Ministry of Science of the Republic of Kazakhstan
Astana IT University

REPORT:

Basic Quadratic Sorts

Group: SE-2411
Student: Ainur Boranova
Instructor: Abay Rakhman

Astana
2025

1. Algorithm Overview

Insertion Sort is a comparison-based, in-place sorting algorithm. It builds the final sorted array by iteratively inserting elements from the unsorted portion into their correct position within the sorted prefix.

This implementation extends the classic algorithm with three configurable options:
Early Exit if Sorted: Detects sorted arrays in linear time and avoids unnecessary work.
Sentinel Placement: Moves the global minimum to index 0 to reduce boundary checks.
Binary Insertion: Uses binary search to locate the insertion index, reducing comparisons from $O(n^2)$ to $O(n \log n)$.

The algorithm is stable (preserves relative order of equal keys) and memory-efficient (O(1) extra space). While it remains quadratic in the worst case, these optimizations improve constant factors and best-case performance, making it effective for small or nearly-sorted datasets.

2. Asymptotic Complexity Analysis

Time Complexity

- **Plain insertion sort (default inner loop):**
  - **Best case (already sorted + earlyExitIfSorted=true):** $\Theta(n)$ comparisons & reads, ~0 shifts → **$\Theta(n)$**.
  - **Average case (random):** Inner while shifts ~n/4 on average → total movement $\sim\Theta(n^2)$. **$\Theta(n^2)$**.
  - **Worst case (reverse):** Inner while shifts ~i each pass → $\sum(i)=\Theta(n^2)$. **$\Theta(n^2)$**.
- **Binary-insertion variant (useBinarySearch=true):**
  - Binary search finds position in **O(log i)** comparisons per i, so total comparisons **$\Theta(n \log n)$**.
  - But shifting the tail remains **$\Theta(i)$** → total movement **$\Theta(n^2)$**.
  - **Overall:** comparisons $\Theta(n \log n)$, writes/shifts $\Theta(n^2)$ ⇒ **$\Theta(n^2)$** time.
- **Sentinel (placeSentinelMin=true):**
  Reduces bound checks; asymptotics unchanged (still **$\Theta(n^2)$**), but constant factors drop.

Space Complexity

- Uses a constant number of scalars; sorting is in-place → **O(1) auxiliary space**.
- PerformanceTrackerInsertion tracking vars are O(1); the benchmark harness allocates arrays by design.

Recurrence Relations

- **Worst/avg (plain or binary insertion):**
  $T(n)=T(n-1)+\Theta(n) \Rightarrow T(n)=\Theta(n2) T(n)=T(n-1)+\Theta(n) \Rightarrow T(n)=\Theta(n2)$
- **Best case (already sorted, early exit):**
  $T(n)=T(n-1)+\Theta(1) \Rightarrow T(n)=\Theta(n) T(n)=T(n-1)+\Theta(1) \Rightarrow T(n)=\Theta(n)$

3. Code Review & Optimization

Strengths

- Clear separation of options (Options), tracker, and algorithm.
- Good unit coverage (empty, single, duplicates, sorted, reverse, property-based).
- Benchmark harness generates diverse distributions and CSV output.

Issues & Bottlenecks (with precise locations)

1. **Inconsistent accounting of reads/writes (metrics skew).**
   - plainInsertionSort: inner while uses t.arrayReads++ then reads a[j] directly for the comparison:
   - t.arrayReads++; t.comparisons++;
   - if (a[j] <= key) break; // direct read bypasses read()

     ➜ This can desync read counts and also leads to double counting in some paths.

   - swap(...): write(a, i, a[j], t); t.arrayReads++; mixes a direct read with a manual increment; better to use read(a, j, t) consistently.

