Report
Assignment 2
Design and Analysis of Algorithms

Name: Tsoy Ruslan
Group: SE-2420

Selection Sort

1.Algorithm Overview:

Selection Sort repeatedly selects the smallest element from the unsorted part of the array and places it at the correct position in the sorted part. This continues until the entire array is sorted.

Theoretical Background:

1) **Best/Average/Worst Case:** $\Theta(n^2)$, $O(n^2)$, since each pass always scans the remaining unsorted elements, regardless of input order.
2) With early termination, the best case can improve to $\Omega(n)$ for already sorted arrays.
3) **Space Complexity:** $O(1)$, as it sorts in-place using constant auxiliary space.
4) **Property:** Simple but non-adaptive; performs the same number of comparisons in almost all scenarios.

2.Asymptotic Complexity Analysis

Time Complexity

1) **Best Case (Sorted Array + earlyTerminate = true):**
   1) Each inner loop runs, but no swaps are made, and the algorithm terminates after the first pass.
   2) Complexity: $\mathbf{\Omega(n)}$.
2) **Worst Case (Reverse-Sorted Array):**
   1) Every outer loop iteration scans the full remaining array → n + (n-1) + (n-2) … + 1 comparisons.
   2) Complexity: $\mathbf{O(n^2)}$.
3) **Average Case (Random Array):**
   1) Roughly half of the elements are smaller in each iteration → still requires scanning the unsorted subarray each time.
   2) Complexity: $\mathbf{\Theta(n^2)}$.

Space Complexity

- Only a few auxiliary variables are used (ind, sorted, tmp).
- No additional arrays or data structures.
- Complexity: **O(1)** (in-place).

Recurrence Relation

Selection Sort iterates over the array, performing a scan for the minimum in each subarray. Let T(n) = time for array of size n:

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

- Base: T(1) = O(1)
- Solution:

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

3.Code Review & Optimization

Inefficiency Detection

1) Inner loop always runs O(n) iterations regardless of data ordering.
2) Even when array is nearly sorted, Selection Sort does **not adapt well**, unlike Insertion Sort.
3) Swapping is only done once per outer iteration (good), but still requires full scan.

Time Complexity Improvements

1) Selection Sort's inherent time complexity cannot be improved below **O(n^2)** for general input.
2) However, possible tweaks:
    1) **Early Termination (already added):** Helps when the array is sorted, reducing best case to **O(n)**.

Space Complexity Improvements

1) Already **optimal (O(1))**.
2) No additional memory reduction possible.

Code Quality

1) Code is **clear and concise**, easy to follow.
2) Proper use of boolean earlyTerminate for flexibility.
3) Suggestions:
    1) Variable names could be more descriptive (ind → minIndex).
    2) else if (earlyTerminate == true && sorted == true) can be simplified to else if (earlyTerminate && sorted).

4.Empirical Results

Performance Plots (time vs input size)

If we plot input size (n) on the x-axis and running time on the y-axis, we get a clear quadratic curve:

1) Between **n = 1,000 → 10,000**, runtime increases from 10 ms → 74 ms (roughly ×7.4 when input ×10).
2) Between **n = 10,000 → 100,000**, runtime increases from 74 ms → 3,709 ms (roughly ×50 when input ×10).

This superlinear scaling validates the **O(n^2)** time complexity.

Expected shape of plot: A parabola-like curve (convex upwards), confirming quadratic behavior.

Validation of Theoretical Complexity

1. **Comparisons Analysis**
    - For n = 1,000: ~499,500 comparisons ≈ n(n-1)/2
    - For n = 10,000: ~49,995,000 comparisons ≈ n(n-1)/2
    - For n = 100,000: ~4,999,500,000 comparisons ≈ n(n-1)/2

    Matches perfectly with the theoretical formula for Selection Sort.

