YEAR 11 GCSE COMPUTER SCIENCE

ANSFORD ACADEMY
LEARNING TOGETHER TO LEAD OUR LIVES

COMPONENT 2 PERSONAL LEARNING CHECKLIST

2.1 Algorithms

2.1.1 Computational Thinking

Principles of Computational Thinking:

- ☐ I understand the core principles of **computational thinking**, which are fundamental to problem-solving in computer science:
  - ☐ **Abstraction**: I know that **abstraction** involves focusing on the important aspects of a problem and ignoring unnecessary details to make it easier to solve.
  - ☐ **Decomposition**: I understand that **decomposition** breaks down complex problems into smaller, more manageable subproblems, making it easier to address each part.
  - ☐ **Algorithmic thinking**: I can explain how **algorithmic thinking** involves designing step-by-step solutions to problems, ensuring that a clear and efficient process is followed to solve them.

Using Computational Thinking to Define and Refine Problems:

- ☐ I can apply **computational thinking** principles to break down complex problems, define them more clearly, and create effective solutions:
  - I use **abstraction** to simplify problems by removing unnecessary details.
  - I apply **decomposition** to break a large problem into smaller, easier-to-manage components.
  - I use **algorithmic thinking** to create clear, step-by-step procedures that address each aspect of the problem.

2.1.2 Designing, Creating and Refining Algorithms

Identify the Inputs, Processes, and Outputs for a Problem:

- ☐ I understand how to identify the **inputs**, **processes**, and **outputs** for a given problem:
  - **Inputs** are the data or values provided to the algorithm.
  - **Processes** are the steps or actions that the algorithm takes to transform the inputs.

- **Outputs** are the results or solutions that the algorithm generates.

Structure Diagrams:

- ☐ I know how to create **structure diagrams** to visually represent the organisation and flow of a problem, showing:
  - The main structure of the problem.
  - How the problem is divided into smaller **subsections**.
  - The links and relationships between different subsections of the problem.

Create, Interpret, Correct, Complete, and Refine Algorithms Using:

- ☐ I can **create** and **interpret** algorithms in different forms, including:
  - ☐ **Pseudocode**: I understand how to write algorithms in pseudocode, which uses structured, human-readable instructions that can be easily translated into code.
  - ☐ **Flowcharts**: I can create **flowcharts** to visually represent the flow of the algorithm, using standard symbols like lines, processes, and decisions.
  - ☐ **Reference Language/High-Level Programming Language**: I can translate algorithms into a programming language, understanding the basic syntax and logic of high-level languages.

Identify Common Errors:

- ☐ I am able to **identify syntax** (e.g. spelling mistakes or incorrect punctuation) and **logic errors** (e.g. incorrect calculations or wrong conditions) in code and suggest fixes.

Trace Tables:

- ☐ I can **create and use trace tables** to follow an algorithm and track how values change as the algorithm executes, helping to identify any errors or inefficiencies.

Use of Nesting for Selection and Iteration:

- ☐ I understand the use of **nesting** (placing one control structure inside another) for both **selection** (if/else statements) and **iteration** (loops), and I can apply this technique to refine algorithms for more complex tasks.

2.1.3 Searching and Sorting Algorithms

Standard Searching Algorithms:

- ☐ I understand how to implement and apply the following common **searching algorithms**:

- ☐ **Binary search**: I know that **binary search** is an efficient algorithm used to find a value in a **sorted** list by repeatedly dividing the search interval in half.
- ☐ **Linear search**: I understand that **linear search** involves checking each element of the list one by one until the desired value is found.

Standard Sorting Algorithms:

- ☐ I understand how to implement and apply the following common **sorting algorithms**:
  - ☐ **Bubble sort**: I can explain how **bubble sort** repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
  - ☐ **Merge sort**: I understand that **merge sort** divides the list into smaller sublists, sorts them, and then merges them back together in the correct order.
  - ☐ **Insertion sort**: I know that **insertion sort** builds the final sorted array one item at a time by inserting elements into their correct position.

Main Steps and Segments of Code:

- ☐ I understand the **main steps** of each searching and sorting algorithm and how the algorithm processes data to achieve the desired result.
- ☐ I can identify the key segments of code in these algorithms, recognising how the logic flows from one step to the next.

