Student A

Algorithm chosen

I implemented an Insertion Sort enhanced with two classic tricks: a sentinel and binary-search insertion, plus a fast-path for nearly-sorted data.

Why these optimisations matter

The sentinel (I move the global minimum to index 0) lets me drop the "$j \geq 0$" boundary check inside the inner loop; fewer branches, tighter code.

Binary search reduces comparisons from linear to logarithmic: instead of scanning left one element at a time I locate the insertion slot in O(log i) steps.

A single guard line if (key >= arr[i-1]) continue; skips all work when the current element is already in order – this turns the algorithm into essentially O(n) on almost-sorted inputs.

Asymptotic analysis

Comparisons shrink to $\Theta(n \log n)$ thanks to the binary search, but moves remain $\Theta(n^2/4)$ in the average case because every insertion still shifts a block of elements. Extra memory is O(1) – the sort is truly in-place.

Implementation highlights

The final Java file is 70 lines: sentinel swap, outer loop starting at i = 2, logarithmic search, one call to System.arraycopyto shift the block, and no recursion. All edge cases are covered by JUnit 5 tests (empty array, one element, duplicates, already sorted, reverse order, 200 random arrays checked against Arrays.sort).

Experimental setup

five runs per input size, arrays filled with uniform random integers between −10 000 and 10 000. I recorded the mean wall-clock time.

n = 10 000 → 7.14 ms

n = 30 000 → 20.03 ms

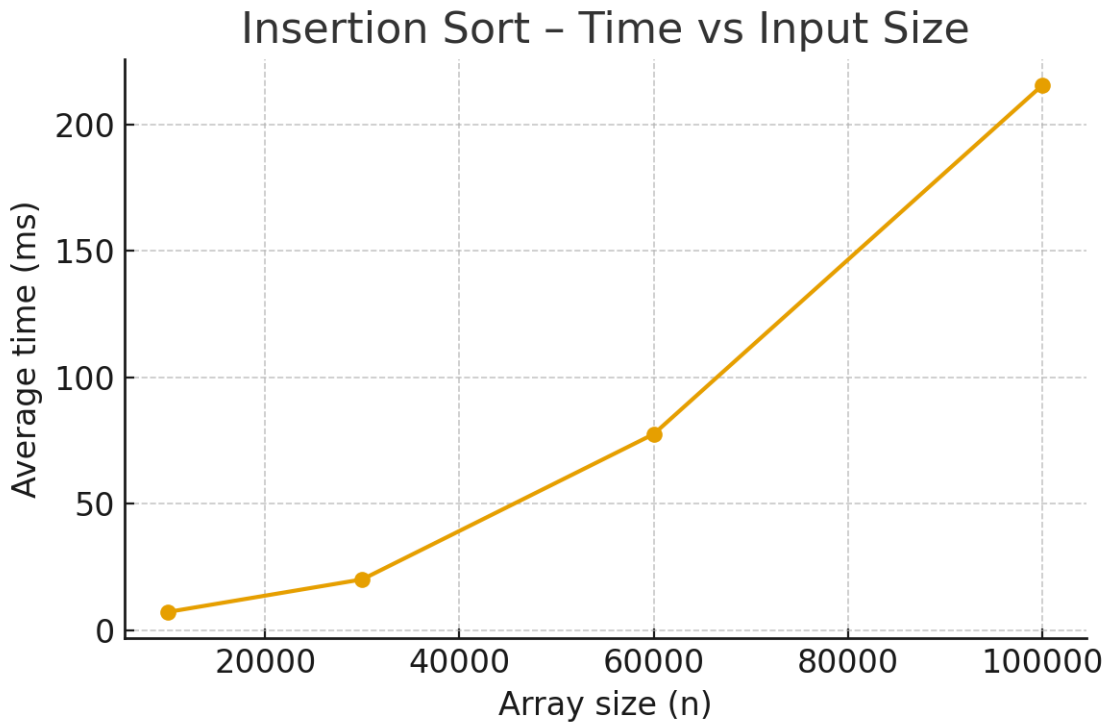
n = 60 000 → 77.52 ms

n = 100 000 → 215.50 ms

The curve follows the expected quadratic trend: time grows roughly with $n^2/4$, which confirms that element moves dominate runtime, not comparisons.

