Introduction to Algorithms

Main Idea:

- This presentation introduces the concept of algorithms and how they build confidence in logical problem-solving.

Key Points:

- Algorithms are the foundation of problem-solving in computing.

- Understanding them enhances logical thinking.

Why Algorithms Matter

Everyday Relevance:

- Algorithms help in solving routine problems — like planning your day or following a recipe.

Programming Perspective:

- An algorithm is like a blueprint you follow before writing actual code.

Mental Skill Development:

- Breaks large, complex problems into smaller, manageable steps, helping you think logically.

What Is an Algorithm? (Basic)

Definition:

- A step-by-step set of instructions to solve a specific problem.

Key Characteristics:

- Finite: Always ends.
- Unambiguous: Every step is clear.
- Effective: Leads to a correct solution.

General Use:

- Doesn’t depend on programming language — can be applied anywhere.

Algorithm vs. Program

Aspect	Algorithm	Program
Definition	Logical plan	Code that runs
Language	None (uses pseudocode)	C++, Python, etc.
Focus	What needs to be done	How it is done
Reusability	Conceptually reusable	Rewritten in each language

Note: Algorithm = idea, logic; Program = code implementation of that logic

Problem-Solving Steps

Step-by-Step Guide:

1. Understand the problem (what is being asked?)

2. Break into sub-problems (divide the work)

3. Identify key operations (what needs to happen)

4. Draw flowchart or pseudocode (visually or textually plan)

5. Code the solution

6. Test and debug

Advanced Tip:

- After breaking the problem down, check if any sub-problem logic can be reused (modular thinking).

Visual Thinking with Flowcharts

Flowchart Symbols:

- Oval: Start / End

- Rectangle: Process or operation

- Diamond: Decision (yes/no)

- Arrow: Flow direction

- Parallelogram: Input or output

Example – “Making Tea” Flowchart: Start → Boil Water → Tea Bag? → Yes → Steep → Pour → End

Purpose: Helps visualize logic clearly before coding.

Pseudocode Examples

Example 1: Calculate Area of a Circle

```
PROCEDURE CalculateArea(Radius)
   PI ← 3.14
   Area ← PI × Radius × Radius
   RETURN Area
END PROCEDURE
```

Example 2: Determine if a Year is a Leap Year

```
PROCEDURE IsLeapYear(Year)
   IF (Year MOD 4 == 0 AND Year MOD 100 ≠ 0) OR (Year
MOD 400 == 0) THEN
      RETURN "Leap Year"
   ELSE
```

```
    RETURN "Not a Leap Year"
  END IF
END PROCEDURE
```

Example 3: Count Vowels in a String

```
PROCEDURE CountVowels(Text)
  Count ← 0
  FOR each Character in Text DO
    IF Character IN ['a','e','i','o','u','A','E','I','O','U'] THEN
      Count ← Count + 1
    END IF
  END FOR
  RETURN Count
END PROCEDURE
```

Example 4: Reverse a List

```
PROCEDURE ReverseList(List)
  Start ← 0
```

```
    End ← LENGTH(List) - 1
    WHILE Start < End DO
        SWAP List[Start] WITH List[End]
        Start ← Start + 1
        End ← End - 1
    END WHILE
    RETURN List
END PROCEDURE
```

Linear Search Example

Goal: Find a value X inside a list L

Pseudocode:

```
FOR each element E in L DO
    IF E == X THEN
        RETURN index of E
    END IF
END FOR
RETURN "Not Found"
```

Efficiency: Time Complexity: O(n), Space Complexity: O(1)

Use case: Good when list is unsorted.

Complexity Basics

Time Complexity:

- $O(1)$: Constant time (e.g., accessing array index)

- $O(n)$: Linear time (e.g., scanning a list)

- $O(n^2)$: Quadratic time (e.g., nested loops)

Space Complexity: Extra memory required to run the algorithm beyond the input

Everyday Analogy: Sorting 5 books = quick, Sorting 50 books = more steps = more complexity

Advanced Complexity Insights

Big-O Notation Types:

- O (Big-O): Worst case scenario (upper bound)

- Θ (Theta): Average case (tight bound)

- Ω (Omega): Best case (lower bound)

Why Important? Helps in choosing the right algorithm for the task — more efficient, faster performance.

Debugging Approach

Debugging Steps:

- Check code step-by-step

- Use print/log statements to view values

- Compare expected vs actual results

Techniques:

- Rubber-duck debugging: Explain your code to an object (or friend!)

- Peer walkthroughs: Review with someone else for fresh perspective