MINISTRY OF EDUCATION AND RESEARCH OF THE REPUBLIC OF MOLDOVA
Technical University of Moldova
Faculty of Computers, Informatics and Microelectronics
Department of Software and Automation Engineering

RUSNAC NICHITA FAF-242

Laboratory work 2: Study and empirical analysis of sorting algorithms. Analysis of quickSort, mergeSort, heapSort, (one of your choice)

of the "Algorithm analysis" course

Checked:
asist. univ. Cristofor Fistic
Department of Software and Automation Engineering,
FCIM Faculty, UTM

Chisinau – 2026

Оглавление

ALGORITHM ANALYSIS

Objective

Study and analyze sorting algorithms Quick Sort, Merge Sort, Heap Sort, and one additional algorithm of choice (QR Sort). The work combines:

- a brief theoretical overview (algorithm idea, complexity, stability, memory usage), and
- an empirical benchmark based on execution time for multiple array sizes and multiple input distributions (random, sorted, reverse sorted).

The goal is not only to "find the fastest algorithm", but to understand why performance changes depending on input ordering, implementation details, and memory behavior.

Tasks

1. Implement the algorithms listed above in a programming language
2. Establish the properties of the input data against which the analysis is performed
3. Choose metrics for comparing algorithms
4. Perform empirical analysis of the proposed algorithms
5. Make a graphical presentation of the data obtained
6. Make a conclusion on the work done.

Theoretical Notes

An alternative to theoretical complexity analysis is empirical analysis. This approach is useful for:

- gaining initial insight into the complexity class of an algorithm;
- comparing the performance of two or more algorithms that solve the same problem;
- comparing different implementations of the same algorithm;
- evaluating how efficiently an algorithm runs on a specific computer system.

When performing empirical analysis, the following steps are typically followed:

1. Define the objective of the analysis.
2. Select the efficiency metric (for example, number of executed operations or execution time of the whole algorithm or a part of it).
3. Define the input data characteristics relevant to the analysis (such as data size or specific input properties).
4. Implement the algorithm in a programming language.
5. Generate multiple input data sets.
6. Execute the program for each input set.
7. Analyze the collected results.

The choice of efficiency metric depends on the goal of the analysis. For instance, if the objective is to estimate the complexity class or verify a theoretical prediction, counting the number of operations is usually appropriate. However, if the goal is to evaluate how well an implementation performs in practice, measuring execution time is more suitable.

After running the program with test data, the results are recorded. Then, depending on the analysis goals, either statistical summaries (such as mean, standard deviation, etc.) are computed, or graphs are plotted using pairs of values (for example, problem size versus efficiency metric).

Introduction

Sorting is one of the most common algorithmic tasks and a standard benchmark problem. It has many correct solutions based on different paradigms (divide-and-conquer, data structures, key-based methods), and it highlights practical tradeoffs between average-case speed, worst-case guarantees, extra memory usage, and stability. Another important reason is that sorting performance can depend strongly on input properties, for example random data versus already sorted or reverse-sorted arrays.

In practice, when choosing a sorting method, we care not only about asymptotic complexity but also about properties such as stability (preserving the order of equal keys), in-place behavior (how much extra memory is needed), and worst-case behavior (whether runtime can degrade badly on adversarial ordering). Adaptivity to partially sorted data is also relevant in many real applications, even though it is not the main focus of this laboratory work.

In this laboratory work, the following are studied:

- Quick Sort (divide-and-conquer, in-place, typically fast on random input but can degrade on adversarial order)
- Merge Sort (divide-and-conquer, stable, predictable O(n log n), needs extra memory)
- Heap Sort (heap data structure, in-place, O(n log n) worst-case)
- QR Sort (non-comparative integer sorting algorithm that divides each input element by a user-specified divisor and uses the QR Theorem)

Additionally, as an extension, the laboratory includes optimized variants of merge sort and quick sort that apply small fixed sorting networks for tiny subarrays. These are not required by the basic task, but they help illustrate how real-world performance can be improved without changing the asymptotic complexity class.

