IGCSE 07 Algorithm design and problem solving (1)

Program development life cycle

Stage:

Analysis: Process of investigation, using abstraction and decomposition to specify what a program does.

Design: Uses structure charts, flowcharts and pseudocode with the program specification from analysis to show to how the program should be developed.

Coding: Writing and iterative testing of the program or suite of programs.

Testing: Testing the completed program to make sure that it works under all conditions.

Maintenance

Computer system

Each **computer system** is made up of software, data, hardware, communications and people.

Each computer system can be divided up into a set of **sub-systems** and each sub-system can be **further divided into subsystems** and so on until each sub-system just performs a single action.

The process of **decomposition** into sub-systems so that a system can be more easily represented and understood is the basis of **top-down design**.

how a problem can be decomposed into its component parts

Any problem that uses a computer system for its solution needs to be decomposed into its component parts. These are **inputs**, **processes**, **outputs** and **storage**.

methods used to design and construct a solution:

1. structure diagrams
2. flowcharts
3. pseudocode.

Structure diagrams

structure diagram shows hierarchically how each computer system can be divided up into a set of sub-systems

Flowcharts

flowchart shows diagrammatically the steps required to complete a task and the order that they are to be performed. These steps, together with the order, are called an algorithm.

Flowcharts

Use	Symbol	Description
Terminator Start/Stop	START STOP	Used at the beginning and end of each flowchart. At least two outputs.
Process	A ← 0 B ← 0	Used to show actions, for example, when values are assigned to variables.
Input/Output	INPUT X	The same flowchart symbol is used to show the input of data and output of information.
Decision	x > B?	Used to decide which action is to be taken next. These can be used for selection and repetition/iteration. There are always two outputs from a decision flowchart symbol.
Flow lines	→	Used to show the direction of flow.

Standard methods of solution:

Totalling:

```
Total ← 0
FOR Count ← 1 TO ClassSize
   INPUT Mark
   Total ← Total + Mark
NEXT Count
```

Counting:

```
PassCount ← 0
FOR Counter ← 1 TO ClassSize
   INPUT Mark
   IF Mark > 50
      THEN
         PassCount ← PassCount + 1
   ENDIF
NEXT Counter
```

Finding maximum, minimum, average

```
Total ← 0
MaxMark ← 0
MinMark ← 100
FOR Count ← 1 TO ClassSize
   INPUT Mark
   IF Mark > MaxMark
      THEN
         MaxMark ← Mark
   ENDIF
   IF Mark < MinMark
      THEN
         MinMark ← Mark
   ENDIF
   Total ← Total + Mark
NEXT Count
Average ← Total / ClassSize
```

Pseudocode

Mathematical operators

Mathematical operators

Use	Symbol
+	Add
-	Substract
*	Mlutiply
/	Divide
^	Raise to the power
()	Group

Comparison operators

Operator	Comparison
>	Add
<	Substract
=	Mlutiply
>=	Divide
<=	Raise to the power
<>	Group
AND	Both
OR	Either
NOT	not

Pseudocode statement

Pseudocode statment	Examples
Assignment A value is assigned to an item/variable using the ← operator.	Cost ← 10 SellingPrice ← Price + Tax

Conditional 1
A condition that can be true or false: IF … THEN … ENDIF or IF … THEN … ELSE … ENDIF
For an IF condition the THEN path is followed if the condition is true, and the ELSE path if it is false (an ELSE may not be required). The end of the statement is followed by ENDIF
	IF Age < 18
	  THEN
	  OUTPUT "Child"
	  ELSE
	  OUTPUT "Adult"
	ENDIF

Conditional 2
A choice between several different values: CASE OF … OTHERWISE … ENDCASE
 	
For a CASE statement, the value of the variable decides the path taken. Several variables are usually specified.
OTHERWISE path is taken for all other values. The statement is ended by ENDCASE
	CASE OF Grade
	  "A" : OUTPUT "Excellent"
	  "B" : OUTPUT "Good"
	  "C" : OUTPUT "Average"
	  OTHERWISE OUTPUT "Improve"
	ENDCASE

Iteration 1 FOR … TO … NEXT a variable is set up, with a start value and an end value, this variable is incremented in steps until the end value is reached and the iteration finishes.	FOR Counter ← 1 to 10 OUTPUT "*" NEXT Counter

Iteration 2
REPEAT … UNTIL is used when the number of repetitions/ iterations is not known, and the actions are repeated UNTIL a given condition becomes true. The actions in this loop are always completed at least once.
	Counter ← 0
	REPEAT
	  OUTPUT "*"
	  Counter ← Counter + 1
	UNTIL Counter >= 10

Input INPUT used for data entry.	INPUT Name INPUT StudentMark
Output OUTPUT or PRINT used to display information.	PRINT "Your name is", Name OUTPUT Name1, "Ali", Name3