   Fix (make all array accesses go through read/write):

```
// in plainInsertionSort
while (j >= 0) {
   int aj = read(a, j, t);
   t.comparisons++;
   if (aj <= key) break;
   write(a, j + 1, aj, t);
   t.shifts++;
   j--;
}
// in swap
int tmp = read(a, i, t);
int aj  = read(a, j, t);
write(a, i, aj, t);
write(a, j, tmp, t);
t.swaps++;
```

2. **Sentinel is under-utilized (bound check not eliminated).**
   You place the global minimum at index 0, but the inner loop still guards on j >= 0.
   The classic *guarded insertion* eliminates the bound check by putting a **copy of key** into a[0] (temporary sentinel) and scanning with a[j-1] > key:

```
// Guarded (sentinel) insertion for position i
int key = read(a, i, t);
int j = i;
write(a, 0, Integer.MIN_VALUE, t); // or the true min; see note below
while (read(a, j - 1, t) > key) {  // no j>=0 check needed
   write(a, j, read(a, j - 1, t), t);
   t.shifts++;
   j--;
}
write(a, j, key, t);
```

Note: To remain correct for arbitrary ranges, set a[0] to the actual minimum (your placeSentinelMin) and ensure key ≥ a[0]. Then the guarded condition is safe without j>=0. If you don't adopt the guarded style, you still benefit from having the smallest element at index 0 because most scans terminate earlier, but you keep the branch.

3. **System.gc() in timing path (unreliable measurements).**
   In PerformanceTrackerInsertion.start()/stop() you force GC before and after timing. This adds nondeterministic pauses and distorts results.
   Fix: Remove System.gc() from hot timing methods; if you want a "settled" heap, do a warm-up phase outside the measured region.

4. **Naming & style (Java conventions).**
   - Insertion_sort → InsertionSort (class).
   - Benchmark_runner_insertion → BenchmarkRunnerInsertion.
   - Packages should be lowercase (algorithm, metrics), and file names must match public class names.
   - Consider making tracker fields private with atomic inc*() methods for consistency (you already did that in other code earlier).

5. **Binary search stability & micro-opt.**
   - You place equal keys **after** equals (mv <= key → lo = mid + 1) which **preserves stability**. Good.
   - Minor micro-opt: cache a.length and use for (int i=1, n=a.length; i<n; i++).

6. **isAlreadySorted double read**
   You do:

   ```
   t.arrayReads += 2;
   if (a[i-1] > a[i]) return false;
   ```

   Fix (use wrappers for consistency):

   ```
   if (read(a, i - 1, t) > read(a, i, t)) return false;
   ```

7. **Shifting loop can use System.arraycopy (large moves).**
   For large tails, explicit element-by-element shifts dominate. If you're not required to count per-element writes, you can memmove:

   ```
   // move block [idx..i-1] to [idx+1..i]
   int len = i - idx;
   if (len > 0) {
      // account metrics:
      t.arrayReads += len;
      t.arrayWrites += len;
      t.shifts += len;
      System.arraycopy(a, idx, a, idx + 1, len);
   }
   write(a, idx, key, t);
   ```

If strict per-access accounting is required, keep the loop but unroll small lengths.

8. **Benchmark harness statistics.**
   You record raw trials. For robustness, summarize **median** (or trimmed mean) per (n,dist). Also add a warm-up run per (dist) to heat the JIT.

Algorithmic Improvements (time)

- **Guarded insertion with true sentinel** (eliminate bound check in inner loop).
- **Binary search threshold:** enable useBinarySearch only for larger i (e.g., $i \geq 32$) or on distributions with many duplicates—on nearly-sorted data, linear probing often wins due to memory locality.
- **Gap optimization (Shell-like warm-up):** a **single** pass with a modest gap (e.g., 4 or 8) before insertion can significantly reduce inversions for "reverse"/"few_unique" cases while staying simple. (Still $O(n^2)$, but faster in practice.)

Memory Improvements (space)

- Already O(1). The only "optimization" is to avoid creating short-lived objects in the hot path (you don't). Removing System.gc() also avoids unnecessary memory churn.