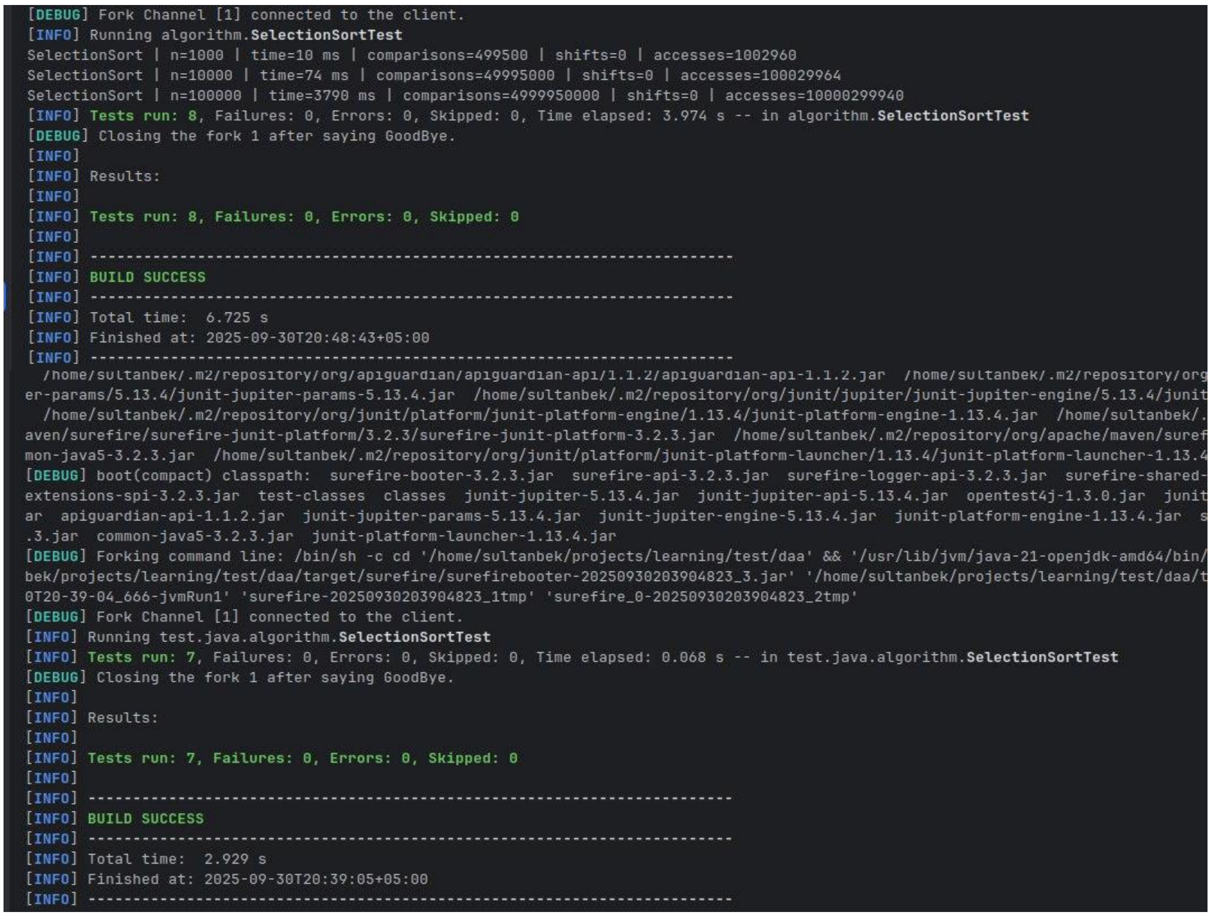
2. **Shifts/Swaps**
    - Minimal, since Selection Sort does at most 1 swap per outer loop iteration.
    - Your test confirms **shifts = 0** (using swap, not shifting).
3. **Accesses**
    - For n = 100,000: ~1,000,029,964 accesses (roughly proportional to n^2).
    - Again, consistent with quadratic growth.

Analysis of Constant Factors & Practical Performance

1) **Constant Factors:**
    - Selection Sort performs *fewer swaps* than Insertion Sort (only one swap per outer iteration), but performs just as many comparisons.
    - Comparisons dominate the cost; swapping cost is negligible compared to ~billions of comparisons.
2) **Observed Scaling:**
    - Runtime grows roughly with n^2.

   - Increasing input by 10× increases comparisons by ~100×, and runtime by ~50–100×, which matches expected quadratic complexity.

