

B2 4 student review

HL Python | End-of-unit review | Original IB-style practice

Use this after learning the relevant topics. Start now with Big O and linear search; return to binary search, sorting and recursion after those lessons. Try the questions on paper before looking at the answers on the final page.

Reading and learning checklist

Objective	Read	You should be able to
B2.4.1	358-361	Explain time and memory growth and justify algorithm choice.
B2.4.2	361-365	Write and trace linear and binary search.
B2.4.3	365-369	Write and trace bubble and selection sort; compare their costs.
B2.4.4-5 HL	370-377	Explain recursion and its applications; write and trace a simple function with one recursive call.

Explain your reasoning

For a trace, record changing values in the requested order. For an explanation, connect what happens to why it matters. For a comparison, refer to both algorithms under the same conditions. For a constructed algorithm, follow its input/output contract and restrictions. The IB FAQ says pseudocode is not required; Big O questions focus on describing growth, not formal calculations.

Big O in plain language

Let n be the amount of input. $O(1)$ means the work does not grow with n ; $O(\log n)$ often comes from repeatedly halving; $O(n)$ from a full scan; $O(n^2)$ from two nested full scans. Doubling n roughly doubles linear work and quadruples quadratic work for large n , ignoring constants and smaller terms.

Big O is a growth bound, not an exact number of seconds. State the case you mean. Worst-case linear search is $O(n)$, but finding the target immediately is $O(1)$.

Worked example

```
total = 0
for value in values:
    total = total + value
```

There is one addition per item: $O(n)$ time. The running total uses $O(1)$ auxiliary space, meaning extra memory beyond the supplied list. Including the input list gives $O(n)$ total space under our fixed-size-item model.

R1 Complexity practice [6]

- (a) Two separate loops each visit every item. State the time complexity and explain why it is not quadratic. [2]
- (b) A function copies the input list, then scans the copy. State its auxiliary space and explain. [2]
- (c) An $O(n)$ algorithm takes longer than an $O(n^2)$ algorithm for $n = 4$. Explain why this does not contradict the growth classifications. [2]

Searching and safe boundaries

Choosing a search

Linear search checks items one at a time and works on unsorted lists. Binary search repeatedly discards half the remaining candidates and needs data sorted by the key being searched, with efficient indexed access. Sorting just to perform one search can cost more than scanning once.

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

// divides and rounds down to an integer. Here it chooses a valid middle index. Return -1 means the target was not found. List indices begin at 0, so len(values) - 1 is the final valid index.

Worked trace

For [3, 8, 12, 17, 23, 31, 44] and target 31, record bounds before each comparison:

low	high	mid	values[mid]	Action
0	6	3	17	Set low to 4
4	6	5	31	Return 5

R2 Search practice [6]

(a) Trace the same function with target 10. Give low, high, mid and values[mid] for each iteration, then the returned value. [4]

(b) The book example on page 364 starts upper at len(numbers) while using an inclusive loop. State the correction and explain which boundary error it prevents. [2]

Test more than the happy path

Try an empty list, one item, first and last items, an absent target below the minimum, an absent target between items and one above the maximum. With duplicates, this binary search returns a matching index, not necessarily the first match.

Sorting and fair comparisons

Two different ways to grow a sorted region

Bubble sort compares neighbouring items and swaps them if out of order. A left-to-right pass places the largest remaining value at the right end. Selection sort scans the unsorted region for its smallest value and swaps it into the next position on the left.

Worked first passes

Start with [5, 2, 4, 1]. Bubble sort first pass: [2, 5, 4, 1], then [2, 4, 5, 1], then [2, 4, 1, 5]. Selection sort first pass: locate 1, then swap it with 5 to give [1, 2, 4, 5]. The intermediate states need not match.

Algorithm / implementation	Best time	Worst time	Extra space
Linear search	$O(1)$	$O(n)$	$O(1)$
Iterative binary search	$O(1)$	$O(\log n)$	$O(1)$
Bubble sort with early exit	$O(n)$	$O(n^2)$	$O(1)$
Bubble sort with fixed passes	$O(n^2)$	$O(n^2)$	$O(1)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(1)$

These space figures exclude the supplied list. A copy made for testing adds $O(n)$ memory to the test program. All complexity claims assume ordinary constant-time indexing and comparisons.

