Computational Thinking and Problem-Solving

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 four key thinking approaches.

Four Computational 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.

> **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. One approach, one justification.

Flowcharts

A **flowchart** is a visual representation of an algorithm, using standardised symbols to show the sequence of steps, decisions, and loops. Flowcharts help communicate an algorithm's logic before writing code or pseudocode.

Standard Flowchart Symbols

Symbol shape	Name	Purpose
Oval (rounded rectangle)	Terminal	Start or End of the algorithm
Rectangle	Process	An action or computation (e.g., `total ← total + arr[i]`)
Diamond	Decision	A yes/no (true/false) question that branches the flow
Parallelogram	Input/Output	Reading data (input) or displaying a result (output)
Arrow	Flow line	Shows the direction of execution

Reading a Flowchart

Execution flows from the Start terminal downward (or in the direction of arrows). At a diamond (decision), the flow splits: one path for the "Yes" (true) branch and one for the "No" (false) branch. Loops are created by having a flow line loop back to an earlier point.

WORKED EXAMPLE

Worked Example — Flowchart for finding the maximum of two numbers:

Start → Input A, B → Decision: A > B? → Yes: Output A → End; No: Output B → End

Written as pseudocode for clarity:

```
input A
input B
if A > B then
    output A
else
    output B
end if
```

The diamond corresponds to `if A > B then`, with the Yes branch going left to output A and the No branch going right to output B.

EXAM ALERT

IB Paper 1 may ask you to draw a flowchart or to trace through a given flowchart and state the output. Always follow the flow line direction carefully. Common mistakes: using the wrong shape (rectangle for a decision instead of a diamond); forgetting to label the Yes/No branches of the diamond; omitting the terminal (Start/End) ovals.

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

IB pseudocode uses 1-based indexing — the first element is `arr[1]`, and an array of N elements runs from `arr[1]` to `arr[N]`. A 1D array named `scores` with 5 elements:

```
scores[1] ← 85
scores[5] ← 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 ← 1
    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	1	3	false	$3 \neq 7$, index ← 2
2	2	9	false	$9 \neq 7$, index ← 3
3	3	7	false	$7 = 7$, found ← true
End	3	7	true	Return 3

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 ← 1
    high ← length(arr)
    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 1–7):

Step	low	high	mid	arr[mid]	Action
1	1	7	4	12	12 < 14, low ← 5
2	5	7	6	23	23 > 14, high ← 5
3	5	5	5	14	14 = 14, found ← true
End	—	—	5	—	Return 5

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 1 to n - 1
        loop j from 1 to n - i
            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 = 1):

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

Pass 2 (i = 2):

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

Pass 3 (i = 3):

j	Comparison	Action	Array state
1	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 1 to n − 1
        minIndex ← i
        loop j from i + 1 to n
            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[1]
    loop i from 2 to length(arr)
        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[1]
2	7	4	7 > 4? Yes	max ← 7
3	2	7	2 > 7? No	—
4	9	7	9 > 7? Yes	max ← 9
5	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 1-based: `arr[1]` is the first element, `arr[N]` is the last in an N-element array

2D Arrays

A **2D array** (two-dimensional array) is an array of arrays — a grid of elements arranged in rows and columns. It is used to represent matrices, game boards, tables, and image pixel grids.

In IB pseudocode, a 2D array is accessed using two indices: `grid[row][col]`. Both indices are 1-based.

```
grid[1][1] ← 5
grid[2][3] ← 9
output grid[1][1]
```

Traversing a 2D array (reading every element):

```
METHOD print2DArray(grid, rows, cols)
    loop i from 1 to rows
        loop j from 1 to cols
            output grid[i][j]
        end loop
    end loop
END METHOD
```

WORKED EXAMPLE

Worked Example — Find the sum of all elements in a 3x3 grid:

