Peer Analysis Report — Insertion Sort (analysis of partner's implementation)

Author: Nikita Tsybus
Group: SE-2421
Partner: Dias Yestemes

1. Algorithm overview (short)

Insertion sort (standard): iterate `i = 1..n-1`, take `key = arr[i]`, scan leftwards until correct insertion position is found, shift elements to the right, and place `key`. It is an *in-place*, stable algorithm. Best case (already sorted): linear time (Θ(n)). Average and worst-case: quadratic (Θ(n^2)).

Binary insertion variant: uses binary search to find insertion index among `arr[0..i-1]` in (O(log i)). However, the cost of shifting elements to make room remains (O(i)). Thus the overall time complexity is still (Θ(n^2)) in average and worst cases; binary search only reduces number of comparisons to locate the spot, not the number of element moves.

The partner implementation supports `Mode.STANDARD` and `Mode.BINARY`. The code measures several metrics via `InsertionSortMetrics` (arrayAccesses, comparisons, shifts, executionTimeNs).

2. Complexity analysis (theoretical)

2.1 Standard Insertion Sort — time

Best case (Ω): array already sorted. For each `i`, the single comparison `key >= arr[i-1]` succeeds and loop continues. Cost: roughly (n-1) comparisons and (O(1)) shifts per iteration -> total time (Θ(n)). Formally: Ω(n).

Worst case (O): array reverse-sorted. For each `i`, we compare key with all previous `i` elements and shift them all -> number of comparisons and moves: (sum(i=1…n-1) i = n(n-1)/2 = Θ(n^2)). So (O(n^2)).

Average case (Θ): For random permutations, expected number of shifts/comparisons per insertion is (Θ(i/2)), leading to total (Θ(n^2)).

Recurrence view is not necessary for basic insertion sort (it is iterative). But if one models expected cost per step (T(n) = T(n-1) + Θ(n)), solution is (Θ(n^2)).

2.2 Binary-Insertion variant — time

Binary search to find insert position for the i-th element costs (Θ(log i)). Shifting still costs (Θ(i)). So for each `i`, cost is (Θ(i) + Θ(log i) = Θ(i)). Sum over `i` yields (Θ(n^2)). Thus binary insertion does not change asymptotic time; it reduces the number of *comparisons* from (Θ(n^2)) to about (Θ(n log n)) (because compare-with-key during binary search is (O(log i)) per insertion), but it does not reduce the number of element moves.

Best case: if array already sorted, binary search costs (Θ(log i)) per step but early exit check in outer loop (the code checks `if (key >= arr[i-1]) continue;`) will avoid search, leading to (Ω(n)).

2.3 Space complexity

Both variants are in-place: auxiliary space (Θ(1)) (a few variables only). The algorithm shifts elements inside the same array—no allocation proportional to n.
---

3. Code review — metrics, and inefficiencies

I inspected the provided `InsertionSort` code carefully. Below are concrete inefficiencies, and suggestions.

3.2 Metric bookkeeping — issues and inconsistencies

* The code increments `m.arrayAccesses` and `m.comparisons` in multiple places. A few problematic points:

1. In the main loop, the code does:
   m.arrayAccesses++;
   int key = arr[i];

   m.arrayAccesses++;
   m.comparisons++;
   if (key >= arr[i - 1]) continue;

The first `m.arrayAccesses++` before `int key = arr[i]` is correct (reading arr[i]).
The second `m.arrayAccesses++` is intended for the access `arr[i-1]`, but it is incremented before the comparison and only once. This is fine if we always access `arr[i-1]` once, but the code then calls `binarySearchInsert` which will itself count array accesses — total accounting is non-uniform across branches (counts depend on whether the early `continue` triggers).
The initial check `if (key >= arr[i - 1])` increments `m.comparisons++` once — but the code does not increment comparisons for the evaluation of the while condition in `linearSearchInsert` while that condition includes `pos >= 0 && arr[pos] > key`. In `linearSearchInsert` the code increments `m.comparisons++` inside the loop but not before the loop for the first failed check; this can lead to undercounting one comparison when loop does not execute.
2. In `linearSearchInsert` comparisons and array accesses are incremented inside the loop:

```
while (pos >= 0 && arr[pos] > key) {
    m.comparisons++;
    m.arrayAccesses += 2;
    pos--;
}
```

This counts two array accesses for `arr[pos]` and `arr[pos] > key` (?) — actually the expression `arr[pos] > key` involves a single read of arr[pos]; `m.arrayAccesses += 2` seems to assume one read and one write or counts both arr[pos] and arr[pos-1] but the code does not access arr[pos-1] in that condition. So it's unclear why `+= 2` — leads to inflated arrayAccesses count.

Conclusion: metric counting is inconsistent and may misrepresent relative costs. For accurate comparison between variants, metric increments must follow a consistent policy: increment `arrayAccesses` exactly when the program actually performs a read or write, and increment `comparisons` exactly when a data comparison between array values occurs. The current code sometimes double-counts or undercounts some events.