Comparison Metric

The comparison metric for this laboratory work will be considered the time of execution of each algorithm (T(n))

Input Format

Experiments are performed on integer arrays generated under three input distributions. For Random input, values are generated as uniformly random integers in [0,10n]. For Sorted (ascending) input, the generated array is already ordered as [0, 1, ..., n-1], which can trigger worst-case behavior in some naive implementations. For Reverse sorted (descending) input, the generated array is [n-1, ..., 0], another adversarial ordering for certain pivot strategies.

To study how performance scales, the notebook uses several predefined size lists: test_small(100, 200, 500, 1000, 1500, 2000, 2500, 3000, 4000, 5000), test_medium(5000, 7500, 10000, 12500, 15000, 17500, 20000, 25000, 30000, 35000), test_large(10000, 20000, 30000, 50000, 70000, 100000, 130000, 160000, 200000, 250000), test_xl(100000, 150000, 200000, 250000, 300000, 350000, 400000, 450000, 500000), test_xxl(500000, 600000, 700000, 800000, 900000, 1000000) and also includes optional extreme stress tests: test_xxxl(1500000, 2000000, 2500000, 5000000, 7500000, 10000000), test_xxxxl(15000000, 20000000, 25000000, 50000000, 75000000, 100000000). These larger sets are useful for pushing the limits, but they may be impractical on some machines due to RAM usage.

IMPLEMENTATION

All algorithms are implemented in Python in a jupyter notebook.

The implementation style is intentionally “educational”: algorithms are coded explicitly (rather than using Python’s built-in sort) to ensure that we measure the algorithmic behavior, thus they are naïve and not as effective as the library implemented ones.

The benchmark wrapper follows the same pattern for all algorithms:

- Generate a base array of size n.
- Copy it (so each algorithm receives identical input).
- Execute the sorting function.
- Measure the elapsed time.
- Store the result for plotting and print a small numerical summary.

Because the algorithms sort in-place, copying is required, otherwise, later algorithms would receive an already sorted array and the comparison would be invalid.

Quick Sort

Quick Sort is a divide-and-conquer algorithm based on partitioning. A pivot element is chosen and the array is rearranged so that elements smaller than (or equal to) the pivot go to the

left part and larger elements go to the right part, after which both parts are sorted recursively. In the notebook, the basic version uses a Lomuto-style partition with the last element as pivot.

On random input, Quick Sort is often very fast in practice because it works in-place and performs simple comparisons and swaps with good cache behavior. Its weakness is input sensitivity: on already sorted or reverse-sorted arrays, choosing the last element as pivot can repeatedly create highly unbalanced partitions, which leads to a worst-case time close to $O(n^2)$. Average behavior is typically $O(n\log n)$, it is not stable, and it uses $O(\log n)$ stack space on average due to recursion.

Implementation:

```
def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

def quick_sort(arr, low, high):
    if low < high:
        p = partition(arr, low, high)
        quick_sort(arr, low, p - 1)
        quick_sort(arr, p + 1, high)
```

Figure 1 – Quick Sort

Merge Sort

Merge Sort is a divide-and-conquer algorithm that splits the array into two halves, sorts each half recursively, and then merges the two sorted halves in linear time. The key advantage is predictability: the same work pattern is performed regardless of the initial ordering of elements.

Because of that structure, Merge Sort has $O(n\log n)$ time in best/average/worst cases and is stable by nature (equal elements preserve their relative order). The tradeoff is memory: the standard merge step uses an auxiliary array, which makes the extra space usage typically $O(n)$.

