SORTING PACKAGE

Overview:

The aim was to design a Python package that implements several sorting algorithms in a clean and scalable manner, following best practices for code modularity and testability. The package includes a parent abstract base class for sorting algorithms, concrete implementations for each algorithm, a factory pattern to select an algorithm based on input parameters, and comprehensive tests to ensure correctness and performance.

1. Abstract Base Class (Interface):

The package begins with an abstract base class, `BaseSort`, which defines the common interface for all sorting algorithms. This class uses Python's `abc` module to enforce that any subclass must implement the `sort` method, which takes the sorting order (ascending or descending), the size of the data, and the data itself.

- **Input validation**: A helper method `_validate_input` is included to ensure that:
    - All elements in the input list are integers.
    - The input list does not exceed the maximum allowed size of 2 * 10^5 elements (a constraint given in the problem).

The `BaseSort` class serves as the foundation for all specific sorting algorithms, ensuring they all conform to the same method signature and input validation.

2. Sorting Algorithms Implementation:

Four different sorting algorithms were implemented using the base class as a template: Bubble Sort, Selection Sort, Quick Sort, and Merge Sort. Each algorithm is implemented in its own class and inherits from the `BaseSort` class.

- **Bubble Sort**: This is a simple comparison-based sorting algorithm. It iterates over the list multiple times, comparing adjacent elements and swapping them if they are in the wrong order. The process repeats until the entire list is sorted.
- **Selection Sort**: This algorithm divides the list into two parts: the sorted part and the unsorted part. In each iteration, the algorithm selects the smallest (or largest, depending on the sorting order) element from the unsorted part and moves it to the sorted part.
- **Quick Sort**: This divide-and-conquer algorithm chooses a pivot element and partitions the list into two sublists: elements smaller than the pivot and elements greater than the pivot. It then recursively sorts the sublists.
- **Merge Sort**: This is another divide-and-conquer algorithm. The list is recursively split into two halves, which are sorted individually and then merged back together in the correct order.

Each algorithm ensures that the sorting logic respects the `ascending` or `descending` parameter and returns a new sorted list without modifying the original data.

3. Factory Class to Select Algorithm:

A `SortFactory` class was designed to provide a mechanism for selecting and using the appropriate sorting algorithm. The class includes a dictionary (`self.algorithms`) that maps algorithm names (such as `"bubble"`, `"quick"`, etc.) to their respective algorithm class instances.

The `sort` method in the `SortFactory` class:

- Accepts the sorting order, algorithm name, size of the data, and the data itself.
- Validates the chosen algorithm.
- Delegates the actual sorting to the appropriate algorithm's `sort` method.

This factory design allows the user to easily switch between different sorting algorithms by specifying the algorithm name as an argument.

4. Test Design:

The test suite was designed to ensure the correctness of the sorting algorithms under various conditions. The testing framework used is `pytest`, which is highly suited for unit testing and provides an easy way to assert conditions and capture errors.

The key test strategies included:

- **Input Validation**: Tests were created to check that the algorithms raise errors when provided with invalid inputs (e.g., non-integer elements or lists that exceed the allowed size).
- **Correctness**: Test cases verify that the sorting algorithms return the correct sorted list for various input types:
    - Random data.
    - Already sorted data.
    - Reversed data.
    - Nearly sorted data (to check performance on edge cases).
    - Lists with few unique elements or lists with all equal elements.
    - Lists containing negative numbers.

Each test case compares the algorithm's output with Python's built-in `sorted()` function, ensuring the algorithms return the correct result.

- **Performance**: Additionally, the test suite includes performance tracking. The execution time of each sorting algorithm is logged into a report file, allowing for performance comparisons across different algorithms and input sizes.

Test parameters were designed to vary:

- **Algorithm**: Bubble sort, Selection sort, Quick sort, Merge sort.
- **Sorting Order**: Ascending and descending.
- **List Size**: From 0 to 1,000 elements.

- **Data Case**: Different types of data such as random, sorted, reversed, etc.

5. Edge Cases:

The following edge cases were considered in the test design:

- Empty lists (size = 0).
- Lists with only one element (size = 1).
- Large datasets close to the maximum size limit (2 * 10^5).
- Lists with negative numbers to ensure proper handling of sign.
- Lists with all equal values to ensure that the sorting logic does not alter them unnecessarily.

6. Conclusion:

The design of this package effectively addresses the problem requirements. It uses object-oriented principles like inheritance and polymorphism to create a scalable system where new sorting algorithms can easily be added in the future. The `SortFactory` class provides a clean interface for selecting and using the algorithms. The test suite ensures the correctness of the implementation by testing various edge cases and logging performance metrics.

This package serves as a robust solution for comparing the performance and correctness of various sorting algorithms while being easily extendable and maintainable.