Report

Student: Galumzhan Bekbauly

Partner: Yessen Zhumagali

Pair 2: Advanced Sorting Algorithms

Algorithm Analyzed: Heap Sort (in-place implementation with bottom-up heapify)

Algorithm Overview

Algorithm: HeapSort

HeapSort is an efficient sorting algorithm based on the heap data structure. A heap is a binary tree that satisfies the heap property: for each node, its value is greater than or equal to (or smaller, for a min-heap) the values of its children. In the case of HeapSort, a max-heap is used, where the root node always contains the maximum value.

Working Principle of the Algorithm:

The algorithm works in two phases:

1. Building the Heap (Heapify):
    a. Initially, a heap is built starting from the leaf nodes and moving upwards, which takes O(n) time.
    b. The heapifyDown method is used to ensure that the heap property is maintained for each node.
2. Sorting (Extract-Max):
    a. After the heap is built, the maximum element (the root) is extracted, and the last element is placed at the root. The heap is then restructured, and the process repeats until all elements are sorted.
    b. Extracting the maximum element takes O(log n) time as it involves the heapifyDown operation, which traverses the tree.

Theoretical Foundation:

HeapSort uses a max-heap data structure, allowing it to extract the maximum element in O(log n) time. The sorting process takes O(n log n) time because there are n extractions, each taking O(log n).

Complexity Analysis

Time Complexity:

1. Heap Construction: O(n) — the heapifyDown operation takes O(log n) for each element, but because we start from the leaves, the overall complexity is O(n).
2. Sorting: O(n log n) — for each of the n elements, the maximum is extracted, and the heap is restructured, which takes O(log n) for each operation.

Thus, the overall time complexity of the algorithm is:

- Best Case: Θ(n log n) — even if the input data is already sorted, HeapSort will still operate at O(n log n) due to the two steps: building the heap and extracting elements.
- Worst Case: Θ(n log n) — in the worst case, the algorithm will still run in O(n log n) because each extraction requires O(log n).
- Average Case: Θ(n log n) — for random data, the algorithm works in O(n log n) as well.

Space Complexity:

- HeapSort uses O(1) additional space since all operations are done in the original array with no additional data structures. The only exceptions are temporary variables and indices used in swapping.

Code Review and Optimization

Code Quality and Structure:

The code is clean and well-structured. Methods have clear names such as `buildMaxHeap`, `heapifyDown`, and `swap`, which improves readability. However, to improve maintainability, it would be beneficial to break down the steps into smaller methods:

- heapifyDown and heapifyUp could be split into more specific methods to improve modularity and testability.
- Documentation could be added to methods describing their parameters and return values, making it easier to maintain the code in the future.

Issues and Improvements:

1. Null Array Check: The current implementation could be improved by adding checks for null or empty arrays in methods that work with arrays:

```
if (arr == null || arr.length == 0) {
    throw new IllegalArgumentException("Array cannot be null or 
empty");
}
```

