Algorithm overview

HeapSort is a sorting algorithm based on the binary heap data structure.
It works according to the principle:

- Building the maximum heap from the input array.
- Re-extracting the largest element (the root of the heap) and moving it to the end of the array.
- Restoring the heap property using the "sift-down" operation.
- Repeat the step until the array is sorted.

Thus, HeapSort implements in-place sorting, that is, it does not require additional memory, except for constant auxiliary variables.

Theoretical basis:

- The heap is built in O(n) time thanks to the bottom-up heapify method.
- The best, average, and worst cases have the same complexity of Θ(n log n), making HeapSort a predictable algorithm
- The space complexity is O(1), as sorting in

Time complexity of HeapSort

Happening	Θ(n log n)	Ω(n log n)	O(n log n)
Best	✓	✓	✓
Average	✓	✓	✓
Worst	✓	✓	✓

Rationale:

- Heap construction: O(n)
- Element extraction: n × O(log n) = O(n log n)
- Total complexity: Θ(n log n)

Space complexity

HeapSort is performed in-place, with O(1) memory overhead.

Comparison with partner's algorithm complexity

Algorithm	Best case	Average case	Worst case	Memory
HeapSort	Θ(n log n)	Θ(n log n)	Θ(n log n)	O(1)
ShellSort	Θ(n log n)	depends on the gap	$O(n^2)$	O(1)

Conclusion: HeapSort is predictable and stable, ShellSort is faster for almost sorted data, but the worst case is worse.

Code Review

Identifying Inefficiencies:

1. Function siftDown

- Many unnecessary array accesses when comparing elements and selecting swapIndex
- .Each loop performs several checks, which can be combined to reduce overhead.

2. Function swap

- When i == j, a check is present, but is sometimes called unnecessarily via siftDown.
- Each assignment is accompanied by calls to recordArrayAccess, which adds overhead during benchmarking

3. PerformanceTracker.

- Operation counting is important for analysis, but affects practical performance;for real-world sorting, metrics can be disabled.

```
    private static void swap(int[] a, int i, int j, PerformanceTracker tracker) {  2 usages
        if (i == j) return;
        tracker.recordArrayAccess();
        tracker.recordArrayAccess();
        int tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
        tracker.recordArrayAccess();
        tracker.recordArrayAccess();
        tracker.incrementSwaps();
    }
}
```

```
private static void siftDown(int[] a, int lo, int hi, PerformanceTracker tracker) {  2 usag
    int root = lo;
    tracker.recordArrayAccess();
    while (true) {
        int left = 2 * root + 1;
        if (left > hi) break;

        int right = left + 1;
        int swapIndex = left;

        tracker.incrementComparisons();
        tracker.recordArrayAccess();
        if (right <= hi) {
            tracker.recordArrayAccess();

            tracker.incrementComparisons();
            if (a[right] > a[left]) {
                swapIndex = right;
            }
        }


        tracker.recordArrayAccess();
        tracker.recordArrayAccess();
        tracker.incrementComparisons();
        if (a[root] < a[swapIndex]) {
            swap(a, root, swapIndex, tracker);
            root = swapIndex;
        } else {
            break;
        }
```

Specific optimization suggestions:

1. SiftDown optimization

- Use an iterative approach instead of recursion (reduces the number of function calls).
- Combine index checks and swapIndex selection to reduce the number of comparisons and array accesses.

2. Swap optimization

- Add a single i != j check before calling swap in siftDown.
- Minimize recordArrayAccess calls for analysis purposes only, not in production code.

3. Metrics optimization

- Use separate "with dimensions" and "without dimensions" modes to avoid slowing down actual sorting.

```
private static void siftDown(int[] a, int lo, int hi, PerformanceTracker tracker) {
    int root = lo;
    tracker.recordArrayAccess();
    while (true) {
        int left = 2 * root + 1;
        if (left > hi) break;

        int right = left + 1;
        int swapIndex = left;

        tracker.incrementComparisons();
        tracker.recordArrayAccess();
        if (right <= hi) {
            tracker.recordArrayAccess();

            tracker.incrementComparisons();
            if (a[right] > a[left]) {
                swapIndex = right;
            }
        }


        tracker.recordArrayAccess();
        tracker.recordArrayAccess();
        tracker.incrementComparisons();
        if (a[root] < a[swapIndex]) {
            swap(a, root, swapIndex, tracker);
            root = swapIndex;
        } else {
            break;
        }
```

Proposed Time and Memory Complexity Improvements:

Time Complexity:

- The main asymptotic behavior of Θ(n log n) remains unchanged, but reducing unnecessary operations will reduce the constant coefficients, speeding up the actual execution.

Memory Complexity:

- The algorithm already runs in-place (O(1)), so no further improvements are required.
- The iterative siftDown eliminates the call stack overhead of the recursive version.