Pre-Requisites of an Algorithm:

- ☐ I know the **prerequisites** for each algorithm, such as whether the list needs to be sorted before applying certain searching or sorting algorithms, and the conditions required for the algorithm to work efficiently.

Apply the Algorithm to a Data Set:

- ☐ I can **apply** a given algorithm to a dataset, ensuring that it is correctly implemented and produces the expected result for that particular set of data.

Identify an Algorithm from Given Code, Pseudocode, or Flowchart:

- ☐ I can identify a given algorithm from its **code**, **pseudocode**, or **flowchart**, and understand how it works to solve the problem at hand.

Flowchart Symbols:

- ☐ I understand the standard **flowchart symbols** used to represent different components of an algorithm:
  - **Line**: Represents the flow of control or data.

- **Input/Output**: Represents input or output operations (e.g., reading data, displaying results).
- **Process**: Represents an operation or action (e.g., a calculation or assignment).
- **Decision**: Represents a decision point, where the algorithm chooses between two options (e.g., if/else).
- **Subprogram**: Represents a call to another procedure or function.
- **Terminal**: Represents the start or end of the algorithm..

2.2 Programming Fundamentals

2.2.1 Programming Fundamentals

The Use of Variables, Constants, Operators, Inputs, Outputs, and Assignments:

- ☐ I understand that **variables** are used to store data that can change during the execution of a program, while **constants** store data that does not change.
- ☐ I can explain how **operators** are used in programming to perform calculations or manipulate data, including:
  - **Arithmetic operators** (e.g., addition, subtraction).
  - **Comparison operators** (e.g., equal to, greater than).
  - **Boolean operators** (e.g., AND, OR, NOT).
- ☐ I know that **inputs** are used to gather data from users or other sources, and **outputs** are used to display results.
- ☐ I understand the concept of **assignments**, where a value is stored in a variable or constant.

The Use of the Three Basic Programming Constructs:

- ☐ I can explain the three basic programming constructs used to control the flow of a program:
  - ☐ **Sequence**: I understand that sequence refers to the execution of instructions in the order they are written.
  - ☐ **Selection**: I can describe how selection allows decisions to be made in a program (e.g., using **if/else** statements).
  - ☐ **Iteration**: I understand how iteration (loops) allows for repeated execution of code. This can be:
    - ☐ **Count-controlled loops**: Where the number of iterations is specified in advance (e.g., for loops).

- ☐ **Condition-controlled loops**: Where the loop continues until a condition is met (e.g., while loops).

The Common Arithmetic Operators:

- ☐ I recognise and can use the following **arithmetic operators**:
  - + for **addition**
  - - for **subtraction**
  - * for **multiplication**
  - / for **division**
  - MOD for **modulo** (remainder after division)
  - DIV for **quotient** (integer result of division)
  - ^ for **exponentiation** (raising to a power)

The Common Boolean Operators:

- ☐ I understand the following **Boolean operators**:
  - AND: Returns true if both conditions are true.
  - OR: Returns true if at least one condition is true.
  - NOT: Reverses the Boolean value (turns true to false, and vice versa).

2.2.2 Data Types

The Use of Data Types:

- ☐ I understand that **data types** specify the type of value a variable can hold, and I can choose the appropriate data type for a given scenario:
  - **Integer**: Used for whole numbers without decimal points.
  - **Real**: Used for numbers with decimal points (also known as floating-point numbers).
  - **Boolean**: Used for logical values, true or false.
  - **Character**: Used to store a single character.
  - **String**: Used to store a sequence of characters (text).

Practical Use of Data Types in a High-Level Language:

- ☐ I can use these data types in a high-level language to store and manipulate data effectively, ensuring that the correct data type is used for each variable to avoid errors.

Ability to Choose Suitable Data Types for Data in a Given Scenario:

- ☐ I can choose the most appropriate data type based on the needs of a program, for example:
    - **Integers** for counting or performing arithmetic on whole numbers.
    - **Real numbers** for calculations requiring decimal precision.
    - **Booleans** for conditions or flags that require a true/false answer.
    - **Strings** for handling text.

