Topic 7: Algorithm Design and Problem Solving

IGCSE Computer Science 0478 | SenpaiCorner | Mr. Wei

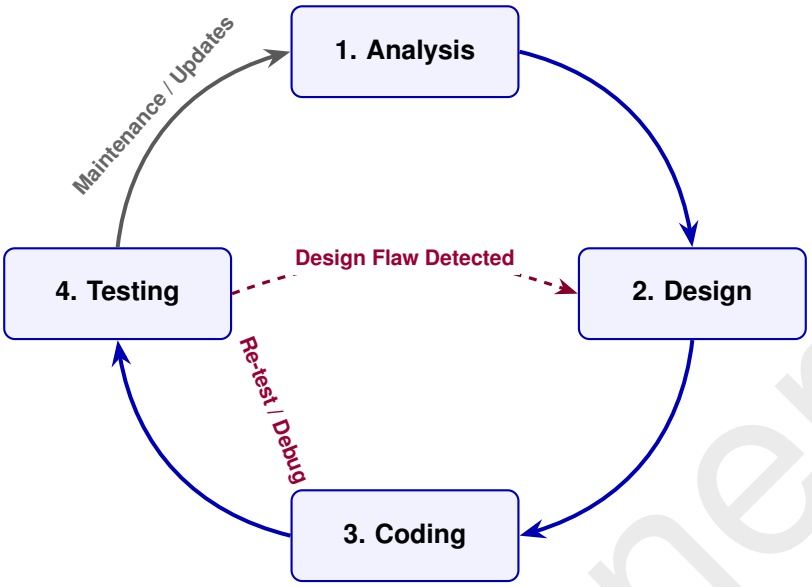
Learning Objectives

- Describe the four stages of the program development life cycle: analysis, design, coding, testing.
- Apply abstraction and decomposition to break complex problems into sub-systems.
- Produce and interpret structure diagrams, flowcharts, and pseudocode.
- Trace and write standard algorithms: linear search, bubble sort, totalling, counting, maximum, minimum, average.
- Identify and apply all six validation check types: range, length, type, presence, format, check digit.
- Distinguish validation from verification; apply visual and double-entry verification.
- Classify test data as normal, abnormal, extreme, or boundary.
- Complete trace tables and perform dry-runs of algorithms.
- Identify logical and syntax errors in algorithms and suggest corrections.
- Write and amend algorithms in pseudocode, flowchart, and program code form.

7.1 The Program Development Life Cycle

The creation of a program follows a structured four-stage process called the **program development life cycle (PDLC)**.

1. **Analysis:** Investigate and define the problem. Key tasks are **abstraction** (filtering out unnecessary detail to focus on what matters) and **decomposition** (breaking the problem into smaller, manageable parts), as well as identifying specific system requirements.
2. **Design:** Plan the solution by producing structure diagrams, flowcharts, and pseudocode before any code is written.
3. **Coding:** Write the program code. Iterative testing is performed *during* this stage as individual modules are developed.
4. **Testing:** Execute the completed program using carefully chosen test data to confirm it handles all expected and unexpected inputs correctly.



Keyword Associations — PDLC Stages

Match exam keywords to stages: **Analysis** → abstraction, decomposition, requirements. **Design** → flowcharts, pseudocode, structure diagrams. **Coding** → writing code, iterative testing. **Testing** → test data, trace tables, error correction.

7.2 Decomposition and Designing a Solution

Decomposition is the process of breaking a large, complex problem into smaller, more manageable sub-systems. Each sub-system can be further decomposed if needed.

