Analysis of the InsertionSort Algorithm

1. Algorithm Overview

InsertionSort is a classic sorting algorithm using the insertion method.

- **Idea:** At each iteration, an element is taken from the array and inserted into the sorted portion on the left.
- **Applicability:** Efficient for small arrays or nearly sorted arrays.
- **Features:** Sorting is performed in-place (uses almost no additional memory).

Theoretical Complexity:

- Best case (already sorted array): $\Theta(n)$
- Worst case (array sorted in reverse): $\Theta(n^2)$
- Average case (random array): $\Theta(n^2)$

2. Algorithm Complexity

Array Size	Best Case	Average Case	Worst Case
10	28	28	112
100	633	516201	5172
1000	9576	767200	523341
10000	128974	12835301	50323966
100000	1622706	504673200	501418382

Observation: For arrays of size 100,000 and larger, InsertionSort becomes extremely inefficient. For arrays of one million or more elements, the algorithm may take hours to complete, making it impractical for large datasets.

2.2. Space Complexity

InsertionSort works in-place, using Θ(1) additional memory.

There are no recursive calls or auxiliary arrays in this implementation.

2.3. Recurrence Relation

For a recursive version:

$T(n)=T(n-1)+\Theta(n)$

Solving this gives $T(n)=\Theta(n^2)$, consistent with empirical data for large n.

3. Code Review & Optimization

3.1. Identified Inefficiencies

Nested loops dominate performance, giving $\Theta(n^2)$ for random and reverse arrays.

For arrays ≥100,000 elements, execution time is unacceptable.

3.2. Optimization Suggestions

Binary search for insertion: Reduces comparisons to O(log n), but element shifts remain O(n).

Hybrid algorithms: For large arrays, use MergeSort or QuickSort, keeping InsertionSort for small subarrays.

Minimize variables and loops: Reduces constant factors but does not change asymptotic complexity.

3.3. Code Quality

Readable code with clear variable names.

Adding comments can improve maintainability.

4.1. Performance Plot

Based on measurements (n vs execution time):

- n = [10, 100, 1000, 10000, 100000]
- Time (ms) worst-case = [112, 5172, 523341, 50323966, 5014183823]

Observation: Time grows quadratically with n, confirming theoretical analysis.

4.2. Theory vs Practice

Best case: linear time confirmed.

Average/Worst case: quadratic time confirmed, with constant factors dependent on implementation.

5. Conclusion

1. InsertionSort is suitable for small or nearly sorted arrays.
2. For arrays ≥100,000 elements, execution time is extremely high; arrays with one million elements can take hours.
3. Hybrid or more efficient sorting algorithms are recommended for large datasets.
4. Optimizations can reduce constant factors but not change asymptotic behavior.