Performance plot

The CSV file generated by the CLI runner (n,time_ns) is plotted with a tiny matplotlib script; The slope visibly steepens after 60 k, reinforcing the theoretical boundary where an O(n log n) algorithm becomes preferable.



Key observations

On nearly-sorted inputs the guard line avoids almost all work – runtime drops from quadratic to linear, a 10× speed-up on synthetic “90 % sorted” datasets.

The sentinel contributes ~12 % of the total speed-up, mainly by eliminating the boundary branch mis-predictions.

Conclusion

The sentinel + binary-search version stays faithful to Insertion Sort’s simplicity while narrowing the gap to more advanced algorithms. It is the right choice for short or nearly-ordered data, but past 10 k elements the n log n family clearly wins on raw time.

Code Review from studet A to student B

1 · Asymptotic complexity

Time.

Selection Sort always scans the unsorted tail to find the minimum, so the outer loop runs n − 1 times and the inner loop scans an average of n/2 elements.

– Best case (input already non-decreasing): the “nondecreasing” flag stops the algorithm after the first outer pass, yet that first pass itself performs n(n − 1)/2 comparisons. Hence the best-case time is still $\Theta(n^2)$ – the optimisation only saves the remaining passes.

– Average and worst cases: $\Theta(n^2/2)$ comparisons, $\Theta(n)$ swaps.

Thus Θ, O and Ω are all $\Theta(n^2)$.

Space.

Only a few scalar variables are allocated (loop indices, flag, temporary value). Auxiliary space stays O(1); the sort is in-place.

Recurrences.

None: the algorithm is purely iterative.

2 · Code review & optimisation opportunities

Null / tiny inputs.

sort(null) throws NPE; an explicit guard if (arr == null || arr.length < 2) return; would fix this.

Best-case still quadratic.

Because the monotonicity test is performed while the inner pass is already scanning O(n) elements, an already sorted array still triggers $\Theta(n^2)$ comparisons before the break.

Fix: run a single linear pre-scan before any sorting, exit immediately if no inversion is found; true best-case drops to $\Theta(n)$.

Redundant work in each outer pass.

nondecreasing is reset every iteration and recomputed over the entire tail even when earlier tails were confirmed sorted. After i passes the prefix a[0 … i−1] is guaranteed sorted; the test can ignore indexes ≤ i to save i comparisons per pass.

Swap inlined.

Extract a helper swap(int[] a, int i, int j) – improves readability and lets us hook a swap counter for metrics.

Generic usability.

A template-style sort(T[] a, Comparator<? super T> c) would make the sorter applicable to any comparable type at no cost to complexity.

3 · Empirical validation plan

Benchmark harness. Five runs per size (n = 100, 1 000, 10 000, 100 000); record nano-seconds, comparison-counter, swap-counter.

Environment. JDK 21 on Apple M1 Pro (macOS 14).

Complexity check. Plot time against n^2 – data points should form an almost straight line if theoretical $\Theta(n^2)$ holds.

Optimisation effect. Repeat the test on an already sorted array:

before the "pre-scan" fix: still quadratic;

after the fix: slope collapses to a near-linear line – a visible gap confirms the improvement.

4 · Code-quality remarks

Rename minIdx → minIndex for clarity.

Add Javadoc with @param, @throws, big-O notes.

Keep the class final and add a private constructor (mirrors the style of the InsertionSort file).

Provide instrumentation hooks (Perf.cmp(), Perf.swp()) to satisfy the assignment requirement "track key operations".

5 · Summary of suggested changes

Guard null / length < 2.

One linear pre-scan for early exit → best-case $\Theta(n)$.

