CAIE Computer Science IGCSE

7 - Algorithm design and problem-solving

Advanced Notes

Program development life cycle

The program development life cycle describes the main stages involved in creating a program. The four stages are: analysis, design, coding, and testing.

Stage	Purpose	Tasks
Analysis	Understand the problem and what the solution needs to achieve.	**Abstraction**: Removing unnecessary details to focus only on what is important. **Decomposition**: Breaking the overall problem into smaller, more manageable sub-problems. **Identification of the problem**: Clearly state what the problem is. **Identification of requirements**: Work out what inputs, processes, and outputs are required.
Design	Plan the solution before writing code.	**Decomposition**: Continue breaking down sub-problems into smaller steps if needed. **Design the solution**: The solution to a problem can be represented in several ways, including structure diagrams, flowcharts, and pseudocode, to provide a plan for the coding stage.
Coding	Translate the design into a working program.	**Writing program code**: Use a programming language to implement the design. **Iterative testing**: Test the code repeatedly as it is written, fixing errors as they occur.
Testing	Check that the program works correctly and meets requirements.	**Test program code:** Run the program code, using test data to make sure that it produces the correct output.

System decomposition

Every computer system is made up of sub-systems. Each sub-system is made up of further sub-systems. Breaking a system into sub-systems makes it easier to design, understand, and test.

Decomposition is the process of breaking a problem into smaller, more manageable parts. Each smaller part can then be solved separately. Each part will usually involve:

- **Inputs:** Data that goes into the system
- **Processes:** Actions or calculations performed on the data
- **Outputs:** Information produced by the system
- **Storage:** Data saved for later use

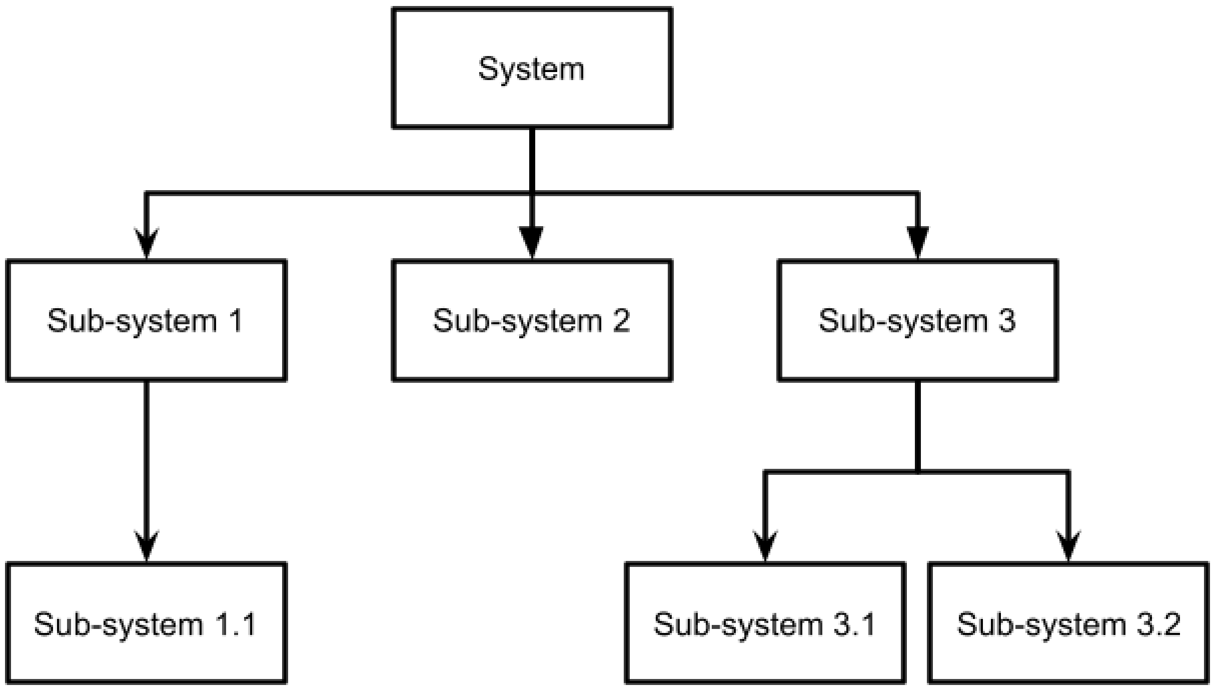
Methods to design and construct a solution to a problem