Nesting 1 Nested IF makes use of two IF statements; the second IF statement is part of the first ELSE or THEN path	IF Age < 18
	  THEN
	  OUTPUT "Child"
	  ELSE
	  IF Age > 65
	  THEN
	  OUTPUT "Senior"
	  ELSE
	  OUTPUT "Adult"
	  ENDIF
	ENDIF

Nesting 2 Nested iteration makes use of two loops; the second loop is inside the first loop.	FOR Number ← 1 to 10
	  OUTPUT Number
	  FOR Counter ← 1 to Number
	  OUTPUT "*"
	  NEXT Counter
	NEXT Number

Linear Search

```
OUTPUT "Enter name to find "
INPUT Name
Found ← FALSE
Counter ← 1
REPEAT
   IF Name = Name[Counter]
      THEN
         Found ← TRUE
      ELSE
         Counter ← Counter + 1
   ENDIF
UNTIL Found OR Counter > ClassSize
IF Found
   THEN
      OUTPUT Name, " found"
   ELSE
      OUTPUT Name, " not found."
ENDIF
```

Bubble sort

```
First ← 1
Last ← ClassSize
REPEAT
   Swap ← FALSE
   FOR Index ← First TO Last - 1
      IF Name[Index] > Name[Index + 1]
         THEN
            Temp ← Name[Index]
            Name[Index] ← Name[Index + 1]
            Name[Index + 1] ← Temp
            Swap ← TRUE
      ENDIF
   NEXT Index
   Last ← Last -1
UNTIL (NOT Swap) OR Last = 1
```

IGCSE 07 Algorithm design and problem solving (2)

Validation and verification

Data entry

Verification	Validation

Double entry	Screen.visual check	Range check	Length check	Type check	Presence check	Format check
The data is entered twice and compared to ensure both entries are the same	A manual check to ensure the data on the screen is the same as the form	Checks that the value of a number is between an **upper value** and a **lower value**	Checks that the data entered is a **reasonable number** or an **exact number** of characters	Checks that the data entered is of a **given data type**	Checks to ensure that some data has been entered and the value has **not been left blank**	Checks that the characters entered conform to a **pre-defined pattern**

Test data

Test data

Normal	Abnormal/ erroneous	Noundary	Extreme
Test data that is **accepted** and the algorithm is expected to work with	Test data that is **rejected** by the algorithm as not suitable	At each boundary two values are required; **one value is accepted** and **the other value is rejected**	The **largest and smallest values** that **normal data** can take

Trace table

A trace table records the results from **each step in an algorithm**;
it shows the value of **each variable every time that it changes**.
Working through an algorithm step by step is called a **dry run**.
A trace table is set up with a column for each variable and a column for any output.

Writing and amending algorithms

1. Make sure that the **problem is clearly specified** – the purpose of the algorithm and the tasks to be completed by the algorithm.
2. Break the problem **down in to sub-problems**. If it is complex, you may want to consider writing an algorithm for each sub-problem. Most problems, even the simplest ones can be divided into:
   - set-up processes
   - permanent storage of data
   - input (if required)
   - processing of data
   - output of results.
3. Decide on **how any data is to be obtained and stored**, what is going to happen to the data and how any results are going to be displayed.
4. Design the structure of your algorithm using a **structure diagram**.
5. Decide on how you are going to construct your algorithm, either using a **flowchart** or **pseudocode**. If you are told how to construct your algorithm, then follow the guidance.
6. Construct your algorithm, making sure that it can be easily read and understood by someone else. Precision is required when writing algorithms, just as it is when writing program code. This involves setting it out clearly and using meaningful names for any data stores.
7. Use several sets of **test data** (normal, abnormal and boundary) to **dry run** your algorithm and show the results in trace tables, to enable you to find any errors.
8. If any **errors** are found, correct them and repeat the process until you think that your algorithm works perfectly.

AND Gate:

logic notation
$$X = A \ AND \ B $$
boolean algebra
$$X = A.B $$

Input	Input	Outpu
A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

OR Gate:

logic notation
$$X = A\ OR\ B $$
boolean algebra
$$X = A\ +\ B $$

Input	Input	Outpu
A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

NAND Gate:

logic notation
$$X = A\ NAND\ B $$
boolean algebra
$$X = \overline{A\ .\ B} $$

Input	Input	Outpu
A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

NOR Gate:

logic notation
$$X = A\ NOR\ B $$
boolean algebra
$$X = \overline{A\ +\ B} $$

Input	Input	Outpu
A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

NOT Gate:

logic notation:
$$X = NOT \ A $$
boolean algebra
$$X = NOT \ \overline{A} $$

Input	Outpu
A	X
0	1
1	0

XOR Gate:

logic notation
$$X = A\ XOR\ B $$
boolean algebra
$$X = (A\ .\ \overline{B}\ ) + (\overline{A}\ .\ B) $$
$$X = \overline{(A\ .\ B) + (\overline{A}\ .\ \overline{B})} $$

Input	Input	Outpu
A	B	X
0	0	0
0	1	1
1	0	1
1	1	0