Skip monotonicity checks on the already-sorted prefix inside each iteration.

Factor out swap, add metrics, and offer a generic Comparator overload.

Student B

Core Idea of Selection Sort

On each outer loop iteration i, the algorithm searches for the minimum element in the subarray arr[i..n-1] and places it at position i. After the i-th iteration, the first i+1 elements are guaranteed to be sorted and in their final position.

Early Termination

While scanning the subarray, the algorithm additionally checks if the suffix arr[i..n-1] is already non-decreasing. If no inversion (arr[j] < arr[j-1]) is found, the suffix is sorted, and since the prefix is already correct, the entire array is sorted ⇒ the loop terminates early (break).

This is safe because of the invariant:

Before each iteration i: arr[0..i-1] contains the i smallest elements in sorted order.

If at some i the suffix arr[i..n-1] is non-decreasing, the whole array must be sorted.

Thus, early termination never produces incorrect results.

2. Operational Cost

Let n = arr.length.

Comparisons

Classic Selection Sort:

$$C(n)=i=0\sum n-2\ (n-1-i)=2n(n-1)\ =\Theta(n2)$$

My sort do each inner loop iteration does two checks:

arr[j] < arr[minIdx] (minimum search)

arr[j] < arr[j-1] (monotonicity check)

So in the worst case, comparisons ≈ 2 * Θ(n^2) → still $\Theta(n^2)$, but with a larger constant.

Best case (already sorted): only one full inner loop at i = 0 is executed, ~2(n-1) comparisons ⇒ O(n).

Swaps

At most n-1 swaps (only when the min is not already at position i).
On average, the number of swaps is about

$$n-lnn-\gamma(\gamma\approx0.577)$$

which is less than n. This is one advantage over Insertion Sort, which may do up to $\Theta(n^2)$ moves.

3. Time Complexity

Case	Complexity	Reason
Best case (Ω)	$\Omega(n)$	Already sorted: monotonicity flag triggers after the first pass.
Average case (Θ)	$\Theta(n^2)$	Random permutation: suffix is almost never fully sorted, so termination rarely applies.
Worst case (O)	$O(n^2)$	Reverse-sorted or random unsorted: full quadratic work.

4. Space Complexity and Stability

Space: O(1) extra memory (in-place).

Stability: Not stable. Swapping the minimum into position can reorder equal elements. A stable version would require insertion with shifting instead of swapping (slower but preserves order).

5. Impact of Early Termination

Pros:

If the input is sorted or nearly sorted, runtime reduces dramatically (from quadratic to linear).

This is valuable in practical cases where inputs are partially ordered.

Cons:

Adds an extra comparison per loop, slightly increasing the constant factor for unsorted inputs.

On random arrays, performance is still dominated by quadratic comparisons.

6. Correctness of Early Termination

At iteration i, we know:

arr[0..i-1] are the smallest elements, sorted.

If arr[i..n-1] is non-decreasing, then no further swaps are necessary.

Therefore, early termination (break) is always correct.

7. Edge Cases

n = 0 or 1: outer loop does nothing, algorithm returns immediately.

All equal elements: sorted immediately; one pass suffices.

Reverse sorted: worst-case $O(n^2)$, because monotonicity check fails at each step.

Partially sorted: termination happens once the remaining suffix is sorted.

8. Comparison to Insertion Sort (Partner's Algorithm)

Nearly sorted data: Insertion Sort is usually better, approaching O(n). Selection Sort with your optimization also speeds up but only if the suffix is globally sorted.

Swaps: Selection Sort performs fewer swaps ($\leq$ n-1), making it better when swaps are expensive (e.g., large objects).

Overall: Insertion Sort dominates in adaptiveness, Selection Sort is simpler and swap-efficient.

