Problem Solving Techniques

PROBLEM SOLVING

Problem Solving

* Problem - Task
* Solving - Creating no. of solutions

Definition

* Systematic approach to define the problem and creating no. of solutions
* Starts: Problem Specifications
  Ends: Correct Program

Real Time Example

Problem - Maths Sum
Solving - Answer / Solution
Technique / Logic - Formula, Shortcut Tricks

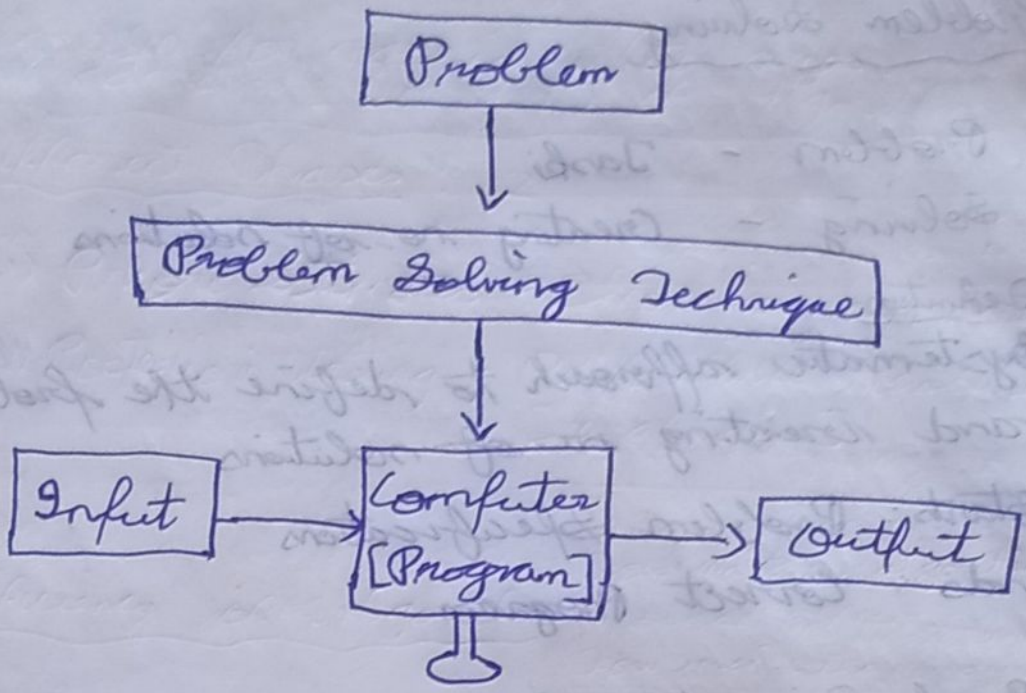
Problem Solving Technique

* Set of Techniques
* Provides logic for solving a problem

Types of Problem Solving Techniques: 4

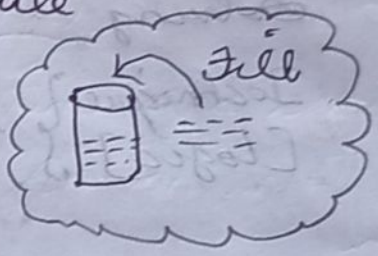
(i) Algorithm
(ii) Flowchart
(iii) Pseudo Codes
(iv) Programs

ALGORITHM



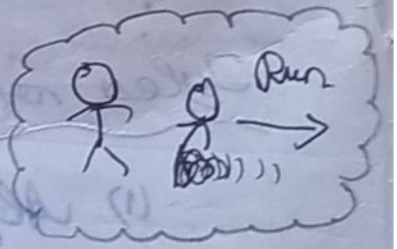
Real Life Example of Algorithm

Example 1: Filling Water Bottle



Step 1: Take water bottle
Step 2: Open lid
Step 3: Fill water in bottle
Step 4: Close lid

Example 2: Walking In Morning



Step 1: Wake up early in the morning
Step 2: Intake any energy drink (Refresh yourself)
Step 3: Wear shoes
Step 4: Run (or) Jog (or) Walk
Step 5: Take rest [Sit / Stand]
Step 6: Come home
Step 7: Refresh again
Step 8: Do other work.

Algorithm

ALGORITHM

Origin of name

Persian Author: Abu Jafar Mohammed ibn Musa al Khowarizmi

Synonym of name

Algorithm: Method to solve a problem

Definition

* Sequence (set) of instructions that describe a method for solving a problem.
* Step by step procedure to complete / accomplish a particular tasks.

Purpose of algorithms

* Developed to store, manipulate and retrieve data from data structures

Algorithmic strategy

* Way of designing an algorithm.

Real-Time Example:

Calculation of Factorial of a number.

Algorithmic Solution

Algorithm that yields [required result] expected output for a valid [legitimate] input in a finite amount of time.

Analysis of Algorithm

An estimation of the complexities of an algorithm for varying input sizes.

Different Phases : 2

(i) Priori estimates - Theoretical performance analysis
(ii) Posteriori testing - Performance Measurement [Evaluation]

Efficiency of Algorithm

(i) Time efficiency
How fast an algorithm runs
(ii) Space efficiency
How much extra memory it uses

* Utilization of complexities of algorithm [computational resources]

Complexity of Algorithm [f(n)]

* Measure of the amount of time and space required by an algorithm for an input of a given size (n).

