

B2 4 Algorithms student handout

IB DP Computer Science HL • Python • Name _____ Date _____

This unit develops a way to judge algorithms as data grows, then applies it to searching and sorting. You will explain efficiency, trace algorithms, write Python and justify an algorithm choice for a particular task.

Section	Focus
B2.4.1	Big O notation, time and space complexity, scalability and efficiency
B2.4.2	Linear search and binary search
B2.4.3	Bubble sort and selection sort

B2 4 1 Understanding growth

The input size n is the amount of data an algorithm processes: here, usually the number of list items. Time complexity describes how the work grows with n . We count useful operations rather than seconds, because timings depend on hardware and implementation.

Big O describes an upper bound on growth for sufficiently large inputs. In this handout, give the tightest common Big O class that describes the stated case. It is not an exact operation count and does not automatically mean worst case: identify the case separately.

Class	Name	Typical pattern
$O(1)$	Constant	A fixed number of steps, such as reading one list index
$O(\log n)$	Logarithmic	Repeatedly halve a candidate range
$O(n)$	Linear	Visit each item once
$O(n \log n)$	Linearithmic	Common in efficient general-purpose sorting
$O(n^2)$	Quadratic	Compare every item with every other item
$O(2^n)$	Exponential	Explore every subset of n choices

Simplify growth without losing the meaning

Ignore constant multipliers and smaller growing terms when classifying large-input growth. For example, $3n + 8$ is $O(n)$, and $2n^2 + 5n + 1$ is $O(n^2)$. Two separate full scans are still $O(n)$; two nested full scans are $O(n^2)$.

Best case is the least work for inputs of size n . Worst case is the most. Average case requires assumptions about how inputs are distributed. We mainly compare best and worst cases in this unit.

Growth labels help predict scalability: whether the algorithm remains practical as n increases. They do not tell us which implementation wins on every small input.

B2 4 1 Space and practical choices

Space complexity

Space complexity describes how memory use grows with n . Total space includes the input and other stored data. Auxiliary space means extra working memory beyond the input. State which measure you mean. Our algorithm comparison tables report auxiliary space.

```
# A: fixed extra storage
largest = values[0]      # assume a nonempty list
for value in values:
    if value > largest:
        largest = value
```

```
# B: storage that grows with the input
copied = []
for value in values:
    copied.append(value)
```

A uses $O(1)$ auxiliary space: it stores only a fixed number of variables. B uses $O(n)$ auxiliary space to hold another list. Both take $O(n)$ time. Both have $O(n)$ total space when the original list is included. These are standard classroom models in which an item occupies fixed-size storage.

An in-place algorithm rearranges the original list using little extra storage. The iterative searches and sorts in this handout use $O(1)$ auxiliary space. Making a copy before sorting adds $O(n)$ storage; recursive calls also use stack space.

Choosing for scalability and efficiency

- Check correctness and preconditions first: is the data sorted by the key being searched?
- Consider data size, expected growth, and how many times the task will run.
- Compare time and memory; include sorting, copying and maintaining order in the whole task.
- Consider input patterns: already sorted, reverse sorted, duplicates, or many updates.
- For close choices, test realistic inputs. Constants and implementation details still matter.

Growth models

These are values of growth functions, not exact timings or exact counts for every algorithm. $\log_2 n$ is the number of halvings needed to reduce n to 1.

n	1	$\log_2 n$	n	$n \log_2 n$	n^2
8	1	3	8	24	64
64	1	6	64	384	4096
1024	1	10	1024	10240	1048576

A binary search on 1024 items can still inspect 11 middle items: after ten reductions, one candidate may remain to be checked. The $O(\log n)$ growth description ignores this additive detail.

B2 4 1 Practice with complexity

Problem 1 Classify and explain

Give the time complexity of A–D and the auxiliary space of each. Assume $n = \text{len}(\text{values})$, ordinary constant-time list indexing, and constant-time print operations for this exercise.

```
# A – assume  $n > 0$   
print(values[0])
```

```
# B  
for x in values:  
    print(x)
```

```
# C  
for x in values:  
    for y in values:  
        print(x, y)
```

```
# D  
k = len(values)  
while k > 1:  
    k = k // 2
```

Snippet	Time	Auxiliary space	Reason

e. A program scans all n items, then separately scans all n items again. Explain why its time complexity is $O(n)$, not $O(n^2)$.

