Computational Thinking, Problem-Solving and Programming

IB SL Study Guide

Contents

Computational Thinking

Computational thinking is a structured approach to solving problems in a way that can be understood and executed by a computer. The IB syllabus identifies five key thinking approaches.

Five Thinking Approaches

Thinking abstractly — identifying and extracting the essential features of a problem while ignoring irrelevant detail. For example, a map application abstracts road geometry but omits the colour of buildings.

Thinking ahead — planning what outputs are needed before designing the process; identifying pre-conditions, post-conditions, and the order in which steps must happen. For example, a sorting algorithm must consider that its output must satisfy a specific ordering property before a step is written.

Thinking procedurally — breaking a complex task into a sequence of clearly ordered steps that, when followed, produce the desired result. For example, writing a recipe as numbered steps.

Thinking concurrently — identifying parts of a problem that can be done simultaneously (in parallel) rather than sequentially. For example, a search engine crawls multiple web pages at the same time rather than one after another.

Thinking logically — using logical conditions and relationships to determine program flow; understanding when conditions are true or false and how that changes behaviour. For example, using `IF score >= 50 THEN output "pass"`.

> **IB TIP**
>
> IB Paper 1 often asks students to "identify the thinking approach" in a scenario and justify it. The key is to match the **specific action described** to the correct approach: abstracting = ignoring detail; thinking ahead = planning outputs first; procedural = sequential steps; concurrent = parallelism; logical = conditions and decisions. One approach, one justification.

Problem Decomposition and Modular Design

Problem decomposition means breaking a large, complex problem into smaller, more manageable sub-problems. Each sub-problem can then be solved independently and the solutions combined.

Modular design applies decomposition to programming: the solution is built as a collection of **modules** (procedures, functions, methods), each responsible for one specific task.

Benefits of modular design:

- Each module can be written, tested, and debugged independently (**unit testing**)
- Modules can be **reused** across different programs
- Large programs can be developed by teams working on different modules simultaneously
- Easier to **maintain** — changing one module does not require rewriting others, provided the interface (inputs and outputs) stays the same
- Makes programs easier to **read and understand**

> **MEMORISE THIS**
>
> **Modular design key benefits: RETUR** Reusability, Easy maintenance, Testability independently, Understanding (readability), Reusability by teams

IB Pseudocode Conventions

IB Computer Science uses a **standardised pseudocode** notation in all exams. You must use this exact syntax in Paper 1 and Paper 2 answers — markers will not accept general-purpose code or other pseudocode styles.

Variables and Assignment

```
count ← 0
name ← "Alice"
total ← total + 5
```

The ← symbol means "is assigned the value of". The right-hand side is evaluated first, then the result is stored in the variable on the left.

Input and Output

```
input name
output "Hello, ", name
output score
```

Conditionals

```
if score >= 50 then
    output "Pass"
else
    output "Fail"
end if
```

Multi-branch:

```
if grade = "A" then
    output "Excellent"
else if grade = "B" then
    output "Good"
else
```

```
    output "Acceptable"
end if
```

Loops

Count-controlled loop (from/to):

```
loop count from 1 to 10
    output count
end loop
```

Condition-controlled loop:

```
loop while answer ≠ "quit"
    input answer
end loop
```

Arrays

Arrays are zero-indexed in IB pseudocode. A 1D array named `scores` with 5 elements:

```
scores[0] ← 85
scores[4] ← 72
output scores[2]
```

Methods and Procedures

```
METHOD greet(name)
    output "Hello, ", name
END METHOD

METHOD add(a, b)
    return a + b
END METHOD
```

Call a method: `greet("Alice")` or `result ← add(3, 5)`

> ⚠ **EXAM ALERT**
>
> The most common pseudocode errors in IB exams are: (1) using `=` for assignment instead of `←`; (2) writing `while` loops without `loop` and `end loop`; (3) using `print` or `console.log` instead of `output`; (4) forgetting `end if` to close a conditional. These are penalised in Paper 1. Practise writing IB pseudocode exactly.

Algorithm Design — Searching

A **searching algorithm** locates a specific item within a collection of data. The two algorithms required for IB SL are **linear search** and **binary search**.

Linear Search

Examines each element of the array in sequence from the first to the last until the target is found or all elements have been checked.

Pre-condition: None — the array does not need to be sorted.

Pseudocode:

```
METHOD linearSearch(arr, target)
    found ← false
    index ← 0
    loop while index < length(arr) and found = false
        if arr[index] = target then
            found ← true
        else
            index ← index + 1
        end if
    end loop
    if found = true then
        return index
    else
        return -1
    end if
END METHOD
```

Time complexity (concept): In the worst case, every element is examined — n comparisons for an array of n elements. This is called **linear** or $O(n)$ complexity.

