# AQA GCSE Computer Science (8525)

## Paper 1: Computational Thinking And Programming Skills

## Revision Booklet

# Topic 1: Fundamentals of Algorithms

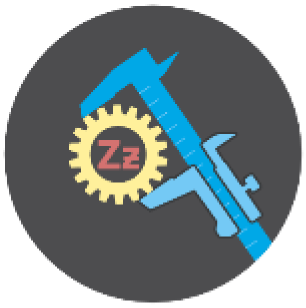
## Algorithms – *A set of steps which can be followed to complete a certain task*

A computer program may implement one or more algorithms.
A computer program is not an algorithm

## Decomposition - *Breaking down large problems into a set of smaller sub problems.*

Each sub-problem accomplishes a clear, identifiable task.
Sub-programs may be further broken down if needed.
Advantages:

- Smaller problems are easier to solve
- Each part can be solved independently
- Each part can be tested independently
- The parts are combined to produce the full problem.
- Allows each smaller problem to be examined in more detail



## Abstraction - *Using symbols and variables to represent a real-world problem using a computer program and removing unnecessary detail*

Example - a program is to be created to let users play chess against the computer.

- Board is created as an array(s).
- Pieces are objects that have positions on the board
- The shape and style of the pieces may not be required.

Advantages:

- Allows the creation of a general idea of how to solve the problem.
- Provides focus on what actually needs to be done.
- Provides a simple view of the problem

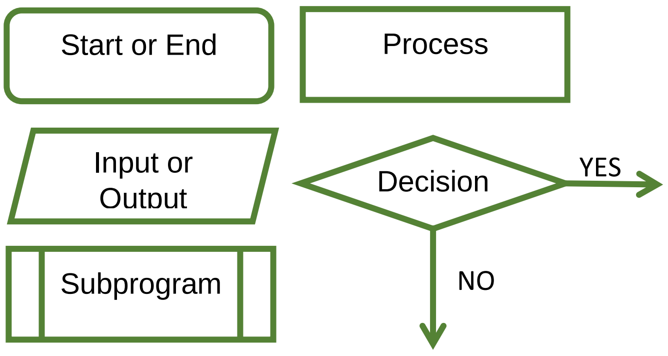
## Pseudocode

- Uses short English words and statements to describe an algorithm.
- Generally, looks a little more structured than normal English sentences.
- Flexible.
- Less precise than a programming language.

```
IF Age is equal to 14 THEN
        Stand up
ELSE Age is equal to 15 THEN
        Clap
ElSE Age is equal to 16 THEN
        Sing a song
ELSE
        Sit on the floor
ENDIF
```

## Flowcharts

- Created to represent an algorithm.
- Show the data that is input, and output.
- Show processes that take place.
- Show any decisions and repetitions that take place.
- Lines show flow through the chart.
- Shapes represent different functions



## Trace Tables – *a method of recording the values used within an algorithm at each stage of processing to help in troubleshooting*

- Tests algorithms for logic errors which occur when the algorithm is executed.
- Simulates the steps of algorithm.
- Each stage is executed individually allowing inputs, outputs, variables, and processes to be checked for the correct value at each stage.
- A great way to spot errors

```
X = 3
Y = 1
while X > 0
    Y = Y + 1
    X = X - 1
print(Y)
```

| Stage | X | Y | Output |
|---|---|---|---|
| 1 | 3 | 1 | |
| 2 | | 2 | |
| 3 | 2 | | |
| 4 | | 3 | |
| 5 | 1 | | |
| 6 | | 4 | |
| 7 | 0 | | |
| 8 | | | 4 |

## Determining the Purpose of Algorithms

- There are several ways to determine the purpose of an algorithm.
- We can dry run the algorithm, by assigning values to its inputs, and working through to see what happens.
- Trace Tables allow us to record these values as the algorithm is run.
- Visual Inspection involves simply looking at the algorithm to determine its purpose.
- Sometimes the algorithm may follow a standard pattern which we can recognise.
- With shorter or simpler algorithms, the purpose may be obvious by simply looking at it.

## Efficiency

There is often more than one algorithm which will solve a given problem
We can compare the efficiency to help us compare algorithms and work out which is the best
The less time an algorithm takes to complete its task, the more efficient it is
Steps which improve efficiency:

- Using repetition (loops) to reduce the amount of code?
- Using arrays instead of declaring many individual variables?
- Using selection statements which only make comparisons until a solution is reached?