In the design stage, the solution to a problem can be represented in the following forms.

Structure diagrams

Structure diagrams can be used to visually represent the process of decomposing a problem. Each level, further down the structure diagram, shows tasks further broken down.

When coding the solution for the problem, each sub-system can be developed independently, allowing work to be made on the system in parallel (by spreading the workload across a team) and making testing and debugging easier.



Pseudocode

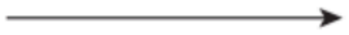
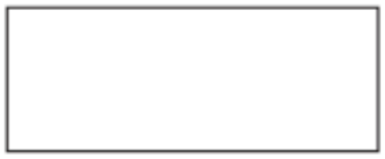
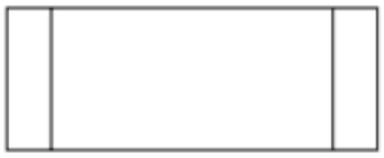
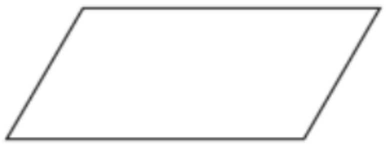
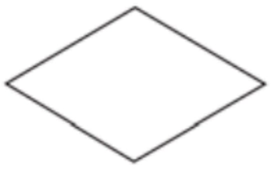
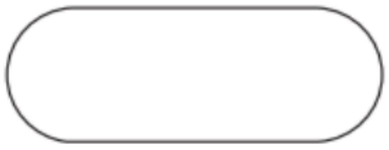
Pseudocode is a text-based way of describing a solution in structured steps. It uses programming-like keywords but is language-independent. There is a specific set of pseudocode keywords available online for the Cambridge IGCSE Computer Science exam.

Here's an example of some pseudocode to ask the user to enter a number repeatedly until the number they enter is less than or equal to 100.