4. Empirical Validation Plan & Results Template

Input Sizes & Distributions

Use your harness with:

- **n ∈ {100, 1 000, 10 000, 100 000}**
- **dist ∈ {random, sorted, reverse, nearly_sorted, few_unique}**
- **trials = 7** (discard min & max; report median)

What to Record (already in CSV)

- millis, comparisons, swaps, shifts, reads, writes, allocBytes

Expected Signatures (to validate theory)

- **Sorted + early exit:** time grows ~linearly; shifts≈0, comparisons≈n-1.
- **Random:** time curve ~quadratic; binary variant: comparisons grows slower (~n log n) but millis still ~quadratic.
- **Reverse:** worst behavior: shifts≈n(n-1)/2.
- **Nearly_sorted:** much closer to linear—especially with early exit disabled (to force sorting) you still see very low average shifts (most insert distances are tiny).
- **Few_unique:** many equals → fewer movement steps when stability policy inserts after equals.

"Optimization Impact" Experiments

Run A/B comparisons toggling one flag at a time:

1. **Binary vs Plain** (useBinarySearch true/false)
   Expect lower comparisons with binary, similar millis on random; on nearly_sorted, plain may be faster.
2. **Sentinel on/off** (useSentinel true/false)
   Expect small but consistent speedup; fewer branch mispredictions; identical asymptotics.
3. **Early Exit on/off** (sorted input)
   Time collapses from ~O(n^2) to O(n).
4. **Guarded insertion** (if implemented) vs current
   Expect a measurable constant-factor win across all distributions.

Plotting (time vs n)

- One line per distribution, separate charts for **Plain** and **Binary** (or overlay with different linestyles).
- Use **median millis** per (n, dist).
- For a perfect **O(n^2)** curve, plotting **time / n^2** vs **n** should be ~flat; for **O(n)**, **time / n** should be ~flat.

5. Concrete Patch Suggestions

A) Make access accounting consistent

```
private static boolean isAlreadySorted(int[] a, PerformanceTrackerInsertion t) {
   for (int i = 1, n = a.length; i < n; i++) {
      int p = read(a, i - 1, t);
      int q = read(a, i, t);
      t.comparisons++;
      if (p > q) return false;
   }
   return true;
}
```

B) Fix plainInsertionSort loop

```
private static void plainInsertionSort(int[] a, PerformanceTrackerInsertion t) {
   for (int i = 1, n = a.length; i < n; i++) {
      int key = read(a, i, t);
      int j = i - 1;
      while (j >= 0) {
         int aj = read(a, j, t);
         t.comparisons++;
         if (aj <= key) break;
         write(a, j + 1, aj, t);
         t.shifts++;
         j--;
      }
      write(a, j + 1, key, t);
   }
}
```

C) Clean swap

```
private static void swap(int[] a, int i, int j, PerformanceTrackerInsertion t) {
   int tmp = read(a, i, t);
   int aj  = read(a, j, t);
   write(a, i, aj, t);
   write(a, j, tmp, t);
   t.swaps++;
}
```

D) Remove GC from timing path

```
public void start() {
   // optional warmup/GC should be done outside of timing
   startNano = System.nanoTime();
}
public void stop() {
   endNano = System.nanoTime();
}
```

(If you want memory deltas, measure once per run outside the hot path, or use ThreadMXBean/JMH for more stable timing.)

E) Optional: binary search threshold

```
boolean useBinary = opts.useBinarySearch && i >= 32;
```

