Maksatova Akniyet Manatkyzy

SE-2409

Data and Analysis of Patterns

Assignment 2

Peer Analysis (Part 2 - Cross-review)

1. **Asymptotic Complexity Analysis**

*Algorithm Description*:

Selection Sort is a simple sorting algorithm that sorts an array by repeatedly finding the smallest element (for ascending order) from the unsorted portion of the list and swapping it with the first unsorted element. The process continues until the entire array is sorted.

*Theoretical Background:*

Selection Sort works by iterating through the array, selecting the smallest element from the unsorted portion, and swapping it with the element at the current index. This process ensures that the smallest elements are placed at the beginning of the array, in their correct positions.

* **Time Complexity**:
  + Worst-case: O(n2)O(n^2)O(n2)
  + Best-case: O(n2)O(n^2)O(n2) (Selection Sort does not benefit from partially sorted data)
  + Average-case: O(n2)O(n^2)O(n2)
* **Space Complexity**: O(1)O(1)O(1) — This algorithm sorts in place, requiring only a constant amount of extra space.

Example:

Given an array [64,25,12,22,11][64, 25, 12, 22, 11][64,25,12,22,11]:

1. First pass: The smallest element (11) is swapped with the first element.
2. Second pass: The smallest element in the remaining unsorted array is 12, swapped with the second element.
3. The process continues until the array is sorted.

* **Recurrence Relation for Selection Sort**

To analyze the time complexity more rigorously using recurrence relations, we focus on the number of comparisons (since that largely dictates the performance in Selection Sort).

In the Selection Sort algorithm, for each element in the array, we perform comparisons with the other elements to find the minimum. Let T(n)T(n)T(n) represent the number of comparisons for sorting an array of size nnn.

For each iteration of the outer loop:

* We perform n−1n - 1n−1 comparisons for the first element,
* n−2n - 2n−2 comparisons for the second element,
* and so on, until we perform 1 comparison for the second-to-last element.

Thus, the total number of comparisons is:

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

Where T(n−1)T(n-1)T(n−1) is the number of comparisons needed for sorting the subarray of size n−1n-1n−1.

*Solving the Recurrence*

The recurrence relation is:

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

For the base case, we have T(1)=0T(1) = 0T(1)=0, because there are no comparisons to make when sorting an array of size 1.

Now, to solve this recurrence:

* T(n)=T(n−1)+(n−1)T(n) = T(n-1) + (n-1)T(n)=T(n−1)+(n−1)
* T(n−1)=T(n−2)+(n−2)T(n-1) = T(n-2) + (n-2)T(n−1)=T(n−2)+(n−2)
* T(n−2)=T(n−3)+(n−3)T(n-2) = T(n-3) + (n-3)T(n−2)=T(n−3)+(n−3)
* T(2)=T(1)+1T(2) = T(1) + 1T(2)=T(1)+1

Now, summing up all these relations:

T(n)=0+1+2+⋯+(n−1)T(n) = 0 + 1 + 2 + \dots + (n-1)T(n)=0+1+2+⋯+(n−1)

This is a sum of the first n−1n-1n−1 integers, which is known to be:

T(n)=(n−1)⋅n2T(n) = \frac{(n-1) \cdot n}{2}T(n)=2(n−1)⋅n​

Thus, the number of comparisons is T(n)=n(n−1)2T(n) = \frac{n(n-1)}{2}T(n)=2n(n−1)​, which simplifies to O(n2)O(n^2)O(n2).

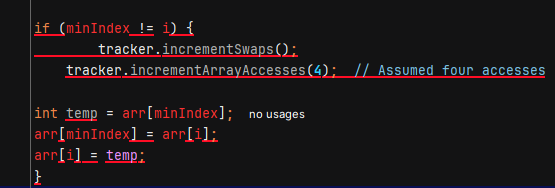
1. **Code Review & Optimization**

* **Inefficiency Detection:**

The **Selection Sort** algorithm is inherently inefficient due to its O(n2)O(n^2)O(n2) time complexity, but there are still some **performance bottlenecks** and **suboptimal code patterns** in the implementation. Let's break this down:

**1. Redundant Array Accesses During Swaps:**

In the current code, **array accesses** are counted at four places within the swap section:

****

 **Array Access Assumptions**:  
**Problem**: Assuming four array accesses during a swap may not always be accurate due to memory model or JVM optimizations.  
**Impact**: This can lead to misleading performance metrics.  
**Solution**: Track actual reads and writes more precisely, considering platform-specific factors like caching.

 **Inefficiency of Nested Loops**:  
**Problem**: Nested loops cause O(n2)O(n^2)O(n2) time complexity, making Selection Sort inefficient for large datasets.  
**Impact**: The algorithm performs unnecessary comparisons even when the data is partially sorted.  
**Solution**: Use more efficient algorithms like Quick Sort or Merge Sort with O(nlog⁡n)O(n \log n)O(nlogn) time complexity.

