Report 1: Peer Analysis Report on Insertion Sort
To: Daniyal
From: Bakdaulet

1. Algorithm Overview
The provided codebase implements the Insertion Sort algorithm. This is a simple, in-place, and stable sorting algorithm that builds the final sorted array one item at a time. It iterates through the elements and, for each element, it "inserts" it into its correct position within the already sorted part of the array. Its primary strength lies in its adaptivity; it is highly efficient for data sets that are already substantially sorted, with a best-case time complexity of $O(n)$.

2. Asymptotic Complexity Analysis (Basic Implementation)
The InsertionSortBasic implementation was analyzed for its time and space complexity.

Time Complexity:
Best Case: $\Theta(n)$ - This occurs when the input array is already sorted. The outer loop runs n-1 times, but the inner while loop condition (arr[j] > key) is immediately false. This results in only n-1 comparisons and no shifts.
Worst Case: $\Theta(n^2)$ - This occurs when the input array is sorted in reverse order. For each element i, the algorithm must compare it with all i-1 elements in the sorted subarray and shift them all one position to the right. The total number of comparisons and shifts is proportional to the sum 1 + 2 + ... + (n-1), which is $n(n-1)/2$, resulting in a quadratic complexity.
Average Case: $\Theta(n^2)$ - For a randomly ordered array, an element at position i will, on average, have to be compared with and shifted past half of the elements in the sorted subarray (i/2). This still results in a quadratic overall complexity.

Space Complexity:
$\Theta(1)$ - The algorithm is performed in-place, requiring only a constant amount of auxiliary space for variables like key and j, regardless of the input size.

3. Code Review and Optimization Opportunity
The InsertionSortBasic code is a clean and correct implementation of the standard algorithm.
The logic is easy to follow, and the use of the PerformanceTracker is correctly integrated.
Identified Performance Bottleneck:
The primary performance bottleneck in the worst and average cases is the method used to find the correct insertion position for the key. The inner while loop performs a linear scan backwards through the sorted subarray:

```
while (j >= 0) {
    tracker.addComparison();
    if (arr[j] > key) {
        // shift element
        j--;
    } else break;
}
```

This linear search contributes directly to the $O(n^2)$ complexity for comparisons.

4. Proposed Optimization: Binary Search Insertion
Suggestion:

Since the subarray arr[0...i-1] is, by definition, always sorted, the linear scan to find the insertion
point can be replaced with a much more efficient Binary Search. A binary search can find the correct
insertion position in O(log i) time instead of O(i) time.
Expected Impact:
Reduction in Comparisons: By replacing the linear search with a binary search, the total number of
comparisons for the entire sort would be reduced from $O(n^2)$ to O(n log n). Overall Time Complexity:
It is crucial to note that while comparisons are optimized, the number of shifts required to make space
for the element remains O(n) in the worst case for each insertion. Therefore, the overall worst-case
time complexity of the algorithm will still be dominated by the shifts and remain $O(n^2)$.
Practical Performance: Despite the same Big-O time complexity, this optimization can yield significant
real-world performance improvements, especially in scenarios where comparison operations are computationally
more expensive than memory writes/shifts.
5. Conclusion
The provided implementation of Insertion Sort is correct and well-written. Its performance aligns with
the established theoretical complexity of the algorithm. A significant optimization is proposed: leveraging binary
search to locate the insertion point, which would drastically reduce the number of comparisons from quadratic to log-linear, improving its practical performance.

Report 2: Simulated Peer Review on Selection Sort
To: Bakdaulet
From: Daniyal

1. Algorithm Overview
The submitted code provides two implementations of the Selection Sort algorithm: a ClassicSelectionSort and
an OptimizedSelectionSort. This algorithm works by repeatedly finding the minimum element from the unsorted
part of the array and placing it at the beginning of the unsorted part. This process is repeated until the entire array is sorted.
2. Asymptotic Complexity Analysis
Time Complexity:
Best, Worst, and Average Case: $\Theta(n^2)$ - Selection Sort's performance is not dependent on the initial order of
the data. The nested loops structure ensures that it always performs the full set of comparisons to find
the minimum element in the remainder of the array. The number of comparisons is always n(n-1)/2, leading
to a consistent quadratic time complexity. The algorithm is not adaptive.
Space Complexity:
$\Theta(1)$ - Both implementations are in-place and use a constant amount of extra memory for variables.
3. Code Review and Analysis of Optimization
Bakdaulet provided two distinct, clean, and correct implementations for comparison.
ClassicSelectionSort: This is a standard, textbook implementation. It correctly finds the minimum element's
index and performs a swap in every pass of the outer loop.
OptimizedSelectionSort: This version introduces a simple yet effective optimization. Before performing the swap,

it checks if the found minimum element is already in its correct position:

```
if (minIndex != i) {
    // perform swap
}
```

Impact of the Optimization:
The core difference between the two versions lies in the number of memory write operations (swaps).
Comparisons: The number of comparisons is identical in both versions ($\Theta(n^2)$). The optimization does not affect this metric.
Swaps:
Classic Version: Performs exactly n-1 swaps, regardless of the input.
Optimized Version: Performs a variable number of swaps, from 0 (for an already sorted array) to a maximum
of n-1 (for a reverse-sorted array where the minimum is never in place).
This optimization is particularly
beneficial for nearly-sorted data, where it can drastically reduce the number of expensive memory write operations.
While it does not change the overall asymptotic time complexity, it improves the algorithm's practical performance by reducing its constant factors.
4. Conclusion
The implementations of Classic and Optimized Selection Sort are well-executed and correct. The provided
optimization effectively minimizes write operations, which is a valuable improvement for certain data
distributions and hardware contexts (e.g., writing to flash memory). The analysis confirms that while
the optimization is beneficial, the algorithm's time complexity remains $\Theta(n^2)$.

Report 3: Joint Comparison Summary
Authors: Bakdaulet & Daniyal

Feature	Insertion Sort	Selection Sort
Time (Best)	$\Theta(n)$	$\Theta(n^2)$
Time (Avg/Worst)	$\Theta(n^2)$	$\Theta(n^2)$
Space	$\Theta(1)$ (in-place)	$\Theta(1)$ (in-place)
Stability	Stable	Unstable
Adaptivity	Yes (fast on nearly-sorted data)	No (performance is independent of initial order)
Comparisons	$O(n^2)$	$\Theta(n^2)$ (always performs the maximum)
Swaps/Writes	$O(n^2)$ shifts	$O(n)$ swaps (very few writes)

Key Differences & Use Cases:
Adaptivity: Insertion Sort is the clear winner for datasets that are known to be small or nearly sorted
due to its O(n) best-case performance. Selection Sort's runtime is fixed regardless of the input's order.
Data Movement: Selection Sort performs a minimal number of swaps (O(n)), making it a preferred choice
when memory write operations are significantly more expensive than reads and comparisons (e.g., with certain types of flash memory).
Insertion Sort, on the other hand, performs $O(n^2)$ data shifts in the worst case, which can be slow.
Conclusion:
While both algorithms have a worst-case complexity of $O(n^2)$, they have distinct performance characteristics.

Insertion Sort is generally preferred for small or nearly-sorted arrays.
Selection Sort is valuable in write-sensitive
scenarios due to its minimal number of swaps.