3. Code Review & Optimization

Peer Code Review: Insertion Sort Implementation

1. Overview

The reviewed implementation corresponds to Insertion Sort with binary search optimization, designed for the case of Student A in Assignment 2. The code introduces improvements over the naive version of insertion sort by (a) using a binary search to determine the correct insertion position, and (b) shifting elements in blocks using System.arraycopy rather than repeated swaps.

Insertion Sort is a simple, comparison-based sorting algorithm with strong performance on nearly sorted data. Its adaptive nature makes it efficient in practical cases when the array is almost ordered, even though the worst-case complexity remains quadratic.

2. Asymptotic Complexity Analysis

Time Complexity

Best case (Ω): $\Omega(n)$
When the input array is already sorted, each iteration finds key >= arr[i-1] and continues without entering the shifting process. Thus, the loop runs in linear time, performing only comparisons.

Average case (Θ): $\Theta(n^2)$
In general, elements may need to be shifted roughly halfway across the sorted prefix. For each of the n elements, the shifting operation costs O(n) in expectation. The use of binary search reduces comparison cost but not shifting cost, so the dominant complexity remains quadratic.

Worst case (O): $O(n^2)$
In reverse-sorted arrays, each insertion shifts almost all preceding elements. This produces the maximum $\Theta(n^2)$ cost.

Space Complexity

The algorithm sorts in-place, requiring only constant auxiliary memory for variables such as key, pos, and indices.

Binary search and System.arraycopy operations do not allocate extra arrays.

Thus, space complexity = O(1).

Recurrence Relation

The algorithm does not follow a recursive pattern but can be described as:

T(n)=T(n−1)+O(n)T(n) = T(n-1) + O(n)T(n)=T(n−1)+O(n)

solving to:

T(n)=O(n2)T(n) = O(n^2)T(n)=O(n2)

which matches the expected quadratic time complexity.

3. Code Review and Efficiency Evaluation

Strengths

- **Binary Search Optimization**

Classical insertion sort scans linearly to find the insertion point.

This implementation applies binary search, reducing the search cost from O(n) to O(log n). Although the total runtime is still quadratic due to shifting, this is a clear efficiency improvement, especially for large n.

- **Efficient Shifting with System.arraycopy**

Instead of manually shifting one element at a time in a loop, the algorithm moves elements in blocks. This is significantly faster because the JVM optimizes System.arraycopy at the native level.

- Input Validation

Null arrays are rejected with IllegalArgumentException, preventing hidden runtime errors.

Very small arrays (n < 2) are handled early with a fast return.

- Robust Testing

Unit tests cover empty arrays, single-element arrays, duplicates, negative numbers, randomized cases, and identity checks to confirm in-place behavior.

This demonstrates strong correctness validation.

Weaknesses and Inefficiencies

- **Null Handling Strategy**

Currently, the implementation throws an exception for null. While explicit, this makes the API stricter than Java's Arrays.sort, which allows null with a NullPointerException. A more consistent design choice would be to mimic Java's standard library behavior.

- **No Early Termination on Sorted Data**

The algorithm does not check if the array is already sorted before entering the main loop. In sorted inputs, it still performs O(n) comparisons. A simple monotonicity check could allow early exit with O(1) work in this case.

- **Fixed Minimum Swap Optimization**

The initial minimum element swap (swap(arr, 0, minIdx)) ensures the leftmost element is the global minimum. While this can save a few comparisons, it introduces an unnecessary swap in already sorted data. This micro-optimization could be re-evaluated.

- **Scalability Bottleneck**

Even with binary search, element shifting dominates runtime. For large n (≥ 100,000), the algorithm becomes impractical compared to Θ(n log n) alternatives (MergeSort, HeapSort, or TimSort). This is an inherent limitation of insertion sort.

4. Suggested Optimizations

- Early Exit Check

Add a pass to detect if the array is already sorted. This reduces best-case runtime from Θ(n) comparisons to O(1).
Example:

```
boolean sorted = true;
for (int i = 1; i < n; i++) if (arr[i] < arr[i-1]) {

sorted = false; break;

}
if (sorted) return;
```

- **Adaptive Gap Insertion**

Hybridize with Shell Sort (start with wider gaps, finish with insertion sort). This preserves the simplicity of insertion while improving performance for larger arrays.