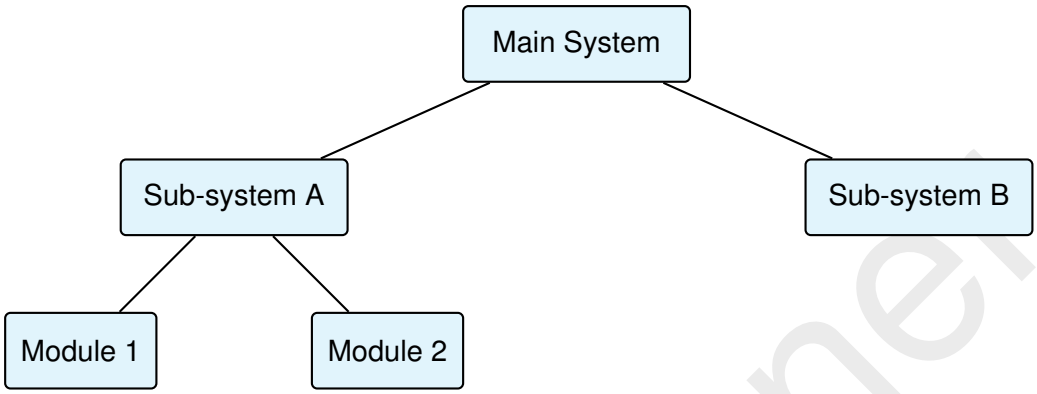
When analysing any system or sub-system, identify its four key components:

Component	Description
Inputs	Data entered into the system.
Processes	How data is manipulated, calculated, or transformed.
Outputs	Results presented to the user or passed to another system.
Storage	Data saved for later retrieval or use.

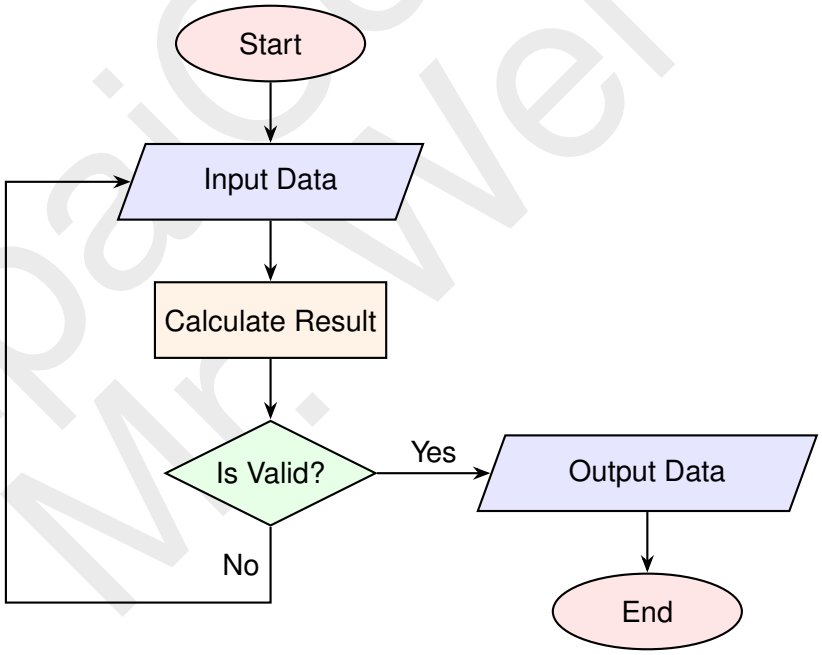
Decomposing a Scenario in the Exam

When asked to decompose a system, structure your answer with explicit headings for **Inputs**, **Processes**, and **Outputs**. This maps directly to mark scheme categories and prevents missed marks.

1. Structure Diagram



2. Flowchart



3. Pseudocode

```
BEGIN
  READ input_data
  REPEAT
    result = PROCESS(input_data)
  UNTIL result is valid
  IF result > 0 THEN
    PRINT result
  END IF
END
```

7.3 Standard Algorithms

An algorithm is a step-by-step set of instructions designed to solve a specific problem. You must be able to state the purpose of a given algorithm, trace it, and write your own.