Types of Complexity of algorithm: 2

(i) Time Complexity

* Amount of computer time taken to run an algorithm
* No. of steps taken by the algorithm to complete the process

(ii) Space Complexity

* Amount of computer memory required to run to its completion.

Space required by an algorithm
= Fixed part + Variable part

(i) Fixed part

* Total space required to store certain data and variables for an algorithm.

Ex: Simple variables and constants

(ii) Variable part

* Total space required by variables
* Variables sizes depend on the problem and its iteration

Ex: Recursion to calculate factorial of given value

Methods for determining efficiency

(i) Speed of the machine
(ii) Compiler and other system software tools
(iii) Operating System
(iv) Programming language
(v) Volume of data required.

Asymptotic Notations of algorithm

* Languages that use meaningful statements about time & space complexity.

(i) Big O - Worst case [Upper Bound]
(ii) Big Ω - Best case [Lower Bound]
(iii) Big Θ - Average case [Lower Bound = Upper Bound]

Qualities of good [best] algorithm

(i) Time - Less (ii) Memory - Less More (iii) Accuracy - More

Characteristics [Properties] of Algorithm

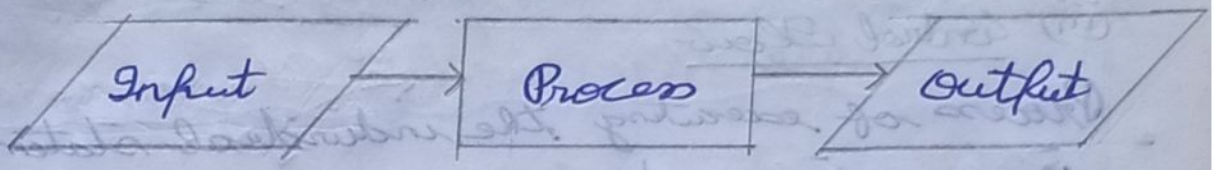
	Property	Description		Property	Description
(i)	Input	Suffly 0/more quantities	(vi)	Correctness	Error free
			(vii)	Simplicity	Easy to implement
(ii)	Output	Atleast 1 quantity Produce	(viii)	Unambiguous	Clear and unambiguous [only 1 meaning]
(iii)	Finiteness	Terminate after finite no of steps	(ix)	Feasibility	Feasible with avalable resources
(iv)	Definiteness	Well defined operations [acceptable]	(x)	Portable	Generic, Indefendent of any programming language
(v)	Effectiveness	Effective carry of every instruction	(xi)	Indefendent	Step-by-step directions - indefendent of any programming code

Space-Time Tradeoff / Time-Memory Tradeoff

* Way of solving a problem in ~~less time by~~ using (i) less time by using more storage space
(ii) less space by spending more time

Searching Techniques	Sorting Techniques
(i) Linear Search	(i) Bubble Sort
(ii) Binary Search	(ii) Selection Sort
	(iii) Insertion Sort
	(iv) Merge Sort (v) Quick Sort

Typical Algorithmic Flowchart



Example for an algorithm

(i) Write an algorithm to "print Good Morning"

Step 1: Start
Step 2: Display "Good Morning"
Step 3: Stop

(ii) Write an algorithm to "fill water bottle"

Step 1: Start
Step 2: Open water bottle cap
Step 3: Fill water bottle with water
Step 4: Close water bottle cap
Step 5: Stop

Various operations on Data - performed in DS

(i) Search (ii) Manipulate (iii) Retrieve

Algorithm Notations: used for calculations & data processing

Building Blocks of Algorithms

(i) Statement

Single action performed in a computer

(ii) State

Transition from one process to another under specified condition within a time.

(iii) Control Flow

Process of executing the individual statements in a given order.

Types: 3

A) Sequence

All instructions are executed one after another

B) Selection

Transfer of program control to a specific part of the program based upon condition.

C) Iteration / Looping

Execution of statements repeatedly upon conditional test

D) Functions

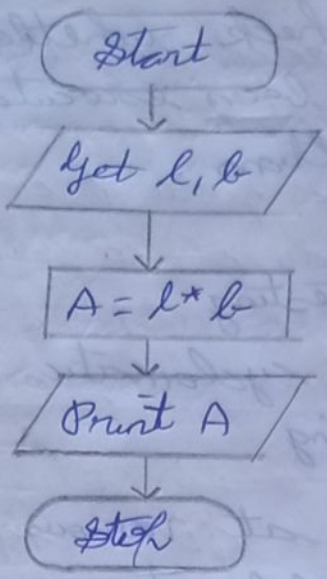
* Sub-Program
* Block of code [set of instructions]
* Performs a particular task.

(v) Modules

(vi) Classes And Objects

Write an algorithm to find area of rectangle.