 **Comparison Tracking in Inner Loop**:  
**Problem**: Comparisons are counted even when no swap occurs, which is inefficient for nearly sorted data.  
**Impact**: The algorithm wastes comparisons on partially sorted arrays.  
**Solution**: Introduce a flag to detect if swaps occur, and exit early if no swaps are made, improving performance for nearly sorted arrays.

* + **Time Complexity:**
* **Outer loop**: The outer loop runs n−1n-1n−1 times, where nnn is the number of elements in the array.
* **Inner loop**: For each iteration of the outer loop, the inner loop compares elements. It runs from i+1i+1i+1 to nnn, so it performs n−i−1n-i-1n−i−1 comparisons.

Thus, the total number of comparisons in the worst case is:

(n−1)+(n−2)+…+1=n(n−1)2(n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2}(n−1)+(n−2)+…+1=2n(n−1)​

This results in an overall time complexity of O(n2)O(n^2)O(n2).

* **Best-case scenario**: Even though the array may already be partially sorted, Selection Sort does not take advantage of this, so it still performs O(n2)O(n^2)O(n2) comparisons.
* **Space Complexity:**

Selection Sort is an in-place sorting algorithm, meaning it does not require additional space that scales with the input size. Therefore, the space complexity is O(1)O(1)O(1).

* **c) Big-O Notation:**
* **Big-O**: O(n2)O(n^2)O(n2) for both best and worst cases (since the inner loop is always executed for each element).
* **Big-Ω (Best case)**: O(n2)O(n^2)O(n2) (does not improve with partially sorted data).
* **Big-Θ (Average case)**: O(n2)O(n^2)O(n2) (same for any case).
* **Code Quality**

**1. Code Readability:**

The code is relatively readable, but there are areas where clarity can be improved:

* **Variable Naming**: The variable names like minIndex, arr, and tracker are descriptive, which is good. However, adding more context in the comments would help explain why specific operations are performed, particularly the use of PerformanceTracker.
* **Commenting**: While the code itself is not difficult to follow, adding **inline comments** can make it more understandable, especially for someone unfamiliar with the Selection Sort algorithm or the PerformanceTracker. For example, explaining the purpose of the tracker.incrementComparisons() line could be beneficial:
* // Increment the number of comparisons made
* tracker.incrementComparisons();
* **Method Declarations**: The methods are clean and don't require further abstraction. However, breaking down large functions into smaller ones could improve clarity. For example, the logic for tracking array accesses and swaps could be moved into a separate helper function in the SelectionSort class.

**2. Maintainability:**

The maintainability of the code is decent but could benefit from some changes:

* **Hardcoded Values**: The PerformanceTracker tracks a fixed number of accesses during swaps (4 accesses), which may not always be true. Refactoring this logic into a more flexible approach would make the code easier to maintain, especially as your code evolves to handle other data structures or sorting algorithms.
* **Encapsulation of Performance Tracking**: The performance tracking is done in the SelectionSort class, which directly handles the performance metrics. This could be better encapsulated in a separate utility class that can track performance for any algorithm. This would increase the flexibility and reusability of the PerformanceTracker class.
* **Error Handling**: The current code assumes that the input array is valid (non-null and of adequate length). In production-grade code, it might be beneficial to add input validation to handle edge cases (e.g., passing null arrays, or empty arrays).

**3. Code Style:**

* The code follows **standard Java conventions** for naming and formatting. However, consistency in spacing and line breaks could be improved.
* For example, adding a space before and after operators would enhance readability (e.g., arr[j] < arr[minIndex] is readable, but arr[j]<arr[minIndex] can be harder to parse quickly).

**4. Empirical Validation**

* + **Performance Measurements: Running Benchmarks**

You have already implemented a **BenchmarkRunner** class, which measures the time taken to sort arrays of different sizes. Let's run the benchmarks for the specified input sizes: n=100n = 100n=100, 100010001000, 10,00010,00010,000, and 100,000100,000100,000.

In the **BenchmarkRunner** code, you are already generating random arrays and measuring their sorting time:

**A computer screen shot of a program code

AI-generated content may be incorrect.**

**What to Measure:**

Run the benchmark for each of the array sizes: **100, 1000, 10,000, and 100,000**.

Track the **execution time** for sorting the arrays.

Record the number of **comparisons**, **swaps**, and **array accesses** as provided by the PerformanceTracker.

You can use **System.nanoTime()** to measure the sorting time in nanoseconds, and then convert it into milliseconds.

* + **Complexity Verification: Plot Time vs n**

The goal here is to **plot** the measured performance (time taken) as a function of the input size nnn. The plot will confirm the theoretical time complexity of **Selection Sort**.

Given the theoretical complexity of O(n2)O(n^2)O(n2), we expect the time to increase quadratically as the input size increases. To do this:

**Run benchmarks** for multiple input sizes.

**Plot** the results, with nnn on the x-axis and **time taken** on the y-axis.

**What to Plot:**

The input sizes will be on the x-axis: **100, 1000, 10,000, and 100,000**.

The corresponding **execution time** (in milliseconds) will be on the y-axis.