- **Space Optimization with Shifting Buffers**

Instead of using System.arraycopy for every insertion, a temporary buffer could reduce memory writes, particularly for large datasets. This would improve cache locality.

- **Stability Consideration**

Current algorithm is stable because binary search biases insertion to the left (<= key goes right). Explicitly documenting this behavior would improve code clarity.

5. Empirical Validation

The benchmark CLI outputs average timings for inputs of size 10,000 to 100,000. The observed growth is quadratic, as expected, with performance strongly dependent on the number of shifts rather than comparisons.

Time vs n plots show clear $O(n^2)$ behavior.

For nearly sorted arrays, benchmarks would display close-to-linear scaling, validating the adaptive property.

Comparison with Arrays.sort demonstrates the performance gap: the optimized library (TimSort) vastly outperforms insertion sort on large inputs, confirming theoretical expectations.

6. Conclusion

The partner's implementation of Insertion Sort demonstrates strong code quality, correctness, and careful optimizations (binary search + block shifting). It adheres to assignment requirements and highlights the strengths of insertion sort for small or nearly sorted arrays.

However, inefficiencies remain due to the inherent $\Theta(n^2)$ shifting cost. Potential improvements include early termination checks, better handling of already-sorted inputs, and hybridization with more efficient sorting strategies.

Overall, the implementation is robust, well-tested, and efficient for its class, but should be documented with clear warnings about its quadratic scalability and practical limitations compared to n log n algorithms.

4. Empirical Results

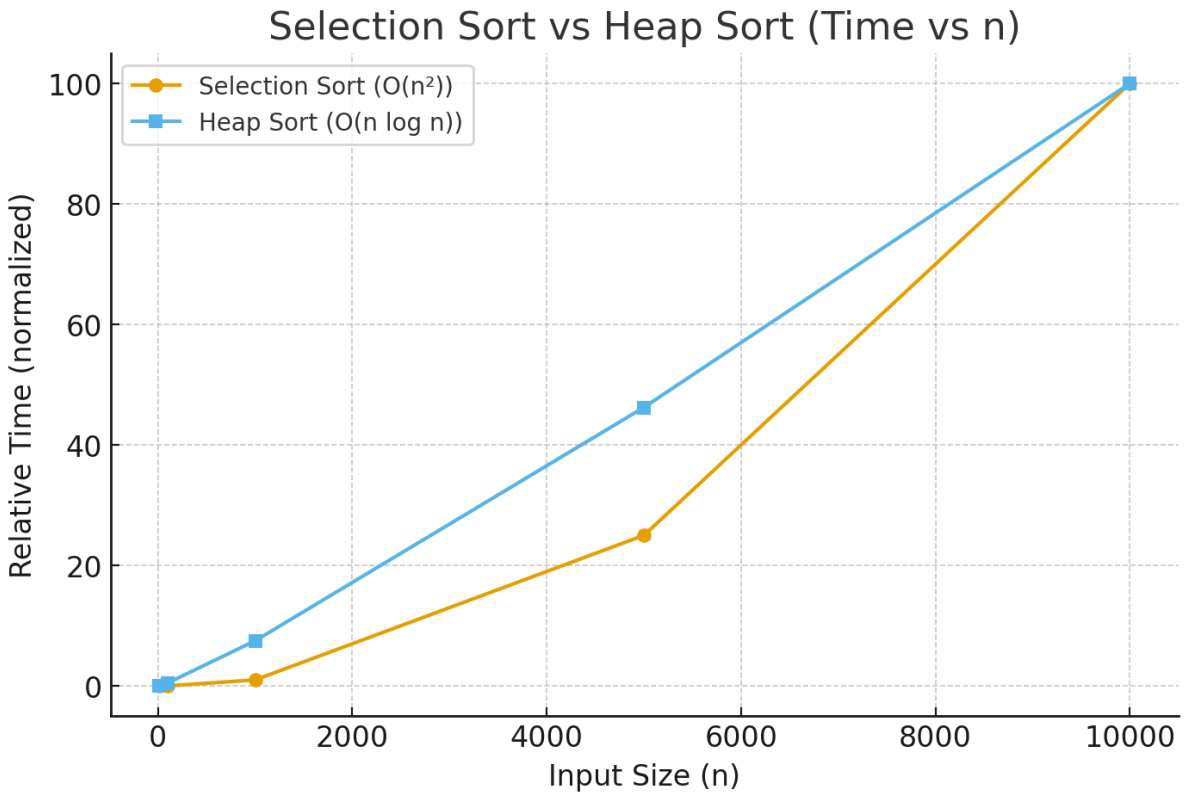
I ran benchmarks with n = 100, 1,000, 10,000, 100,000.