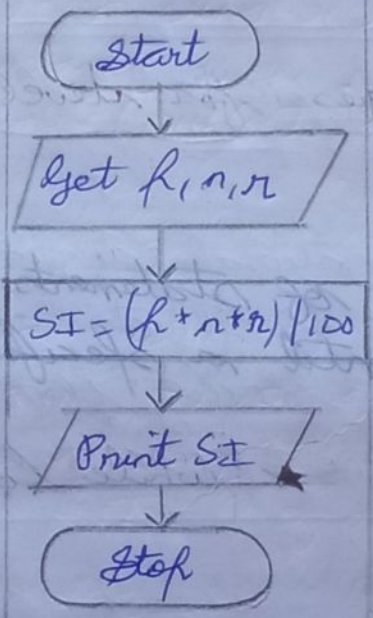
Step 1: Start
Step 2: get l, b values
Step 3: Calculate $A = l * b$
Step 4: Display A
Step 5: Stop



BEGIN
READ l, b
CALCULATE $A = l * b$
DISPLAY A
END

Write an algorithm to calculate simple interest

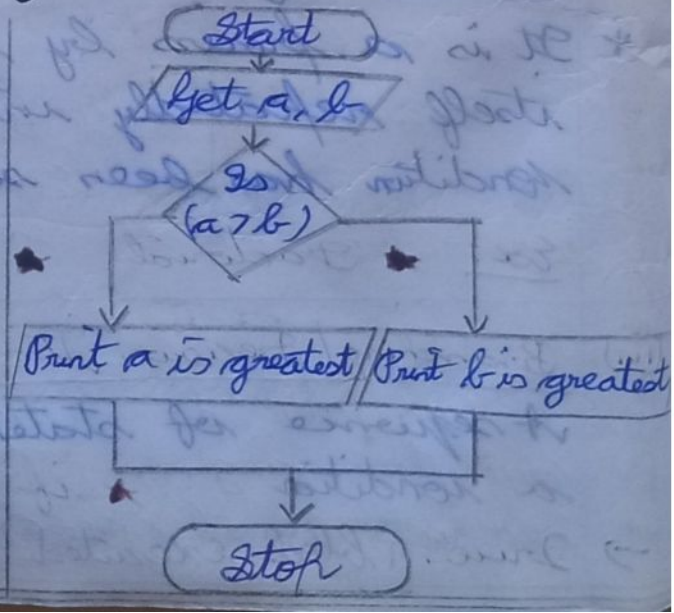
Step 1: Start
Step 2: Get p, n, r values
Step 3: Calculate $SI = \frac{(p * n * r)}{100}$
Step 4: Display SI
Step 5: Stop



BEGIN
READ p, n, r
CALCULATE SI
$SI = (p*n*r)/100$
DISPLAY SI
END

To check greatest of two numbers

Step 1: Start
Step 2: Get a, b value
Step 3: Check if $(a > b)$ print a is greater
Step 4: Else print b is greater
Step 5: Stop

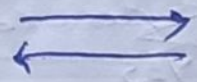
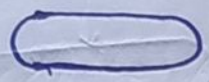
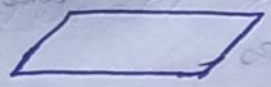
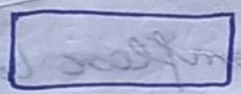
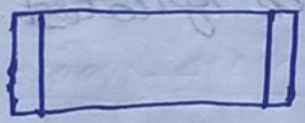


Flow Chart

FLOW CHART

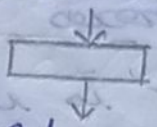
(Visual)

* Graphical (or) Pictorial representation of algorithm (logic) for problem solving
* Steps are drawn in the form of different shapes of boxes
* Logical flow is indicated by interconnecting arrows

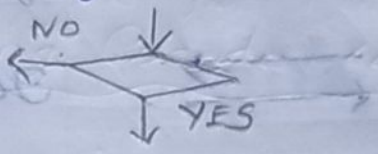
Symbols	Symbol Name	Description
	Flow lines	Connect symbols
	Terminal	Start, pause (or) Stop (Halt)
	Input / output	Information entering (or) leaving the system
	Processing	Arithmetic & Logical instructions
	Decision	Represents a decision to be made
	Connector	Join different flowlines
	Sub-function	Call function

Rules for drawing flowchart

* Clear, Neat & easy to follow
* Logical Start and Finish
* Only 1 flow line should come out from a process symbol



* Only 1 flow line should enter a decision symbol while 2/3 lines may leave the decision symbol



* Only 1 flow line is used with a terminal symbol

* Write briefly and precisely within standard symbols
* Avoid intersection of flowlines.

Advantages

* Communication
* Effective Analysis
* Proper Documentation
* Efficient Coding
* Proper Debugging
* Efficient Program Maintenance

Disadvantages

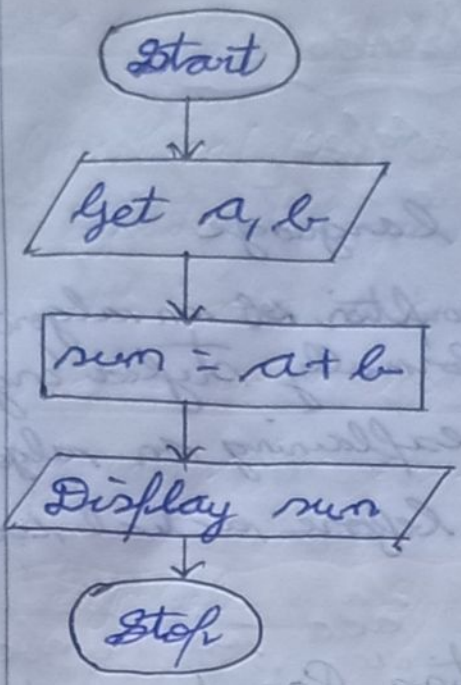
* Complex Logic
* Alterations and Modifications
* Reproduction
* Cost * Time
* No Standards

