

Laboratory Session 02

Asymptotic complexity and recurrences

1 Longest increasing run (iterative)

Given a list A of integers, return the length of its **longest contiguous, strictly increasing** run. A run consists of consecutive positions; elements cannot be skipped. **Equal consecutive values break a run.** Return 0 for an empty list. Use one left-to-right scan and a constant number of variables.

State the corresponding asymptotic time complexity at the end of the procedure.

Examples

Input	Expected Output
<code>longest_increasing_run([3, 5, 7, 2, 4, 6, 8, 1])</code>	4
<code>longest_increasing_run([1, 2, 2, 3])</code>	2
<code>longest_increasing_run([9, 7, 5])</code>	1
<code>longest_increasing_run([])</code>	0

In the first example, the longest run is $[2, 4, 6, 8]$. In the second, $[1, 2]$ and $[2, 3]$ are separate runs.

2 Count equal pairs (iterative)

Given a list A of integers, return the **number of pairs of distinct positions that contain equal values**. Count each pair only once, considering only positions (i, j) with $i < j$. Return 0 if the list contains fewer than two elements.

State the corresponding asymptotic time complexity at the end of the procedure.

Examples

Input	Expected Output
<code>count_equal_pairs([4, 1, 4, 4, 1])</code>	4
<code>count_equal_pairs([7, 7, 7])</code>	3
<code>count_equal_pairs([1, 2, 3])</code>	0
<code>count_equal_pairs([5])</code>	0
<code>count_equal_pairs([])</code>	0

In the first example, using zero-based indices:

- The value 4 occurs at positions 0, 2, and 3, forming the pairs $(0, 2)$, $(0, 3)$, and $(2, 3)$.
- The value 1 occurs at positions 1 and 4, forming the pair $(1, 4)$.

Therefore, the total number of matching pairs is 4.

In the second example, using zero-based indices, three occurrences of 7 form three pairs $(0, 1)$, $(0, 2)$ and $(1, 2)$.

3 Find the first one (recursive)

Given a list A containing only 0s and 1s in **nondecreasing order** (all 0s come before all 1s), write a recursive function that **returns the position of the first 1**. Return -1 if the list is empty or contains only 0s. Use a recursive helper operating on an index interval. Do not use loops, list slicing, `index`, or a library search function.

State the corresponding asymptotic time complexity at the end of the procedure.

Examples

Input	Expected Output
<code>first_one([0, 0, 0, 1, 1, 1])</code>	3
<code>first_one([1, 1, 1])</code>	0
<code>first_one([0, 0, 0])</code>	-1
<code>first_one([0, 1])</code>	1
<code>first_one([])</code>	-1

In the first example, the 1s occur at positions 3, 4, and 5. The function returns 3, the position of the first 1.

4 Prefix sums (recursive)

Given a list A of integers, return a new list P in which **each element is the sum of all elements of A up to that position**:

$$P[i] = A[0] + A[1] + \cdots + A[i].$$

You may use list slicing and concatenation in this exercise. **Do not modify** A . If the list is empty or contains one element, return a copy. Include their costs in your analysis.

Examples

Input	Expected Output
<code>prefix_sums([3, 1, 4, 2])</code>	<code>[3, 4, 8, 10]</code>
<code>prefix_sums([2, -3, 5])</code>	<code>[2, -1, 4]</code>
<code>prefix_sums([7])</code>	<code>[7]</code>
<code>prefix_sums([])</code>	<code>[]</code>

In the first example if $A = [3, 1, 4, 2]$, then:

$$P = [3, 3 + 1, 3 + 1 + 4, 3 + 1 + 4 + 2] = [3, 4, 8, 10].$$

Hint: Split the list into two approximately equal halves. For $[3, 1, 4, 2]$, the two halves are $[3, 1]$ and $[4, 2]$.

5 Instructions

How to Annotate Your Code

To report the asymptotic computational complexity, place a Python comment containing a $\text{\LaTeX}$ formula at the end of each function, after its final return statement. Keep the comment indented inside the function. For example, if your analysis establishes a linear upper bound, write:

```
def example(A):
    # ... your implementation ...
    return result
    # Time complexity (worst case):  $O(n)$ 
```

The dollar signs and backslash commands are literal text in the Python comment. For other bounds, use valid $\text{\LaTeX}$ such as $O(n^2)$ or $O(\log n)$. Include the tight bound and your derivation in the comments immediately before the final complexity comment. Annotate any helper functions too, specifying their input-size parameter.

How to Complete This Lab

In this lab, you are provided with this PDF containing the exercise instructions, the necessary libraries (if there are libraries to use) and expected Input/Output examples (e.g., Input: `power(2, 3)` → Output: `8`).

Along with this document, you have received two Python files:

- `exercises.py`: This file contains the function signatures with empty bodies for all exercises. For example:

```
def power(base: int, exponent: int) -> int: pass
```

Your task is to replace the `pass` keyword with your actual implementation. Do not change the function names or their signatures. You can add other functions to support your code.

- `main.py`: This file contains a suite of automated tests. Once you have written your code in `exercises.py`, run `main.py`. It will automatically import your functions, test them against various inputs, and report if your outputs are correct (True for correct result, False otherwise).