OCR Computer Science GCSE

2.1 – Algorithms

Advanced Notes

2.1.1 Computational thinking

Key Concepts

Term	Definition
Decomposition	The process of **breaking down a problem** into smaller, more manageable sub-problems that each accomplish a specific task.
Abstraction	The process of **removing unnecessary details** to focus on the **essential parts** of a problem.
Algorithmic Thinking	The ability to create a clear **set of instructions (algorithms)** to find the solution to a problem.
Pattern Recognition	The process of **identifying similarities or recurring features** in problems and solutions.

Key Computational Thinking Concepts:

- Decomposition:
    - This is the process of breaking down a complex problem into smaller, more manageable sub-problems. The solutions of these sub-problems can then be recombined to solve the complex problem.
    - Each sub-problem should have a clear, identifiable task. These sub-problems can sometimes be broken down further themselves.
    - Why use it?
        - It makes complex problems easier to understand
        - Easier to design and implement solutions for
        - Easier to debug and maintain
        - Allows sub-problems to be split across a team of developers
- Abstraction:
    - Abstraction is the process of removing unnecessary detail from a problem.
    - Why use it?
        - It helps focus on the essential aspects of the problem
        - Time is not wasted on developing unnecessary components
- Algorithmic thinking:
    - This is the ability to create a set of instructions to find the solution to a problem.
    - Why use it?
        - It helps develop clear, step-by-step solutions to problems
        - Easy to identify the most efficient way to solve a problem

- Pattern recognition:
    - Pattern recognition is the process of identifying similarities or recurring features in problems and solutions.
    - Why use it?
        - Helps to identify common features in solutions, such as decisions (selection) or loops (iteration)
        - Allows existing code to be reused, saving time

2.1.2 Designing, creating and refining Algorithms

Algorithm Representation

1. **Inputs**: What data the algorithm receives.
2. **Processing**: How the data is transformed or calculated during the algorithm.
3. **Outputs**: The result or outcome produced by the algorithm, often presented to the user.

You should be able to:

- Identify the **input**, **processing**, and **output** stages in a given algorithm.
- **Explain the purpose** of a simple algorithm.
- Use **trace tables** to follow the flow of an algorithm and determine what it does.

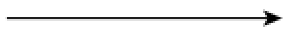
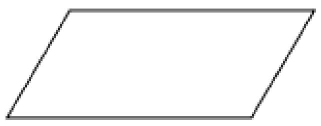
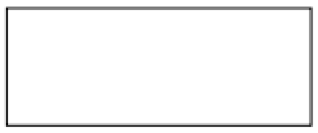
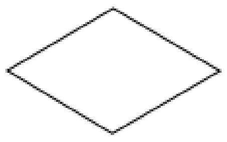
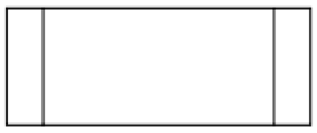
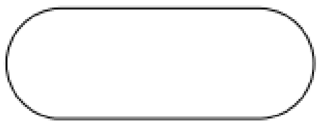
Trace Tables

A trace table is used to track the value of variables and how they change as the algorithm runs step by step. They help the user understand how an algorithm works, check for logical errors, and predict the output of a program.

Representing Algorithms

- Algorithms can be represented in various forms:
    - **Pseudo-code**: A simplified, informal way of describing an algorithm that is closer to human language than programming code, but structured like code.
        - If you are expected to write pseudo-code, or if it is given in an exam, it will be the OCR Exam Reference Language
    - **Program Code:** High level programming languages, such as Python.
    - **Flowcharts**: Diagrams that use symbols / shapes to represent the steps and decision-making processes of an algorithm.
    - For your exam, you are permitted to write code in either pseudo-code or Python, or represent algorithms on a flowchart if asked.

These are the following **flowchart** symbols:

	**Line**		**Input/ Output**
	**Process**		**Decision**
	**Sub program**		**Terminal**

- **Line:** Used to connect flowchart symbols and determine the flow of the algorithm.
- **Input / Output:** Represented by a parallelogram – used when data is entered or printed out.
- **Process:** Represented by a rectangle – shows an operation being performed, such as a calculation.
- **Decision:** Represented by a diamond – used when a yes/no or True/False question is asked, leading to different paths.
- **Sub-program:** Represented by a rectangle with two vertical lines – refers to a predefined procedure or function.
- **Terminal (Start / Stop):** Represented by an oval – used at the beginning and end of a flowchart.