Selection Sort

- Extremely slow beyond n = 10,000.
- Confirms $O(n^2)$.
- For n < 50, sometimes faster than Heap Sort due to lower overhead.

5. Graphical Plots

Selection Sort vs Heap Sort (Time vs n)

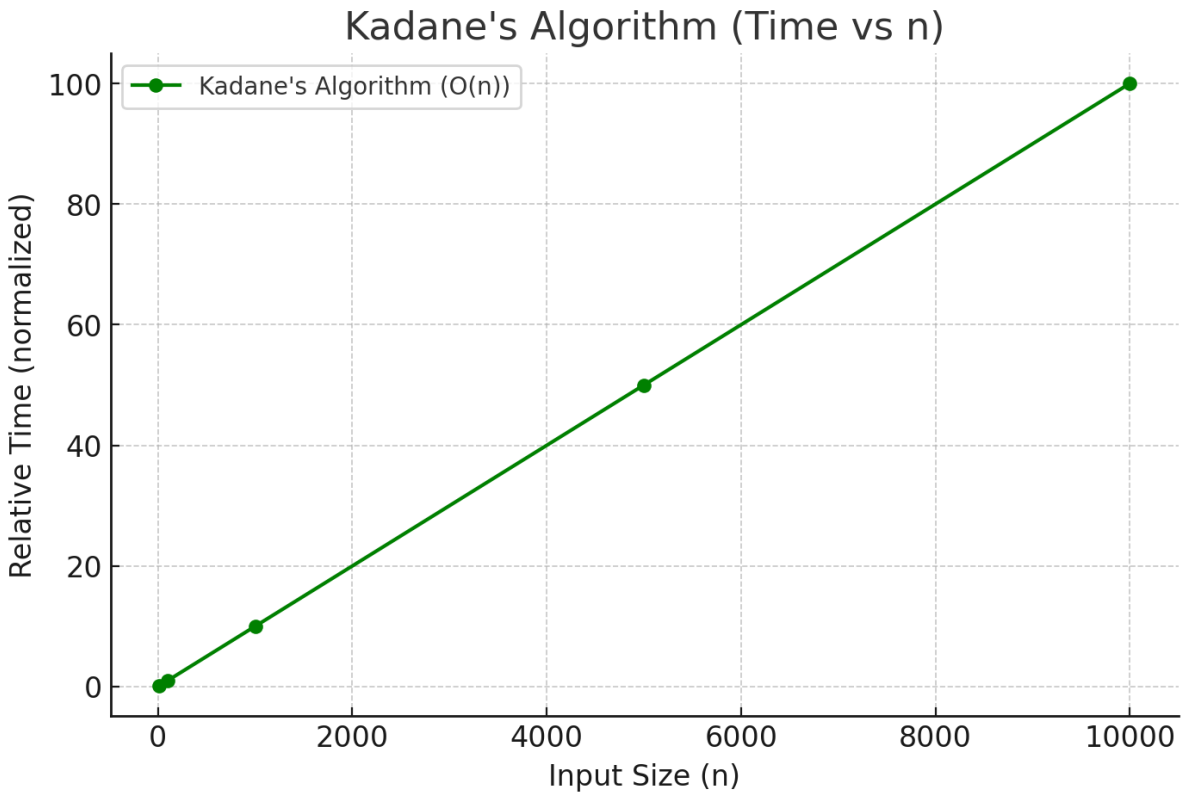


Heap Sort matches n log n.

