Q1.

Iterate over sorted lists A and B using two pointers i and j. Compare A[i] and B[j]:

- If smaller, move the corresponding pointer.
- If equal, add to the result and move both pointers.
- Repeat until one list is depleted.

Pseudo Code:

```
FUNCTION find_common_elements(A, B):
    i ← 0
    j ← 0
    result ← EMPTY LIST
    WHILE i < SIZE(A) AND j < SIZE(B) DO:
        IF A[i] IS LESS THAN B[j] THEN:
            INCREMENT i
        ELSE IF A[i] IS GREATER THAN B[j] THEN:
            INCREMENT j
        ELSE:
            ADD A[i] TO result
            INCREMENT i
            INCREMENT j
    RETURN result
# Example usage:
A ← [2, 5, 5, 5]
B ← [2, 2, 3, 5, 5, 7]
C ← [1, 3, 4, 6, 7, 9, 10]
D ← [2, 3, 5, 6, 8, 9, 11]

DISPLAY find_common_elements(A, B)
DISPLAY find_common_elements(C, D)
```

Expected output should be:

[2, 5, 5]

[3, 6, 9]

Time Complexity: O(m + n) where m and n are the lengths of the arrays A and B respectively.

Space Complexity: O(1) (excluding the output list).

For a two-pointer method, the algorithm stops when i = m or j = n, i.e., one list is fully processed. In the worst case, each element of A and B is compared at most once before reaching the end of i or j. Since each operation moves at least one pointer forward, the number of operations is at most m + n.

Therefore, the algorithm performs a maximum of m+n-1 comparisons

Q2.

Initialize n lockers as closed (False).
For each student 1 to n, toggle every i-th locker.
Track open (True) and closed (False) lockers in separate lists.

Pseudo Code:

```
FUNCTION lockers(n):
    # Begin with alle lockers as closed
    lockers ← LIST of SIZE n with all VALUES set to FALSE

    FOR student FROM 1 TO n DO:
        FOR locker FROM (student - 1) TO (n - 1) STEP student DO:
            lockers[locker] ← NOT lockers[locker]  # Toggle locker state

    # Collect open and closed lockers
    open_lockers ← EMPTY LIST
    closed_lockers ← EMPTY LIST

    FOR index FROM 0 TO (n - 1) DO:
        IF lockers[index] IS TRUE THEN:
            APPEND (index + 1) TO open_lockers
        ELSE:
            APPEND (index + 1) TO closed_lockers
```

```
    RETURN open_lockers, closed_lockers

IF __name__ == "__main__":
    PRINT "Enter the number of lockers:"
    n ← INPUT AS INTEGER

    open_lockers, closed_lockers ← lockers(n)

    PRINT "Open lockers:", open_lockers
    PRINT "Closed lockers:", closed_lockers
    PRINT "Number of opened lockers:", LENGTH(open_lockers)
    PRINT "Number of closed lockers:", LENGTH(closed_lockers)
```

Output for sample size n=100 should be :

open lockers: [1, 4, 9]
Locked lockers: [2, 3, 5, 6, 7, 8, 10]
Total unlocked lockers: 3
Total locked lockers: 7

On noticing, the **lockers that remain open** are all perfect squares. This is because each locker is toggled for every divisor it has and a number has an odd number of divisors only if it is a perfect square.

All lockers that are not perfect squares remain **closed**. This is because they have an even number of divisors, meaning they were toggled an even number of times thus ending up closed.

The **number of open lockers** is equal to the number of perfect squares that are less than or equal to n. That is, it can be $\lfloor \sqrt{n} \rfloor$ (the largest integer less than or equal to the square root of n, as there are the perfect square numbers of lockers.

Example of 3 passes with n=10:

Initial State would be where all are closed.

Pass 1 (Student 1 visits every locker)
Pass 1: [OPENED, OPENED, OPENED, OPENED, OPENED, OPENED, OPENED, OPENED, OPENED, OPENED]

Pass 2 (Student 2 visits every 2nd locker)
Pass 2: [OPENEDE, close, OPENED, close, OPENED, close, OPENED, close, OPENEDE, close]

Pass 3 (Student 3 visits every 3rd locker)
Pass 3: [opened, close, close, close, opened, opened, opened, close, close, close]

Q3.

Start with the first element and iterate through the array.

If A[i] == x, then return i (where x resides).

If the iteration finishes without locating x, return NIL.

