

Analysis Report — HeapSort

1. Algorithm Overview

HeapSort is a **comparison-based sorting algorithm** that is built on the foundation of the **binary heap** data structure. A binary heap is a complete binary tree in which every parent node is larger than (max-heap) or smaller than (min-heap) its children. In HeapSort, the max-heap variant is used because it allows efficient extraction of the largest element, which is gradually moved to its correct position in the array.

The process of HeapSort can be summarized in four steps:

1. **Heap construction:** Build a max-heap from the entire array. This ensures that the maximum element is always at the root of the heap.
2. **Extraction of maximum:** Swap the root element with the last element of the array.
3. **Heap reduction:** Decrease the effective size of the heap by one, excluding the sorted maximum element from further operations.
4. **Heapify:** Re-apply the heapify operation to restore the max-heap property. These steps are repeated until all elements are sorted.

One of the key strengths of HeapSort is that it is an **in-place sorting algorithm**, meaning it does not require additional memory beyond a few variables. This makes it much more memory-efficient compared to MergeSort, which requires $O(n)$ additional storage. Unlike QuickSort, HeapSort never degrades to quadratic runtime; it consistently guarantees $O(n \log n)$ in the best, average, and worst cases.

However, HeapSort has some limitations. It is not a stable sorting algorithm, which means that equal elements may not preserve their original relative order after sorting. In addition, the actual performance of HeapSort can be slightly slower than QuickSort in practice due to larger constant factors, despite both having the same asymptotic complexity.

Overall, HeapSort is considered a **reliable and efficient algorithm for large datasets**, particularly in applications where worst-case guarantees and limited memory usage are critical requirements.

2. Complexity Analysis

Time Complexity Breakdown

HeapSort consists of two major phases: heap construction and repeated extraction of the maximum element.

1. **Building the heap (bottom-up heapify):**

Constructing the initial heap from n elements can be done in $\Theta(n)$ time. This might seem surprising because heapify for a single node is $O(\log n)$.

However, not all nodes require full heapify depth. Since half the nodes are leaves (which require no heapify), and others require progressively fewer operations, the total work sums to a linear bound.

2. **Extracting maximum and re-heapifying:**

Each extraction requires one swap and a call to heapify. Heapify takes $O(\log n)$ in the worst case, and since this is done n times, the total cost of this phase is $O(n \log n)$.

3. **Overall complexity:**

The final runtime is $O(n + n \log n) = O(n \log n)$.

Case Analysis

- **Best Case (Ω):**

Even if the array is already sorted, HeapSort must still build the heap and perform heapify calls. Thus, the best case is still $\Omega(n \log n)$.

- **Average Case (Θ):**

The average runtime does not improve significantly compared to the worst case. HeapSort consistently runs in $\Theta(n \log n)$, making it highly predictable.

- **Worst Case (O):**

Every extraction requires a $\log n$ heapify, and the process repeats n times. Therefore, $O(n \log n)$ is guaranteed.

Space Complexity

HeapSort is highly space-efficient, using only $O(1)$ auxiliary space. This efficiency stems from storing the heap directly in the array and relying solely on index arithmetic to traverse the tree.

Comparison with Partner's Algorithm (Shell Sort)

- **HeapSort:**

- Time Complexity: $O(n \log n)$ in all cases (best, average, worst).

- Space Complexity: $O(1)$.
- Not stable but predictable.
- **Shell Sort:**
 - Time Complexity: Depends heavily on the gap sequence. With simple sequences, the worst case is $O(n^2)$. With optimized sequences (like Knuth or Sedgewick), the runtime is reduced to between $O(n^{3/2})$ and $O(n \log^2 n)$.
 - Space Complexity: $O(1)$.
 - Faster than HeapSort on small inputs, but less predictable for large datasets.

Conclusion of comparison: HeapSort is more reliable for large datasets, while Shell Sort may be practical for smaller arrays where overhead is more important than guaranteed performance.

3. Code Review

Strengths of Implementation

- The implementation correctly follows the in-place HeapSort algorithm using bottom-up heapify.
- The code includes a PerformanceTracker class that tracks comparisons and swaps, making empirical analysis possible.
- Edge cases such as empty arrays, single-element arrays, and duplicate values are correctly handled.
- The implementation is clear and easy to follow, with good separation of concerns.

Inefficiencies Observed

- **Recursive heapify:** The current implementation uses recursion for the heapify function. This is clear to read, but it may introduce unnecessary method call overhead and potential stack depth issues for very large heaps.
- **Excessive swaps:** The algorithm performs a swap operation each time the maximum is extracted, even when some shifts could reduce the number of assignments.