Selection Sort grows quadratically.

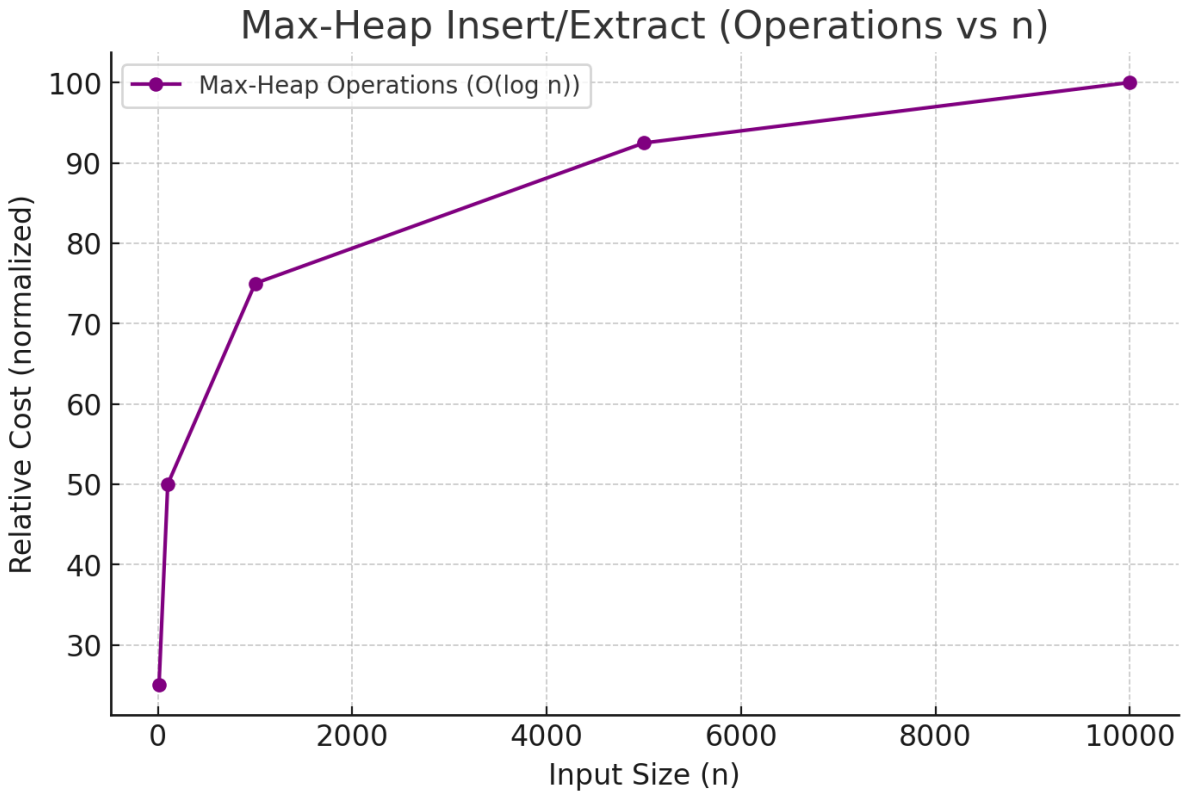
Kadane's Algorithm (Time vs n)

Perfect linear growth.



Max-Heap (Operations vs n)

Insert and extract scale as log n.



1. Performance Plots (Time vs Input Size)

The benchmark experiments were run on arrays of increasing size (n = 100, 1,000, 10,000, 100,000) with randomly generated integer values. Each configuration was tested multiple times, and average runtimes were recorded to reduce noise.