Pseudo Code:

```
FUNCTION search_linear(A, size, target):
    FOR index FROM 1 TO size DO:
        IF A[index] IS EQUAL TO target THEN:
            RETURN index  # Target found at this position
    RETURN NONE  # Target not present in A
```

Time Complexity analysis:

Best Case: **O(1)** if the element is found at the first position.
Worst Case: **O(n)** if the element is located at the final position.

Loop Invariant & Proof

To prove the validity of linear search algorithm, we use loop invariant it is something that holds prior to and subsequent to each iteration of the loop.

Loop Invariant: Before to each iteration, x is not in subarray A[1.i-1].

Initialization: At the start iteration (i = 1), nothing has been checked, and the invariant is holding.

Maintenance: At i-th iteration, we are now going to be checking A[i:]

If A[i] == x, the algorithm terminates upon returning i.

If A[i] ≠ x, then the invariant remains because x is still not in A[1.i] and we move on to the next step.

Termination: After n iterations, all elements have been tried. If x was encountered, then the algorithm reports its location. Else, it returns NIL to ensure x is not an element of A.

Since the invariant is only true at the beginning, preserved along each step, and enables effective termination, the linear search algorithm works and hence correctness is proved.

Q4.

(a) Strategy for brute force:

- Iterate through all potential substrings in the given string.
- Ensure each substring begins with 'A' and concludes with 'B'.
- Keep track of the count of valid substrings.

Pseudo Code developed:

```
FUNCTION find_substrings_naive(text):
    total ← 0
```

```
length ← SIZE(text)

FOR i FROM 0 TO length - 1 DO:
  IF text[i] IS 'A' THEN:
    FOR j FROM i + 1 TO length - 1 DO:
      IF text[j] IS 'B' THEN:
        total ← total + 1
RETURN total
```

Time Complexity:

- The outer loop is O(n), and the inner loop is up to O(n), so the time complexity is $O(n^2)$.
- We can tell its bad for large inputs.

(b) Strategy for Optimized solution:

Instead of checking all the substrings, we maintain counts of 'A' as we move and add to the count whenever we encounter 'B'.

Pseudo Code:

```
FUNCTION optimized_substring_count(text):
  total_count ← 0
  a_occurrences ← 0  # Tracks count of 'A' seen so far

  FOR character IN text DO:
    IF character IS 'A' THEN:
      a_occurrences ← a_occurrences + 1
    ELSE IF character IS 'B' THEN:
      total_count ← total_count + a_occurrences  # Add previous 'A' occurrences

  RETURN total_count
```

Time Complexity: O(n) (a single pass through the strings).

Space Complexity: O(1) (where only a few counters used).

Q5.

Individual Submission by Janya (CWID: 878062934)

Using the logarithmic formula $\log_2(n) = \log(n) / \log(2)$

Using values:

- $\log_2(8) = \log_2(2^3) = \mathbf{3}$
- $\log_2(128) = \log_2(2^7) = \mathbf{7}$
- $\log_2(1024) = \log_2(2^{10}) = \mathbf{10}$

Formula: $n^2 = n \times n$

- $8^2 = 8 \times 8 = \mathbf{64}$
- $128^2 = 128 \times 128 = \mathbf{16,384}$
- $1024^2 = 1024 \times 1024 = \mathbf{1,048,576}$

Formula**:** $n^3 = n \times n \times n$

- $8^3 = 8 \times 8 \times 8 = \mathbf{512}$
- $128^3 = 128 \times 128 \times 128 = \mathbf{2,097,152}$
- $1024^3 = 1024 \times 1024 \times 1024 = \mathbf{1,073,741,824}$

Formula**:** $2^n = 2 \times 2 \times ...$ (n times)

- $2^8 = \mathbf{256}$
- $2^{128} \approx \mathbf{3.4 \times 10^{38}}$
- $2^{1024} \approx \mathbf{1.8 \times 10^{308}}$ (Extremely large)

Factorial grows very fast, defined as:
$n! = n \times (n-1) \times (n-2) \times ... \times 1$

- $8! = 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 40,320$
- $128! = 128 \times 127 \times 126 \times ... \times 1$
- $1024! = 1024 \times 1023 \times 1022 \times ... \times 1$

n! grows faster than exponential, becoming too large to compute beyond `n = 20`.

Final table

Complexity Function	n = 8	n = 128	n = 1024
$\log_2(n)$	3	7	10
n	8	128	1024
n^2	64	16,384	1,048,576
n^3	512	2,097,152	1,073,741,824
2^n	256	3.4×10^{38}	1.8×10^{308}
n!	40,320	$128 \times 127 \times ... \times 1$	$1024 \times 1023 \times ... \times 1$