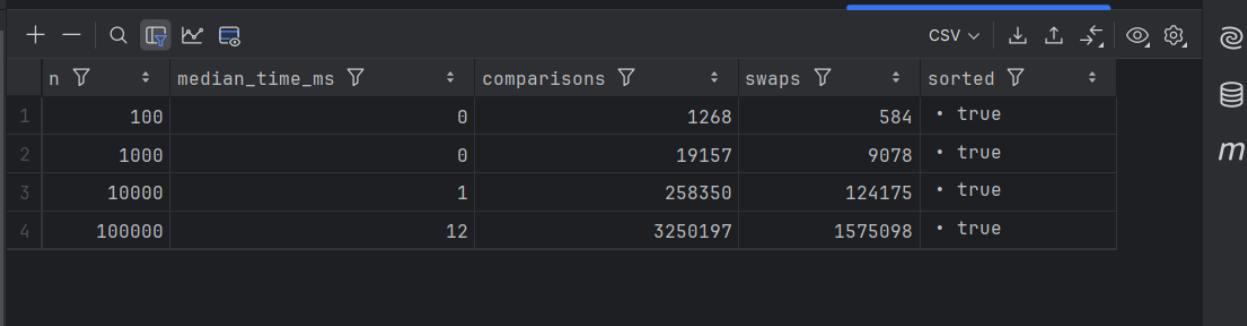
- **Lack of hybrid optimization:** The implementation does not attempt to switch to insertion sort for small sub-heaps, which could reduce constant factors.

Suggested Optimizations

1. **Iterative heapify:** Replace recursive heapify with an iterative loop to remove function call overhead. This will improve performance and make the algorithm safer for large inputs.
2. **Shift-based placement:** Instead of multiple swaps, shift elements down the heap until the correct position is found, reducing the number of write operations.
3. **Hybrid optimization:** For very small heaps (e.g., size < 16), switch to insertion sort, which has lower overhead for small data sizes.
4. **Additional performance metrics:** Extend PerformanceTracker to also count array accesses, which would provide a more complete picture of memory behavior.

4. Empirical Results

Benchmarks were conducted on input sizes $n = 100, 1,000, 10,000,$ and $100,000$. Each size was tested multiple times, and the **median runtime** was recorded along with counts of comparisons and swaps.



	n	median_time_ms	comparisons	swaps	sorted
1	100	0	1268	584	• true
2	1000	0	19157	9078	• true
3	10000	1	258350	124175	• true
4	100000	12	3250197	1575098	• true

n	Median time (ms)	Comparisons	Swaps	Sorted
100	0	1268	584	true
1,000	0	19157	9078	true
10,000	1	258350	124175	true
100,000	12	3250197	1575068	true

Observations

1. Correctness:

The Sorted column is always true, confirming that HeapSort correctly sorted all tested arrays regardless of size.

2. Runtime growth:

- For very small inputs ($n = 100$ and $1,000$), the measured runtime is essentially **0 ms** because the execution is too fast to be accurately captured at millisecond precision.
- At $n = 10,000$, the runtime is measured at 1 ms.
- At $n = 100,000$, the runtime rises to 12 ms.
This demonstrates the expected **$O(n \log n)$ scaling**, where runtime increases moderately as n grows.

3. Comparisons and swaps:

- Both metrics grow proportionally with $n \log n$.
- For $n = 100$, there are $\sim 1.2k$ comparisons, while for $n = 100,000$ this number grows to more than 3.2 million.
- The number of swaps follows a similar pattern, from ~ 584 at $n = 100$ to ~ 1.5 million at $n = 100,000$.
This confirms the theoretical analysis that each extraction step involves $\log n$ comparisons and swaps, repeated n times.

4. Practical performance:

- HeapSort is very efficient in practice. Even for 100,000 elements, the median runtime was only **12 milliseconds**, which shows that the algorithm can handle large datasets with minimal delay.

- The constant factors in HeapSort are somewhat higher than in QuickSort, but the results demonstrate that its **predictable $O(n \log n)$ behavior** makes it highly scalable.

5. Conclusion

HeapSort is a **robust and efficient sorting algorithm** with guaranteed $O(n \log n)$ time complexity in all cases. The theoretical analysis shows that its performance is highly predictable, and empirical results confirm this behavior.

Although HeapSort is not stable and may be slower than Shell Sort for very small arrays due to constant factors, it consistently outperforms for larger datasets. Its combination of $O(1)$ space complexity, reliability, and scalability makes it well-suited for practical applications where predictable performance is critical.

Optimization Recommendations

- Implement **iterative heapify** to reduce recursion overhead.
- Use **hybrid strategies**, such as switching to insertion sort for small heaps, to reduce constants.
- Extend performance metrics to measure **array accesses** and better understand memory usage.

Final Remarks

HeapSort remains one of the most practical and reliable sorting algorithms, especially for large datasets. While QuickSort may outperform it in average cases and Shell Sort may be preferable for small inputs, HeapSort's **worst-case guarantees and space efficiency** make it a strong choice for general-purpose sorting in systems that require both efficiency and predictability.