Analysis Report: Selection Sort

Analysis Author: Shomanov Rakhat (Student A, Pair 1)

Analyzed Algorithm: SelectionSort.java (implementation by Student B, Tomiris Nauryzbai, Pair 1)

1. Algorithm Overview

Selection Sort is a simple, intuitive, comparison-based sorting algorithm. Its operation involves repeatedly traversing the unsorted part of an array to find the minimum (or maximum) element and swapping it with the first element of that part. This process is repeated for the remaining subarray until the entire array is sorted.

Key characteristics of the algorithm:

- **In-place:** Does not require additional memory proportional to the size of the input data.
- **Unstable:** May change the relative order of equal elements.
- **Low adaptivity:** The execution time is practically independent of the initial order of the data.

2. Asymptotic Complexity Analysis

The analysis is based on the provided implementation in SelectionSort.java.

Time Complexity

The algorithm consists of two nested loops.

- The outer loop for (int i = 0; i < n - 1; i++) executes **n-1** times.
- The inner loop for (int j = i + 1; j < n; j++) searches for the minimum element. The number of iterations decreases with each step of the outer loop: (n-1), (n-2), ..., 1.

The total number of comparisons if (arr[j] < arr[minIndex]) can be calculated as the sum of an arithmetic progression:
$C(n)=i=0\Sigma n-2(n-i-1)=(n-1)+(n-2)+\cdots+1=2(n-1)n=2n2-n$
The number of swaps is always constant and equal to **n-1**, as exactly one swap occurs in each iteration of the outer loop.

- Best Case: $\Omega(n^2)$
  Even if the array is already sorted, the algorithm cannot "know" this. It will still perform the full number of comparisons to find the minimum element in each pass. The complexity remains quadratic.
- Average Case: $\Theta(n^2)$
  For a randomly ordered array, the number of comparisons and swaps remains the same.
- Worst Case: $O(n^2)$

For a reverse-sorted array, the algorithm will perform the same number of operations.

Conclusion: The time complexity of the algorithm is insensitive to the initial data order and is always quadratic.

Space Complexity

The algorithm requires a fixed amount of additional memory to store variables (n, i, minIndex, j, temp). This amount does not depend on the size of the input array n.

Conclusion: The space complexity is **O(1)**.

3. Code Review and Optimization Suggestions

1. The implementation correctly demonstrates the Selection Sort algorithm. To fully meet the assignment requirements, it could be improved by adding an early termination optimization for example, a check to stop when the remaining elements are already sorted. This small enhancement would make the solution more efficient and closer to the intended task.
    - **Suggestion:** Implement a check after the inner loop. If minIndex has not changed (i.e., minIndex == i), it means the first element of the subarray is already the minimum. A flag isSorted could be added, and if an entire pass of the outer loop completes without any swaps, the array is sorted, and the process can be terminated.
2. The PerformanceTracker class currently measures only the total execution time, which is a good start. To fully meet the assignment requirements, it would be great to also include additional metrics such as the number of comparisons, swaps, and array accesses. This would provide a more complete view of the algorithm's performance.
    - **Suggestion:** Refactor the PerfomanceTracker class to include counters for each operation (comparisons, swaps, arrayAccesses). These counters should be incremented within the SelectionSort.sort() algorithm. This will allow for a deep analysis, not just a superficial time measurement.
3. The current implementation successfully tests the algorithm on random data, which is a solid starting point. To fully meet the assignment requirements, it would be helpful to also include tests on sorted, reversed, and nearly-sorted data that would allow for a more complete performance comparison. Additionally, the array sizes are currently hardcoded. Introducing a simple command-line interface to specify the array size and data type would make the testing process more flexible
    - **Suggestion:** Modify the BenchmarkRunner to be similar to my implementation: it should accept the array size and data type from the command line and generate the corresponding arrays for testing.

4. Empirical Validation

The partner provided benchmark results in the README.md for random arrays up to size 10,000.

Size	Time (ms)
100	0.49
1000	3.31
5000	15.73
10000	36.06

These results indeed demonstrate a **quadratic relationship**. For example, when n increases by a factor of 10 (from 1000 to 10000), the execution time increases by a factor of approximately 100 (from ~0.33 ms to ~36 ms), which is close to the theoretical estimate of $(10n)^2 / n^2 = 100$.

Plan for Full Empirical Validation:

1. Implement the suggested improvements in BenchmarkRunner and PerformanceTracker.
2. Run benchmarks for sizes n = 100, 1000, 10000, 100000.
3. Run benchmarks for all data types: random, sorted, reversed, nearly-sorted.
4. Plot graphs of execution time versus n. It is expected that all curves will be nearly identical and follow a parabolic trajectory, confirming the $O(n^2)$ complexity.

5. Conclusion

The implementation of the SelectionSort.java algorithm demonstrates a clear understanding of the fundamental sorting principles and is logically correct. The program performs as expected and successfully passes all provided unit tests, showing that the main algorithmic logic is stable and reliable. However, while the core functionality is correct, the project only partially fulfills some of the key assignment requirements. Specifically, the current implementation lacks the early termination optimization that could improve efficiency in partially sorted arrays. In addition, the PerformanceTracker currently measures only the total execution time, whereas the assignment requires collecting additional metrics such as the number of comparisons, swaps, and array accesses to allow for a more detailed performance analysis. Testing is currently limited to random data, which is a solid start, but further experiments on sorted, reversed, and nearly-sorted arrays would provide a broader and more accurate comparison of algorithm behavior under different conditions. Another area for improvement is the inclusion of a command-line interface that would allow users to specify the array size and data type dynamically instead of relying on hardcoded values. Overall, the project demonstrates a solid foundation and a correct understanding of Selection Sort.