A Level OCR Computer Science

Unit	YEAR 13

PAPER 2	Learning Objective
Computational Thinking	
Thinking Abstractly	The term 'abstraction', and its purpose in the design and creation of computer programs.
	The benefits of abstraction and how these apply to a specific scenario.
	Candidates may be given a scenario and be asked how abstraction can be applied to it, how it has been applied, or how further abstraction can be applied.
	How reality differs to abstraction, and an understanding of differences between reality and abstraction.
	Candidates may be given a scenario/example and be asked how the abstraction differs from the reality.
Thinking Ahead	That situations require inputs and output, and that outputs can be both digital or in a hard copy format.
	Candidates may be given a description, diagram, or code for a scenario, and they will need to demonstrate an understanding of what inputs and outputs are needed, and/or are used in that specific scenario.
	For a description of a program, candidates need to be able to determine what else they need to know before they can produce a solution; for example, what information is missing and what else will affect that solution.
	The purpose, benefits and drawbacks of reusable program components.
	How these components can be reused, and for a given scenario/program, candidates will need to be able to identify the subprograms that will be needed.
	Candidates may then be required to write code for these reusable components.
	The purpose of caching in programming, and how it can be used when writing a program.

	To apply knowledge of caching to a scenario to demonstrate an understanding of how it can be used.
	The benefits and drawbacks of using caching in a program.
Thinking Procedurally	To deconstruct a program and identify the component parts that will make it up; for example, listing the parts or completing a structure diagram.
	Candidates may be given some component parts and be asked to complete these from a written description or pseudocode for a program.
	To identify the steps that will need to take place to complete the algorithm or program, and be able to write these in a suitable format, or put a given list into the correct order to produce a working program.
	To write pseudocode or draw a flow chart to show this sequencing of steps.
	For a given scenario, candidates need to be able to identify where sub-procedures may be used, and then write appropriate pseudocode for these sub-procedures, making use of parameters where appropriate.
	Candidates may be given a structure diagram that they will need to interpret or complete to identify these sub-procedures.
Thinking Logically	That decisions are made within programs. Candidates need to be able to identify where these decisions will take place within an algorithm or program, and be able to understand what these decisions are, and the impact of these decisions on the algorithm/program and the next (and final) outcomes from the algorithm/program.
	That there can be many different routes through a program, and how decisions influence these routes and outcomes.
Thinking Concurrently	What is meant by thinking concurrently. Candidates need to be able to work out which parts of a program can be developed to take place (be processed) at the same time, and which parts are dependent on other parts.
	The benefits and trade offs that are brought from concurrent processing, and be able to apply these to a given scenario.
Programming Techniques	
	To understand the constructs of sequence, iteration and branching.

Programming Techniques	Use these constructs independently of each other, and combine them to produce a solution. These include the selection statements of if (include elseif and else) and select case statements, and both condition based iteration (e.g. while, repeat until) and count controlled iteration (e.g. for) – as well as how condition based can be used as count controlled iteration.
	To read code using these constructs, create code using these constructs, and trace code (for example using a trace table).
	The use and need for variables in a program, and must understand the difference, benefits and drawbacks of both global and local variables.
	Recognise where local and global variables are used, and the impact that these have on the program; for example, the amount of memory used by the program.
	How a program using global variables can be changed to use local variables – and vice-versa.
	What is meant by modular code, and how this can be produced using functions and procedures.
	The differences between functions and procedures and how each is used within a program.
	To read, trace and write code using functions and procedures.
	The purpose and use of parameters within a program, and how they are used in functions and procedures.
	To be able to read, trace and write code that makes use of parameters.
	The difference between passing a parameter by value and by reference. Candidates need to understand the benefits and drawbacks of each, recommending which should be used for a given situation.
	To read, trace and write code that makes use of parameters passed both by value and by reference.
	Have experience of using an IDE to produce code.
	How an IDE can be used to produce code. Candidates should understand the range of features and tools that are within an IDE that can be used to help produce and debug a program.
	The principle of recursion and the key features that produce a recursive algorithm such as a stopping condition. Candidates need to be able to read and trace recursive functions, write recursive functions, and translate a recursive function to an iterative solution and vice-versa.

	The benefits and drawbacks of using both a recursive and iterative solution.
	The purpose of object-oriented code. Candidates need to have an understanding of classes, objects, properties, attributes, and methods, and need to understand the difference between private and public properties, attributes and methods.
	Encapsulation and the use of get and set methods to access private properties.
	The purpose and principles of inheritance, super-classes, parent-classes and sub-classes.
	Polymorphism and how it can be applied to classes.
	To read, trace and write code that makes use of these object-oriented techniques.
	To interpret class diagrams to produce class definitions.
	To identify where object-oriented programming can be used in a solution, and derive an object-oriented solution for a given scenario.
Computational Methods	To determine if a problem can be solved using computational methods, such as decomposition, abstraction, calculations, and storage of data.
	To recognise a problem from a description of a scenario, decompose the problem, and use abstraction to design a solution.
	How divide and conquer can be used within a task to split the task down into smaller tasks that are then tackled.
	How tasks can be carried out simultaneously to produce a solution.
	The purpose of backtracking within an algorithm; for example, when traversing a tree.
	To read, trace and write code that makes use of backtracking for a given scenario.
	What is meant by data mining, and how data mining is used in a situation.
	The complexities within data mining and how a program will search for and interrogate the data.
	What is meant by heuristics, and how they can be used within a program; for example, the A* algorithm.
	Candidates should have some experience of programming a simple heuristic, and be able to apply their knowledge to a given scenario to explain the purpose and benefits of using heuristics in a solution.
	The principles and purpose of performance modelling, and how it is used in the production of software.