7.3.1 Linear Search

Checks each item in a list *sequentially* from the first element until the target item is found or the end of the list is reached.

```
Found <- FALSE
Index <- 1

WHILE Index <= 10 AND Found = FALSE
    IF List[Index] = Target THEN
        Found <- TRUE
        OUTPUT "Found at position ", Index
    ENDIF
    Index <- Index + 1
ENDWHILE

IF Found = FALSE THEN
    OUTPUT "Item not found"
ENDIF
```

7.3.2 Bubble Sort

Repeatedly steps through a list, *comparing adjacent elements* and swapping them if they are in the wrong order. Each full pass moves the largest unsorted element to its correct position. The algorithm stops when a complete pass produces no swaps.

```
Swapped <- TRUE

WHILE Swapped = TRUE
    Swapped <- FALSE
    FOR Index <- 1 TO 9
        IF List[Index] > List[Index + 1] THEN
            Temp <- List[Index]
            List[Index] <- List[Index + 1]
            List[Index + 1] <- Temp
            Swapped <- TRUE
        ENDIF
    NEXT Index
ENDWHILE
```

Bubble Sort Termination Condition

The outer loop must continue until a full pass produces **no swaps** — not simply a fixed number of passes. A fixed count may leave the list partially unsorted.

7.3.3 Totalling, Counting, and Average

- **Totalling:** Initialise a variable to 0 before the loop; add each value to it inside the loop.
- **Counting:** Initialise a variable to 0; increment it each time a condition is met.
- **Average:** Computed as Total ÷ Count after the loop.

```
Total <- 0
Count <- 0

FOR Index <- 1 TO 10
    OUTPUT "Enter a number:"
    INPUT Number
    Total <- Total + Number
    Count <- Count + 1
NEXT Index

Average <- Total / Count
OUTPUT "Total: ", Total, "  Average: ", Average
```

7.3.4 Finding Maximum and Minimum

Initialise the tracking variable to a value lower than any possible input (for maximum) or higher than any possible input (for minimum). Update it each time a more extreme value is found.

```
MaxScore <- -1

FOR Index <- 1 TO 5
    OUTPUT "Enter student score:"
    INPUT Score
    IF Score > MaxScore THEN
        MaxScore <- Score
    ENDIF
NEXT Index

OUTPUT "Highest score: ", MaxScore
```

Initialising Max/Min Variables

Do not initialise `MaxScore` to 0 if input values could be negative — the result will always be incorrect. Use a known lower bound such as -1 or the value of the first element.

7.4 Validation and Verification

Data entered into a computer must be checked to ensure it is both sensible and accurate. These are two distinct mechanisms.

7.4.1 Validation

Validation is an *automatic check performed by the computer* to ensure input data is reasonable and follows accepted criteria. It does **not** guarantee data is correct — only that it is plausible.

Check Type	Description and Example
Range check	Verifies a value falls within a specified upper and lower boundary. *E.g. age must be between 1 and 120.*
Length check	Verifies the number of characters is correct. *E.g. a password must be at least 8 characters long.*
Type check	Verifies data is of the correct data type. *E.g. age must be numeric, not alphabetic.*
Presence check	Verifies that a required field has not been left blank.
Format check	Verifies data matches a predefined pattern. *E.g. an email address must contain an '@' symbol.*
Check digit	An additional digit appended to a number (e.g. a barcode) calculated from the other digits to detect data entry errors.

Pseudocode Example — Range Check:

```
REPEAT
    OUTPUT "Enter a percentage score (0-100):"
    INPUT Score
    IF Score < 0 OR Score > 100 THEN
        OUTPUT "Invalid score. Please try again."
    ENDIF
UNTIL Score >= 0 AND Score <= 100
```

7.4.2 Verification

Verification checks that data entered matches the *original source data* exactly. It catches transcription errors — values that might pass validation but were typed incorrectly.

Verification Method	Description
Visual check	The user manually reads data on screen and compares it against the original source document.
Double entry check	The user enters the data twice. The computer compares both entries; a mismatch triggers an error.

Pseudocode Example — Double Entry Verification:

```
OUTPUT "Enter your new password:"
INPUT Pass1
OUTPUT "Re-enter your new password:"
INPUT Pass2

IF Pass1 = Pass2 THEN
    OUTPUT "Password successfully set."
ELSE
    OUTPUT "Passwords do not match. Verification failed."
ENDIF
```

Validation vs Verification — Never Confuse These

Validation checks if data is *sensible* — the computer does this automatically.

