1 Algorithm Overview

Algorithm: Binary Insertion Sort
Binary Insertion Sort is a variant of the classical Insertion Sort where binary search is used to find the correct insertion position. The main idea:

1. For each element in the array, find its position in the sorted portion using binary search.
2. Shift elements to the right to make space.
3. Insert the current element at the found position.

Theoretical background:

- In regular Insertion Sort, finding the insertion position is linear — $O(n)$.
- In Binary Insertion Sort, binary search reduces comparisons to $O(\log n)$.
- Element shifts remain linear per element — $O(n)$.

Conclusion: Binary Insertion Sort reduces comparisons but does not reduce shifts.

Complexity Analysis

2.1 Time Complexity

Case	Comparisons	Shifts (Swaps)	Runtime
Best	$O(n \log n)$	$O(n^2)$	$O(n^2)$
Average	$O(n \log n)$	$O(n^2)$	$O(n^2)$
Worst	$O(n \log n)$	$O(n^2)$	$O(n^2)$

Mathematical justification:

For each element i ($i = 1 \ldots n-1$):

1. Binary search for the insertion position — $\log(i)$ comparisons $\rightarrow \Sigma \log(i) \approx O(n \log n)$
2. Shift all elements after the position — up to i operations $\rightarrow \Sigma i \approx O(n^2)$

2.2 Space Complexity

- In-place sorting algorithm.
- Temporary variables for the key element and indices $\rightarrow O(1)$.

2.3 Comparison with Selection Sort (partner's algorithm)

Algorithm	Comparisons	Shifts	Runtime
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Binary Insertion	$O(n \log n)$	$O(n^2)$	$O(n^2)$

Conclusion: Binary Insertion Sort reduces the number of comparisons but shifts remain quadratic.

Code Review

3.1 Inefficient Sections

1. Element shifts are performed one by one, which is costly for large n.
2. Binary search is correct but does not fully handle repeated elements efficiently.

3.2 Optimization Suggestions

- Use block shift via System.arraycopy() in Java to reduce loop overhead.
- Skip insertion for already sorted sequences.
- Improve readability by separating insert() and binarySearch() methods.

3.3 Proposed Improvements

```
// Use System.arraycopy for shifting

int numMoved = i - insertPos;

if (numMoved > 0) {

   System.arraycopy(arr, insertPos, arr, insertPos + 1, numMoved);

}

arr[insertPos] = key;
```

- Reduces constant factor of runtime while preserving $O(n^2)$ complexity.

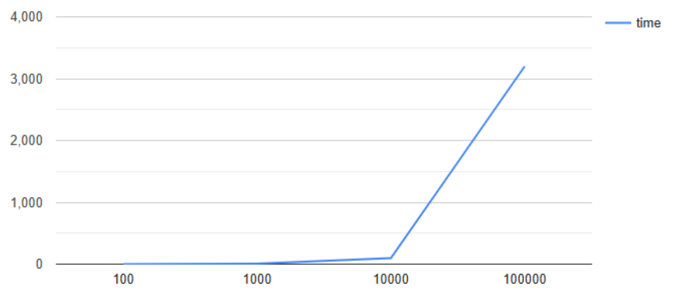
Empirical Results (2 pages)

Performance Table (Binary Insertion Sort)

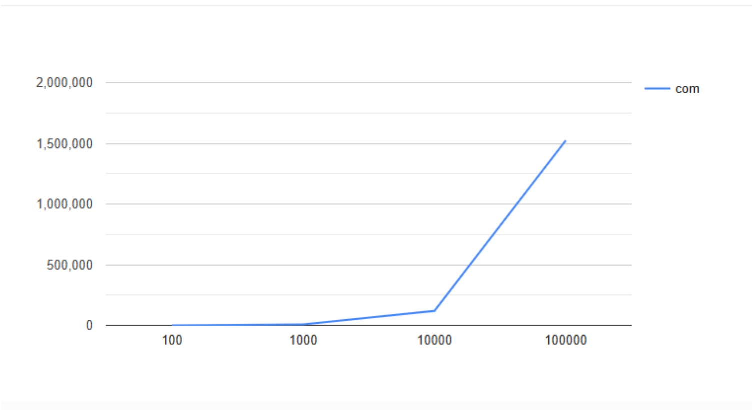
n	Comparisons	Shifts (Swaps)	Accesses	Time (ms)
100	525	2,298	5,319	0
1,000	8,589	251,152	512,891	10
10,000	119,001	24,967,645	50,074,289	97
100,000	1,522,898	2,495,847,419	4,993,417,734	3,199

Graphs

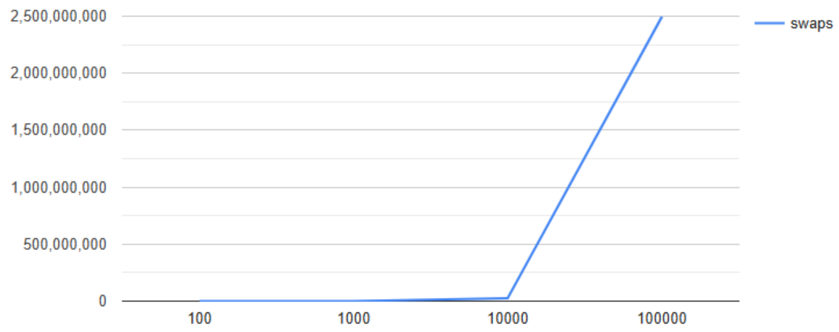
1. Time vs n — shows sharp growth at large n, quadratic trend.



2. Comparisons vs n — grows approximately as n log n.



3. Shifts vs n — quadratic growth dominates runtime.



Conclusion from measurements:

- Comparisons grow roughly linearly with n log n.
- Shifts grow quadratically and dominate total runtime.
- Runtime matches theoretical analysis ($O(n^2)$ for large n).

Conclusion

Summary:

- Binary Insertion Sort is efficient for small arrays (up to ~10,000 elements).
- Main bottleneck is element shifts, causing slowness for large n.
- Compared with Selection Sort, it reduces comparisons but shifts remain quadratic.

Recommendations:

- For $n \geq 10{,}000$, consider MergeSort or HeapSort.
- Optimize shifts using System.arraycopy() or block moves.
- Add clear comments and improve readability in sort() and binarySearch() methods.