## Searching Algorithms

**Linear Search**

1) Check the first value
2) If it is desired value
    a) Stop
3) Otherwise check the second value
4) Keep Going until all elements have been checked or the value is found

**Binary Search**

1) Put the list in order.
2) Take the middle value.
3) Compare it to the desired value.
    a) If it is the desired value.
        i) Stop.
    b) If it is larger than the desired value.
        i) Take the list to the left of the middle value.
    c) If it is smaller than the desired value.
        i) Take the list to the right of the middle value.
4) Repeat step 3 with the new list.

| | Linear Search | Binary Search |
|---|---|---|
| Pros | • Works with unsorted lists<br>• Not affected by changes to the list<br>• Works well for small lists | • More efficient<br>• Efficient for large lists |
| Cons | • Slower<br>• Inefficient for large lists | • Does not work with unsorted lists |

## Sorting Algorithms

**Bubble Sort**

1) Take the first element and second element
2) Compare the two
   a) If element 1 > element 2
      i) Swap them over
   b) Otherwise
      i) Do nothing
   c) Move to the next pair in the list
   d) If there are no more elements return to step (1)
   e) Otherwise, return to step (2)
3) Repeat until you have worked through the whole list without making any changes

**Merge Sort**

1) Split the list into individual elements.
2) Merge the elements together in pairs, putting the smallest element first.
3) Merge two pairs together, putting the smallest first.
4) Keep merging until all pairs are in order.

| | Bubble Sort | Merge Sort |
|---|---|---|
| Pros | • Simplest and easiest to code<br>• Uses less memory | • Far more efficient and faster<br>• Consistent running time |
| Cons | • Slower with larger lists<br>• Inefficient and slow | • Uses more memory<br>• More complexed to program |

# Topic 2: Programming

## Data Types

Data can come in many forms, and different types of data need to be treated differently.
Defining a data type, tells the program what kind of data will be held within a variable.
Data types are treated differently, for example, we cannot use maths (such as multiplication) in strings.

| Data type | Purpose | Example |
|---|---|---|
| Integer | Whole numbers | 27 |
| Real | Decimal numbers | 27.5 |
| Character | A single alphanumeric character | A |
| String | One or more alphanumeric characters | ABC |
| Boolean | TRUE/FALSE | TRUE |

## Variables - *a box in which data may be stored. The value can be changed as needed whilst the program is running*

- Different types e.g. string, decimal, etc.
- Allows the program to store data such as an input for later use

## Constants – *a fixed value used by the program such as pi. The value cannot be changed whilst the program is running*

- Allows easy use of fixed values without having to store them in the program
- Allows a fixed value to be easily referenced over and over again

## Declaring – *a process to 'create' a variable or constant before it can be used*

- Provides both the name and type of the variable
- Most programming languages require variables and constants to be defined before use

## Assignment – *assigning a value to a variable*

- Once a variable has been declared, we can assign a value to it.
- We can assign different values to the same variable as the variable is run.
- Assigning a value to a variable replaces the value which was previously in it.

## Sub Programs - *Small programs which form part of a larger program.*

**Procedures** are sets of instructions stored under a single name (identifier).
**Functions** are like procedures but will always return a value to the main program.
Advantages:

- Used to save time and simplify code
- Allows the same code to be used several times without having to write it out each time
- Usually small in size, so easier to write and test.
- Easy for someone else to understand.
- Can be saved separately as modules and used again in other programs.
- Saves time because code that has already been written and tested can be reused

## Iteration – *Repeating a set of steps several times.*

**Definite (Count Controlled):**

- Repeats the same code a set number of times
- Uses a variable to track how many times the code has been run
- This variable can be used in the loop
- At the end of each iteration the variable is checked to see if the code should be run again
- FOR sets how many times the code should be repeated
- STEP sets how the variable should increment
- ENDFOR shows the end of the loop

```
FOR a ← 1 TO 5 STEP 2
 OUTPUT a
ENDFOR
# will output 1, 3, 5
```

**Indefinite (Condition Controlled):**

- Uses a condition to determine how many times code should be repeated.
- The condition can be placed at the start of the loop or at the end.
- The condition is checked each run through the loop is run to see if the code should be run again.

```
a ← 1
WHILE a < 4
 OUTPUT a
 a ← a + 1
ENDWHILE
# will output 1, 2, 3
```

```
a ← 1
REPEAT
 OUTPUT a
 a ← a + 1
UNTIL a = 4
# will output 1, 2, 3
```

## Selection

- Allows the program to make decisions
- Uses conditions to change the flow of the program
- Selections may be nested one inside another
- **IF** statements perform comparisons sequentially and so the order is important
- The **ELSE** statement allows us to use multiple conditions.

```
a ← 1
IF (a MOD 2) = 0 THEN
 OUTPUT 'even'
ENDIF
```

```
a ← 1
IF (a MOD 2) = 0 THEN
 OUTPUT 'even'
ELSE
 OUTPUT 'odd'
ENDIF
```