Types of Errors

Type	Description	Example
Syntax Error	Breaks the rules of the language (won't run)	Missing colon in Python
Logic Error	Code runs but produces the wrong result (harder to spot)	Using + instead of *

2.1.3 Searching and sorting Algorithms

What Are Searching Algorithms?

Searching algorithms are step-by-step methods used to find a specific value (target) in a list of data.

Linear Search

How It Works:

- Starts at the first item in the list.
- Check each item one by one until the target is found or the end is reached.

Characteristics:

- Works on any list – sorted or unsorted.
- Simple to implement.
- Slow for large lists (especially if the item is at the end or not there).

Example 1 – (item found):

Let's suppose we are searching for the number '13' in the list below:

15	7	3	5	13	8
15	7	3	5	13	8
15	7	3	5	13	8
15	7	3	5	13	8
15	7	3	5	13	8

Here the target (13) was found at index 4, and so the algorithm returns 4 to indicate that the target (13) was found at that location – remember lists and arrays are 0-indexed, meaning they start at 0!

Note: the algorithm stops as soon as the item is found to avoid unnecessary computation.

Example 2 – (item not found):

Let's suppose we are searching for the number '6' in the list below:

15	7	3	5	13	8
15	**7**	3	5	13	8
15	7	**3**	5	13	8
15	7	3	**5**	13	8
15	7	3	5	**13**	8
15	7	3	5	13	**8**

In this case, the target (6) was not found in the list. The algorithm would **return -1** or output an appropriate message to indicate that the target (6) does not exist in the list.

Binary Search

How It Works:

- Can only be used on **sorted lists**.
- Repeatedly divides the list in half, comparing the middle element to the target. This is known as divide and conquer.
- Narrows down the search range, by discarding the half of the list not containing the target. This is repeated until the item is found, as the middle element, or the list can't be divided further (contains a single element)

Characteristics:

- Much faster than linear search for large, sorted lists.
- More complex logic to implement.
- Requires the list to be sorted.

Example:

Let's suppose we are searching for the number '8' in the list below.

4	7	8	11	15	18	20	24	30	51

Since this is an even length list, there is no middle, so you must either choose the middle-left or middle-right. You must pick one and stick with it. In this case we will go with middle-left.

4	7	8	11	15	18	20	24	30	51

In this case, the search item (8) is not equal to the middle item (15), so we compare and find that the search item is smaller than the middle item – **this means the search item must be on the left half of the list**. We can discard the right side of the list (highlighted in grey).

4	7	8	11	~~15~~	~~18~~	~~20~~	~~24~~	~~30~~	~~51~~

The current list is [4, 7, 8, 11], and as we initially picked middle-left for even length lists, the middle item would now be 7.
The search item (8) is not equal to the middle item (7), but it is greater than the middle item, which means the **search item must be on the right half of the current list**, and so we discard the left side.

~~4~~	7	8	11	~~15~~	~~18~~	~~20~~	~~24~~	~~30~~	~~51~~

The current list is [8, 11] and so the middle item would be 8 – this is in fact our search item and so its index in the list is returned, in this case it is 2.

~~4~~	~~7~~	8	11	~~15~~	~~18~~	~~20~~	~~24~~	~~30~~	~~51~~

What Are Sorting Algorithms?

Sorting algorithms are methods used to arrange data (usually numbers or strings) into a specific order—typically ascending or descending.

Bubble Sort

How It Works:

- Repeatedly goes through the list, comparing 2 adjacent items and swapping them if they are in the wrong order.
- This is done as multiple passes until no swaps are needed — indicating the list is sorted.

Characteristics:

- Simple to understand and implement.
- Inefficient for large datasets as it requires multiple passes.
- It takes a long time if the list is in reverse order.

Example:
Sort the list [3, 2, 9, 10, 7] in ascending order using bubble sort.

