

Laboratory Session 03

Sorting

1 Compression Sort

Write a function `compression_sort(A)` that takes a list A of integers and returns it **modified in-place** containing the elements sorted in non-decreasing order, **with all duplicates removed**.

You must implement a modified version of the Insertion Sort algorithm. Do not modify the original list A and do not use library functions like `set()` or `sort()`. State the corresponding asymptotic time complexity at the end of the procedure.

Examples

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

2 Descending Bubble Sort

Write a function `descending_bubble_sort(A)` that implements the Bubble Sort algorithm to sort a list A of integers in **descending order** (from largest to smallest).

You must include an **early stopping mechanism**: keep track of the swaps made during each pass. If a complete pass is made without any swaps, the algorithm must terminate immediately without executing the remaining passes. Modify the list *in-place*. State the corresponding asymptotic time complexity at the end of the procedure.

Examples

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

3 3-Way Merge Sort

Write a recursive function `three_way_merge_sort(A)` that sorts a list A of integers in non-decreasing order using a modified Merge Sort algorithm.

Instead of dividing the list into two halves, **divide the list into three approximately equal parts**. Recursively sort each part, and then merge the three sorted parts into a single sorted list. You may return a new list or modify it *in-place*. State the corresponding asymptotic time complexity at the end of the procedure.

Examples

Input	Expected Output
<code>three_way_merge_sort([3, 1, 4, 1, 5, 9, 2, 6, 5])</code>	<code>[1, 1, 2, 3, 4, 5, 5, 6, 9]</code>
<code>three_way_merge_sort([10, 9, 8, 7, 6, 5])</code>	<code>[5, 6, 7, 8, 9, 10]</code>
<code>three_way_merge_sort([1, 2])</code>	<code>[1, 2]</code>
<code>three_way_merge_sort([])</code>	<code>[]</code>

4 Mean-Pivot QuickSort

Write a recursive function `mean_pivot_quicksort(A)` that sorts a list A of integers using a modified QuickSort algorithm.

Instead of choosing the first, last, or a random element as the pivot, **calculate the arithmetic mean** of the elements in the current sub-array. Then, choose the element in the sub-array whose value is **closest to this mean** as the pivot. Partition the sub-array around this pivot and sort recursively. Modify the list *in-place*. State the corresponding asymptotic time complexity at the end of the procedure.

Examples

Input	Expected Output
<code>mean_pivot_quicksort([3, 1, 4, 1, 5, 9])</code>	<code>[1, 1, 3, 4, 5, 9]</code>
<code>mean_pivot_quicksort([10, 9, 8, 7, 6])</code>	<code>[6, 7, 8, 9, 10]</code>
<code>mean_pivot_quicksort([1, 2])</code>	<code>[1, 2]</code>
<code>mean_pivot_quicksort([])</code>	<code>[]</code>

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)` $\rightarrow$ 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).