## Nested Selection and Iteration – *using more than one iteration or selection, one within the other.*

- This provides more flexibility and choices.
- We can nest statements many layers deep.
- This allows for more complex decisions to be made.

```
IF GameWon THEN
  … Instructions here …
  IF Score > HighScore THEN
     … Instructions here …
  ENDIF
  … Instructions here …
ENDIF
```

```
WHILE NotSolved
   … Instructions here ...
   FOR i ← 1 TO 5
       … Instructions here …
   ENDFOR
   … Instructions here …
ENDWHILE
```

## Choosing Names

- Names of variables, constants and sub programs should be meaningful.
- They should give an indication as to the purpose and function.
- This makes it much easier to understand the code

## Operators – *Allow us to work with data, to change it and compare it*

**Arithmetic Operators**

| | |
|---|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division |
| MOD | Modulus (the remainder from a division, e.g. 12 MOD 5 gives 2) |
| DIV | Quotient (integer division, e.g. 21 DIV 5 gives 4) |
| ^ | Exponentiation (to the power of, e.g. 3^3 gives 27) |

**Comparison Operators**

| | |
|---|---|
| == | Equal to |
| != | Not equal to |
| < | Less than |
| <= | Less than or equal to |
| > | Greater than |
| >= | Greater than or equal to |

**Boolean Operators**

AND - two conditions must be met for the statement to be true
OR - at least one condition must be met for the statement to be true
NOT – inverts the result, e.g. NOT(A AND B) will only be false when both A and B are true

## Data Structures – *defines how data flows via inputs, processes and outputs*

## Arrays - *An ordered collection of related data*

- Each element in the array has a unique index, usually starting at 0
- All elements must be the same type of data
- Arrays are usually a fixed size
- **1 Dimensional arrays** are like a simple list, each element needs a single index number
- **2 Dimensional arrays** are like tables, with each element needing two index numbers
- 2 Dimensional arrays are usually used to store properties of objects, with objects in rows and properties in columns
- Fruits[1] references element 1 in the 1D Fruits array
- Tools[0,2] references element 0,2 in the Tools array

```
RECORD Car
     make : String
     model : String
     reg : String
     price : Real
     noOfDoors : Integer
ENDRECORD
```

## Records – *a method for storing related data*

Allows different types of related data to be stored together.
Several records can be held together in an array.
Each field in a record can contain different types of data.

## Input – *getting data into the program*

To work effectively, most programs require users to input data.
This is often done using a keyboard.
Data is usually stored in a variable after being input.
Inputted data can then be processed by the program

```
a ← USERINPUT
```

```
OUTPUT a
OUTPUT a, g
```

## Output – *allows the program to show the user the results of processing*

Keeps the user informed about what the program is doing.
Data is usually outputted to the computer's screen.
The program can output several things at the same time.

## String Handling and Manipulation

Strings hold alphanumeric characters, what we would consider normal text.
Much of the data a program processes will be in string format.
The program can change (manipulate) strings in many different ways.

| Technique | Explanation | Example |
|---|---|---|
| Length | Gives the length of the string as an integer | `LEN('computer science')` `# evaluates to 16(including space)` |
| Character Position | Tells us where a character is located within a string. | `POSITION('computer', 'm')` `# evaluates to 2` |
| Substring | Returns a part of a string. We must specify how many charters to return and where to start. | `SUBSTRING(2, 9, 'computer science')` `# evaluates to 'mputer s'` |
| Concatenation | Joins one or more strings together. | `'computer' + 'science'` `# evaluates to 'computerscience'` |
| Converting to Character Code | Converts a given character to the character code. | `CHAR_TO_CODE('a')` `# evaluates to 97 using ASCII/Unicode` |
| Convert to a Character | Converts a given character code to an actual character. | `CODE_TO_CHAR(97)` `# evaluates to 'a' using ASCII/Unicode` |
| String to Integer | Converts a number stored in a string to an integer number which can be processed using mathematical operators | `STRING_TO_INT('16')` `# evaluates to the integer 16` |
| String to Real | Converts a number stored in a string to a real number number which can be processed using mathematical operators | `STRING_TO_REAL('16.3')` `# evaluates to the real 16.3` |
| Integer to String | Converts an integer to a string, allowing it to be processed by string operators such as substring | `INT_TO_STRING(16)` `# evaluates to the string '16'` |
| Real to String | Converts a real number to a string, allowing it to be processed by string operators such as substring | `REAL_TO_STRING(16.3)` `# evaluates to the string '16.3'` |

## Random Numbers

```
diceRoll ← RANDOM_INT(1, 6)
# will randomly generate an
# integer between 1 and 6
# inclusive
```

A program may need to generate a random letter or number within a given range.
This allows the program to behave in an unpredictable way.
This is very useful for games.
Random numbers are also used to generate secure passwords.