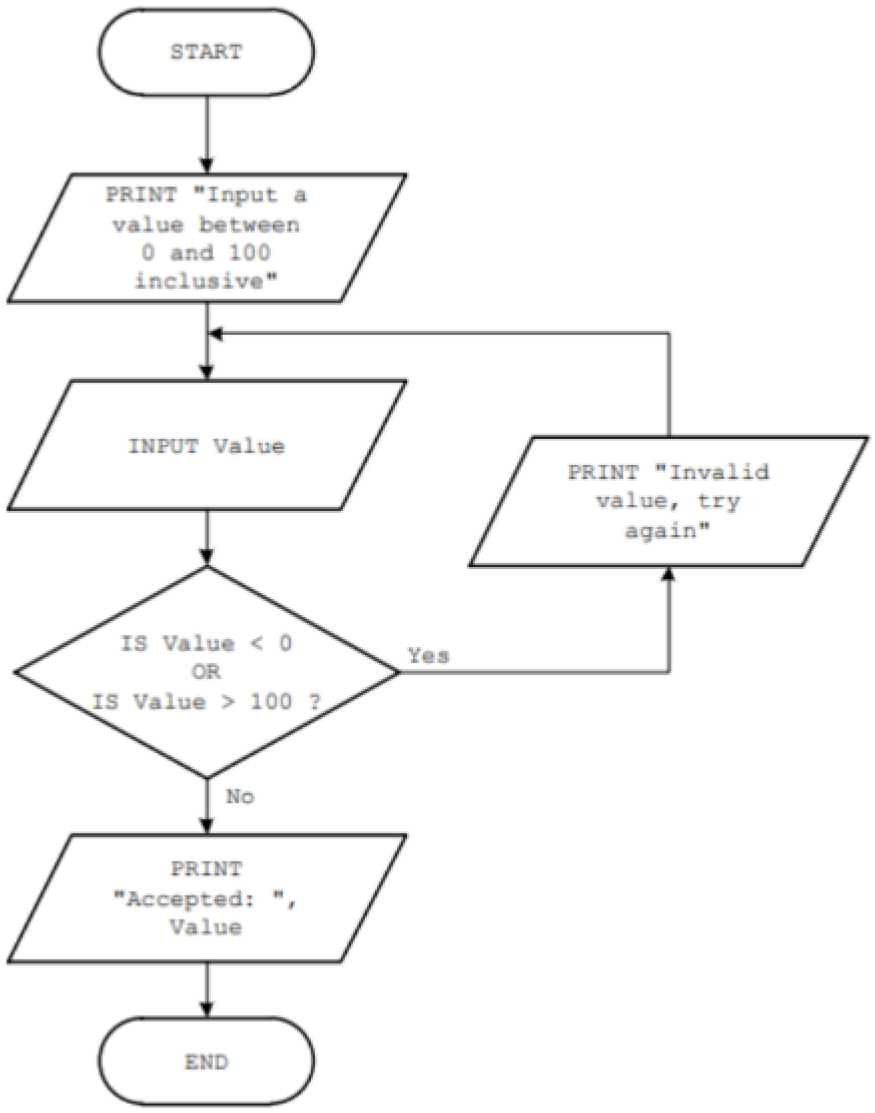
```
INPUT Number
WHILE Number > 100 DO
    OUTPUT "The number is too large"
    INPUT Number
ENDWHILE
OUTPUT "The number is acceptable"
```

Flowcharts

Flowcharts are diagrams that show the step-by-step flow of a process. You should be aware of the following flowchart symbols:

	**Flow line**	An arrow represents control passing between the connected shapes.
	**Process**	This shape represents something being performed or done.
	**Subroutine**	This shape represents a subroutine call that will relate to a separate, non-linked flowchart.
	**Input/Output**	This shape represents the input or output of something into or out of the flowchart.
	**Decision**	This shape represents a decision (Yes/No or True/False) that results in two lines representing the different possible outcomes.
	**Terminator**	This shape represents the 'Start' and 'Stop' of the process.

Making use of these symbols, here's a typical flowchart to outline a process for inputting a value and checking if it falls within a specific range (0 to 100 inclusive).


START
PRINT "Input a value between 0 and 100 inclusive"
INPUT Value
IS Value < 0 OR IS Value > 100 ?
Yes
PRINT "Invalid value, try again"
No
PRINT "Accepted: ", Value
END

Standard methods of solution

The following methods of solution can be used to solve many different problems.

Linear Search

Linear search is a searching algorithm, used to find items in a list. You can think of it as going along a bookshelf one by one until you come across the book you're looking for. Sometimes the algorithm gets lucky and finds the desired element almost immediately, while in other situations, if the desired element is towards the end of a list or the list is incredibly long, the algorithm is incredibly inefficient.

Example

Find the position of Apple in the data.

0	1	2	3	4
Banana	Orange	Apple	Kiwi	Mango

First we inspect position 0, and find Banana. Not the element we're after.

0	1	2	3	4
Banana	Orange	Apple	Kiwi	Mango

Next we inspect position 1, finding Orange. Again, not what we're looking for.

0	1	2	3	4
Banana	Orange	Apple	Kiwi	Mango

Next we look at position 2 and find Apple, this is the data we're looking for and so the algorithm returns the index of Apple (2) and terminates.