Implementation:

```
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m

    L = [0] * n1
    R = [0] * n2

    for i in range(n1):
        L[i] = arr[l + i]
    for j in range(n2):
        R[j] = arr[m + 1 + j]
```

```
    i = j = 0
    k = l

    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

def mergeSort(arr, l, r):
    if l < r:
        m = l + (r - l) // 2
        mergeSort(arr, l, m)
        mergeSort(arr, m + 1, r)
        merge(arr, l, m, r)
```

Figure 2 – Merge Sort

Heap Sort

Heap Sort is based on the binary heap data structure. The array is first converted into a max-heap, after which the maximum element is repeatedly swapped to the end of the array and the heap property is restored (heapify) on the reduced heap.

Heap Sort has a strong guarantee: its time complexity is $O(n\log n)$ in the best, average, and worst cases, and it is in-place (constant extra memory). In practice, it can be slower than Quick Sort on random data because heap operations often jump around memory, which is less cache-friendly. Heap Sort is not stable(swaps can change the relative order of equal keys).

Implementation:

```
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2

    if l < n and arr[l] > arr[largest]:
        largest = l
```

```
    if r < n and arr[r] > arr[largest]:
        largest = r

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heapSort(arr):
    n = len(arr)

    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # Swap max to end
        heapify(arr, i, 0)
```

Figure 3 – Heap Sort

QR Sort

QR Sort is a novel non-comparative integer sorting algorithm introduced by Bushman, Tebcherani, and Yasin (2024). It operates based on the Quotient-Remainder Theorem and utilizes two passes of a stable sorting subroutine (typically Counting Sort). The algorithm works by computing two keys for each element si: a remainder $r_i = (s_i - \min(S)) \pmod d$ and a quotient $q_i = \lfloor (s_i - \min(S))/d \rfloor$, where d is a chosen divisor. The sequence is first stably sorted by the remainders, and then stable sorted by the quotients, resulting in a fully sorted array.

The theoretical time and space complexity of QR Sort is O(n+d+m/d), where n is the input size and m is the range of values. By selecting the optimal divisor $d = \sqrt{m}$, the complexity is minimized to $O(n + \sqrt{m})$. This allows the algorithm to achieve linear time O(n) when the range m is not excessively large relative to n (specifically when $m \leq O(n^2)$). Being non-comparative, it overcomes the O(nlogn) lower bound of comparison-based sorts, provided the input distribution allows specifically for efficient counting sort passes.

Implementation:

```
def counting_sort_with_keys(A, B, C, K, copy_B_to_A):
    """
    Counting Sort using externally provided keys.
    A - input array
    B - auxiliary array (same size as A)
    C - counting array (size = key range)
    K - key array (same size as A)
    copy_B_to_A - if True copy sorted B back to A
    """

    n = len(A)

    # Reset counting array
    for i in range(len(C)):
```

```
        C[i] = 0

    # Count occurrences
    for i in range(n):
        k = K[i]
        C[k] += 1

    # Prefix sums
    for k in range(1, len(C)):
        C[k] += C[k - 1]

    # Build output (stable)
    for i in range(n - 1, -1, -1):
        k = K[i]
        B[C[k] - 1] = A[i]
        C[k] -= 1

    # copy back
    if copy_B_to_A:
        for i in range(n):
            A[i] = B[i]

def qr_sort(A, d):
    """
    QR Sort using divisor d.
    Sorts using remainder then quotient (stable).
    """

    n = len(A)
    if n <= 1:
        return A

    min_a = min(A)
    max_a = max(A)

    max_quot = (max_a - min_a) // d

    B = [0] * n

    # Remainder keys
    Cr = [0] * d
    R = []
    for v in A:
        R.append((v - min_a) % d)

    if max_quot == 0:
        counting_sort_with_keys(A, B, Cr, R, True)
        return A

    # First pass: sort by remainder into B
    counting_sort_with_keys(A, B, Cr, R, False)
```

```
    # Quotient keys
    min_b = min(B)
    Q = []
    for v in B:
        Q.append((v - min_b) // d)

    max_q = max(Q)
    Cq = [0] * (max_q + 1)

    # Second pass: sort by quotient into A
    counting_sort_with_keys(B, A, Cq, Q, False)

    return A
```