3.3 Performance / algorithmic inefficiencies

Shifting approach: `shiftRight` performs per-element assignment in a loop:

```
 for (int j = to; j > from; j--) {
     arr[j] = arr[j - 1];
 }
 arr[from] = key;
```

This is a correct and simple approach but costly in practice because each element is moved one by one. For large arrays, using `System.arraycopy(...)` to move the block `arr[from..to-1]` one position right is often faster because it's implemented natively and can use optimizations. Example replacement:

```
System.arraycopy(arr, from, arr, from + 1, to - from);
arr[from] = key;
```

This reduces the Java-level loop overhead and reduces per-element bytecode operations, often giving big constant-factor speedups.

Binary search is only half the story: The `Mode.BINARY` variant reduces the number of comparisons to find insertion position, but does not reduce the shifting cost. Because shifts dominate memory traffic and copy time, binary insertion often yields modest or no overall speed improvement for large arrays. The empirical data (see section 4) confirms this: binary variant reduces comparisons but does not proportionally reduce total runtime.

3.4 Code style & maintainability

The code is concise and readable; methods are small and focused (`linearSearchInsert`, `binarySearchInsert`, `shiftRight`).
Suggest renaming some counters to be unambiguous: `arrayAccesses` -> `arrayReads`/`arrayWrites` or track them separately.
Consider making `shiftRight` use `System.arraycopy` and handling the special-case `from == to` cleanly.

4. Empirical validation

4.1 Key numeric observations

Selection Sort:

Random, n=10000: 39.383 ms; comparisons ~49,292,295; swaps ~8,813
Reverse, n=10000: 45.352 ms

Insertion Sort:

Random, n=10000: 125.544 ms
Reverse, n=10000: 250.025 ms
Sorted, n=10000: 0.050 ms

Insertion Sort (binary mode):

Random, n=10000: 644.915 ms
Reverse, n=10000: 618.150 ms

> Observed pattern: Selection Sort often runs faster than Insertion Sort on large random and reverse inputs in these benchmarks. At n=10000, Selection random time 39.4ms vs Insertion standard 125.5ms — Insertion ~3.2x slower on random data; reverse case Insertion ~5.5x slower.

4.2 Why is Selection faster in these runs?

Comparisons: Selection performs ~n(n-1)/2 comparisons (≈ 50M for n=10k) — same order as insertion comparisons. But swaps in selection are only O(n) (one swap per outer loop), while insertion executes many element moves (shifts) — each shift is a write operation that is expensive relative to a read or compare. In particular, insertion shifts ≈ number of inversions in array; for reversed array it's maximal (~n(n-1)/2), leading to many writes.
The heavy cost of memory writes (shifts) explains why insertion is much slower in practice despite comparable or fewer comparisons in some measurements. The `shiftRight` loop performs many element assignments — replacing it with `System.arraycopy` (which delegates to optimized native code) would reduce time significantly.

4.3 Binary-insertion tradeoffs

The binary mode reduces comparisons substantially but does not reduce number of shifts; empirical data shows comparisons drop while time may not (and in provided numbers binary mode can be even slower — possibly because the reported times are inconsistent units).
Conclusion: binary insertion helps only when comparisons are hugely dominating costs and shifts are negligible (rare). For standard arrays of primitive `int`, shifts (memory writes) often dominate.

4.4 Complexity verification

The log-log time vs n plot for each algorithm and input type shows near-quadratic scaling for random and reversed inputs — slope near 2 on log-log scale, which confirms (Θ(n^2)) average/worst behavior.
For sorted input, Insertion Sort shows near-linear scaling, as expected (best case (Θ(n))). Selection remains quadratic even on sorted input (does not exploit sortedness).

5. Optimization suggestions (concrete)

Below are practical, implementable suggestions.

1) Use `System.arraycopy` for shifting

Expected effect: large constant-factor speedup, because `System.arraycopy` uses optimized native code.

2) Remove unnecessary metric overhead inside inner loops (or make metrics optional)

Metric counter increments inside the hot inner loops (comparisons, arrayAccesses) add overhead that distorts timing. My option:

Keep metrics but disable metrics in time-sensitive JMH runs; use separate runs to gather metrics.

3) Avoid unnecessary array accesses / cache values

And avoid `m.arrayAccesses += 2` unless you truly did two separate accesses (read + write).

6. Empirical comparison summary (Selection vs Insertion)

Using the provided numbers (converted where appropriate):

Random input (n=10k):

Selection Sort: ~39.4 ms
Insertion Sort (standard): ~125.5 ms
Interpretation: Insertion performs many shifts (writes) — more expensive than Selection's occasional O(n) swaps. Selection is faster because writes are far fewer.

Reverse-sorted input (n=10k):

Selection: ~45.4 ms
Insertion: ~250.0 ms
Interpretation: reverse case is worst for insertion (maximal shifts), selection still only does O(n) swaps but O(n^2) comparisons; but comparisons seem cheaper than mass writes.

Conclusion: though both algorithms are (Θ(n^2)), practical performance depends strongly on the ratio of comparisons vs writes. For arrays of primitives, writes are expensive; selection's fewer writes make it competitive and often faster for large random / reverse inputs.