First Pass

SWAP

3	2	9	10	7

2	3	9	10	7

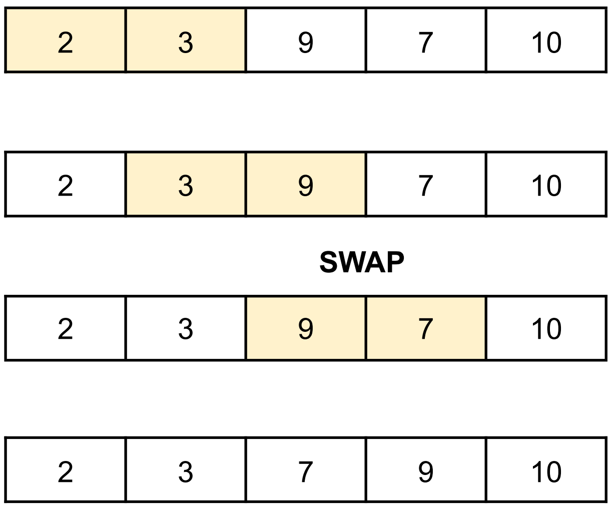
2	3	9	10	7

SWAP

2	3	9	10	7

2	3	9	7	10

Second Pass



The last two items (7 and 10) do not need to be compared – this is because we know the largest number would have been sorted to the top of the list in the first pass, so can be skipped in later passes! In each new pass, one fewer comparison is needed; the largest unsorted item is sorted on each pass.

Note: the algorithm would perform a third pass, to check that no more swaps needed to be made, before declaring the list ordered.

Merge Sort

How It Works:

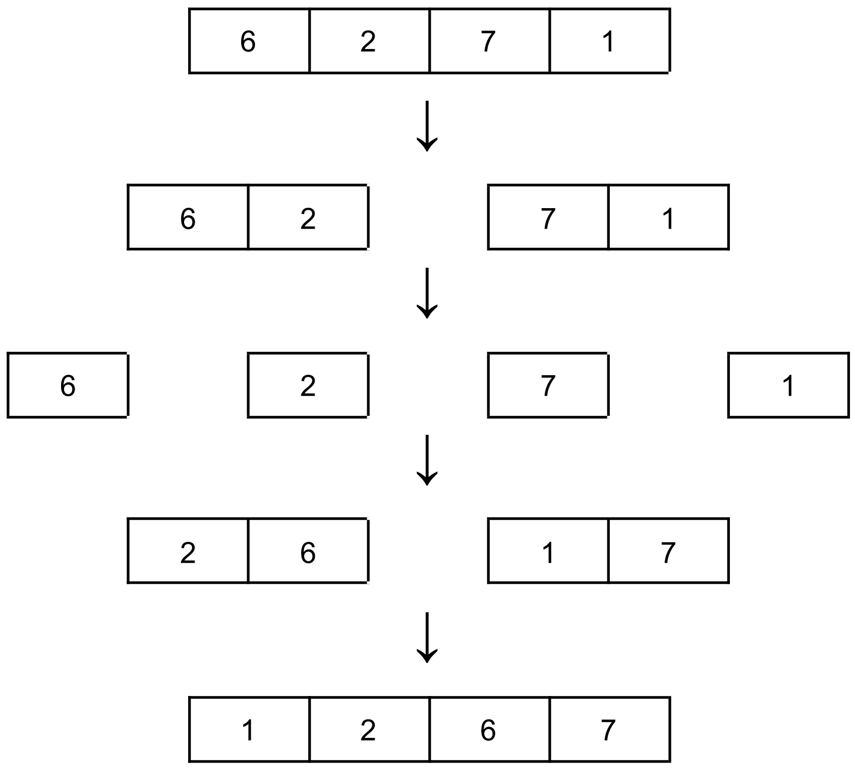
- A divide and conquer algorithm:
- Continuously splits the list in half until each sublist has one item.
- Merges the sublists back together in the correct order by comparing each element in the sublists.

Characteristics:

- Faster than bubble sort for large datasets.
- Uses more memory due to recursion and temporary lists.
- More complex to implement.

Example

Sort the list [6, 2, 7, 1] in ascending order using merge sort.



Insertion Sort

How It Works:

- Start at the second item in the list and insert it into the correct position on its left.
- Continue with the rest of the items, in order, by inserting each item into the correct position until the list is sorted.

Characteristics:

- Simple to understand and implement.
- Efficient for small-medium sized lists.

Example:
Sort the list [5, 3, 9, 4, 7] in ascending order using insertion sort.

Start with the second item (index 1) in the list '3', as it is smaller than 5, we insert it before 5.

5	3	9	4	7

Now looking at the next item '9', as it is larger than all items on its left, it does not move.

3	5	9	4	7

The next item '4' is larger than 3, but smaller than 5, so inserted between them.

3	5	9	4	7

The last item '7' is larger than 5, but smaller than 9, so inserted between them.

3	4	5	9	7

The list is now sorted.

3	4	5	7	9