IB CS Pseudocode Reference — Approved Notation

IB SL Study Guide

Contents

> **IB COMPUTER SCIENCE SL:** All Guides Computational Thinking System Fundamentals Computer Organization

Introduction

IB Computer Science uses a standardised pseudocode notation for all exam questions and internal assessments. This notation is designed to be **language-neutral** — it does not favour students who know Python, Java, or any specific programming language. Using Python or Java syntax in your exam answers **will lose marks**.

> ⚠ **EXAM ALERT**
>
> **Common mistakes that cost marks:**
>
> - Writing `x := 5` instead of `x ← 5`
> - Writing `x = x + 1` instead of `x ← x + 1` (the single `=` is the **comparison** operator, not assignment)
> - Forgetting to write `end if`, `end loop`, or `END METHOD`
> - Using Python methods like `.append()` instead of IB Collection methods like `.addItem()`
> - Writing `//` for integer division instead of `div`

This guide provides the complete IB pseudocode notation with worked examples and side-by-side Python comparisons to help you avoid these mistakes.

Basic Syntax

Variable Assignment

IB pseudocode uses the **left arrow** ← for assignment. This means "assign the value on the right to the variable on the left".

IB Pseudocode:

```
x ← 10
name ← "Alice"
total ← total + 5
```

Python Equivalent:

```
x = 10
name = "Alice"
total = total + 5
```

⚠ EXAM ALERT

Critical: The single equals sign = in IB pseudocode is the **comparison operator** (like == in Python). Never use = for assignment.

Wrong: `x = 10` Right: `x ← 10`

Wrong: `if x = 10 then` (this is correct — it's a comparison!) Right: `if x = 10 then`

Output

Use the `output` keyword to display a value or message.

IB Pseudocode:

```
output "Hello, world!"
output total
output "The sum is: ", sum
```

Python Equivalent:

```
print("Hello, world!")
print(total)
print("The sum is:", sum)
```

Input

Use the `input` keyword to read a value from the user.

IB Pseudocode:

```
input name
input age
```

Python Equivalent:

```
name = input()
age = int(input())
```

Comments

Use `//` for single-line comments.

IB Pseudocode:

```
// This is a comment
x ← 10 // Assign 10 to x
```

Python Equivalent:

```
# This is a comment
x = 10  # Assign 10 to x
```

Arithmetic Operators

Operator	Meaning	Example	Result
+	Addition	7 + 3	10
−	Subtraction	7 − 3	4
*	Multiplication	7 * 3	21
/	Division (real result)	7 / 2	3.5
div	Integer division	7 div 2	3
mod	Remainder (modulus)	7 mod 3	1

⚠ EXAM ALERT

Critical: Use `div` for integer division, **not** `//` (which is a comment marker). Writing `7 // 2` will be interpreted as "7" followed by a comment containing "2" — not a division operation.

Wrong: `mid ← (low + high) // 2` Right: `mid ← (low + high) div 2`

Comparison and Logical Operators

Comparison Operators

Operator	Meaning	Example
=	Equal to	x = 10
≠	Not equal to	x ≠ 10
<	Less than	x < 10
>	Greater than	x > 10
≤	Less than or equal to	x ≤ 10
≥	Greater than or equal to	x ≥ 10

IB TIP

IB exams will accept `!=` as an alternative to ≠ if you cannot type the mathematical symbol, but using the official symbol is preferred.

Logical Operators

Operator	Meaning	Example
and	Logical AND	x > 0 and x < 10
or	Logical OR	x < 0 or x > 10
not	Logical NOT	not found

IB Pseudocode:

```
if age ≥ 18 and hasLicense = true then
    output "You can drive"
end if
```

Python Equivalent:

```
if age >= 18 and hasLicense == True:
    print("You can drive")
```

Control Structures

IF Statement

IB Pseudocode:

```
if condition then
    // statements
end if
```

With ELSE:

```
if condition then
    // statements when true
else
    // statements when false
end if
```

With ELSE IF:

```
if condition1 then
    // statements
else if condition2 then
    // statements
else
    // statements
end if
```

⚠ EXAM ALERT

Critical: Every `if` **must** end with `end if`. Forgetting this is a common error that costs marks. IB pseudocode does not use indentation alone to define blocks — the explicit `end if` is required.