```
METHOD sumGrid(grid)
    total ← 0
    loop i from 1 to 3
        loop j from 1 to 3
            total ← total + grid[i][j]
        end loop
    end loop
    return total
END METHOD
```

For grid = [[1,2,3],[4,5,6],[7,8,9]]: total = 1+2+3+4+5+6+7+8+9 = 45.

EXAM ALERT

2D array questions in IB exams often ask you to trace a nested loop. Track both loop variables (i and j) simultaneously in your trace table — add a column for each. The inner loop runs completely for each iteration of the outer loop.

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
- **2D Array** — grid structure; nested loops to traverse; row-column indexing
- **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

HL Abstract Data Structures

Binary Trees

A **binary tree** is a hierarchical data structure where each node has at most two children, referred to as the **left child** and **right child**. The topmost node is the **root**; nodes with no children are **leaves**.

Binary Search Tree (BST): A special binary tree where, for every node:

- All values in the **left subtree** are less than the node's value
- All values in the **right subtree** are greater than the node's value

This property enables efficient searching, insertion, and deletion.

Three traversal orders for binary trees:

Traversal	Order	Visits
In-order	Left → Root → Right	Produces a sorted sequence for a BST
Pre-order	Root → Left → Right	Useful for copying or serialising the tree
Post-order	Left → Right → Root	Useful for deleting the tree or evaluating expression trees

WORKED EXAMPLE

Worked Example — In-order traversal of the BST above:

In-order visits: 1, 3, 4, 6, 7, 8, 10, 13, 14

Note this produces the values in ascending sorted order — confirming the BST property.

Searching in a BST:

1. Start at the root
2. If the target equals the current node, found
3. If the target is less, go left; if greater, go right
4. Repeat until found or reach a null pointer (not present)

Average time complexity: $O(\log n)$ for a balanced BST — each step halves the remaining search space, just like binary search on an array.

EXAM ALERT

HL exams may ask you to insert values into a BST and draw the resulting tree, or to perform a named traversal and list the output order. For insertion: follow the same comparison rules as searching until you reach a null position, then insert the new node there. The order of insertion determines the tree's shape.

HL Recursion

Recursion is a programming technique where a function calls itself as part of its own definition. A recursive function must have:

1. **Base case** — a condition under which the function returns a result without calling itself again (prevents infinite recursion)
2. **Recursive case** — the function calls itself with a simpler or smaller version of the problem

How Recursion Works

Each recursive call creates a new **stack frame** on the call stack, storing the local variables and the return address for that call. When the base case is reached, the stack

unwinds — each frame returns its result to the caller below it.

WORKED EXAMPLE

Worked Example — Recursive factorial:

```
METHOD factorial(n)
    if n = 0 then
        return 1
    else
        return n * factorial(n - 1)
    end if
END METHOD
```

Trace for factorial(4):

- factorial(4) calls factorial(3)
- factorial(3) calls factorial(2)
- factorial(2) calls factorial(1)
- factorial(1) calls factorial(0)
- factorial(0) returns 1 (base case)
- factorial(1) returns 1 * 1 = 1
- factorial(2) returns 2 * 1 = 2
- factorial(3) returns 3 * 2 = 6
- factorial(4) returns 4 * 6 = 24

Recursion vs iteration:

Attribute	Recursion	Iteration
Mechanism	Function calls itself	Loop repeats
Memory	Each call uses stack space; risk of stack overflow	Fixed memory use
Readability	Elegant for tree/graph problems	Often simpler for linear problems
Use cases	Tree traversal, divide-and-conquer, factorials	Array processing, simple loops

IB TIP

Any recursive algorithm can be rewritten iteratively and vice versa. IB HL questions sometimes ask you to “convert a recursive algorithm to an iterative one” or to “identify the base case and recursive case” in given code. Always identify both explicitly.

HL Big-O Notation

Big-O notation describes the **time complexity** of an algorithm — how the number of operations grows as the input size n increases. It focuses on the dominant term as n

becomes large, ignoring constants and lower-order terms.

Common Complexity Classes

Big-O	Name	Example	Growth
$O(1)$	Constant	Array access by index	Does not grow with n
$O(\log n)$	Logarithmic	Binary search, BST search	Grows very slowly
$O(n)$	Linear	Linear search, finding max	Proportional to n
$O(n \log n)$	Linearithmic	Merge sort, quicksort (average)	Slightly worse than linear
$O(n^2)$	Quadratic	Bubble sort, selection sort	Grows rapidly
$O(2^n)$	Exponential	Recursive subset generation	Extremely rapid growth

Analysing Algorithm Complexity

- **Single loop** over n elements: $O(n)$
- **Nested loop** (loop inside a loop), each running n times: $O(n^2)$
- **Halving the search space** each step: $O(\log n)$
- **Fixed number of steps** regardless of input: $O(1)$

WORKED EXAMPLE

Worked Example — Determine the complexity of bubble sort:

Bubble sort uses two nested loops. The outer loop runs $n - 1$ times. For each iteration of the outer loop, the inner loop runs up to $n - 1$ times. Total comparisons in the worst case: approximately $\frac{n(n-1)}{2}$.

Dominant term: n^2. Therefore bubble sort is $O(n^2)$.

MEMORISE THIS

Search and sort complexity to memorise:

- Linear search: $O(n)$
- Binary search: $O(\log n)$
- Bubble sort: $O(n^2)$
- Selection sort: $O(n^2)$
- Binary tree search (balanced): $O(\log n)$

EXAM ALERT

When comparing algorithms in HL, always state the Big-O class and explain what it means in context. For example: “Binary search is $O(\log n)$, meaning that for an array of 1,000,000 elements, at most 20 comparisons are needed, whereas linear search would require up to 1,000,000 comparisons in the worst case.”

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[1]
    loop i from 2 to length(arr)
        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 1 to length(arr)
        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 1 to length(arr) + 1` instead of `length(arr)`), using `>` where `>=` is needed, or failing to initialise the accumulator variable before the loop.

Video Resources

▸Watch: IB CS Topic 4 — Pseudocode & Problem-Solving VIDEO

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]