Example for a flowchart

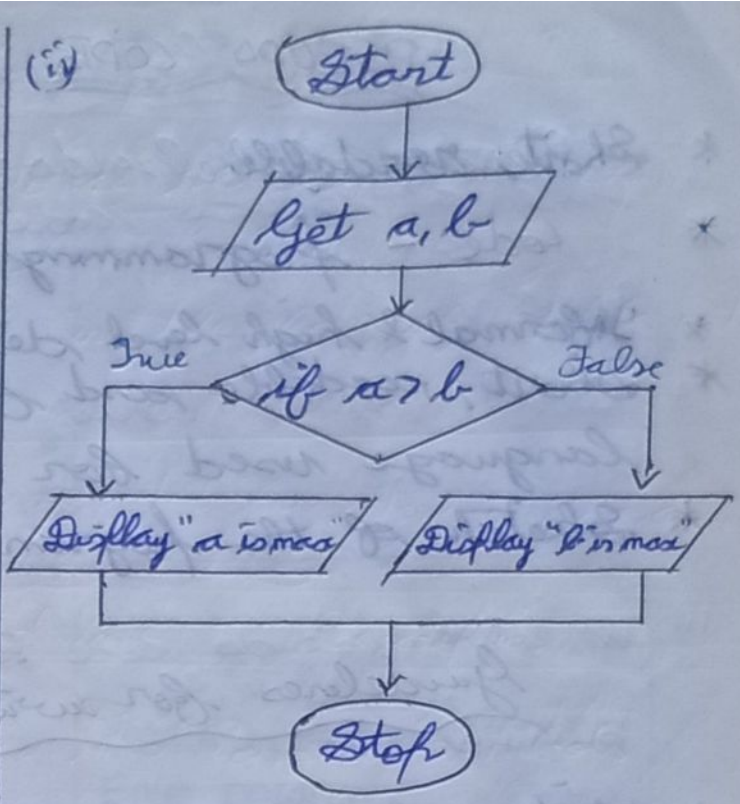
(i) Draw a flowchart to "add two numbers"

(ii) Draw a flowchart to "check greatest among 2 numbers.

(i)



(ii)



Algorithm (or) Pseudocode	Flowchart
* Written statement of algorithm	* Visual representation of algorithm
* Not visual	* Visual
* Easy to modify	* Difficult to modify
* Takes less time to design	* Takes more time to design
* More frequently used	* Less frequently used
* Not easy to understand and communicate	* Easy to understand and communicate

Pseudo Code

PSEUDO CODE

* Pseudo - false
* code - programming language
* Informal & high level description of an algorithm
* Short, readable and formally styled English language used for explaining an algorithm
* Sketch of the program before actual coding

Guidelines for writing Pseudo code

* Write one statement for line
* Capitalize initial keyword
* Indent to hierarchy
* End multiline structure
* Keep statements language independent

Common keywords used in Pseudo code

(i) // - Comment
(ii) BEGIN, START - First statement
(iii) END, STOP - Last statement
(iv) INPUT, GET, READ - Inputting data
(v) OUTPUT, PRINT, DISPLAY - Displaying data
(vi) COMPUTE, CALCULATE - Calculation of result of given expression
(vii) ADD, SUBTRACT, MULTIPLY, DIVIDE, INITIALIZE - Addition, Subtraction, Multiplication, Division, Initialization
(viii) IF, ELSE, ENDIF - Make Decision statements
(ix) FOR, ENDFOR, WHILE, ENDWHILE - Make Looping statements

Syntax of Pseudo code

if-else

```
IF (condition) THEN
  statement
  ....
ELSE
  statement
  ...
ENDIF
```

while

```
WHILE (condition) DO
   statement
   ...
ENDWHILE
```

do while

```
DO
   statement
   ...
   WHILE (condition)
END DOWHILE
```

for

```
FOR (start-value to end-value) DO
   statement
   ...
ENDFOR
```

repeat

```
REPEAT_UNTIL
```

Advantages

* Independent of any language [language independent]
* Used mostly
* Easy translation into programming language
* Easy modification
* Easy

Disadvantages

* No visual representation
* No accepted standards / standard syntax
* Not machine readable - no compilation / execution
* More difficult to follow the logic.

Example for a pseudo code

(i) Write a pseudo code to print "Good Morning"

```
BEGIN
PRINT "Good Morning"
END
```

(ii) Write a pseudo code to add two numbers.

```
BEGIN
GET a, b
ADD c = a+b
PRINT c
END
```

Algorithm	Flowchart	Pseudo code
* Sequence of instructions to solve a problem	* Graphical representation of algorithm	* High Level Description (or) Language Representation of algorithm
* User needs knowledge to write algorithm	* User not need knowledge of program to draw / understand flowchart	* User not need knowledge of program language to write / understand pseudo code

Programming Language

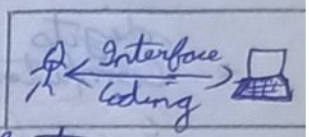
PROGRAM / PROGRAMMING LANGUAGE

Program (or) Code (or) Sniffet (or) Routine

Set of instructions to control computer's operation

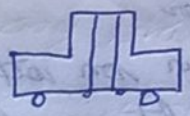
Programming Language / Coding

* Set of symbols and rules (syntax) [instructions] for instructing a computer to perform specific task.
* To communicate and simplify things



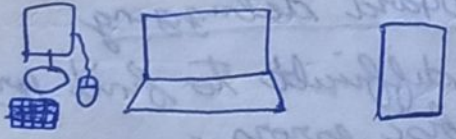
* Interface b/w humans and computer
* The user has to communicate with the computer using language which it can understand.
* The programmers have to follow all the specified rules before writing program using programming language.
* Most fun thing.

