Three parts of computational thinking

- Abstraction
- Decomposition
- Algorithmic thinking

Abstraction = Process of removing unnecessary details to simplify a problem or system, focusing only on the relevant information needed to solve it

Purpose of Abstraction:

- To reduce complexity.
- To make a problem or system easier to understand.
- To allow developers to focus on the core aspects of a problem without being distracted by irrelevant details.

Decomposition = Decomposition is the process of breaking down a complex problem or system into smaller, more manageable parts that are easier to solve or understand.

Advantages of decomposition:

- Problems are easier to solve
- Different people can work on different parts of a problem at a time → Reduces development time
- Program components developed in one program can be used in other programs

Algorithm = Step-by-step set of instructions designed to perform a specific task or solve a problem.

A computer program is an implementation of an algorithm

Pseudocode:

- Informal description
- No standardised syntax
- Intended for human reading

1. Linear Search

- **Time Complexity**:
  - **Best Case**: O(1) (when the target element is the first in the list).
  - **Worst Case**: O(n) (when the target element is at the end or not found).
- **Advantages**:
  - Simple to implement.
  - Works on both sorted and unsorted data.
- **Disadvantages**:
  - Inefficient for large datasets since it checks every element.

2. Binary Search

Describe the steps a binary search will follow to look for a number in a sorted list.

Any four bullet points for 1 mark each • Select / choose / pick **middle** number (or left/right of middle as even number) and … • **…**check if selected number is **equal to / matches** target number *(not just compare)* • …if searched number is **larger**, discard left half // if searched number is **smaller**, discard right half • Repeat until **number found** • … or remaining list is of size 1 / 0 (number not found)	4 **(AO1 1b)**

- **Time Complexity**:
  - **Best Case**: O(1) (when the target element is the middle one).
  - **Worst Case**: O(log n) (divides the list in half each step).
- **Advantages**:
  - Much faster than linear search for large datasets.
  - Efficient on sorted data.
- **Disadvantages**:
  - Only works on sorted data (requires extra effort if data isn't sorted).
  - More complex to implement.

1. Initialise two pointers (Left pointer at the start of the array and right pointer for the end of the array)
2. Calculate midpoint of array by doing:

- mid = (l+r)//2

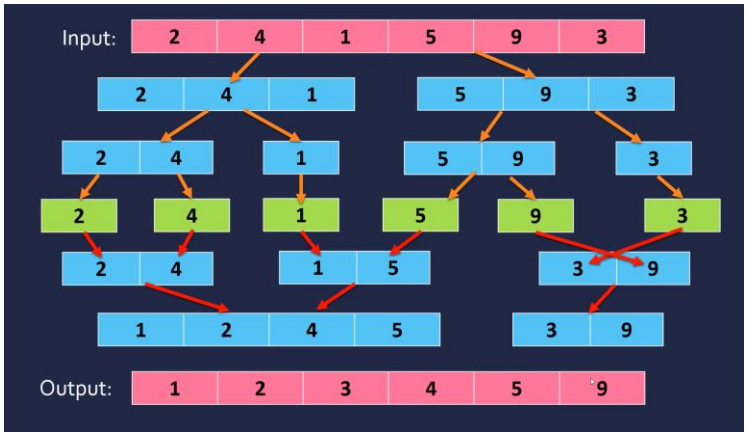
3. Check if midpoint is smaller or greater than target
4. If midpoint > target then repeat on right side else check on left side

Bubble sort – Performs passes

- **Time Complexity:**
  - **Best Case: O(n) (when the list is already sorted).**
  - **Worst Case: $O(n^2)$ (when the list is in reverse order).**
- **Advantages:**
  - **Simple to understand and implement.**
  - **Doesn't need additional memory (in-place sorting).**
  - **Good for small datasets**
- **Disadvantages:**
  - **Very inefficient for large datasets.**
  - **Slow compared to more advanced sorting algorithms.**

Merge sort:

- Divide and conquer



- **Time Complexity:**
  - **Best Case: O(nlogn).**
  - **Worst Case: O(nlogn) (always divides and combines).**
- **Advantages:**
  - **Efficient for large datasets.**
  - **Works well with linked lists.**
  - **Always guarantees O(nlogn) performance.**
- **Disadvantages:**
  - **Requires additional memory for splitting and merging.**
  - **More complex to code compared to bubble sort.**

Insertion sort:

- Creates a new list and appends new elements and sorts them at the same time

- **Time Complexity**:
  - **Best Case**: O(n) (when the list is already sorted).
  - **Worst Case**: O(n^2) (when the list is in reverse order).
- **Advantages**:
  - Simple to implement.
  - Works *well for small or nearly sorted data*sets.
  - In-place sorting (no extra memory needed).
- **Disadvantages**:
  - Inefficient for large datasets.
  - Comparatively slower than merge sort.

Flowchart = Method of representing the sequences in an algorithm in the form of a diagram