Note: if the item was not found, a suitable error message should be output

Bubble sort

Sorting algorithms are designed to take a number of elements in any order and output them in a logical order. Bubble sort is a common form of sorting algorithm that makes comparisons and swaps between adjacent elements. The largest element in the unsorted part of the input is said to "bubble" to the top of the data with each iteration of the algorithm.

The algorithm starts at the first element in an array and compares it to the second. If they are in the wrong order, the algorithm swaps the pair. Otherwise, the algorithm moves on. The process is then repeated for every adjacent pair of elements in the array until the end of the array is reached (at which point the largest element is in the last position of the array).

This is referred to as one pass of the algorithm. For an array with n elements, the algorithm will perform a maximum of n passes through the data, at which point the input is sorted and can be returned.

Example

Use bubble sort to arrange the elements in the array into ascending order.

4	9	1	6	7

The two first elements are compared. They are in the correct order, so the algorithm moves on.

4	9	1	6	7
4	1	9	6	7

The second two elements are in the wrong order, so they are swapped.

4	1	9	6	7
4	1	6	9	7

Again, the next two elements are in the wrong order, so they are swapped.

4	1	6	9	7
4	1	6	7	9

The last two elements are also in the wrong order, so are swapped.

This marks the end of the first pass of the algorithm.

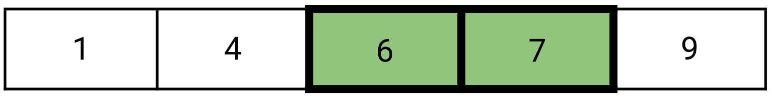
The second pass starts at the beginning of the array and compares the first two elements. They are in the wrong order, so are swapped.

4	1	6	7	9
1	4	6	7	9

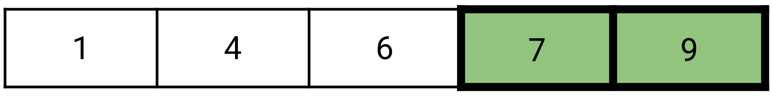
The next two elements are in the correct order, so the algorithm moves on.

1	4	6	7	9

Again the comparison reveals that the elements are in the correct order.



The last pair are also in the correct order, so no swap is made.



This is the end of the second pass, and we can see that the data is sorted. However, the algorithm wouldn't terminate here. It would carry out another final pass before terminating, to confirm that the data is sorted by checking that no swaps are made.

Totalling

Totalling is used to work out the sum of a set of values. There are lots of scenarios where totalling can be used, such as adding up a player's score in each round of a game to find their total score.

To find the total, create a variable that is initialised as 0, and then add each value to the total using a loop.

Counting

Counting is used to count how many times something occurs.

To count the number of times something occurs, create a variable that is initialised as 0, and increase the counter by 1 each time the condition is met. This is often used within WHILE loops to control the number of repetitions.

Finding minimum and maximum values

To identify the largest or smallest value in a set:

- Start with the first value as the current maximum or minimum.
- Compare each new value with the current maximum/minimum and update if needed.

Finding the average value

To calculate the mean value in a set of numbers, use totalling to find the sum of all numbers. Then divide the total by the count of numbers.
To find the median, select the item at the middle index (which can be found by dividing the maximum index by 2).

Validation

Validation is the process of checking that input data is reasonable and sensible before it is accepted by the system. It prevents errors caused by incorrect or unrealistic data being entered. There are several different types of validation check.

Range check

Ensures data is within a specified range.

Example: Age must be between 0 and 120.

Length check

Ensures data has the correct number of characters.

Example: A UK postcode should not exceed 8 characters.

Type check

Ensures data is of the correct data type.

Example: An age field must be numeric, not text.

Presence check

Ensures data is actually entered (not left blank).

Example: A name field must not be empty.

Format check

Ensures data is in the correct format or pattern.

Example: An email address must contain "@".

Check digit

Extra digit calculated from the other digits, used to detect errors in entry.

Example: Commonly used in ISBN numbers and barcodes.

Verification

Verification is the process of checking that data has been correctly entered into the system, matching the original source. It prevents errors that occur during data entry (for example, typing mistakes).