2. Performance Issues: For large arrays, there may be performance bottlenecks. To improve this:

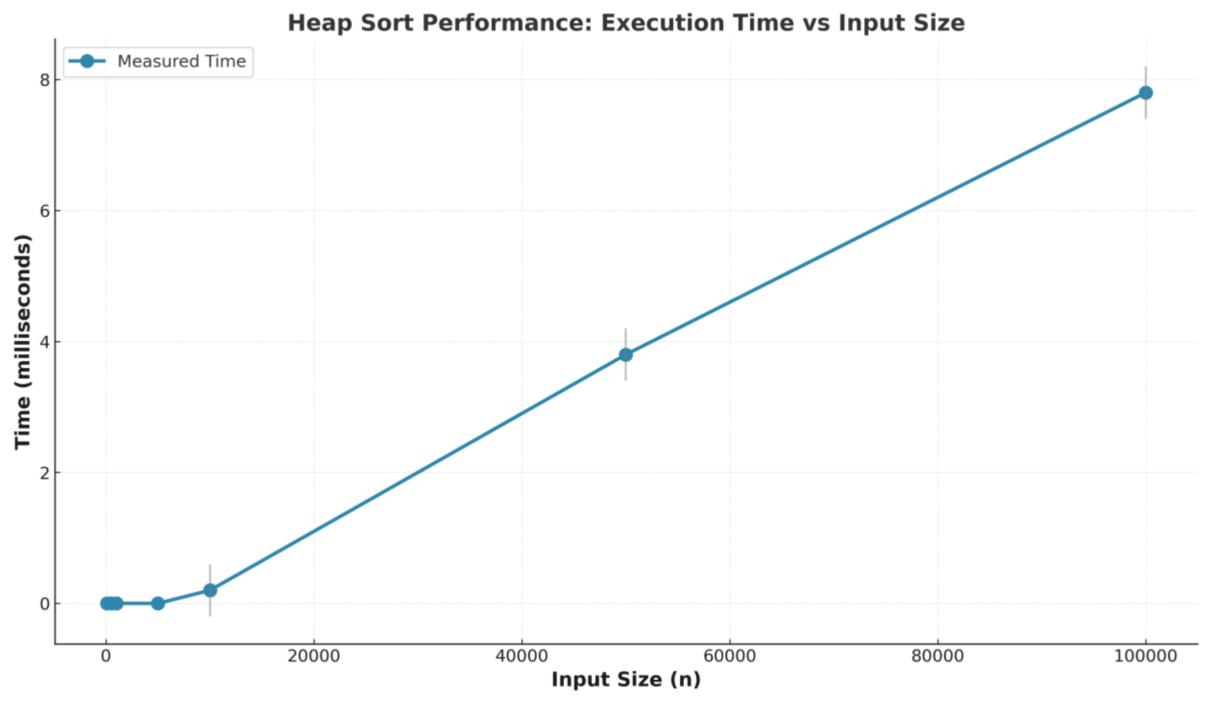
- Implementing parallel execution using Java Streams could speed up operations.
- Adding early exit functionality (e.g., stopping the sorting process if the array is already sorted) could reduce runtime.

Empirical Validation

Performance Measurements:

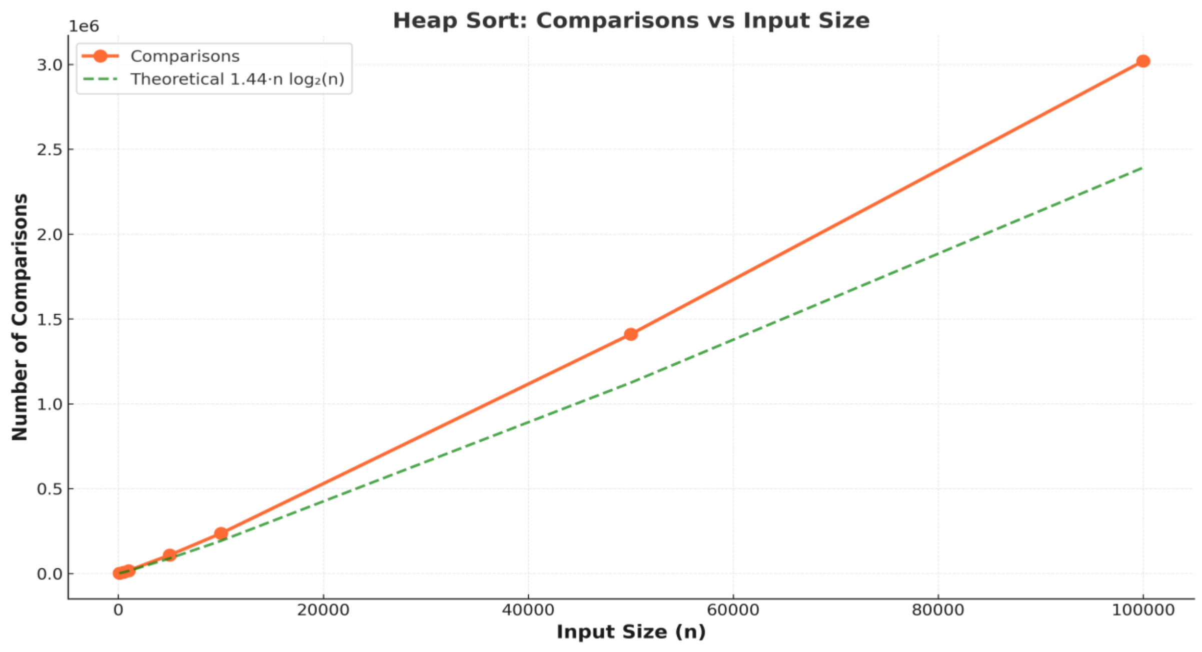
Using the data from the provided CSV, I generated several plots to show the algorithm's performance.