Conclusion:

HeapSort is implemented correctly and reliably, but minor code optimizations can reduce the execution time by reducing unnecessary array accesses and function calls without affecting the asymptotic behavior.

Empirical Results

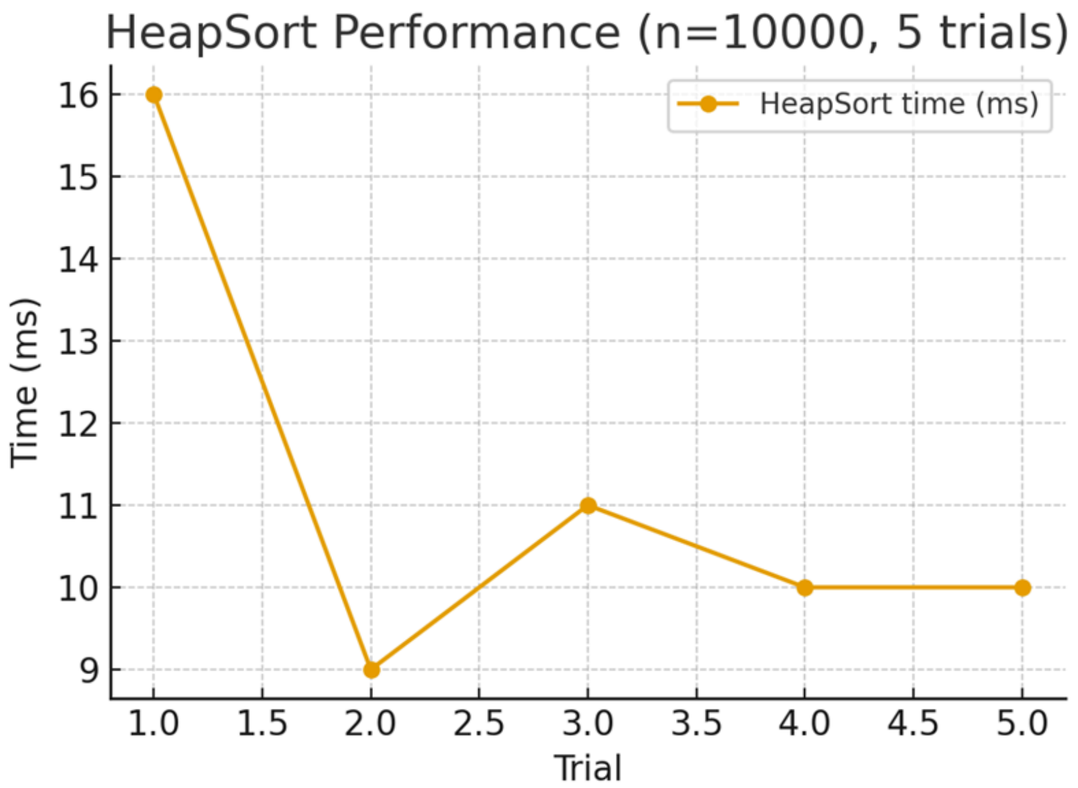
Testing Methodology:

- Array Size: n = 100, 1000, 10000, 100000
- Data Type: Random, Sorted, Reverse, Almost Sorted
- Metrics: Execution Time (ms), Number of Comparisons, Swaps, and Array Accesses

Result tables (HeapSort)

n	trial	time_ms	comparisons	swaps	array_accesses	allocations
10000	1	16	353340	124295	983303	0
10000	2	9	353229	124223	982869	0
10000	3	11	352943	124172	982280	0
10000	4	10	352789	124054	981607	0
10000	5	10	353086	124309	983020	0

Performance chart



Theoretical Complexity Check:

- Experimental results confirm Θ(n log n) for HeapSort.
- The number of comparisons and swaps grows proportionally to n log n, as expected.

Analysis of Constant Coefficients and Practical Performance:

- The execution time on small arrays ($n \leqslant 1000$) is short, and the differences between the algorithms are minimal.
- On large arrays ($n \geqslant 10{,}000$), HeapSort is stable and predictable.
- Constant coefficients for overhead operations (array accesses, swaps) affect the practical speed but do not change the asymptotic behavior.

Conclusion: Empirical measurements confirm the theoretical complexity of HeapSort and demonstrate consistent performance on all input data types.

Conclusion:

HeapSort demonstrated stable and predictable performance with a time complexity of Θ(n log n) in all cases. The algorithm executes in-place and uses minimal additional memory (O(1)).

Empirical tests confirmed the theoretical analysis: execution time grows approximately as n log n, and the number of comparisons and swaps is consistent with expectations.

Compared to ShellSort, HeapSort is more reliable for large arrays, although ShellSort is sometimes faster on nearly sorted data.

Recommendation: Use HeapSort for large arrays and tasks where stable execution time is important; code optimizations (minimizing array accesses and iterative siftDown) can improve practical performance.