Verification checks if data is *accurate to the original source* — a human comparison or repeat-entry is required.

A validated value can still be wrong: entering age 25 when the real age is 52 passes any range check.

7.5 Test Data

To ensure an algorithm works correctly under all conditions, it must be tested with different **categories** of test data.

Type	Description	Example (range 1–100)
Normal	Data the program should accept and process correctly.	45, 72, 10
Abnormal	Data the program should reject (wrong type or clearly out of scope).	`hello`, -5, 200
Extreme	The largest or smallest *acceptable* value.	1 and 100
Boundary	A pair: the edge accepted value **and** the closest rejected value.	1 & 0 ; 100 & 101

Extreme vs Boundary — Key Distinction

Extreme data is a single edge-accepted value (e.g. 100 for a range of 1–100).

Boundary data is a *pair*: the edge accepted value and the immediately rejected value (e.g. 100 *and* 101). Writing only one value when asked for boundary data loses marks.

7.6 Trace Tables and Dry-Runs

A **trace table** documents a **dry-run** of an algorithm — manually stepping through the code and recording the value of every variable at each step without running it on a computer.

Procedure:

1. Write each variable name (including the loop counter) as a column header.
2. Trace through the code line by line.
3. Each time any variable changes value, write the new value on a *new row* in that column.
4. Never erase previous values — the full history must be visible for marks to be awarded.

Example — Totalling loop with inputs 3, 7, 5 (3 iterations):

Index	Number	Total	Count
		0	0
1	3	3	1
2	7	10	2
3	5	15	3

Average $= 15 \div 3 = 5$

Completing Trace Tables in the Exam

Always include the loop counter as its own column. Examiners award marks per iteration — a missing row loses marks even if the final value shown is correct.

Identifying Errors in Algorithms

You must be able to read an algorithm, identify errors, and suggest corrections. Common error patterns:

- **Off-by-one errors:** A loop runs one too many or one too few times. E.g. `Index > 10` instead of `Index >= 10`.
- **Uninitialised accumulator variables:** Totalling or counting variables not set to 0 before the loop begins.
- **Missing user prompts:** An `INPUT` statement with no preceding `OUTPUT` to tell the user what to enter.

Off-by-One Errors in Loop Bounds

To process indices 1 to 10, the loop condition must be `Index <= 10`, *not* `Index < 10`. Always verify both the start and end values with a single manual iteration before finalising your answer.

7.7 Writing and Amending Algorithms

When writing or amending algorithms you must express logic correctly in **pseudocode**, **flowcharts**, or program code as the question requires.

Requirements for well-written algorithms:

- Initialise all variables — especially totals and counts — to 0 before any loop begins.
- Include an `OUTPUT` prompt *before every* `INPUT` statement.
- Use correct mathematical and logical operators (`>`, `<=`, `AND`, `OR`, `MOD`, `DIV`). Writing "greater than" in plain English is **not** acceptable.
- Ensure all loop bounds and conditions are precise — verify with a one-iteration mental trace.

Paper 2 Scenario Question — 15-Mark Checklist

Before finishing any algorithm: (1) initialise all totalling/counting variables at the very start; (2) add an `OUTPUT` prompt before every `INPUT`; (3) use correct pseudocode operators not plain English; (4) perform a one-iteration trace to verify loop bounds are correct. These are simple guaranteed marks.

Worked Example — Student Mark Tracker

Task: Read 5 student marks. Output the highest mark and the average.

Key decisions: initialise `Total` to 0 and `MaxMark` to a value lower than any possible mark; prompt before each input; calculate average *after* the loop; update `MaxMark` on every iteration where the current mark exceeds it.

```
Total <- 0
MaxMark <- -1

FOR Index <- 1 TO 5
    OUTPUT "Enter mark for student ", Index, ":"
    INPUT Mark
    Total <- Total + Mark
    IF Mark > MaxMark THEN
        MaxMark <- Mark
    ENDIF
NEXT Index

Average <- Total / 5
OUTPUT "Highest mark: ", MaxMark
OUTPUT "Average mark: ", Average
```