Understanding of Casting:

- ☐ I understand that **casting** is the process of converting one data type into another (e.g., converting a real number into an integer), and I can apply this when necessary to ensure compatibility in calculations or data storage.

2.2.3 Additional Programming Techniques

The Use of Basic String Manipulation:

- ☐ I can manipulate **strings** in programming, including:
    - **Concatenation**: Combining two or more strings together to form a single string.
    - **Slicing**: Extracting a portion of a string by specifying the starting and ending positions.

The Use of Basic File Handling Operations:

- ☐ I understand how to perform basic **file handling operations** in a program:
    - **Open**: To open a file for reading or writing.
    - **Read**: To read data from a file.
    - **Write**: To write data to a file.
    - **Close**: To close the file after operations are completed.

The Use of Records to Store Data:

- ☐ I understand that **records** are used to store related data together in a structured way, and can contain fields of different data types, such as storing a person's name, age, and address in a single record.

The Use of SQL to Search for Data:

- ☐ I can use **SQL** (Structured Query Language) to search for and retrieve specific data from a database:
    - **SELECT**: Used to specify which fields of data to retrieve.
    - **FROM**: Specifies the table from which to retrieve the data.
    - **WHERE**: Used to filter data based on specific conditions.

The Use of Arrays:

- ☐ I can work with **arrays** (or equivalent data structures) to store multiple values of the same type:
  - ☐ **One-dimensional arrays (1D)**: Used to store a list of items in a linear structure.
  - ☐ **Two-dimensional arrays (2D)**: Used to store data in a table-like format, with rows and columns (useful for emulating database tables or matrices).

The Use of Subprograms (Functions and Procedures):

- ☐ I understand the use of **subprograms** (functions and procedures) to create structured, modular code:
  - ☐ **Functions**: I know that functions are used to perform specific tasks and return a value.
  - ☐ **Procedures**: I know that procedures are used to perform tasks without returning a value.

Where to Use Functions and Procedures Effectively:

- ☐ I know when to use **functions** to return values after performing a calculation or task, and when to use **procedures to** perform tasks that do not require a return value.

The Use of Variables and Constants in Functions and Procedures:

- ☐ I understand that within functions and procedures, variables and constants can be **local** (only accessible within the function or procedure) or **global** (accessible throughout the program).
- ☐ I can pass **arrays** into functions or procedures and return arrays, allowing data to be manipulated or processed.

Random Number Generation:

- ☐ I can use **random number generation** to create random values within a specified range, which can be useful for tasks like simulations, games, or generating random test data.

2.3 Producing Robust Programs

2.3.1 Defensive Design

Defensive Design Considerations:

- ☐ I understand the concept of **defensive design**, which involves designing programs to anticipate and handle potential issues and misuse that could cause errors or system failures.
- ☐ I know that **anticipating misuse** involves considering how users might use (or misuse) a program, and designing it in such a way that the program can handle unexpected inputs or behaviours safely.
- ☐ I understand that **authentication** is a method of verifying the identity of a user, ensuring that only authorised users can access certain parts of a system, such as using **usernames** and **passwords**.

Input Validation:

- ☐ I know that **input validation** is the process of ensuring that the data entered into a program is both **correct** and **safe** before it is processed.
- ☐ I understand how to **design input validation** systems that check user input against predefined rules or conditions to ensure the data is valid and does not cause errors in the system.

Maintainability:

- ☐ I understand that **maintainability** refers to how easily a program can be modified or updated to fix bugs, add new features, or improve performance.
- ☐ I know that using **subprograms** (such as functions or procedures) helps make code more organised, reusable, and easier to maintain.
- ☐ I understand that **naming conventions** ensure that variables, functions, and other elements of the program have clear and consistent names, making the code easier to read and maintain.
- ☐ I understand the importance of **indentation** to visually structure the code, making it easier to understand and modify. Proper indentation helps to highlight the logical flow of the program.
- ☐ I understand that **commenting on the** code is essential for explaining what the code does, why certain decisions were made, and how parts of the program work. I know how to apply comments effectively to improve the readability and maintainability of the code.