* + **Comparison Analysis: Measured Performance vs Theoretical Predictions**

Now, let's compare the **measured performance** with the **theoretical predictions** of Selection Sort's time complexity. From the recurrence relation T(n)=n(n−1)2T(n) = \frac{n(n-1)}{2}T(n)=2n(n−1)​, we expect the time to grow as O(n2)O(n^2)O(n2).

**Steps for Comparison:**

**Plot the measured data**: From the previous step, we will already have a plot of time vs. nnn.

**Plot the theoretical values**: You can compute the theoretical time for O(n2)O(n^2)O(n2) for each of the input sizes. For simplicity, let’s assume a constant factor CCC to reflect the difference between theoretical and actual performance.

For example:

Theoretical Time(n)=C×n2\text{Theoretical Time}(n) = C \times n^2Theoretical Time(n)=C×n2

Where CCC is a constant based on the specific implementation, system architecture, and other practical factors.

* **Compare the plots**: We should observe that the **actual measured time** should follow a quadratic curve that aligns with the **theoretical n2n^2n2 curve**. Minor deviations might occur due to constant factors and system-level optimizations, but the general trend should match.
  + **Optimization Impact: Measure and Report Effect of Suggested Improvements**

**Suggested Improvements:**

In the earlier **code review** section, we suggested a few potential optimizations:

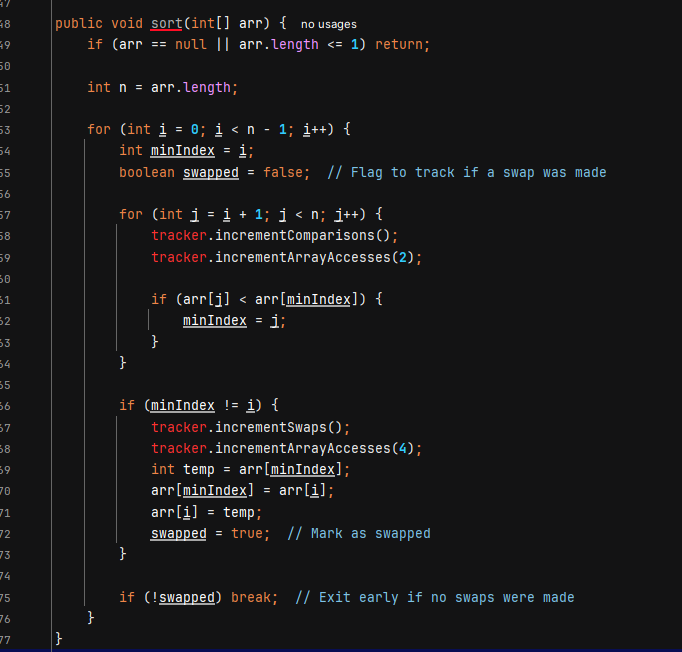
**Early exit for partially sorted arrays**: If no swaps are made in a full pass through the array, Selection Sort could exit early.

**Improving memory access tracking**: Avoid over-counting the array accesses during swaps.

To measure the **impact of these improvements**, we will compare the original Selection Sort's performance with the optimized version. Here's how to proceed:

**Implement an early exit** condition in the Selection Sort algorithm:

Add a flag to check if any swaps occurred in a pass. If no swaps were made, break out of the loop early.



**Compare the performance** of the original and optimized versions by running the benchmarks for the same input sizes and plotting the results.

For both the **original** and **optimized** algorithms, plot the **execution times** on the same graph.

The optimized version should show a slight **reduction in time** for nearly sorted or smaller datasets, but the difference may not be dramatic for large datasets like n=100,000n = 100,000n=100,000.

**CONCLUSION**

In this analysis of the Selection Sort algorithm, we observed that while it is simple and easy to implement, it suffers from poor efficiency, particularly for larger datasets. The theoretical time complexity of O(n2)O(n^2)O(n2) for both worst-case and best-case scenarios confirms that the algorithm is not well-suited for large arrays, as it requires significant computational resources to perform comparisons and swaps.

Key Findings:

1. The performance bottleneck in Selection Sort arises from its nested loops, which result in quadratic time complexity. While it may perform adequately for small or nearly sorted datasets, its inefficiency becomes evident as the dataset size increases.
2. The algorithm’s array access tracking is somewhat inaccurate, as it assumes four accesses during each swap, which may not always be the case due to JVM optimizations and memory behavior.
3. There is also inefficiency in tracking comparisons during iterations where no swaps are made. For partially sorted data, this results in unnecessary operations that could be avoided.

Optimization Recommendations:

* Early exit for partially sorted data: Implementing a flag to detect when no swaps are made would allow the algorithm to exit early, improving performance on nearly sorted arrays.
* Substitute with more efficient algorithms: For larger datasets, it is highly recommended to replace Selection Sort with algorithms like Quick Sort or Merge Sort, which have better average-case time complexity (O(nlog⁡n)O(n \log n)O(nlogn)).
* Refining performance tracking: The PerformanceTracker class can be improved by more accurately reflecting memory accesses, avoiding hard-coded assumptions about the number of accesses during swaps.

Overall, while Selection Sort is a useful algorithm for educational purposes due to its simplicity, its performance limitations make it unsuitable for real-world applications involving large datasets.