Real Time Example



Favourite Car (or) Fastest Car

Don't know how to drive it



Laptop / PC / Mobile

Keyboard - Typing

Don't know how to code (programming)

Types of Programming Language : 3

(i) Machine language (or) Low-level language (or) Binary language

Invented by: Gottfried Leibniz

* Two-symbol system ⇒ 0's and 1's
* The different instructions are formed by taking different combinations of 0's and 1's
* The binary code assigns a pattern of binary digits [bits] to each character, instruction, etc.
* Computer understands only this language.

Example:

Character	ASCII value
A	65

Decimal no.	Binary no.				
	8	4	2	1	method
$(12)_{10}$ ⇒ 12	1	1	0	0	⇒ $(1100)_2$
$(7)_{10}$ ⇒ 7	0	1	1	1	⇒ $(0111)_2$

Advantages

* **Translation free**
  → No conversion process
* **High speed & efficiency**
  → All data present in binary format
  → No need of translator
  → Execution time is saved

Disadvantages

* **Machine Dependent**
  → Differs from computer to computer
  → May not run on other type of computer architecture
* **Time-consuming process**
  → takes more time to write binary program.
* **Hard debugging**
  → difficult to find for) fix errors.

(ii) Assembly Language (or) Intermediate language

* Symbolic representation of machine language
* Symbolic programming language [logically equivalent]
* Mnemonic codes [Symbolic notation - MIPS format] to represent machine language instructions
* Also called low-level language
* Comes under Computer Architecture & Organization.

Ex: ADD a, b (or) ~~Add~~ add $t, a, b
beq rs, rt, L1

Translator: Assembler

Assembler

* Program (or) Translator
* Translates assembly language into machine language.

Advantages

* Simple
* Easy to read, write, understand and use
* Easy debugging
  → easy to locate and correct errors
* Memory efficient
  → requires less memory
* Less instruction
  → requires less instructions to get the result
* low-level embedded system

Disadvantages

* Machine dependent
  → depends on the architecture of that computer and OS
* Hard to learn
  → Have hardware knowledge to create applications.
  → Difficult syntax
* Less efficient
  → more execution time as translation is needed [assembler - conversion]
*

Key	Machine language	Assembly language
Understanding	Understand by computers	Understand only by humans not computers
Data Representation	Binary format [0's & 1's], Hexadecimal and Octadecimal	Mnemonics format [Add, Sub, Mov, End etc.]
Level of understanding	Very difficult	Easy compare to machine language
Modifications & Error fixing	Cannot be done	Can be done
Memorizable	Very difficult	Easy compare to machine language
Execution	Fast	Slow
Translator	No need	Need Assembler
Hardware Dependency	Dependent on hardware	Dependent on machine
Portability	Portable	Not portable
Ex	00101100	Ex: MOV AX, 12H

(iii) **High-level language**

* English words and symbols [natural language]
* Strong abstraction from the details of the computer.
* The specified rules are to be followed while writing program in high level language.

Ex: C program - printf("Hello World");
Python program - print("Hello World")
Java program - System.out.println("Hello World");

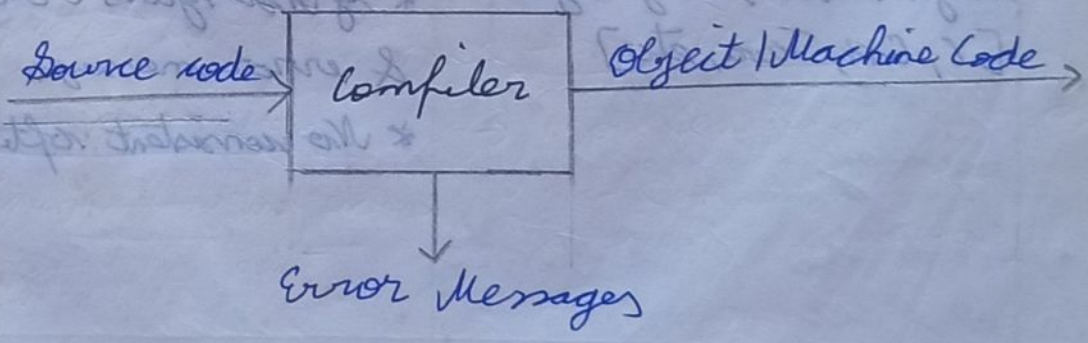
Translator:

(i) Compiler (ii) Interpreter

* Translator [compiler / interpreter] converts high level language code into machine language code.

(i) **Compiler**

* Program that translates whole source code (high level language) into object code (machine language)
* If any error is found, it display error message on the screen.