Figure 4 – QR Sort

Optimized Quick Sort

This variant integrates AlphaDev's assembly-optimized sorting networks (DeepMind) as base cases for small subarrays (typically sizes 3 to 8) encountered during recursion. As described by Aly, Jensen, and ElAarag (2025), replacing the standard base case with these specialized networks can yield significant performance improvements. Their research highlights that converting to sorting networks at specific thresholds (e.g., configurations covering sizes 3 to 5 or 6 to 8) can provide up to a 1.5x speedup on sorted arrays compared to classical implementations. The implementation in this lab combines these networks with a median-of-three pivot strategy and Hoare partitioning to further mitigate worst-case scenarios and reduce overhead.

Implementation:

```
# Sorting Network helpers (you can extend networks here)
# Networks are lists of comparator pairs (i, j) using 0-based indices inside the
subarray.

SORTING_NETWORKS = {
    # size 2
    2: [(0, 1)],
    # size 3 (simple)
    3: [(0, 1), (1, 2), (0, 1)],
    # size 4 (bitonic network)
    4: [(0, 1), (2, 3), (0, 2), (1, 3), (1, 2)],
    # size 5 (works, not minimal)
    5: [(0, 1), (3, 4), (2, 4), (2, 3),
        (0, 3), (0, 2), (1, 4), (1, 3), (1, 2)],
    # size 8 (classic bitonic sorting network)
    8: [
        (0, 1), (2, 3), (4, 5), (6, 7),
        (0, 2), (1, 3), (4, 6), (5, 7),
        (1, 2), (5, 6), (0, 4), (1, 5), (2, 6), (3, 7),
        (2, 4), (3, 5),
        (1, 2), (3, 4), (5, 6),
    ],
```

```
}

def apply_sorting_network(arr, left, right):
    size = right - left + 1
    net = SORTING_NETWORKS.get(size)
    if net is None:
        return False  # no network for this size
    for i, j in net:
        a = left + i
        b = left + j
        if arr[a] > arr[b]:
            arr[a], arr[b] = arr[b], arr[a]
    return True
```

Figure 5 – Sorting network helper

```
# Optimized Quick Sort with Sorting Networks

def _median_of_three_index(arr, low, high):
    mid = low + (high - low) // 2
    a, b, c = low, mid, high

    # Return index of median value among arr[a], arr[b], arr[c]
    if arr[a] < arr[b]:
        if arr[b] < arr[c]:
            return b
        return c if arr[a] < arr[c] else a
    else:
        if arr[a] < arr[c]:
            return a
        return c if arr[b] < arr[c] else b


def _hoare_partition(arr, low, high, pivot_index):
    pivot = arr[pivot_index]
    i = low - 1
    j = high + 1

    while True:
        i += 1
        while arr[i] < pivot:
            i += 1

        j -= 1
        while arr[j] > pivot:
            j -= 1

        if i >= j:
            return j
```

```
        arr[i], arr[j] = arr[j], arr[i]


def optimized_quick_sort(arr, low=0, high=None):
    if high is None:
        high = len(arr) - 1
    if low >= high:
        return

    size = high - low + 1
    if size in SORTING_NETWORKS:
        apply_sorting_network(arr, low, high)
        return

    pivot_index = _median_of_three_index(arr, low, high)
    p = _hoare_partition(arr, low, high, pivot_index)
    optimized_quick_sort(arr, low, p)
    optimized_quick_sort(arr, p + 1, high)
```

Figure 6 – Optimized quick sort

Optimized Merge Sort

Similar to the Quick Sort optimization, this approach utilizes AlphaDev's fixed-size sorting networks to handle small subarrays effectively at the bottom of the recursion tree. According to recent benchmarks by Aly et al. (2025), a configuration using networks for sizes 6, 7, and 8 can achieve a 1.5x speedup for random inputs and up to 2x for sorted arrays compared to classical Merge Sort. This hybrid strategy bridges the gap between theoretical gains and practical efficiency by bypassing the overhead of recursive calls for small data chunks, leveraging the highly optimized instruction sequences discovered by AI.