B2 4 1 Applying complexity

Problem 2 Predict growth and memory

a. When n grows from 100 to 1000, approximately how do linear and quadratic work change? Use growth rates, not measured seconds.

b. A function copies all n items into a new list, then runs an in-place sort on the copy. What auxiliary space does the whole function use, and why?

c. A is $O(n)$ and B is $O(n^2)$. Must A be faster for $n = 5$? Explain.

Problem 3 Choose with reasons

You have one short unsorted list to search once. A colleague proposes sorting it before binary search. Explain your preferred approach. Then explain what changes if a large list is already sorted and will be searched thousands of times.

B2 4 2 Linear and binary search

Searching finds a target value. Our functions return a matching zero-based index, or -1 when the target is absent. Linear search works on any order; this binary search requires ascending order by the same key being searched.

Linear search recap

Inspect items from the start until a match is found or the list ends. In [18, 5, 27, 9], target 27 takes three item checks and returns index 2.

```
FUNCTION linearSearch(A, target)
  FOR index FROM 0 TO LENGTH(A) - 1 INCLUSIVE
    IF A[index] = target THEN
      RETURN index
    END IF
  END FOR
  RETURN -1
END FUNCTION
```

Binary search step by step

1. Set low = 0 and high = the last index. Both bounds are included.
2. While $\text{low} \leq \text{high}$, choose $\text{mid} = (\text{low} + \text{high}) \text{ DIV } 2$.
3. If $A[\text{mid}]$ equals the target, return mid.
4. If $A[\text{mid}]$ is smaller, set $\text{low} = \text{mid} + 1$. Otherwise set $\text{high} = \text{mid} - 1$.
5. Repeat. When $\text{low} > \text{high}$, no candidates remain; return -1 .

Sorted order makes discarding safe. If the middle value is too small, everything to its left is also too small. The failed middle item is excluded, so the range shrinks. Equal values are allowed.

```
FUNCTION binarySearch(A, target)
  low ← 0
  high ← LENGTH(A) - 1
  WHILE low <= high
    mid ← (low + high) DIV 2
    IF A[mid] = target THEN
      RETURN mid
    ELSE IF A[mid] < target THEN
      low ← mid + 1
    ELSE
      high ← mid - 1
    END IF
  END WHILE
  RETURN -1
END FUNCTION
```

Classroom pseudocode convention: indices start at 0; DIV is integer division; FOR bounds shown as INCLUSIVE include both endpoints. A FOR loop with no indices to visit runs zero times.

B2 4 2 Worked trace and practice

Use A = [3, 8, 12, 17, 23, 31, 38, 44, 56, 62, 75].

Index	0	1	2	3	4	5	6	7	8	9	10
Value	3	8	12	17	23	31	38	44	56	62	75

Worked example: find 56. Record bounds at the start of each iteration; record updates in the final column. Count one inspected middle item as one check.

Step	low	high	mid	A[mid]	Decision and next bounds
1	0	10	5	31	Go right; low = 6, high = 10
2	6	10	8	56	Found; return 8

Problem 4 Is binary search applicable

Can the ascending-order algorithm be used directly? Justify each answer: a. [2, 6, 14, 20]; b. [20, 2, 14, 6]; c. student IDs in records sorted by surname; d. [20, 14, 6, 2].

Problem 5 Trace two searches

a. Search A for 23. Complete the trace, then give the returned index and number of checks.

Step	low	high	mid	A[mid]	Decision and next bounds

Returned index _____ Checks _____

b. Search A for 24. Complete the trace and state the final bounds, loop condition and return value.

Step	low	high	mid	A[mid]	Decision and next bounds

B2 4 2 Python and debugging

Problem 6 Complete and test

Use this linear search as a reference. Complete the binary search blanks in your editor. Predict and then run the tests below with both functions.

```
def linear_search(values, target):
    for index in range(len(values)):
        if values[index] == target:
            return index
    return -1

def binary_search(values, target):
    low = 0
    high = _____ # A
    while _____: # B
        mid = _____ # C
        if values[mid] == target:
            return _____ # D
        elif values[mid] < target:
            low = _____ # E
        else:
            high = _____ # F
    return _____ # G
```

values	target	Linear prediction	Binary prediction
[3, 8, 12, 17, 23]	3		
[3, 8, 12, 17, 23]	23		
[3, 8, 12, 17, 23]	10		
[]	7		
[7]	7		
[7]	9		
[2, 4, 4, 4, 9]	4		

Explain why the duplicate test can return different indices. Check a result against -1 before using it as an index: in Python, `values[-1]` means the last item.

