Q3: Sorting Design Report

1. The package was designed using the **Strategy Pattern** to ensure the code is modular, extensible, and easy to maintain. This pattern allows us to define a family of algorithms (sorting), encapsulate each one, and make them interchangeable.

1.1 Strategy Pattern Components

- **The Interface (Strategy):** src/algorithms/base_sort.py
  - The BaseSort class is an Abstract Base Class (ABC).
  - It defines a common interface, sort(), that all sorting algorithms *must* implement. This guarantees that any new algorithm we add will be compatible with the main system.

- **The Concrete Strategies:** src/algorithms/*.py
  - The classes BubbleSort, SelectionSort, QuickSort, and MergeSort are the concrete strategies.
  - Each class inherits from BaseSort and provides its own specific implementation of the sorting logic.

- **The Context:** src/sorter.py
  - The Sorter class is the ”Context.” It is the single point of entry for the user.
  - It holds a dictionary that maps algorithm names (like "bubble") to an instance of the corresponding strategy class.
  - When the user calls Sorter.sort(), it performs validation and then *delegates* the sorting task to the chosen strategy object.

This design fulfils Requirement 1 (base class) and 2 (implementations) and makes Requirement 4 (class to call algorithms) clean and efficient.

2. Test Case Design

The test suite in test/test_sorting.py was designed to be robust and cover all stated requirements. The pytest framework was used, specifically its parametrize feature, to efficiently test all algorithms against all test cases.

2.1 Reaching the Test Cases

The core test cases (TEST_CASES) were developed by considering all common and edge-case scenarios for a sorting algorithm:

- **Empty List (id="empty"):** The most basic edge case. The algorithm should not crash and should return an empty list.

- **Single Element (id="single"):** Another simple edge case. Should return the same list.
- **Pre-sortedList(id="pre-sorted"):** Tests if the algorithm can efficiently handle data that is already in order.
- **Reversed List (id="reversed"):** This is often the worst-case scenario for algorithms like Bubble Sort or some Quick Sort pivots.
- **Duplicates (id="duplicates"):** Ensures the algorithm correctly handles duplicate values and doesn't drop any.
- **Negatives (id="negatives"):** Confirms the algorithm works with negative numbers.
- **Mixed List (id="mixed_with_zero"):** A standard, realistic case with positive, negative, and zero values.

2.2 Testing All Requirements

Specific test functions were created to map directly to the project's requirements:

1. **test_all_algorithms():** This function uses pytest.mark.parametrize to run *every* test case against *every* algorithm. It also explicitly tests both **ascending** (Requirement 6a) and **descending** order.
2. **test_large_random_list():** This test addresses Requirement 5 (large list) and 8 (correctness). It generates a large list of random INT32-range numbers, sorts it, and compares the result to Python's built-in sorted() function to guarantee accuracy.
3. **test_validation_errors():** This function ensures all constraints are met. It uses pytest.raises to confirm that the Sorter class correctly throws errors for:
    - An unknown algorithm name (Req 6b).
    - A list containing non-integers (Req 7).
    - A mismatched size_of_list (Req 6c).
    - A number outside the INT32 range (Req 5).
4. **test_original_list_is_not_modified():** This test explicitly verifies Requirement 8 ("Output should be a new list"). It passes a list to the sorter, then asserts that the original list variable was not changed.