Implementation:

```
# Optimized Merge Sort with Sorting Networks
def _merge(arr, left, mid, right):
    temp = []
    i, j = left, mid + 1

    while i <= mid and j <= right:
        if arr[i] <= arr[j]:
            temp.append(arr[i])
            i += 1
        else:
            temp.append(arr[j])
            j += 1

    while i <= mid:
        temp.append(arr[i])
        i += 1
    while j <= right:
        temp.append(arr[j])
        j += 1
```

```
    for k in range(len(temp)):
        arr[left + k] = temp[k]


def optimized_merge_sort(arr, left=0, right=None):
    if right is None:
        right = len(arr) - 1
    if left >= right:
        return

    size = right - left + 1
    if size in SORTING_NETWORKS:
        apply_sorting_network(arr, left, right)
        return

    mid = left + (right - left) // 2
    optimized_merge_sort(arr, left, mid)
    optimized_merge_sort(arr, mid + 1, right)
    _merge(arr, left, mid, right)
```

Figure 7 – Optimized merge sort

RESULTS

Hope Mr. Fistic wouldn't downgrade me but I need to change a little the structure. I decided that it would be better to compare and plot all algorithms one against another(and perform like a battle royale, There Can Be Only One) and print some data just in case. So I have 6 sorting algorithms and the first sets, let's plot them.