Wrong:

```
if x > 0 then
    output "positive"
```

Right:

```
if x > 0 then
    output "positive"
end if
```

WORKED EXAMPLE

Worked Example — Determine grade from score:

IB Pseudocode:

```
input score
if score ≥ 90 then
    grade ← "A"
else if score ≥ 80 then
    grade ← "B"
else if score ≥ 70 then
    grade ← "C"
else if score ≥ 60 then
    grade ← "D"
else
    grade ← "F"
end if
output grade
```

Python Equivalent:

```
score = int(input())
if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"
elif score >= 70:
    grade = "C"
elif score >= 60:
    grade = "D"
else:
    grade = "F"
print(grade)
```

Loops

IB pseudocode has **three types of loops**, each with a distinct structure.

LOOP WHILE

Repeats while a condition is true. The condition is checked **before** each iteration.

IB Pseudocode:

```
loop while condition
    // statements
```

```
end loop
```

Example:

```
count ← 1
loop while count ≤ 5
    output count
    count ← count + 1
end loop
```

Python Equivalent:

```
count = 1
while count <= 5:
    print(count)
    count = count + 1
```

> ⚠ EXAM ALERT
>
> **Critical:** Every `loop` **must** end with `end loop`. Do not forget this terminator.

LOOP UNTIL

Repeats until a condition becomes true. The condition is checked **after** each iteration, so the loop body always runs **at least once**.

IB Pseudocode:

```
loop until condition
    // statements
end loop
```

Example:

```
input password
loop until password = "secret"
    output "Incorrect. Try again."
    input password
end loop
output "Access granted"
```

Python Equivalent:

```
password = input()
while password != "secret":
    print("Incorrect. Try again.")
    password = input()
print("Access granted")
```

> IB TIP

> **LOOP WHILE vs LOOP UNTIL:**
>
> - `loop while condition` → repeat as long as the condition is **true**
> - `loop until condition` → repeat as long as the condition is **false** (stop when it becomes **true**)
> - `loop until` always runs at least once; `loop while` may not run at all if the condition is initially false

LOOP FROM…TO

A counted loop that repeats a fixed number of times. The loop variable automatically increments by 1 each iteration.

IB Pseudocode:

```
loop variable from start to end
    // statements
end loop
```

Example:

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

This is equivalent to:

```
i ← 1
loop while i ≤ 10
    output i
    i ← i + 1
end loop
```

Python Equivalent:

```
for i in range(1, 11):  # Note: range(1, 11) produces 1 to
    print(i)
```

> ⚠ EXAM ALERT
>
> **Critical difference from Python:**
>
> - IB: `loop i from 1 to 10` includes **both 1 and 10** (inclusive on both ends)
> - Python: `range(1, 10)` includes 1 but **excludes** 10 (so you need `range(1, 11)` to match IB)

WORKED EXAMPLE

Worked Example — Sum of numbers 1 to 100:

IB Pseudocode:

```
total ← 0
loop i from 1 to 100
    total ← total + i
end loop
output total
```

Python Equivalent:

```
total = 0
for i in range(1, 101):
    total = total + i
print(total)
```

Result: 5050

Arrays

IB arrays are **1-based** — the first element is `arr[1]`, not `arr[0]`.

Array Declaration and Access

IB Pseudocode:

```
arr ← [5, 10, 15, 20]
output arr[1]  // Outputs 5
arr[3] ← 99
```

Python Equivalent (0-based):

```
arr = [5, 10, 15, 20]
print(arr[0])  # Outputs 5
arr[2] = 99
```

EXAM ALERT

Critical: IB arrays are **1-based**. The first element is `arr[1]`, and the last element of an N-element array is `arr[N]`.

Wrong (Python-style): `arr[0]` Right (IB-style): `arr[1]`

Array Length

Use `length(arr)` to get the number of elements.

IB Pseudocode:

```
arr ← [3, 7, 2, 9]
n ← length(arr)  // n = 4
```

Python Equivalent:

```python
arr = [3, 7, 2, 9]
n = len(arr)  # n = 4
```

Traversing an Array

IB Pseudocode:

```
arr ← [10, 20, 30, 40, 50]
loop i from 1 to length(arr)
    output arr[i]
end loop
```

Python Equivalent:

```python
arr = [10, 20, 30, 40, 50]
for i in range(len(arr)):
    print(arr[i])
```

2D Arrays

A 2D array is accessed using two indices: `grid[row][col]`. Both indices are **1-based**.

IB Pseudocode:

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

Traversing a 2D Array

IB Pseudocode:

```
rows ← 3
cols ← 4
loop i from 1 to rows
    loop j from 1 to cols
        output grid[i][j]
    end loop
end loop
```

Python Equivalent (0-based):

```
rows = 3
cols = 4
for i in range(rows):
    for j in range(cols):
        print(grid[i][j])
```

Collections

IB defines a **Collection** as an abstract data structure with specific methods for adding and iterating over items. Collections are **not** the same as arrays — they are dynamic and unordered.