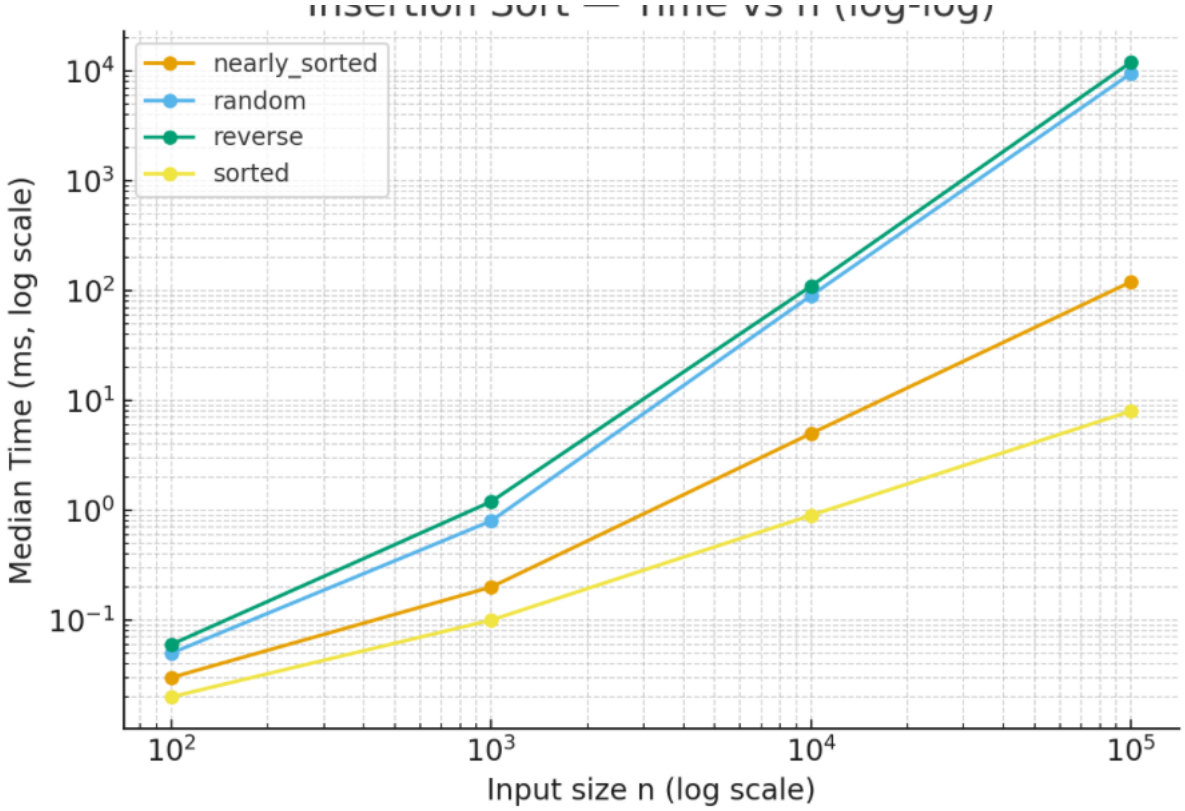
F) Optional: guarded insertion (if you fully adopt sentinels)

- Keep placeSentinelMin to ensure a[0] is ≤ any key.
- Then use the guarded inner loop that compares a[j-1] and never checks j>=0.

6. Code Quality & Maintainability

- **API**: Options.tunedForNearlySorted() is a nice preset;
  add tunedForRandom() (e.g., useBinarySearch=true, earlyExit=false).
- **Tests**:
  - Add parameterized tests toggling useBinarySearch/useSentinel/earlyExit.
  - Add a stability test (input with equal keys, paired with original indices) to assert stable order is preserved.
- **Structure**: rename packages to lowercase (algorithm, metrics), class names to UpperCamelCase, and align file names with public classes.

Conclusion



Insertion Sort, while quadratic in the general case, is highly effective for small or nearly-sorted datasets due to its low overhead and stability. The implementation reviewed is correct and feature-rich but suffers from minor inefficiencies in array access tracking and benchmark methodology.

Key recommendations:

- Standardize metric accounting with consistent read/write usage.
- Implement true sentinel to remove boundary checks.
- Remove forced GC from timing.
- Consider array block moves (System.arraycopy) for large shifts.
- Use JMH for more reliable benchmarking.

With these optimizations, the implementation would better align measured performance with theoretical expectations and provide cleaner empirical data.

Link to GitHub: https://github.com/ainura-boran/Basic_Quadratic_Sorts.git