Visual check

The user compares the data entered with the original source, checked by eye.

Example: Proofreading a typed-in address against a paper form.

Double entry check

Data is entered twice and compared by the system. If the two entries do not match, the user must re-enter the data.

Example: Entering and confirming a new password.

Test data

Testing is used to make sure that a program works correctly and handles all possible inputs. Different types of test data are used to check how the program responds.

Normal data

Data that is typical, valid, and within the expected range. Used to check the program works under usual conditions.

Example: If age must be between 0 and 120, normal data could be 35.

Abnormal data

Data that is not acceptable and should be rejected. Used to check the program prevents incorrect inputs, which could be common due to user misuse.

Example: If age must be between 0 and 120, abnormal data could be -5 or "hello".

Extreme data

The largest and smallest values that are still acceptable. Used to check the program handles the very limits correctly.

Example: If age must be between 0 and 120, extreme data could be 0 and 120.

Boundary data

Includes both:

- The largest/smallest acceptable value (extreme).
- The next value just outside the limit (invalid/abnormal).

This ensures the program accepts the correct values and rejects the wrong ones without causing any errors.

Example: If age must be between 0 and 120, boundary data could be:
Acceptable: 0, 120
Rejected: -1, 121

Trace Tables and Dry-Runs

A dry-run is when you manually go through an algorithm step by step without running it on a computer. It helps to check the logic, spot errors, and understand how variables change.

A trace table is used to record what happens during a dry-run. It shows the values of variables, any outputs, and any user prompts at each step of the algorithm. They are useful for testing and debugging an algorithm and to help a user understand its flow.

Example algorithm (pseudocode)

```
total ← 0
FOR i ← 1 TO 3
    PROMPT "Enter a number"
    INPUT number
    total ← total + number
NEXT i
OUTPUT total
```

Example trace table

This trace table is used to track variables, any outputs, and any user prompts at each step of the above algorithm, with inputs 5, 7, and 2.

i	number	total	Prompt shown	Output
1		0	"Enter a number"	
1	5	5		
2		5	"Enter a number"	
2	7	12		
3		12	"Enter a number"	
3	2	14		
		14		14

Identifying and correcting errors

When developing algorithms, errors can occur. Being able to spot errors and correct them is an important skill. There are a few main types of error - you don't need to know the names specifically, but it's useful to be aware of the types, common examples, and how to fix them.

Type of Error	Example	How to Fix
Syntax error	Missing keyword: `IF x = 5 THEN` (no `ENDIF`) Wrong spelling: `PRNT "Hello"` instead of `PRINT "Hello"`	Carefully check pseudocode or code against the rules of the language. Add missing keywords or correct spelling.
Logic error	Using wrong operator: `IF mark > 50 THEN` instead of `IF mark >= 50 THEN` Wrong calculation: `total ← number` instead of `total ← total + number`	Use dry-runs, trace tables, and test data to spot unexpected behaviour. Correct conditions or calculations so they match the intended design.
Runtime error	Dividing by zero: `average ← total / count` when `count = 0` Invalid index: Accessing item 11 in a list of 10 items	Add validation to prevent invalid input or actions (e.g. check that `count ≠ 0`). Ensure loops and indexes stay within valid ranges.

Writing and amending algorithms for given problems or scenarios

An algorithm is a set of instructions to perform a specific task. In your exam, you'll be asked to write or amend algorithms for given problems or scenarios.

There are different ways to express an algorithm:

- **Pseudocode:** a text-based way of describing a solution in structured steps. It uses programming-like keywords but is language-independent. There is a specific set of pseudocode keywords available online for the Cambridge IGCSE Computer Science exam.
- **Program code:** The actual code written in a specific programming language that can be executed by a computer.
- **Flowcharts:** Diagrams that show the step-by-step flow of an algorithm. Make sure that you know the flowchart symbols provided earlier in these notes.