R3 Sorting practice [6]

- (a) Starting afresh from [6, 3, 5, 2], write the list after each complete left-to-right bubble pass. Use a shrinking right boundary and stop after a pass with no swaps. [3]
- (b) Starting afresh from that same list, write the result of the first selection-sort pass. [1]
- (c) A student says selection sort is $O(n)$ because it makes at most $n - 1$ swaps. Explain the missing work. [2]

Common mistakes

A pass is a complete sweep through the relevant region, not one comparison. Selection sort must reset its minimum index at the start of each pass. Bubble sort must reset its swapped flag for every pass if it uses an early exit. Give both algorithms fresh copies of the same input when comparing them.

Recursion for HL

Recursion means a function calls itself. A base case produces a result without another recursive call. Each recursive step must move towards that case. Pending calls wait on the call stack, which uses memory.

Worked example with one recursive call

```
def sum_to(n):  
    # Assume n is a non-negative integer.  
    if n == 0:  
        return 0  
    return n + sum_to(n - 1)
```

sum_to(3) waits for sum_to(2), which waits for sum_to(1), which waits for sum_to(0). The base case returns 0. The waiting calls then return 1, 3 and 6. This return process is sometimes called unwinding.

Under the classroom model, this has $O(n)$ time and $O(n)$ auxiliary stack space. A loop can calculate the same total with $O(1)$ auxiliary space. Recursion is not automatically faster or smaller.

Applications you should understand

Quicksort partitions data around a pivot and recursively sorts smaller partitions. Fractals repeat a similar pattern at smaller scales. A tree contains smaller subtrees that can be processed recursively. Explain how the repeated smaller problem helps, and consider stack depth and memory limits.

For B2.4.5, practise writing and tracing simple non-branching recursion: at most one recursive call in each activation. Understanding a branching application does not mean you must construct it for this objective.

R4 Recursion practice [6]

- (a) Trace sum_to(4): list calls down to the base case and values returned as they unwind. [3]
- (b) Explain why changing $n - 1$ to n prevents this function from returning for $n = 4$. [2]
- (c) State its auxiliary space complexity for non-negative input n . [1]

Before the assessment

Check that you can explain an answer as well as calculate it. For every algorithm, identify its assumptions, a boundary input and the result the caller should receive. Practise handwriting Python with clear indentation.

Sources: IB Computer science guide, February 2025, first assessment 2027, printed pp. 39-40 and 50; IB 2027 specimen HL Paper 2 (Python) and markscheme; IB Computer science FAQ, 26 January 2026; Baumgarten, Ganea and Turland, Computer Science for the IB Diploma, printed pp. 358-377. Page references are printed page numbers. Original classroom materials; not an official IB publication.

Review answers

Use these after attempting the questions. The suggested marks are for this original practice, not official IB marking instructions.

R1 [6]

(a) $O(n)$ [1]; the scans add to $2n$ operations rather than multiplying because neither is inside the other [1].
(b) $O(n)$ auxiliary space [1]; the additional list grows with the input [1]. (c) Constants, implementation or fixed overhead can dominate on small inputs [1]; Big O describes growth as n becomes large rather than an exact time at $n = 4$ [1].

R2 [6]

low	high	mid	values[mid]	Action
0	6	3	17	high = 2
0	2	1	8	low = 2
2	2	2	12	high = 1; exit

(a) One mark per correct trace row and one for return -1 [4]. (b) `upper = len(numbers) - 1` [1]; with an inclusive loop this avoids treating the length as a valid index and accessing beyond the list [1].

R3 [6]

(a) Pass 1: [3, 5, 2, 6]; pass 2: [3, 2, 5, 6]; pass 3: [2, 3, 5, 6]. One mark per complete pass [3]. (b) [2, 3, 5, 6] [1]. (c) Each pass still scans the unsorted region for its minimum [1]; the comparison counts add to $(n - 1) + \dots + 1 = n(n - 1)/2$, so time is $O(n^2)$ [1].

R4 [6]

(a) Calls: 4, 3, 2, 1, 0 [1]; base return 0 [1]; later returns 1, 3, 6, 10 [1]. (b) The argument never decreases towards 0 [1]; calls continue without reaching the base case, eventually exceeding Python's recursion limit [1]. (c) $O(n)$ auxiliary space for the pending call stack [1].

Turn a mistake into a specific next step

Record the question, the idea you missed and one corrected explanation or trace. Then answer a similar question with different data. A raw score is useful, but the corrected reasoning shows what you learned.