

INTRODUCTION TO PROBLEM SOLVING & PROGRAMMING

Problem solving in programming means using logical thinking and systematic steps to find a solution to a problem and converting that solution into a computer program.

A computer cannot think on its own – it follows the instructions given by a programmer.

Therefore, the programmer must first understand the problem clearly and then design a proper method to solve it.

✓ STEPS FOR PROBLEM SOLVING

To solve any problem using a computer, we generally follow these steps:

1. Understanding the Problem

- Read the problem carefully.
- Identify what is given (input) and what result is expected (output).
- Understand the constraints and purpose of the problem.

Example:

Find the sum of two numbers.

Input: two numbers

Output: their sum

2. Analyzing the Problem

- Break the problem into smaller parts.
 - Find what operations or calculations are needed.
 - Think logically about how to approach the solution.
-

3. Designing the Solution

This step includes planning how the solution will work.

Common tools used:

- Algorithm (step-by-step written instructions)
- Flowchart (diagram showing flow of operations)

4. Writing the Algorithm

- Write clear steps that explain how to solve the problem.
 - This is done in simple English-like statements.
-

5. Coding / Programming

- Convert the algorithm into a programming language such as Python, C, Java, etc.
 - Use correct syntax and rules of the language.
-

6. Testing and Debugging

- Run the program to check if it is working correctly.
 - Find and correct any errors (bugs).
 - Test with different inputs.
-

7. Documentation and Maintenance

- Writing explanations, comments, and instructions for future use.
 - Updating the program when needed.
 - Fixing new issues over time.
-

✓ INTRODUCTION TO ALGORITHM

An algorithm is a finite, step-by-step set of instructions to solve a particular problem.

Characteristics of a Good Algorithm

1. **Clear and unambiguous**
Each step should be clear and easy to understand.
2. **Finite**
It must end after a limited number of steps.
3. **Input**
An algorithm should clearly define the required inputs.
4. **Output**
It must produce at least one output.

5. Effective

Each step must be simple and possible to perform.

Example of an Algorithm

Problem: Find the sum of two numbers.

Algorithm:

1. Start
 2. Read number A
 3. Read number B
 4. Calculate Sum = A + B
 5. Display Sum
 6. Stop
-

Another Example: Find the largest of two numbers

1. Start
 2. Input A and B
 3. If A > B, print A is largest
 4. Else print B is largest
 5. Stop
-

If you want, I can also add flowchart diagrams, extra examples, or short exam notes for quick revision.

You said:

Understanding and preparing Flowcharts ✓ Evolution of programming language ✓ Types of programming language

ChatGPT said:

Here is a clear and exam-ready explanation for:

- ✓ Understanding and Preparing Flowcharts
 - ✓ Evolution of Programming Languages
 - ✓ Types of Programming Languages
-

1. UNDERSTANDING AND PREPARING FLOWCHARTS

A flowchart is a diagrammatic representation of the sequence of steps in a process or program. It uses different symbols to show the flow of control and operations.

Flowcharts make it easier to understand, analyze, and communicate a program's logic.

✓ Purpose of Flowcharts

- To visualize the logic of a program
 - To detect errors and improve the design
 - To communicate steps clearly
 - To help in debugging and documentation
-

✓ Common Flowchart Symbols

| Symbol | Meaning |
|---------------|-------------------------------|
| Oval | Start / Stop |
| Parallelogram | Input / Output |
| Rectangle | Process / Calculation |
| Diamond | Decision (Yes/No, True/False) |
| Arrow | Flow of control |
| Circle | Connector |

✓ Steps to Prepare a Flowchart

1. Identify the problem clearly.
 2. Determine inputs and outputs.
 3. Break the process into simple steps.
 4. Arrange the steps in logical order.
 5. Use proper flowchart symbols to represent each step.
 6. Draw arrows to show the flow.
 7. Review the flowchart for correctness and clarity.
-

✓ Flowchart Example: Add two numbers

```
Start
  ↓
Input A, B
  ↓
```

```
Sum = A + B
↓
Output Sum
↓
Stop
```

2. EVOLUTION OF PROGRAMMING LANGUAGES

Programming languages have evolved over time to become easier, faster, and more powerful.