Problem 7 Diagnose boundary errors

A broken version changes three lines: `while low < high`; `low = mid`; `high = mid`. All other lines are unchanged. Trace on paper before running it.

a. What happens for `[7]`, target `7`? b. For `[3, 8]`, target `8`, give the first two (low, high, mid) triples. c. Correct all three lines and explain why the range must shrink.

B2 4 3 Bubble sort

Sorting puts items in an order, here ascending. Bubble sort compares adjacent values and swaps them if they are out of order. A pass scans the remaining unsorted part from left to right; its largest value reaches the right-hand end.

After each pass, the sorted suffix grows by one. Later passes do not need to inspect it. Our optimized version also stops if a complete pass makes no swaps.

Worked example

Start [5, 1, 4, 2]. Pass 1: compare 5 and 1 → [1, 5, 4, 2]; compare 5 and 4 → [1, 4, 5, 2]; compare 5 and 2 → [1, 4, 2, 5]. The final 5 is now fixed.

Pass	List after pass	Comparisons	Swaps	Outcome
1	[1, 4, 2, 5]	3	3	Continue
2	[1, 2, 4, 5]	2	1	Continue
3	[1, 2, 4, 5]	1	0	Stop

Pass 3 has no swaps, so the early-stop condition is met. A list may become sorted before the algorithm has performed a pass that confirms it.

Pseudocode

```

PROCEDURE bubbleSort(A)
  FOR end FROM LENGTH(A) - 1 DOWN TO 1 INCLUSIVE
    swapped ← FALSE
    FOR j FROM 0 TO end - 1 INCLUSIVE
      IF A[j] > A[j + 1] THEN
        SWAP A[j], A[j + 1]
        swapped ← TRUE
      END IF
    END FOR
    IF swapped = FALSE THEN
      RETURN
    END IF
  END FOR
END PROCEDURE

```

Efficiency

Best case: already sorted, so the first pass makes $n - 1$ comparisons and no swaps, then stops: $O(n)$ time. Worst case: reverse sorted, requiring roughly n^2 comparisons and swaps: $O(n^2)$ time. Auxiliary space is $O(1)$.

Without early stopping, this loop structure still performs all passes on sorted data and has $O(n^2)$ best-case time. Always connect a complexity claim to the implementation.

B2 4 3 Selection sort

Selection sort finds the smallest item in the unsorted part, then swaps it into the first position of that part. After each pass, the sorted prefix grows by one. The rest is still unsorted.

Worked example

Start [5, 1, 4, 2]. In pass 1, the smallest value is 1 at index 1. Swap it with index 0 to give [1, 5, 4, 2]. In each later pass, scan the whole remaining unsorted part before swapping.

Pass	Minimum index	List after pass	Comparisons	Swaps
1	1	[1, 5, 4, 2]	3	1
2	3	[1, 2, 4, 5]	2	1
3	2	[1, 2, 4, 5]	1	0

Minimum index refers to the index before that pass's swap. A self-swap is skipped, so pass 3 uses zero swaps. It still compares the remaining values.

Pseudocode

```

PROCEDURE selectionSort(A)
  FOR start FROM 0 TO LENGTH(A) - 2 INCLUSIVE
    minIndex ← start
    FOR j FROM start + 1 TO LENGTH(A) - 1 INCLUSIVE
      IF A[j] < A[minIndex] THEN
        minIndex ← j
      END IF
    END FOR
    IF minIndex != start THEN
      SWAP A[start], A[minIndex]
    END IF
  END FOR
END PROCEDURE

```

Efficiency and differences

Selection sort scans for a minimum even if the data is already sorted. Both best- and worst-case time are $O(n^2)$. It uses $O(1)$ auxiliary space and at most $n - 1$ actual swaps with this version.

Both sorts make at most $(n - 1) + (n - 2) + \dots + 1$ comparisons. Bubble sort may swap after every comparison; selection sort swaps at most once per pass. Few swaps can matter when moving records is expensive, but scanning time still grows quadratically.