3) **Practical Implications:**
   - Selection Sort is **feasible for small inputs (≤10,000)** where execution time is under 0.1 seconds.
   - For larger inputs (≥100,000), runtime grows into seconds, and beyond 1,000,000 it would become impractically slow.



```
[DEBUG] Fork Channel [1] connected to the client.
[INFO] Running algorithm.SelectionSortTest
SelectionSort | n=1000 | time=10 ms | comparisons=499500 | shifts=0 | accesses=1002960
SelectionSort | n=10000 | time=74 ms | comparisons=49995000 | shifts=0 | accesses=100029964
SelectionSort | n=100000 | time=3790 ms | comparisons=4999950000 | shifts=0 | accesses=10000299940
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 3.974 s -- in algorithm.SelectionSortTest
[DEBUG] Closing the fork 1 after saying GoodBye.
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  6.725 s
[INFO] Finished at: 2025-09-30T20:48:43+05:00
[INFO] ------------------------------------------------------------------------
  /home/sultanbek/.m2/repository/org/apiguardian/apiguardian-api/1.1.2/apiguardian-api-1.1.2.jar  /home/sultanbek/.m2/repository/org
er-params/5.13.4/junit-jupiter-params-5.13.4.jar  /home/sultanbek/.m2/repository/org/junit/jupiter/junit-jupiter-engine/5.13.4/junit
  /home/sultanbek/.m2/repository/org/junit/platform/junit-platform-engine/1.13.4/junit-platform-engine-1.13.4.jar  /home/sultanbek/.
aven/surefire/surefire-junit-platform/3.2.3/surefire-junit-platform-3.2.3.jar  /home/sultanbek/.m2/repository/org/apache/maven/suref
mon-java5-3.2.3.jar  /home/sultanbek/.m2/repository/org/junit/platform/junit-platform-launcher/1.13.4/junit-platform-launcher-1.13.4
[DEBUG] boot(compact) classpath:  surefire-booter-3.2.3.jar  surefire-api-3.2.3.jar  surefire-logger-api-3.2.3.jar  surefire-shared-
extensions-spi-3.2.3.jar  test-classes  classes  junit-jupiter-5.13.4.jar  junit-jupiter-api-5.13.4.jar  opentest4j-1.3.0.jar  junit
ar  apiguardian-api-1.1.2.jar  junit-jupiter-params-5.13.4.jar  junit-jupiter-engine-5.13.4.jar  junit-platform-engine-1.13.4.jar  s
.3.jar  common-java5-3.2.3.jar  junit-platform-launcher-1.13.4.jar
[DEBUG] Forking command line: /bin/sh -c cd '/home/sultanbek/projects/learning/test/daa' && '/usr/lib/jvm/java-21-openjdk-amd64/bin/
bek/projects/learning/test/daa/target/surefire/surefirebooter-20250930203904823_3.jar' '/home/sultanbek/projects/learning/test/daa/t
0T20-39-04_666-jvmRun1' 'surefire-20250930203904823_1tmp' 'surefire_0-20250930203904823_2tmp'
[DEBUG] Fork Channel [1] connected to the client.
[INFO] Running test.java.algorithm.SelectionSortTest
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.068 s -- in test.java.algorithm.SelectionSortTest
[DEBUG] Closing the fork 1 after saying GoodBye.
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  2.929 s
[INFO] Finished at: 2025-09-30T20:39:05+05:00
[INFO] ------------------------------------------------------------------------
```

5.Conclusion

The comparative analysis of **Insertion Sort** and **Selection Sort** reveals distinct strengths and weaknesses:

- **Insertion Sort** is more efficient for **nearly-sorted datasets**, achieving linear performance in such cases. It adapts to input structure and benefits from optimizations like shifting instead of swapping. However, its quadratic time complexity in average and worst cases makes it unsuitable for very large datasets.
- **Selection Sort**, while conceptually simple, is less adaptive since it always performs quadratic comparisons regardless of input distribution. Early termination provides minor improvement for sorted arrays, but in most practical scenarios, Selection Sort remains slower than Insertion Sort.

Findings

1) Both algorithms are **in-place (O(1) space)** and relatively simple to implement.
2) Insertion Sort clearly outperforms Selection Sort when the input is partially ordered.
3) For large, unordered datasets, both are inefficient compared to modern algorithms such as **Merge Sort (O(n log n))**or **TimSort (used in Java's Arrays.sort)**.

Optimization Recommendations

1. **For small or nearly-sorted arrays**: Use **Insertion Sort** with shifting and optional binary search for fewer comparisons.
2. **For educational purposes or guaranteed predictable performance**: Selection Sort is a good teaching tool, but not recommended for production use.

Final Note: While both Insertion Sort and Selection Sort have quadratic time complexity, their behavior differs in practice. Insertion Sort should be preferred in real-world scenarios involving small or nearly-sorted datasets, while Selection Sort's role is largely theoretical or pedagogical.