Types of compiler: 2

(i) One-pass compiler (ii) Multi-pass compiler

Features of compiler

* It compiles a large amount of code in less time
* It needed less amount of memory area to compile the source language
* It can compile only the modified code segment if frequent alterations are needed in the source code.
* While managing the hardware interrupts, best compilers connect closely with the operating system.

Advantages

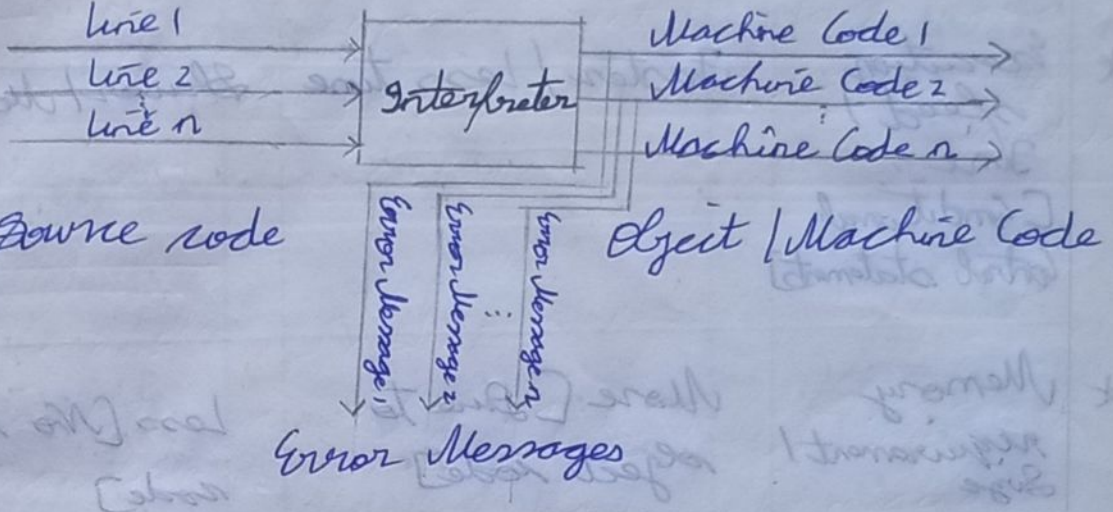
* Translation in single run
* Less time consumption
* More CPU utilization
* Concurrent error checking (syntatic & semantic)
* Easily supported by high-level languages [C, C++, Java, etc]

Disadvantages

* Not flexible
* More space consumption
* Difficult error localization
* Compilation need for every modification
* Not portable
* It Not gives diagnostic & error messages
* No consistant optimization

(ii) Interpreter

* Program that translates source code into machine code in line by line manner
* Executes 1 statement immediately before translating the next statement.
* When an error is found, the execution of the program is halted and error message is displayed on the screen.



Advantages

* Readability :-
→ easier to learn & understand
* Machine independent
→ Portable b/w machines
* Easy debugging
→ easy to find & correct error.

Disadvantages

* less efficient
→ More execution time
→ More memory space

Key	Compiler	Interpreter
* Input / Execution type	Entire program (Whole code)	[Single instruction] Line by line code of program
* Intermediate Object Code	Generated	Not generated
* Execution speed / Time [Conditional control statements]	Faster / Less time	Slower / More time
* Memory requirement / Size	More [Due to object code]	Less [No object code]
* Compilation / Interpretation of program	Compilation for everytime	Interpretation for everytime
* Flexibility	Not flexible	Flexible
* CPU utilization	More	Less
* Reorganization of program	Presence of error cause the whole program to be re-organized	Presence of error causes only part of program to be reorganized
* Error Display	After entire program is checked	For every instruction
* Localization of error	Difficult	Easy
* Examples	C, C++, Java	Python, Ruby, R, PHP

	Key	High level language	Low level language
*	Type of language environment	Programmer-friendly language	Machine-friendly language
*	Memory/time efficiency	Less	High
*	Understanding	Easy	Difficult
*	Debugging/Modification	Simple	Complex
*	Maintenance	Simple	Complex
*	Portability	Portable	Non-portable
*	Machine Dependency	Independent [run on any platform]	Dependent
*	Translator	Compiler (or) Interpreter	Assembler
*	Usage in programming	Widely used	Not commonly used
*	Abstraction	Much more	Little or none
*	Hardware Knowledge	Not required	Required (mandatory)
*	Facilities at hardware level	Not provide	Provide [very close to hardware]
*	Statement-Instruction Relationship	A single statement may execute several instructions	The statements can be directly mapped to process instructions
*	Examples	Pascal, C, C++, Java, Python, R, Ruby, Go, etc	Machine language & Assembly language

Computer System Level Hierarchy

* Combination of different levels that connects the computer with the user and that makes the use of the computer.
* How the computational activities are performed on the computer
* It shows all the elements used in different levels of system.

Levels of Computer System : 7

Level	Computer System	Example
Level 6	User	Executable Programs (Applications, Websites)
Level 5	High Level language	C, C++, Java, Python
Level 4	Assembly language	Assembly Code [MIPS]
Level 3	System Software	Operating System (OS)
Level 2	Machine / Binary language [Low Level language]	Instruction Set Architecture
Level 1	Control Unit	Microcode [Microprogrammed or Hardwired]
Level 0	Digital logic	Circuits, gates, etc.