Collection Methods

Method	Description
`collection.addItem(item)`	Add an item to the collection
`collection.resetNext()`	Reset the iterator to the start
`collection.hasNext()`	Returns `true` if there are more items to iterate
`collection.getNext()`	Returns the next item and advances the iterator

> ⚠ **EXAM ALERT**
>
> **Critical:** Do **not** use Python list methods like `.append()` or `.pop()` when writing IB pseudocode. Use the official IB Collection methods.
>
> Wrong: `names.append("Alice")` Right: `names.addItem("Alice")`

WORKED EXAMPLE

Worked Example — Add items to a collection and iterate:

IB Pseudocode:

```
names ← new Collection()
names.addItem("Alice")
names.addItem("Bob")
names.addItem("Carol")

names.resetNext()
loop while names.hasNext()
    current ← names.getNext()
    output current
end loop
```

Output:

```
Alice
Bob
Carol
```

Python Equivalent (using a list as a substitute):

```
names = []
names.append("Alice")
names.append("Bob")
names.append("Carol")

for current in names:
    print(current)
```

Stacks

A **stack** is a Last-In, First-Out (LIFO) data structure. Items are added and removed from the **top** only.

Stack Methods

Method	Description
`stack.push(item)`	Add an item to the top of the stack
`stack.pop()`	Remove and return the top item
`stack.isEmpty()`	Returns `true` if the stack is empty

IB Pseudocode:

```
s ← new Stack()
s.push(10)
s.push(20)
s.push(30)
output s.pop()  // Outputs 30
output s.pop()  // Outputs 20
```

Queues

A **queue** is a First-In, First-Out (FIFO) data structure. Items are added at the **back** and removed from the **front**.

Queue Methods

Method	Description
`queue.enqueue(item)`	Add an item to the back of the queue
`queue.dequeue()`	Remove and return the item at the front
`queue.isEmpty()`	Returns `true` if the queue is empty

IB Pseudocode:

```
q ← new Queue()
q.enqueue(10)
q.enqueue(20)
q.enqueue(30)
output q.dequeue()  // Outputs 10
output q.dequeue()  // Outputs 20
```

Sub-programs (Methods)

IB pseudocode uses the `METHOD` keyword to define sub-programs (functions/procedures). Methods can take **parameters** and **return** a value.

Method Declaration

IB Pseudocode:

```
METHOD methodName(parameter1, parameter2)
    // statements
    return value
END METHOD
```

Method Call

```
result ← methodName(arg1, arg2)
```

⚠ EXAM ALERT

Critical: Every `METHOD` **must** end with `END METHOD`. The entire method definition is bookended by these keywords.

WORKED EXAMPLE

Worked Example — Method to calculate the area of a rectangle:

IB Pseudocode:

```
METHOD calculateArea(width, height)
    area ← width * height
    return area
END METHOD

// Calling the method
result ← calculateArea(5, 10)
output result  // Outputs 50
```

Python Equivalent:

```
def calculateArea(width, height):
    area = width * height
    return area

# Calling the function
result = calculateArea(5, 10)
print(result)  # Outputs 50
```

WORKED EXAMPLE

Worked Example — Method with no return value (procedure):

IB Pseudocode:

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

printGreeting("Alice")  // Outputs: Hello, Alice
```

Python Equivalent:

```
def printGreeting(name):
    print("Hello,", name)

printGreeting("Alice")  # Outputs: Hello, Alice
```

Side-by-Side Comparisons

Example 1: Find the Maximum of Two Numbers

IB Pseudocode	Python
`if a > b then`	`if a > b:`
`max ← a`	`max = a`
`else`	`else:`
`max ← b`	`max = b`
`end if`	*(no explicit end needed)*

Example 2: Sum of Array Elements

IB Pseudocode	Python
`total ← 0`	`total = 0`
`loop i from 1 to length(arr)`	`for i in range(len(arr)):`
`total ← total + arr[i]`	`total = total + arr[i]`
`end loop`	*(no explicit end needed)*

Note: In Python, `arr[i]` works directly because `range(len(arr))` produces 0-based indices matching Python's 0-based arrays. To match IB's 1-based arrays, you'd need `range(1, len(arr)+1)` and access `arr[i-1]`.

Example 3: User Input Validation

IB Pseudocode	Python
`input num`	`num = int(input())`
`loop until num ≥ 0`	`while num < 0:`
`output "Must be non-negative"`	`print("Must be non-negative")`
`input num`	`num = int(input())`
`end loop`	*(no explicit end needed)*

Worked Algorithm Examples

1. Linear Search

Problem: Search for a target value in an unsorted array. Return the **index** if found, or `-1` if not found.

IB 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
```

Python Equivalent:

```
def linearSearch(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
```

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

2. Bubble Sort

Problem: Sort an array in ascending order using bubble sort.