WORKED EXAMPLE

Trace table — linear search for target = 7 in `[3, 9, 7, 1, 5]`:

Step	index	arr[index]	found	Action
1	0	3	false	3 ≠ 7, index ← 1
2	1	9	false	9 ≠ 7, index ← 2
3	2	7	false	7 = 7, found ← true
End	2	7	true	Return 2

Binary Search

Repeatedly halves the search space by comparing the target to the middle element. Requires the array to be **sorted** beforehand.

Pre-condition: Array must be sorted in ascending order.

Pseudocode:

```
METHOD binarySearch(arr, target)
    low ← 0
    high ← length(arr) - 1
```

```
    found ← false
    loop while low <= high and found = false
        mid ← (low + high) div 2
        if arr[mid] = target then
            found ← true
        else if arr[mid] < target then
            low ← mid + 1
        else
            high ← mid - 1
        end if
    end loop
    if found = true then
        return mid
    else
        return -1
    end if
END METHOD
```

Time complexity (concept): Each step halves the remaining elements. For n elements, at most $\log_2 n$ comparisons are needed — much faster than linear search for large arrays.

WORKED EXAMPLE

Trace table — binary search for target = 14 in `[2, 5, 8, 12, 14, 23, 38]` (indices 0–6):

Step	low	high	mid	arr[mid]	Action
1	0	6	3	12	12 < 14, low ← 4
2	4	6	5	23	23 > 14, high ← 4
3	4	4	4	14	14 = 14, found ← true
End	—	—	4	—	Return 4

EXAM ALERT

A common error is applying binary search to an **unsorted** array and claiming it will still work — it will not. Always state the pre-condition: "the array must be sorted in ascending order". A second common error is writing `mid = (low + high) / 2` — use `div` for integer division in IB pseudocode to avoid fractional indices.

Algorithm Design — Sorting

A **sorting algorithm** rearranges the elements of an array into a specified order (typically ascending). Two algorithms are required for IB SL.

Bubble Sort

Repeatedly compares adjacent pairs of elements and swaps them if they are in the wrong order. After each full pass, the largest unsorted element "bubbles" to its correct

position at the end.

Pseudocode:

```
METHOD bubbleSort(arr)
    n ← length(arr)
    loop i from 0 to n − 2
        loop j from 0 to n − i − 2
            if arr[j] > arr[j + 1] then
                temp ← arr[j]
                arr[j] ← arr[j + 1]
                arr[j + 1] ← temp
            end if
        end loop
    end loop
END METHOD
```

WORKED EXAMPLE

Trace table — bubble sort on `[5, 3, 8, 1]`:

Pass 1 (i = 0):

j	Comparison	Action	Array state
0	5 > 3? Yes	Swap	[3, 5, 8, 1]
1	5 > 8? No	No swap	[3, 5, 8, 1]
2	8 > 1? Yes	Swap	[3, 5, 1, 8]

Pass 2 (i = 1):

j	Comparison	Action	Array state
0	3 > 5? No	No swap	[3, 5, 1, 8]
1	5 > 1? Yes	Swap	[3, 1, 5, 8]

Pass 3 (i = 2):

j	Comparison	Action	Array state
0	3 > 1? Yes	Swap	[1, 3, 5, 8]

Sorted result: [1, 3, 5, 8]

Selection Sort

Finds the minimum element in the unsorted portion of the array and swaps it into the next sorted position. Repeats until the entire array is sorted.

Concept (pseudocode):

```
METHOD selectionSort(arr)
    n ← length(arr)
    loop i from 0 to n − 2
```

```
        minIndex ← i
        loop j from i + 1 to n − 1
            if arr[j] < arr[minIndex] then
                minIndex ← j
            end if
        end loop
        if minIndex ≠ i then
            temp ← arr[i]
            arr[i] ← arr[minIndex]
            arr[minIndex] ← temp
        end if
    end loop
END METHOD
```

Key difference from bubble sort: Selection sort makes at most $n - 1$ swaps (one per pass); bubble sort may make many more swaps. However, both make $O(n^2)$ comparisons.

> **IB TIP**
>
> IB Paper 1 often gives a partially-sorted array and asks you to show the state after one or two passes of a named algorithm. For bubble sort: show the state after each full pass. For selection sort: show which element was placed in position after each pass. State the algorithm name explicitly before beginning your trace.

Trace Tables

A **trace table** is a tool for manually simulating the execution of an algorithm, tracking the value of each variable at each step. Trace tables are used in IB exams to demonstrate understanding of algorithm execution.