Categories (or) Classification of High Level Languages

I) Basis : Application

ii) Commercial programming languages

* Dedicated to the commercial domain
* Specifically designed for solving business-related programs.
* Used in organization for processing, handling the data related to payroll, accounts payable and task building applications.

Example: COBOL

(ii) Scientific programming languages

* Dedicated to the scientific domain
* Specially designed for solving different scientific and mathematical problems.
* Used to develop programs for performing complex calculation during scientific research.

Example: FORTRAN, C, Python, R, Matlab

(iii) Special Purpose programming languages

* Dedicated to a particular domain area
* Specially designed for performing some dedicated functions.

Example: SQL - interact with database programs
HTML - formatting of webpages

(iv) General Purpose programming languages

* Dedicated to general domain area.
* Specially designed for developing different types of software application regardless of their application area.

Example: BASIC, C, C++, Java, Python

II.) Basis: Design Paradigm

* Paradigm = Organizing principle of program - approach to programming

(i) Markup programming language

* Artificial language that uses annotations to text in document that define how the text is to be displayed. [to present information]
* Interpreted by the browser

Example: HTML, XML, XHTML

(ii) Concurrent / parallel programming language

* Computer programming technique executed as a means of structuring a program
* It provides for the execution of processes (or) operations concurrently (or) simultaneously (or) parallelly (or) threads of execution
* Execution takes either within single computer or across no. of systems

Example: Java, Scala, C, C++, Joule, Limbo

(iii) Scripting Programming Languages

* For a runtime system, it control on application
* It automates the execution of frequent tasks that would otherwise be performed individually by a human operator.
* Scripts can execute independent of any other application.
* Mostly embedded in the application.
* Used in communicating with external program.

Example: Python, Ruby, JavaScript, AppleScript, VBScript.
Perl

(iv) Compiled Programming Languages

* Implementations are typically compilers and not interpreters.
* It converts whole source code into machine code.

Example: C, C++, C#, Java

(v) Interpreted Programming Languages

* Implementations are typically interpreters and not compilers.
* It translates source code into machine code and executes immediately without compiling it.

Example: Python, PHP, Pascal

(vi) Logic Programming language

* Based on specifying rules and facts
* Could be applied in AI and computational linguistics
* Lays a stress on 'what' more than 'how', as a mark of revolution
* Can adequately be used only for solving the very complex problem which has a set of very clear and definite rules.

(vi) Functional (or) Modular Programming Language

* Sub division of a computer program into separate sub programs (modules)
* Examples of modules:
  * procedures * sub routines * functions
* Defines every computation as a mathematical evaluation
* Focus on the mathematical calculations on which they bound.

Example: Clean, Haskell, C

(vii) Procedural Programming Language [Imperative]

* Programming model derived from structured programming, based upon the concept of calling procedure.
* It implies specifying the steps that the programs should take to reach to an intended state.
* A procedure is a group of statements that can be referred through a procedure call
* Procedures help in the reuse of codes
* Makes the programs structured and easily traceable for program flow.

Example: HyperTalk, MATLAB.

(viii) Object - oriented Programming language

* Programming paradigm/model based on the concept of objects.
* Objects contain data in the form of attributes and code in the form of methods (procedures)
* Objects interact with real world.

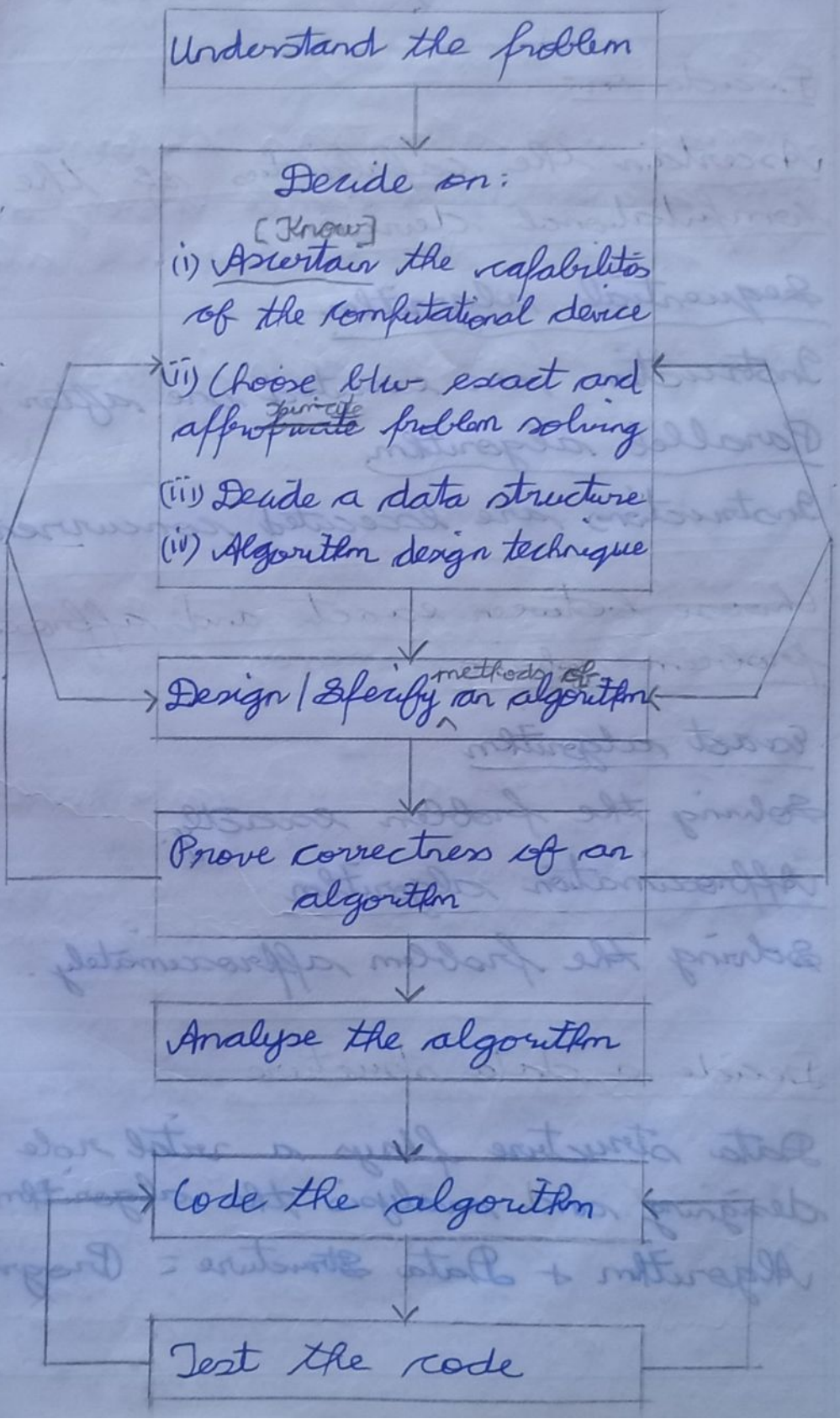
Example: Java, Moto, Objective-C, C++, Java, Python, PHP.