IB 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
```

Python Equivalent:

```python
def bubbleSort(arr):
    n = len(arr)
    for i in range(n - 1):
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

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]

3. Binary Search

Problem: Search for a target value in a **sorted** array using binary search. Return the **index** if found, or -1 if not found.

Pre-condition: The array must be sorted in ascending order.

IB 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
```

Python Equivalent:

```
def binarySearch(arr, target):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

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

Critical: Binary search only works on a **sorted** array. Always state the pre-condition: "the array must be sorted in ascending order". Use `div` for integer division when calculating `mid`, **not** `/` (which produces a real number) or `//` (which is a comment marker).

4. Recursive Factorial

Problem: Calculate the factorial of a non-negative integer n using recursion.

$n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$

$0! = 1$ (by definition)

IB Pseudocode:

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

Python Equivalent:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

WORKED EXAMPLE

Trace — 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`

Result: 24

IB TIP

Recursion requirements:

1. **Base case** — a condition where the method returns without calling itself (prevents infinite recursion)
2. **Recursive case** — the method calls itself with a simpler/smaller input

> For factorial: base case is `n = 0`; recursive case is `n * factorial(n - 1)`.

Common Syntax Errors Summary

Error	Wrong	Right
Assignment operator	`x = 10`	`x ← 10`
Comparison in condition	*(actually correct)* `if x = 10 then`	`if x = 10 then`
Integer division	`mid ← (low + high) // 2`	`mid ← (low + high) div 2`
Forgetting terminators	`if x > 0 then output x`	`if x > 0 then output x end if`
Python list methods	`arr.append(5)`	`collection.addItem(5)`
0-based indexing	`arr[0]`	`arr[1]`
Loop range inclusivity	`loop i from 1 to 9` to get 1–10	`loop i from 1 to 10`

MEMORISE THIS

Key IB Pseudocode Rules:

1. Assignment uses ←, not =
2. Comparison uses =, not ==
3. Integer division uses `div`, not `//`
4. Every `if` ends with `end if`
5. Every `loop` ends with `end loop`
6. Every `METHOD` ends with `END METHOD`
7. Arrays are 1-based: `arr[1]` is the first element
8. Collections use `.addItem()`, `.getNext()`, `.hasNext()`, `.resetNext()`
9. Stacks use `.push()`, `.pop()`, `.isEmpty()`
10. Queues use `.enqueue()`, `.dequeue()`, `.isEmpty()`

Practice Questions

▶ **Question 1:** Write an IB pseudocode method to find the sum of all elements in an array.

▶ **Question 2:** Write an IB pseudocode method to count how many times a target value appears in an array.

▶ **Question 3:** Write an IB pseudocode method to reverse an array in place.

▶ **Question 4:** Write an IB pseudocode method to check if an array is sorted in ascending order.

▶ **Question 5:** Write an IB pseudocode method to find the minimum value in an array.

▶ **Question 6:** Write an IB pseudocode method to calculate the Fibonacci number at position n using recursion.

Exam Strategy Tips

> **IB TIP**
>
> **Paper 1 algorithm questions — step-by-step approach:**
>
> 1. **Read the question carefully** — identify whether you need to write pseudocode, trace a given algorithm, or answer conceptual questions
> 2. **Identify the required operations** — searching, sorting, counting, summing, etc.
> 3. **Write the method signature** — `METHOD name(parameters)`
> 4. **Declare and initialise variables** — use ← for assignment
> 5. **Write the loop structure** — choose `loop while`, `loop until`, or `loop from...to`
> 6. **Write the conditions** — use = for comparison, not ==
> 7. **Add terminators** — `end if`, `end loop`, `END METHOD`
> 8. **Check for common errors** — assignment operator, integer division, array indexing, missing terminators

> **IB TIP**
>
> **Trace table questions:**
>
> - Create a column for each variable mentioned in the algorithm
> - Add a column for any output produced
> - Execute the algorithm **one line at a time**, recording each variable change
> - If a loop is involved, show each iteration as a separate row
> - If a condition is checked, show the result of the comparison
> - Be methodical — examiners award marks for correct intermediate values, even if the final answer is wrong

> **IB TIP**
>
> **Internal Assessment (IA):**
>
> You **may** use Python, Java, or any programming language for your IA code. However, if you include pseudocode in your documentation (e.g., planning section, algorithm design), you **must** use IB-approved pseudocode notation. Mixing Python syntax into pseudocode will be marked as poor practice.