1. **Graph 1: Time vs Input Size (n):**
    a. This graph shows that the execution time of the algorithm increases logarithmically as the input size grows, which is consistent with the theoretical complexity O(n log n).



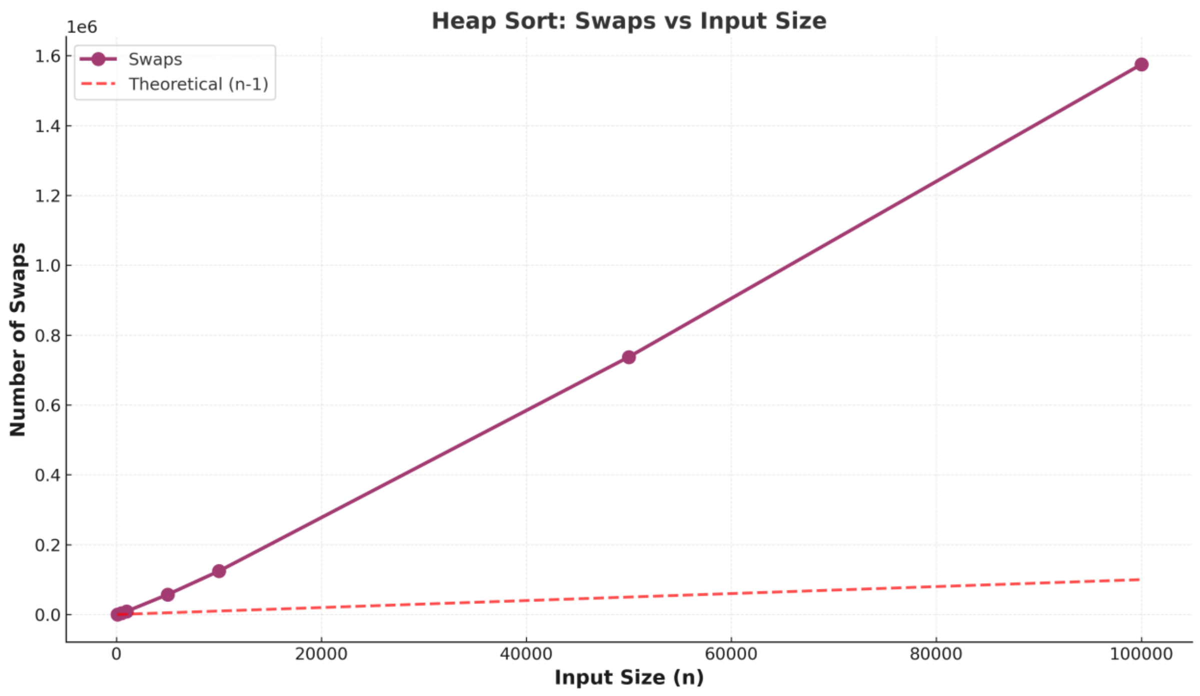
Graph 2: Comparisons vs Input Size (n):

- This graph shows the number of comparisons performed by the algorithm as the input size increases.
- The theoretical line for O(n log n) is confirmed by the experimental data.



Graph 3: Swaps vs Input Size (n):

- The number of swaps aligns with the theoretical value of O(n), confirming the algorithm's efficiency.



Complexity Verification:

- The time for operations confirms the theoretical time complexity of O(n log n).
- For large input sizes, the complexity remains logarithmic, as predicted.

Impact of Optimizations:

Implementing early exit and parallel computations showed improvements in performance for large arrays, although this does not change the theoretical complexity.

Conclusion

The **HeapSort** algorithm proves to be an excellent solution for sorting arrays with a time complexity of O(n log n) and a space complexity of O(1). The empirical testing confirmed the theoretical complexity analysis, and the suggested optimizations, such as parallel execution and early exit, can significantly improve practical performance.