In Python, range stops before its end value. Translate inclusive pseudocode endpoints carefully. A pass is a whole scan; a comparison is one test between list values; a swap exchanges two values.

B2 4 3 Python sorting tasks

Reference implementation of bubble sort

```
def bubble_sort(values):
    for end in range(len(values) - 1, 0, -1):
        swapped = False
        for j in range(end):
            if values[j] > values[j + 1]:
                temp = values[j]
                values[j] = values[j + 1]
                values[j + 1] = temp
                swapped = True
        if not swapped:
            break
    return values
```

Problem 8 Complete selection sort

Complete A–D using the pseudocode. Both functions change the supplied list in place and return the same list for convenience.

```
def selection_sort(values):
    for start in range(len(values) - 1):
        min_index = _____ # A
        for j in range(_____): # B
            if _____: # C
                min_index = _____ # D
        if min_index != start:
            temp = values[start]
            values[start] = values[min_index]
            values[min_index] = temp
    return values
```

For both sorts, predict and check the output for [], [7], [1, 2, 3, 4], [4, 3, 2, 1], and [3, 1, 3, 2]. Verify that no values are lost or added. Record results in your editor.

To compare fairly, give each algorithm a fresh copy of the original list. Otherwise the second algorithm receives an already sorted list. The copy uses $O(n)$ extra memory in the test program.

Problem 9 Debug the sorting code

a. In `bubble_sort`, `swapped = False` is moved outside the outer loop. Explain what may be lost after a pass that swaps. b. In `selection_sort`, `min_index = start` is replaced with `min_index = 0`. Trace [1, 2, 3] and explain the error.

B2 4 3 Trace and compare the sorts

Problem 10 Trace both algorithms

Start a fresh copy of [4, 2, 3, 1] for each algorithm. Record every executed pass, including a no-swap pass if one occurs. Count only comparisons between list values, not loop conditions. A self-swap is not counted. Leave unused rows blank.

Bubble pass	List after pass		Comparisons	Swaps	Stop or continue

Selection pass	Minimum index	List after pass	Comparisons	Swaps

Give the total comparisons and swaps for each algorithm. Which made fewer swaps on this input?

Problem 11 Explain what the counts mean

a. For [1, 2, 3, 4], give the total comparisons and swaps for each supplied sort. Why are the best-case time complexities different?

b. A student says “Selection sort is $O(n)$ because it makes at most $n - 1$ swaps.” Explain the missing work.

c. After bubble sort’s first pass, which end contains an item in its final position? After selection sort’s first pass, which end does?

Optional extension

Add comparison and swap counters to both functions. Check your Problem 10 totals. Try larger sorted and reverse-sorted lists. Explain why comparison counts are more informative about growth than a single timing on a tiny input.

B2 4 Review and algorithm choice

Algorithm	Best time	Worst time	Auxiliary space	Important condition
Linear search	$O(1)$	$O(n)$	$O(1)$	Any input order
Binary search	$O(1)$	$O(\log n)$	$O(1)$	Sorted; direct access by index
Bubble sort with early stop	$O(n)$	$O(n^2)$	$O(1)$	Adjacent swaps; stops after a no-swap pass
Selection sort	$O(n^2)$	$O(n^2)$	$O(1)$	Scans full unsorted part; at most one swap per pass

These bounds describe the supplied iterative implementations. Best-case search bounds assume a nonempty input. Empty lists involve no item checks. For a sorted list of 1024 items, linear search may check 1024 items; binary search checks at most 11.

Problem 12 Justify the choice

For each scenario, choose from the four algorithms above and justify your answer with relevant preconditions, time, memory or data movement. Answer on paper or in your editor.

a. Locate one target in a short unsorted list that changes frequently.

b. Make many searches of a large, already sorted list of student IDs.

c. Sort an already sorted list, choosing only between the supplied sorts.

d. Sort a small unsorted list of records where swaps are much more costly than value comparisons.

e. Someone wants to sort millions of records routinely using one of these two sorts. Explain the scalability concern and what kind of growth rate you would investigate instead.

Exit check

In your own words: distinguish time from space complexity; explain why binary search needs sorted data; and explain why two $O(n^2)$ sorting algorithms can still behave differently.

Success check: I can trace and implement all four algorithms, state the case and space convention when giving complexity, and justify a choice using the needs of the task.