Algorithmic Problem Solving

Algorithmic Problem Solving

* Solving problem that require the formulation of an algorithm for the solution.

Flowchart for Algorithmic Problem Solving



1) Understand the problem

* Finding the valid input of the problem that the algorithm solves.
* Specifying exactly the set of inputs the algorithm needs to handle.
* A correct algorithm works correctly for all legitimate inputs.

2) Decide on:

(i) Ascertain the capabilities of the computational device

* Sequential algorithm

Instructions are executed one after another

* Parallel algorithm

Instructions are executed concurrently

(ii) Choose between exact and approximate problem solving

* Exact algorithm

Solving the problem exactly.

* Approximation algorithm

Solving the problem approximately.

(iii) Decide a data structure

* Data structure plays a vital role in designing and analysis the algorithms.
* Algorithm + Data Structure = Programs

(iv) Algorithm Design Techniques / Strategy / Paradigm

* General approach to solving problems algorithmically that
* Applicable to a variety of problems from different areas of computing.

Utmost importance of learning algorithm design techniques

* Provide guidance for designing algorithms for new problems
* Algorithms are the cornerstone of computer science.

(v) Design / Specify Methods of an algorithm

★ Pseudocode

* Mixture of natural language and programming language-like constructs ★
* Usually more precise than natural language
* Often yields more [short] succint algorithm descriptions ★

★ Flowchart

* In the earlier days of computing, the dominant vehicle for specifying algorithms was a flowchart.
* Expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.

★ Programming Language / Coding

* Implementation [Conversion] of algorithm written in particular computer language
* Fed into an electronic computer directly.

(vi) Prove correctness of an algorithm

* After specification of an algorithm, to prove that the algo be an algorithmic solution
* A common technique for proving correctness is to use mathematical induction [logic].
* An algorithm's iterations provide a natural sequence of steps needed for such proofs.
*

Incorrect algorithm

* Tracing the algorithm's performance for a few specific inputs can be a very worthwhile activity but it cannot prove the algorithm's correctness conclusively
* Incorrect algorithm fails in just 1 instance of its input.

(vii) Analyse the algorithm

★ Efficiency (Performance) → Time efficiency, Space efficiency → accuracy
* Run faster in less memory accurately
★ Simplicity
* Precise definition and investigation with mathematical expressions
* Simple algorithms are easier to understand, easier to code and usually contain fewer bugs.

(viii) Code the algorithm

* Destiny to programming an algorithm as computer programs
* Presents both a peril and an opportunity
* A working program provides an additional opportunity in allowing an empirical analysis of the underlying algorithm based on timing the program on several inputs & then analysing the results obtained

(ix) **Test the code**

* A way to check whether each & every line of code has been executed without any errors & exceptions.
* It involves:
  - ★ dynamic testing
  - ★ calculating cyclomatic complexity
  - ★ static testing
* Carried out at various levels:
  - ★ code development
  - ★ code inspection
  - ★ unit testing

Simple strategies for developing algorithms

(i) **Iteration**

A sequence of statements is executed repeatedly until a specified condition is true

★ for loop ★ while loop

(ii) **Recursion**

* A function that calls itself is called recursive function.
* It is a process by which a function calls itself repeatedly until some specified condition has been satisfied

Ex: ★ Factorial ★ Fibonacci series ★ Tower of Hanoi

(iii) **Branching / Decision Making**

A sequence of statements executed based on a condition ★ if ★ if..else ★ switch case

→ True: 1 block executed → False: Another block executed