How to Construct a Trace Table

1. Identify all variables used in the algorithm
2. Create a column for each variable (and one for any output)
3. Execute the algorithm one line at a time, recording each change
4. Record only the values that change at each step; leave unchanged cells blank

WORKED EXAMPLE

Trace table — find the maximum value in `[4, 7, 2, 9, 5]`:

```
METHOD findMax(arr)
    max ← arr[0]
    loop i from 1 to length(arr) - 1
        if arr[i] > max then
            max ← arr[i]
        end if
    end loop
    return max
END METHOD
```

i	arr[i]	max	Condition	Action
—	—	4	—	Initialise max ← arr[0]
1	7	4	7 > 4? Yes	max ← 7
2	2	7	2 > 7? No	—
3	9	7	9 > 7? Yes	max ← 9
4	5	9	5 > 9? No	—
End	—	9	—	Return 9

Collection Types

The IB syllabus requires understanding of four abstract data structures and their basic operations.

Arrays

An **array** is an ordered, fixed-size collection of elements of the same type, accessed by index.

- Access any element directly by index: $O(1)$
- Inserting or deleting in the middle is slow (requires shifting elements)
- IB arrays are zero-indexed: `arr[0]` is the first element

Stacks

A **stack** is a Last-In, First-Out (LIFO) collection. Think of a stack of plates — you add and remove only from the top.

Operation	Description
`push(item)`	Add item to the top of the stack
`pop()`	Remove and return the top item
`peek()`	View the top item without removing it
`isEmpty()`	Returns true if the stack has no items

Real-world uses: undo/redo in editors, call stack in program execution, browser back button history.

Queues

A **queue** is a First-In, First-Out (FIFO) collection. Think of a line at a checkout — the first person to join is the first to be served.

Operation	Description
`enqueue(item)`	Add item to the back of the queue
`dequeue()`	Remove and return the item at the front
`isEmpty()`	Returns true if the queue has no items

Real-world uses: print job queues, CPU scheduling, network packet queues.

Linked Lists

A **linked list** is a dynamic collection of **nodes**, where each node stores a data value and a **pointer** (reference) to the next node. Unlike arrays, linked lists do not require contiguous memory.

- **Advantage:** Inserting or deleting a node requires only updating pointers — no shifting of elements
- **Disadvantage:** Accessing element n requires traversing from the head — $O(n)$ access time
- The last node's pointer is `null` (or `NIL`), indicating the end of the list

MEMORISE THIS

Data structure — access pattern:

- **Array** — fast random access by index; fixed size
- **Stack** — LIFO; add/remove from top only
- **Queue** — FIFO; add at back, remove from front
- **Linked list** — dynamic size; fast insert/delete; slow random access

Standard Algorithms

Beyond searching and sorting, the IB syllabus requires students to be able to write or trace algorithms for these common tasks.

Finding Minimum and Maximum

```
METHOD findMin(arr)
    min ← arr[0]
    loop i from 1 to length(arr) - 1
        if arr[i] < min then
            min ← arr[i]
        end if
    end loop
    return min
END METHOD
```

The maximum algorithm is identical but uses `>` and a variable named `max`.

Counting Occurrences

```
METHOD countOccurrences(arr, target)
    count ← 0
    loop i from 0 to length(arr) - 1
        if arr[i] = target then
            count ← count + 1
        end if
    end loop
    return count
END METHOD
```

> **IB TIP**
>
> In Paper 1, these standard algorithms are sometimes presented with minor errors and you are asked to identify the bug. Common inserted bugs include: off-by-one errors in loop bounds (`from 0 to length(arr)` instead of `length(arr) - 1`), using `>` where `>=` is needed, or failing to initialise the accumulator variable before the loop.

Practice Questions

▶ Q1 — A programmer writes a procedure that compresses image files and a separate procedure that uploads them to a server, then calls them in sequence. Identify the computational thinking approach demonstrated and justify your answer. [2 marks]

▶ Q2 — Write IB pseudocode for a procedure that accepts an array of integers and outputs the sum of all elements. [4 marks]

▶ Q3 — Trace the binary search algorithm on the sorted array `[1, 4, 6, 10, 15, 21, 30]` searching for target = 10. Show the trace table. [4 marks]

▶ Q4 — Show the state of the array `[6, 2, 9, 4]` after each pass of a bubble sort. [4 marks]

▶ Q5 — State two differences between a stack and a queue. [4 marks]

▶ Q6 — Write IB pseudocode for a linear search that returns true if a target value is found in an array, and false otherwise. [4 marks]