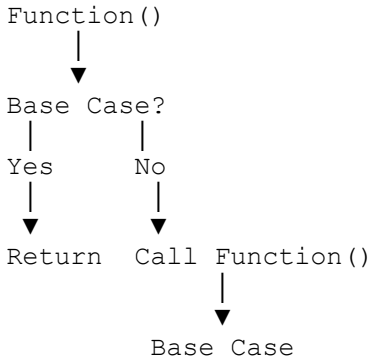
---

3. Divide and Conquer



---

4. Recursion



---

5. Bubble Sort

```
5 3 8 2

Pass 1
5 3 → Swap
3 5 8 2

8 2 → Swap
3 5 2 8

Pass 2
3 5 2 8
3 2 → Swap
2 3 5 8

Sorted
```

---

6. Selection Sort

```
64 25 12 22
```

```
Minimum =12

12 25 64 22

Minimum =22

12 22 64 25

Minimum =25

12 22 25 64
```

---

7. Insertion Sort

```
7 4 5 2

7

4 7

4 5 7

2 4 5 7
```

---

8. Merge Sort

```
       8 4 6 2

       /      \
    8 4        6 2

   /  \       /  \
  8   4      6   2

Merge

4 8     2 6

Merge

2 4 6 8
```

---

9. Quick Sort

```
8 4 6 2 7

Pivot = 6

4 2 |6| 8 7

↓

2 4 6 7 8
```

---

10. Binary Search

```
Sorted Array

10 20 30 40 50 60

          ↑
        Middle

If key < middle → Left

If key > middle → Right

Repeat until found.
```

---

11. Time Complexity Chart

```
Fast
 │
 │  O(1)
 │  O(log n)
 │  O(n)
 │  O(n log n)
 │  O(n²)
 │  O(2ⁿ)
 │  O(n!)
 │
Slow
```

---

12. Algorithm Paradigms