**REPORT**

**Student:** *Sanzhar Sagatov (Student A)*  
**Partner:** *Tlegen Tolegenuly (Student B)*

**Insertion Sort**

**1. Algorithm Overview**

**Description**

Insertion Sort is a comparison-based, in-place, stable sorting algorithm. It builds the final sorted array one element at a time by taking the next element and inserting it into the correct position among the previously sorted elements. For an array A[0..n-1], insertion sort processes indices from left to right, maintaining that A[0..i-1] is sorted and inserting A[i] into that sorted prefix.

**Relevance for nearly-sorted data**

Insertion Sort is *adaptive*: its running time depends on the number of inversions in the input. For nearly-sorted arrays (few elements out of order), insertion sort can approach linear behavior. Because of this property, it is frequently used as the base-case sorter in hybrid algorithms (e.g., Timsort, MergeSort with insertion for small subarrays).

**Common Optimizations for nearly-sorted inputs**

* **Binary insertion (binary search for insertion position):** reduces comparisons from O(n) per insertion to O(log n) comparisons, but does not reduce the number of element moves (shifts).
* **Use System.arraycopy / block moves:** uses native bulk-copy to shift ranges which can be faster than manual element-by-element movement in Java.
* **Early-exit scanning:** when scanning from right to left, detect that a portion is already sorted and break early.
* **Sentinel or guard element:** avoids an extra index check in inner loop.
* **Adaptive thresholding in hybrids:** only use insertion sort for subarrays of length ≤ k (typical k between 16 and 64).

**2. Complexity Analysis**

**Notation**

Let n be the number of elements. Let I denote the number of *inversions* in the input (pairs (i,j) with i<j and A[i] > A[j]). An already-sorted array has I = 0; a reverse-sorted array has I = n(n-1)/2 (Θ(n²)).

**Time complexity (detailed)**

**Basic insertion sort (naïve implementation)**

* **Worst case (reverse-sorted):**
  + Each insertion requires shifting i elements for position i → total moves:  
    (\sum\_{i=1}^{n-1} i = n(n-1)/2 = Θ(n^2)).
  + **Worst-case time:** **Θ(n²)** (and thus O(n²), Ω(n²)).
* **Best case (already sorted):**
  + Each insertion compares once and performs no shifts.\
  + **Best-case time:** **Θ(n)** (O(n), Ω(n)).
* **Average case (random permutation):**
  + Expected number of inversions is (n(n-1)/4 = Θ(n²)). Running time is **Θ(n²)**.

**Adaptive expression using inversions**

* The number of element moves is proportional to the number of inversions I. Cost can be expressed as **Θ(n + I)** (comparisons and moves combined). For nearly-sorted arrays I can be small, so cost approaches linear.

**Binary insertion (reduces comparisons)**

* Comparisons reduced to O(log n) per insertion → total comparisons O(n log n). However element shifts remain proportional to I, so **time = Θ(n log n + I)**. For nearly-sorted arrays where I = o(n log n), this may be dominated by O(n log n) comparisons.

**Hybrid approach (e.g., insertion sort for small runs within mergesort/Timsort)**

* Overall complexity depends on the higher-level algorithm; insertion cost on subarrays of size ≤ k is bounded by O(k²) per subarray. Chosen properly, overall asymptotic cost of the hybrid matches the outer algorithm with improved constants.

**Big-O/Θ/Ω summary**

* **Naïve insertion sort:**
  + Best: Θ(n) (Ω(n), O(n))
  + Average: Θ(n²)
  + Worst: Θ(n²)
* **Insertion with binary search:**
  + Best: Θ(n) (still linear if already sorted)
  + Average/Worst: Θ(n²) in terms of data movement; comparisons are O(n log n)
* **Adaptive expression:** Θ(n + I)

**Space complexity**

* **Auxiliary space:** **O(1)** in-place; stable (relative order of equal elements preserved) provided shifting is done correctly.
* **No recursion** used — constant additional memory: a few temporaries (key element, indices).
* Using System.arraycopy does not change asymptotic auxiliary memory (still O(1) extra), but it may allocate small temporary buffers internally in some JVM implementations (usually negligible and still O(1)).

**Recurrence relations (where applicable)**

Insertion sort is primarily iterative; a recursive formulation (for analysis) can be written as:

[ T(n) = T(n-1) + Θ(n) ]

Solving gives (T(n) = Θ(n^2)) for worst case. This recurrence is a linear homogeneous recurrence with a non-homogeneous Θ(n) term; telescoping sum yields the quadratic bound.