## Subroutine *– a reusable piece of code used to perform a common task*

```
SUBROUTINE printMessage()
 OUTPUT 'This is a message'
ENDSUBROUTINE
printMessage()
```

Computer programs often need to perform the same task many times.
Writing out the same code over and over again is inefficient, prone to errors and hard to read and maintain.
Subroutines allow the code to be written only once but easily executed many times.

**Parameters**

Allow us to provide (pass) data to the subroutine for processing.
Subroutines can accept more than one parameter.
They are passed using the syntax `SUBROUTINE subroutineName(prarmeter1, pameter2)`
The subroutine must be passed the same number of parameters as it was created to expect.
Providing too many or not enough parameters will cause errors.

**Returning Values**

The subroutine will often need to return an output back to the main program.
This allows the result of processing carried out in the subroutine to be used in the main program.
This is done using the `Output 'return value'` code in a subroutine.
This is important because any variables used within the subroutine are not available to the main program.

**Local Variables**

Variables created within a subroutine are called Local Variables.
They only exist whilst the subroutine is running.
They are only accessible to that subroutine and not to other subroutines or the main program.
This makes the code easier to debug, as the variable will only exist within the subroutine.
This improves efficiency since the variable does not exist when the subroutine is not running, freeing up memory.

**Global Variables**

Global variables can be used anywhere within the code.
Because they can be altered anywhere in the program they are harder to troubleshoot.

## Structured Programming Approach

Makes programs easier and quicker to write, test, debug, and change.
Makes code easier to understand and think about.
Combines different skills and techniques.
The use of decomposition and subroutines form a key part of it.

## Data Validation *– rules to make sure data is in the correct format*

Users may enter incorrect data.
Computer programs should be able to check for this and take appropriate action.
Validation applies rules to data, only data which meets the rules is allowed.
This reduces the risk that an incorrect input will crash the program.
Validation does not ensure that the data is correct, but that it is in the correct format.
We can use comparison and string manipulation operators combined with selection and subroutines to implement validation.

**Range Check** – makes sure that the data is within a certain range, this is usually used for numbers and dates. For example, a child’s date of birth must be between the current date, and the current date minus 18 years.

**Length Check** – makes sure the data is the correct length, and not too long or short. For example, the first line of an address must be at least one character long, but not more than 50 characters.

**Presence Check -** makes sure a value has actually been entered. For example, an email address must be entered to sign up for a newsletter.

**Format Check** – makes sure the data is in the correct format. For example, a date must be in the format DD/MM/YYYY

**Type Check** – Makes sure the data is of the correct data type. For example, a quantity must be an integer.

Below is an example of a validation subroutine to have the user input a number between a lower and upper value provided as parameters.

```
SUBROUTINE get_input(lower, upper)
    OUTPUT 'Enter a number between ' + lower + ' and ' + upper
    number_as_string ← USERINPUT
    # continue to loop until number is returned
    WHILE True
        TRY
            number ← STRING_TO_INT(number_as_string)
            IF number < lower OR number > upper THEN
                OUTPUT 'Number not within bounds'
            ELSE
                RETURN number
            ENDIF
        CATCH
            OUTPUT 'Not an integer'
        ENDTRY
    ENDWHILE
ENDSUBROUTINE
```

## Authentication *– confirming the user is who they say they are*

This is commonly accomplished by asking the user to enter a username and password.
The username and password are stored persistently, such as in a text file.
Techniques such as iteration, selection and comparison operators allow us to prompt the user for these inputs then compare them to the stored data.

## Testing

Newly written programs often contains bugs which stop them working properly.
Testing allows the programmer to locate and remove these bugs, making sure the program meets its' needs.

**Test Data**
Different types of data will need to be entered into the program to test it is working correctly.
This data should cover a range of different data which the program might have to deal.
It should cover possible data, which is normal and allowed, and impossible, which is not allowed.
**Normal Data** – is typical data that the program is likely to encounter and should be able to process without error.
**Boundary Data** – is data at the limit of what the program will and will accept.
**Erroneous Data** – is data which is not valid.

**Example Test Data for Subroutine Which Validates Input is Between 1 and 100**
**Normal Data:** 10, 50, 75
**Boundary Data:** 0, 1, 100, 101
**Erroneous Data:** 200, -500, John, Four

## Types of Error

**A program with a syntax error will not run. A program with a logic error will run but it will not perform as expected.**

**Syntax Errors**
When the code does not follow the syntax rules of the programming language used. This stops the program from running.
Examples:
- Misspellings or typos
- Using a variable before it has been declared
- Missing or incorrect use of brackets

**Logic Errors**
The program runs but does not do what it should.
Examples:
- Incorrectly using logical or Boolean operators
- Creating infinite loops
- Incorrectly using brackets in calculations

Using the same variable name at different points for different purposes