	The principle of pipelining and how it is used within programming; for example, the result from a process feeds into the next process.
	How visualisation can be used to create a mental model of what a program will do or work, and that from this they can plan ahead what is going to happen or what they will need to do.
Algorithms	
	To write algorithms using flow charts, pseudocode and program code.
	To follow the code as shown in the OCR pseudocode guide, but are not expected to write code in this syntax.
	Candidate's code is not expected to be syntactically correct, but must use appropriate code structures.
	The need for standard sorting algorithms.
	How the sorting algorithms bubble and insertion work, and the situations when each can, and cannot be used.
	To use the algorithms to sort data, and complete, write and correct algorithms to perform each sorting algorithm.
	The need for standard searching algorithms.
	How the searching algorithms binary and linear work, and the situations when each can, and cannot be used.
	To use the algorithms to search data sets for specific values that may or may not exist in the data set.
	When each searching algorithm can and cannot be used.
	To complete, write and correct algorithms to perform each searching algorithm.
	Using the data structures, stacks and queues.
	The differences and similarities between stacks and queues.
	To add and remove data from both stacks and queues.
	How pointers are used within stacks and queues.
	How stacks and queues can be implemented in a computer system; for example, through the use of an array with pointers.
	To read, correct and write algorithms to add and remove data items, and manipulate data items in a stack and queue.
	How the choice of algorithm can be affected by the data set.
	The impact of specific algorithms on speed and memory use.

Algorithms

To write algorithms using flow charts, pseudocode and program code.
To follow the code as shown in the OCR pseudocode guide, but are not expected to write code in syntax.
Candidates' code is not expected to be syntactically perfect, but must use appropriate structures and techniques.
That there are a range of possible solutions to a task, and that these algorithms may be different in respect to their execution time and the amount of memory they make use of.
To compare different algorithms for a given data set, and demonstrate an understanding of which is more efficient in terms of speed and/or memory.
To compare the use of one or more algorithms against several different data sets to determine how they will differ in their use of memory and speed of execution.
How the efficiency of an algorithm is measured using Big O notation.
The meaning of constant, linear, polynomial, exponential and logarithmic complexity. Candidates need to be able to recognise and draw each of these complexities of using a graph, and be able to read and write the notation.
Candidates need to know the best and worst case complexities for the searching and sorting methods.
The difference between best case, average case and worst case complexities, and how and why these can differ for an algorithm.
The situations where queues, stacks, trees etc. can be used, and be able to recommend and justify their use in specific scenarios or programs.
A stack as a dynamic data structure.
To add and remove items to a stack.
Candidates need to be able read, trace and write code to implement a stack structure (including adding and removing items).
How a stack can be implemented using a different data structure, such as a static array.
A queue as a dynamic data structure.
To add and remove data to/from a queue.
To read, trace and write code to implement a queue structure (including adding and removing items).
How a queue can be implemented using a different data structure, such as a static array.
A tree structure, both binary and multi branch trees.

To add and remove data to/from a tree.
To read, trace and write code to implement a tree structure (including adding and removing items).
How a tree can be implemented using a different data structure, such as a linked list.
Why and how trees are traversed.
How a depth-first (post-order) traversal works, and be able to perform the traversal on a tree.
To read, trace and write code for a post-order traversal.
How a breadth-first traversal works, and be able to perform the search on a tree.
To read, trace and write code for a breadth-first traversal on a tree.
A linked list as a dynamic data structure.
To add, remove and search for data to/from/in a linked list.
To read, trace and write code to implement a linked list (including adding, removing and search for items).
The need for searching and sorting algorithms.
Pre-conditions required to perform a specific algorithm.
How a bubble sort works, and be able to perform a bubble sort on a set of data.
To read, trace and write code to perform a bubble sort.
How a merge sort works and be able to perform a merge sort on a set of data.
To read, trace and write code to perform a merge sort.
How a quick sort works and be able to perform a quick sort on a set of data.
To read, trace and write code to perform a quick sort.
How Dijkstra’s shortest path algorithm works.
To calculate the shortest path in a graph or tree using Dijkstra’s shortest path algorithm.
To read and trace code that performs Dijkstra’s shortest path algorithm.
How the A* algorithm works.
To calculate the shortest path in a graph or tree using the A* algorithm.
To read and trace code that performs the A* algorithm.
How a binary search works and be able to perform a binary search on a set of data.
To read, trace and write code to perform a binary search.
How a linear search works and be able to perform a linear search on a set of data.

	To read, trace and write code to perform a linear search.