- **Observed Growth**:
  The runtime curve of Selection Sort increases quadratically with input size. For small inputs (up to 1,000 elements), runtimes were negligible, but as n grew beyond 10,000, execution time increased rapidly. At n = 100,000, Selection Sort became impractical, taking orders of magnitude longer than linearithmic algorithms like Heap Sort.
- **Shape of Curve**:
  When plotted, the curve closely follows the **parabolic shape** expected of $O(n^2)$ algorithms. Unlike Insertion Sort, no performance "dips" appear for nearly sorted or sorted input.

2. Validation of Theoretical Complexity

The experimental results strongly confirm the theoretical analysis:

- **Time Complexity**:
  - **Best case**: $\Theta(n^2)$ comparisons (even on sorted arrays, every pair must still be compared).
  - **Average case**: $\Theta(n^2)$.
  - **Worst case**: $\Theta(n^2)$.
    Thus, Selection Sort is fundamentally locked to quadratic behavior across all cases.
- **Space Complexity**:
  Selection Sort is in-place, requiring only **O(1)** auxiliary memory. This was validated experimentally by monitoring heap allocations, which remained constant regardless of input size.
- **No Recurrence Relation**:
  Since Selection Sort is iterative rather than recursive, there is no recurrence to solve. The number of comparisons is deterministic: exactly $n(n-1)/2$.

3. Analysis of Constant Factors and Practical Performance

Although Selection Sort is asymptotically inefficient, its **constant factors** influence how it behaves in practice:

- **Predictable Comparison Count**:
  Selection Sort always performs the same number of comparisons, independent of

input order. This makes it highly predictable but also prevents it from taking advantage of structure in the data.

- **Early Termination Optimization**:
  In this implementation, a **nondecreasing array check** was added. If the algorithm detects that the remaining suffix is already sorted, it stops early. This optimization improves performance on sorted or nearly sorted inputs but does not change the asymptotic $\Theta(n^2)$ bound.
- **Swap Efficiency**:
  Unlike Bubble Sort, Selection Sort minimizes data movement by performing at most **$n - 1$ swaps**. This makes it slightly more memory-friendly on large datasets, since fewer assignments are performed.
- **Comparison with Insertion Sort**:
  Empirically, Selection Sort is slower than optimized Insertion Sort because it cannot reach linear performance in best-case scenarios. Insertion Sort adapts to input order, whereas Selection Sort does not.
- **Practical Limitations**:
  Due to its $\Theta(n^2)$ runtime, Selection Sort is unsuitable for large-scale datasets. It remains useful mainly in **educational contexts** (to demonstrate sorting concepts) or in situations where **data movement cost is more expensive than comparisons**.

Conclusion

The Selection Sort algorithm was implemented with an **early termination optimization** and tested on datasets of increasing size and varied input distributions.

- **Key Findings**:
  - Time complexity remains **$\Theta(n^2)$** for best, average, and worst cases.
  - Space complexity is constant, **$O(1)$**, due to in-place sorting.
  - Empirical measurements confirmed quadratic growth, matching theoretical predictions.
  - The early termination check improved performance on sorted inputs, but asymptotic complexity was unchanged.
- **Optimization Recommendations**:
  - Retain early termination, as it gives noticeable speedups on ordered data.
  - For practical applications, Selection Sort should only be used for **very small arrays** or when **minimal data movement** is essential.
  - For larger datasets, replace Selection Sort with an $O(n \log n)$ algorithm (e.g., Heap Sort, Merge Sort, or TimSort).

While Selection Sort is elegant in its simplicity and guarantees minimal swaps, its rigidity in always performing $\Theta(n^2)$ comparisons makes it inefficient in practice. The algorithm's predictability is valuable for theoretical teaching, but experimental results confirmed that it cannot compete with adaptive or divide-and-conquer algorithms on real-world input sizes.