**3. Code Review & Optimization**

Below I analyze a canonical Java implementation and point out inefficiencies and practical optimizations.

**Inefficiency detection**

1. **Per-element shifting in Java loop:** shifting by one in a while loop does repeated bound checks and array loads/stores; for large moves a System.arraycopy (block move) may be faster.
2. **Comparison-heavy inner loop:** in nearly-sorted arrays comparisons are cheap, but in random/worst inputs the inner loop executes many times.
3. **Suboptimal insertion point search:** linear scan from i-1 downwards costs O(i) comparisons; binary search reduces comparisons but not moves.
4. **No sentinel/guard:** each inner loop iteration checks j >= 0 which is a branch; a sentinel can remove one check.
5. **Using boxed types (Integer) or comparator overhead:** if code uses Integer[] or Comparator<T> wrappers, this adds boxing/unboxing or virtual calls — avoid when not needed.

**Suggested algorithmic optimizations**

1. **Binary insertion search + System.arraycopy**
   * Use Arrays.binarySearch (or custom binary search) on a[0..i) to find insertion index, then perform a single System.arraycopy to shift the block. This reduces comparisons while keeping moves.
2. **Adaptive early-exit bookkeeping**
   * While scanning, if the remaining prefix is already ordered, skip extra work. For example, detect long increasing runs and skip them.
3. **Hybrid strategy**
   * Use insertion sort for small runs inside a higher-level divide-and-conquer sorter (common pattern). This keeps asymptotics of the outer algorithm while improving constants.
4. **Use primitive arrays and avoid comparators**
   * For int[] use primitive operations; for objects prefer T extends Comparable<T> but inline comparators carefully.

**Space Complexity Improvements**

* Keep algorithm in-place (already O(1)). Avoid temporary arrays or lists. When using System.arraycopy, you still remain O(1) extra.

**Code Quality & Maintainability**

* **Readability:** Keep helper methods (e.g., binarySearchInsertPos) small and well-documented. Use descriptive names and Javadoc for public methods.
* **API design:** Provide both in-place void sort(int[] a) and a int[] sortedCopy(int[] a) if immutability is desired.
* **Testing:** Add unit tests for edge cases: empty array, single element, all equal elements, duplicate-heavy arrays, nearly-sorted arrays, reverse-sorted arrays.
* **Benchmark harness:** Provide a main or JMH microbenchmarks for robust timing (JMH recommended for JVM microbenchmarks to avoid warmup and JIT effects).

**4. Empirical Validation**

**Note:** I ran preliminary benchmarks and saved plots to a PDF (/mnt/data/peer\_analysis\_report.pdf) that contains time-vs-n log-log plots for several algorithms. Below I describe methodology, expected results, and interpretation. If you want, I can produce a dedicated Java benchmarking harness (JMH) and include raw data tables.

**Measurement methodology (recommended for Java)**

* **Benchmark harness:** Use JMH (Java Microbenchmark Harness). If JMH is not available, use System.nanoTime() with careful warmup and multiple trials:
  + Run several warmup iterations (e.g., 5) to trigger JIT compilation.
  + For measured runs, perform multiple trials (e.g., 10) and record the mean and standard deviation.
  + Garbage collector: minimize interference by pre-allocating any large objects and invoking System.gc() between large trials if needed (note: GC introduces noise).
* **Input distributions:** test across:
  + Already sorted
  + Nearly-sorted (few swaps or small number of random inversions)
  + Random uniform
  + Reverse-sorted (worst case)
  + Many duplicates (e.g., small key range)
* **Sizes:** n ∈ {100, 1\_000, 10\_000, 100\_000} — note insertion sort will be slow for random 100\_000; consider limiting to 10\_000 for some algorithms to keep test time reasonable.
* **Metrics:** wall-clock time (ms), comparisons (if instrumented), data moves (array writes), memory usage (optional).

**Expected empirical curves**

* **Already-sorted:** near-linear; times scale roughly O(n).
* **Nearly-sorted (k inversions):** time ≈ Θ(n + k). If k scales sublinearly, runtime approaches linear.
* **Random:** quadratic curve on linear scale; on log-log plot slope ≈ 2 (since time ∝ n²).
* **Binary insertion + arraycopy:** comparisons curve ~ O(n log n) (slope ≈ 1 \* log n behaviour on log-linear scales), but data movement remains tied to inversions.