AQA Computer Science GCSE

3.1 Fundamentals of Algorithms

Advanced Notes

3.1.1 Representing Algorithms

Key Concepts

Term	Definition
Algorithm	An algorithm is a well-defined **sequence of steps** or instructions designed to solve a particular problem or perform a task. *It is a plan, not actual code.*
Computer Program	A computer program is a **set of instructions** written in a programming language **to perform a task**, often based on one or more algorithms.
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.

What is an Algorithm?

- An algorithm is a sequence of steps that can be followed to complete a task.
    - Baking a cake is an example of an algorithm, as you follow a sequence of steps!
- It's crucial to understand that a computer program is an *implementation* of an algorithm. The algorithm is the plan and the program is the code that carries out that plan. An algorithm itself is *not* a computer program.
    - Similarly, baking a cake is not a computer program! However, if you write code for a machine to bake a cake on its own, that would be a computer program.

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

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 pseudo-code is given in an exam, it will be the AQA standard version.
  - **Program Code:** Actual code written in a specific programming language (e.g., Python, C#, VB.NET).
  - **Flowcharts**: Diagrams that use symbols to represent the steps and decision-making process of an algorithm.
  - Exam questions will always specify the expected form of your response (e.g., pseudo-code, program code, or a flowchart).

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** and **visual inspection** to follow the flow of an algorithm and determine what it does.

3.1.2 Efficiency of Algorithms

What is Algorithm Efficiency?

Algorithm efficiency refers to how well an algorithm performs, particularly in terms of the time it takes to complete a task.

Efficiency refers to how well an algorithm uses resources, particularly time and memory.

You need to be able to compare time efficiency of algorithms in simple terms.

- Example: You can sort a list using bubble sort, merge sort, or other sorting algorithms.
- All algorithms can successfully sort a list, but they may vary greatly in speed depending on:
    - The size of the data set
    - The nature of the data
    - The algorithm's design

Key idea: *Just because an algorithm works doesn't mean it's the best choice.*

Time Efficiency (Execution Speed)

Two algorithms solving the same task may:

- Finish in different amounts of time
- Use a different number of steps
- Respond differently to large inputs

Example – Sorting:

- Bubble Sort: Simpler to implement but slower for large lists.
- Merge Sort: More complex but faster on large lists.

Example – Searching:

- Linear Search: Good for unsorted data but checks each item one by one.
- Binary Search: Much faster but only works on sorted data.

Comparing Efficiency – How to Do It

When comparing algorithms:

1. Count how many steps they take to complete.
2. Consider how they handle larger inputs.
3. Consider conditions or requirements for the input data
4. Ask: *Does one consistently finish faster?*

Efficiency increases when:

- The number of steps is lower.
- The algorithm avoids unnecessary operations.

Summary Table

Searching Algorithms:

Feature	Linear Search	Binary Search
Data Requirement	Any data (unsorted)	Requires sorted data
Time Efficiency	Slower (checks one by one)	Faster (halves list each step)
Complexity	Simple	More complex to implement

Sorting Algorithms:

Feature	Bubble Sort	Merge Sort
Simplicity	Very simple	More advanced
Speed (small data)	Acceptable	Fast
Speed (large data)	Slow	Very fast

Exam tips: If asked to compare algorithms, mention both speed and conditions (e.g., sorted/unsorted data). Use examples if possible!

3.1.3 Searching 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.
- Narrow down the search range until the item is found, as the middle element, or the list can't be divided further.

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

Now as 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~~

Comparison Table: Linear vs Binary Search

Feature	Linear Search	Binary Search
Requires Sorting	No	Yes
Search Method	Sequential (one by one)	Halves the list repeatedly
Simplicity	Easy to write	More complex logic
Speed on large data	Slower	Much faster
Best for...	Small or unsorted data	Large, sorted data

3.1.4 Sorting Algorithms

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

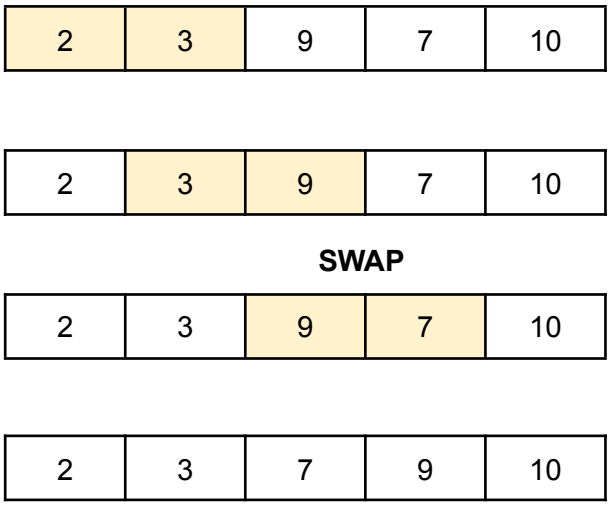
2	3	9	10	7

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!

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

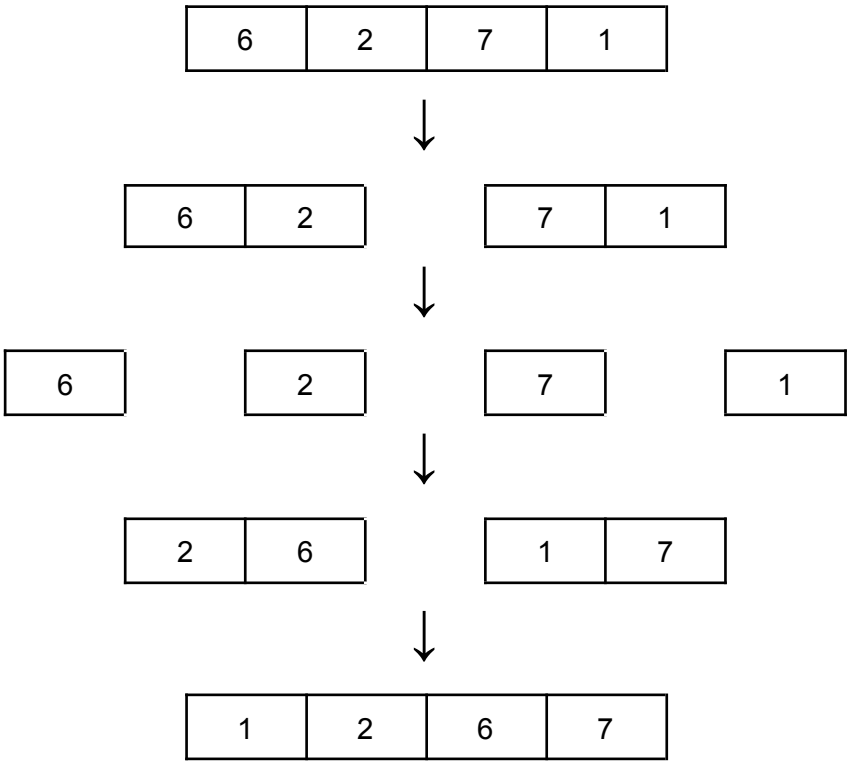
How It Works:

- A divide and conquer algorithm:
    1. Continuously splits the list in half until each sublist has one item.
    2. Merge sublists back together in the correct order.

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.



Comparison Table: Bubble Sort vs Merge Sort

Feature	Bubble Sort	Merge Sort
Simplicity	Very simple	More complex
Memory usage	Minimal	Higher (uses recursion)
Speed on small data	Acceptable	Fast
Speed on large data	Slow	Very fast
Technique	Repeated swapping	Divide and conquer