2.3.2 Testing

The Purpose of Testing:

- ☐ I understand that the **purpose of testing** is to ensure that a program works correctly, meets the requirements, and performs as expected in all situations.

- [ ] I know that testing is an ongoing process that helps to identify and fix errors or bugs in the program before it is deployed.

Types of Testing:

- [ ] I can explain the difference between **iterative testing** and **final/terminal testing**:
  - [ ] **Iterative testing** involves testing modules or parts of the program during development to identify and fix errors early in the process.
  - [ ] **Final/terminal testing** is conducted after the program is fully developed to ensure the entire program works as intended and meets the specified requirements.

Identify Syntax and Logic Errors:

- [ ] I know that **syntax errors** occur when the code violates the grammatical rules of the programming language, making the program unable to run or compile. These errors must be fixed before the program can be executed.
- [ ] I understand that **logic errors** occur when the program runs but produces incorrect or unexpected results. These errors do not prevent the program from running, but they cause it to behave incorrectly.

Selecting and Using Suitable Test Data:

- [ ] I understand the different types of **test data** and their importance in ensuring the program works correctly:
  - [ ] **Normal test data** refers to valid data that the program should accept and process correctly without causing errors.
  - [ ] **Boundary test data** refers to data that is at the edge of being valid, such as the minimum or maximum values allowed. This ensures the program handles these extreme cases correctly.
  - [ ] **Invalid test data** refers to data that is of the correct data type but should be rejected by the system because it does not meet the expected conditions or requirements.
  - [ ] **Erroneous test data** refers to data of the wrong type (e.g., text entered where a number is expected), which the program should reject to avoid errors.

Refining Algorithms:

- [ ] I understand the importance of **refining algorithms** based on test results, adjusting the logic or structure of the code to handle edge cases, improve efficiency, or ensure the program meets the requirements.

Ability to Identify Suitable Test Data for a Given Scenario:

- ☐ I can identify the most appropriate test data for a given scenario, ensuring that it covers all possible cases, including normal, boundary, invalid, and erroneous data.

Ability to Create and Complete a Test Plan:

- ☐ I understand the process of **creating a test plan**, which includes:
  - ☐ Defining the **test cases** (e.g., what inputs will be tested).
  - ☐ Specifying the **expected outcomes** for each test.
  - ☐ Recording the **actual outcomes** of each test.
  - ☐ Determining if any issues or errors were found and how they should be fixed.

2.4 Boolean Logic

2.4.1 Boolean Logic

Simple Logic Diagrams Using the Operators AND, OR, and NOT:

- ☐ I understand the purpose of **logic gates** in Boolean logic, which are used to perform logical operations on one or more binary inputs to produce a binary output.
- ☐ I can create and interpret **logic diagrams** that use the basic **Boolean operators**:
  - **AND** (Conjunction): The output is true if and only if both inputs are true.
  - **OR** (Disjunction): The output is true if at least one input is true.
  - **NOT** (Negation): The output is the inverse of the input (true becomes false and vice versa).

Truth Tables:

- ☐ I understand that a **truth table** is used to represent the output of a logical operation for all possible input combinations.
- ☐ I can construct and interpret **truth tables** for **AND**, **OR**, and **NOT** operations by systematically listing all possible input combinations and determining the corresponding output.

Combining Boolean Operators Using AND, OR, and NOT:

- ☐ I can combine multiple **Boolean operators** in a logic diagram or truth table to solve more complex problems.
- ☐ I understand how to use **AND**, **OR**, and **NOT** in combination to create compound logical expressions and can evaluate these expressions to determine the final output.

Applying Logical Operators in Truth Tables to Solve Problems:

- ☐ I can apply **logical operators** in **truth tables** to solve problems by evaluating the output for each combination of inputs.
- ☐ I understand how to create, complete, or edit a truth table based on a given logical expression or real-world scenario.

Knowledge of the Truth Tables for Each Logic Gate:

- ☐ I understand the standard **truth tables** for each of the basic logic gates (AND, OR, NOT).