They can be grouped into generations:

✓ 1st Generation (1GL): Machine Language

- Written in binary (0s and 1s)
- Very difficult to understand
- Machine-dependent
- Fast execution

Example: 10110101 11001010

✓ 2nd Generation (2GL): Assembly Language

- Uses symbolic codes (mnemonics)
- Easier than machine language
- Needs an assembler to convert to machine code
- Still hardware dependent

Examples: MOV, ADD, SUB

✓ 3rd Generation (3GL): High-Level Languages

- English-like statements
- Programmer-friendly
- Machine-independent
- Uses compiler or interpreter

Examples: C, C++, Java, Python, Fortran, BASIC

✓ 4th Generation (4GL): Very High-Level Languages

- Focus on reducing coding effort
- More like human language
- Often used in database handling and report generation

Examples: SQL, MATLAB, Oracle Reports, Query languages

✓ 5th Generation (5GL): Artificial Intelligence Languages

- Used for AI, machine learning, neural networks
- Focus on problem-solving and constraint-based logic

Examples: Prolog, Mercury, Lisp

3. TYPES OF PROGRAMMING LANGUAGES

Programming languages can be classified in multiple ways:

A. Based on Level of Language

1. Low-Level Languages

- Machine language
 - Assembly language
 - Fast but difficult to write
-

2. High-Level Languages

- C, C++, Java, Python, PHP
 - Easy to read and write
 - Machine-independent
-

3. Very High-Level Languages

- SQL, R, MATLAB
 - Used for specific applications
-

B. Based on Programming Paradigm

1. Procedural Languages

- Step-by-step instructions
- Functions and procedures

Examples: C, Pascal, BASIC

2. Object-Oriented Languages

- Based on objects and classes
- Supports encapsulation, inheritance, polymorphism

Examples: Java, Python (OOP), C++, C#, Ruby

3. Functional Languages

- Based on functions and immutability

Examples: Haskell, Scala, Lisp

4. Scripting Languages

- Used for automation and small tasks

Examples: Python, JavaScript, Perl, PHP, Shell Script

5. Logic Programming Languages

- Works on rules and facts

Examples: Prolog

✓ If you want, I can also prepare:

- ❖ Flowchart diagrams for more examples
- ❖ Very short notes for exam revision
- ❖ MCQs with answers
- ❖ PDF format notes

You said:

Pseudo coding • Program • Compiler • interpreter

ChatGPT said:

Below is a clear, simple, and exam-ready explanation for:

- ✓ Pseudo Code
 - ✓ Program
 - ✓ Compiler
 - ✓ Interpreter
-

1. PSEUDO CODE (PSEUDOCODING)

Pseudo code is a simple, informal, English-like description of the steps in a program.

It is not written in any specific programming language.

It is used to plan and design a program before writing the actual code.

✓ Features of Pseudo Code

- Written in plain English, not actual code
 - Easy to understand for anyone
 - Helps in designing algorithms
 - No strict rules or syntax
 - Used as a bridge between algorithm and program
-

✓ Example of Pseudo Code: Add two numbers

```
START
READ A, B
SET SUM = A + B
DISPLAY SUM
END
```

✓ Example: Find largest of two numbers

```
START
INPUT A, B
IF A > B THEN
```

```
    PRINT "A is greater"
ELSE
    PRINT "B is greater"
END IF
END
```

2. PROGRAM

A program is a set of instructions written in a programming language that a computer can understand and execute.

Programs are written using languages like Python, Java, C, C++, etc.

✓ Characteristics of a Program

- Has a specific purpose
 - Written using proper syntax
 - Provides instructions to solve a problem
 - Should be correct, efficient, and error-free
-

✓ Example Program (in Python)

```
a = int(input("Enter A: "))
b = int(input("Enter B: "))
sum = a + b
print("Sum =", sum)
```

3. COMPILER

A compiler is a software tool that translates the entire source program (high-level code) into machine code at once, before execution.

✓ Features of a Compiler

- Translates the whole program at once
- Shows all errors after compilation
- Faster execution

- Produces an executable file
-

Examples of Compiler Languages

- C
 - C++
 - Java
 - Go
 - Rust
-

Diagram: How a Compiler Works

Source Code → Compiler → Machine Code → Execution

4. INTERPRETER

An interpreter is a software tool that executes the program line-by-line.

It translates and runs each line immediately.

✓ Features of an Interpreter

- Translates one line at a time
 - Shows errors instantly
 - Slower execution compared to compiler
 - No separate executable file is produced
-

Examples of Interpreted Languages

- Python
 - JavaScript
 - PHP
 - Ruby
 - SQL
-

Diagram: How an Interpreter Works

Source Code → Interpreter → Output (line by line)

Compiler vs Interpreter (Quick Comparison)

| Feature | Compiler | Interpreter |
|---------------|-------------------------|-----------------|
| Translation | Whole program at once | Line-by-line |
| Error Display | After full scan | Immediately |
| Speed | Faster | Slower |
| Output | Creates executable file | No executable |
| Examples | C, C++, Java | Python, JS, PHP |