Recognition of Each Gate Symbol:

- ☐ I can recognise the standard symbols for **AND**, **OR**, and **NOT** gates in logic diagrams:
  - ☐ **AND**: A flat-ended shape with two inputs and one output.
  - ☐ **OR**: A curved-ended shape with two inputs and one output.
  - ☐ **NOT**: A triangle shape with a circle at the output, representing inversion.

Understanding of How to Create, Complete, or Edit Logic Diagrams and Truth Tables for Given Scenarios:

- ☐ I can create logic diagrams based on a given Boolean expression or scenario, using the correct symbols for each gate.
- ☐ I can complete or modify **truth tables** for different logic gate combinations, ensuring that all possible input combinations are covered and the correct output is provided.

Ability to Work with More Than One Gate in a Logic Diagram:

- ☐ I can combine multiple gates in a **logic diagram**, using **AND**, **OR**, and **NOT** gates in various combinations to create more complex logical expressions.
- ☐ I can evaluate the output of combined gates using **truth tables** to determine the correct result for any given input.

2.5 Programming Languages & The IDE

2.5.1 Languages

Characteristics and Purpose of Different Levels of Programming Language:

- ☐ I understand the differences between **high-level** and **low-level** programming languages, including their characteristics and purposes:

- ☐ **High-level languages** are designed to be easy for humans to read and write, using simple syntax and abstracting away hardware details. These languages are platform-independent, meaning they can run on different types of computers without modification.
- ☐ **Low-level languages** are closer to machine code and provide more control over hardware. They are specific to a particular type of computer architecture and are often used for system-level programming, such as operating systems or embedded systems.

The Purpose of Translators:

- ☐ I understand the role of **translators** in converting code written in a high-level programming language into machine-readable instructions that the computer can execute.
    - **Translators** are essential because computers can only understand machine code, while humans typically use high-level languages to write programs.

The Characteristics of a Compiler and an Interpreter:

- ☐ I can explain the differences, benefits, and drawbacks of using a **compiler** or an **interpreter** to translate high-level code:
    - ☐ **Compiler**: A compiler translates the entire program into machine code at once, creating a standalone executable file. This makes compiled programs fast to run, but the compilation process can take time. Additionally, any errors must be fixed and recompiled before the program can run.
    - ☐ **Interpreter**: An interpreter translates code line-by-line and executes it immediately, without creating a separate executable file. This allows for easier debugging, as errors are identified immediately, but it may result in slower execution because the code is being translated as it runs.

2.5.2 The Integrated Development Environment (IDE)

Common Tools and Facilities Available in an Integrated Development Environment (IDE):

- ☐ I understand the common tools and facilities provided by an **IDE** that help programmers develop, debug, and execute programs efficiently:
    - ☐ **Editors**: I know that **editors** in an IDE are used to write and edit the source code, often with features like syntax highlighting, code completion, and indentation.
    - ☐ **Error diagnostics**: I understand that **error diagnostics** in an IDE help programmers identify syntax and logical errors in their code. These tools can display error messages and suggest possible fixes.

- [ ] **Run-time environment**: I know that an IDE includes a **run-time environment** that allows programmers to execute and test their code within the same environment, making it easier to debug and iterate on the program.
- [ ] **Translators**: I understand that **translators** (such as compilers and interpreters) are often integrated into the IDE, enabling seamless translation of high-level code into machine code.

How Each of the Tools and Facilities Listed Can Be Used to Help a Programmer Develop a Program:

- [ ] I know how each of the tools in an IDE helps a programmer:
  - [ ] The **editor** allows for quick and efficient code writing, with tools to assist with formatting and reducing errors.
  - [ ] **Error diagnostics** quickly point out issues in the code, helping to identify syntax and logical mistakes early in the development process.
  - [ ] The **run-time environment** allows for testing the program within the IDE itself, providing immediate feedback on its performance and functionality.
  - [ ] **Translators** within the IDE make it easier to compile or interpret code directly from the development environment, streamlining the development process.

Practical Experience of Using a Range of These Tools Within at Least One IDE:

- [ ] I have practical experience using a range of tools in at least one **IDE**, where I can use the editor to write code, apply error diagnostics to identify issues, run the program in the integrated run-time environment, and use the translator to compile or interpret the code.