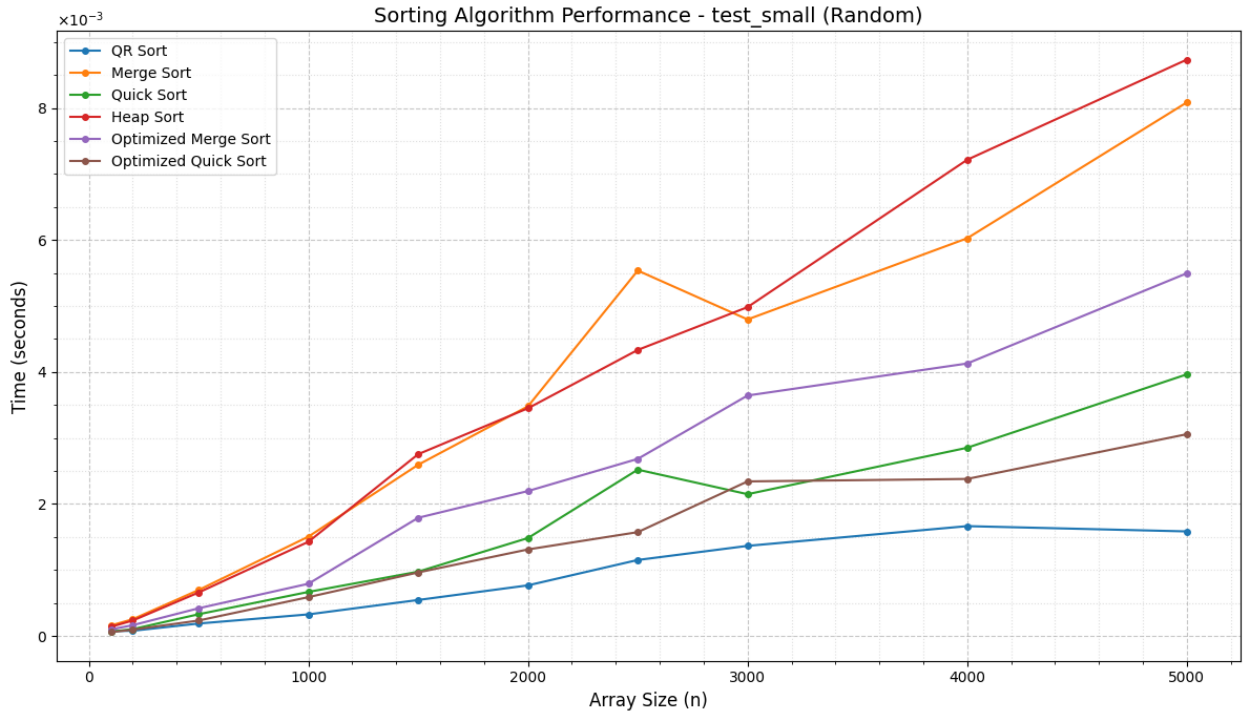


Figure 8 – results of the algorithms on test_small (Random)

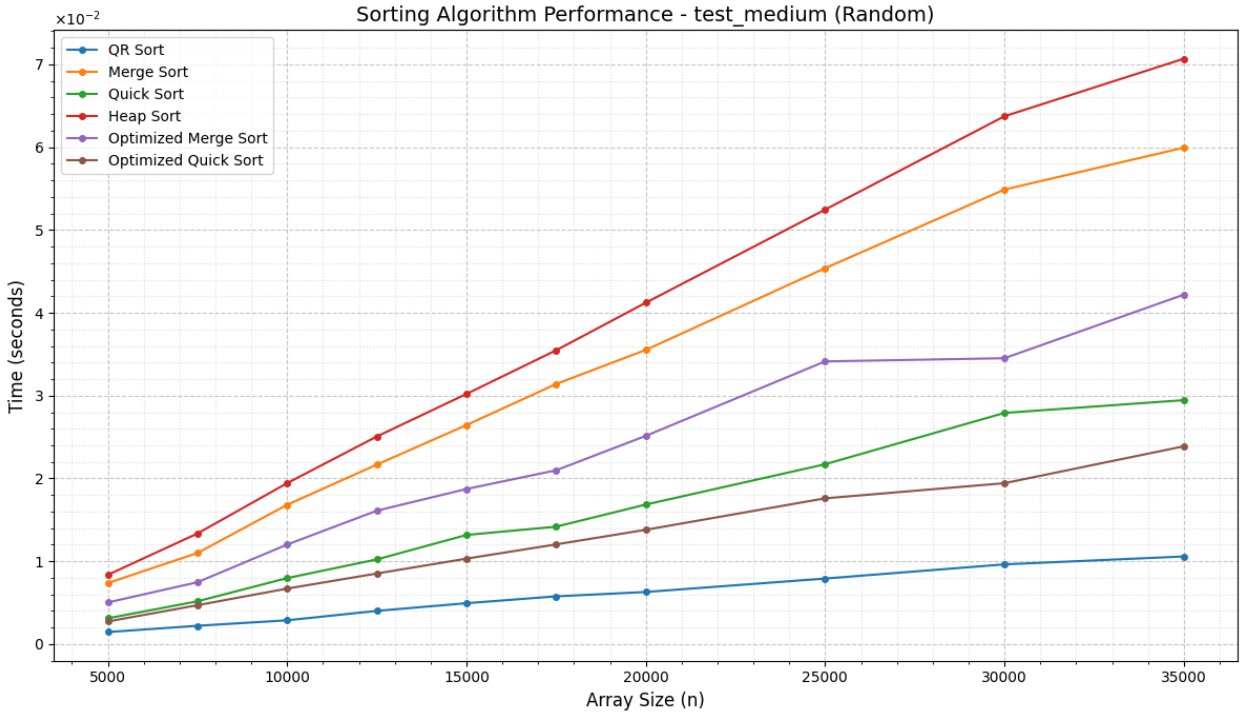


Figure 9 – results of the algorithms on test_medium (Random)

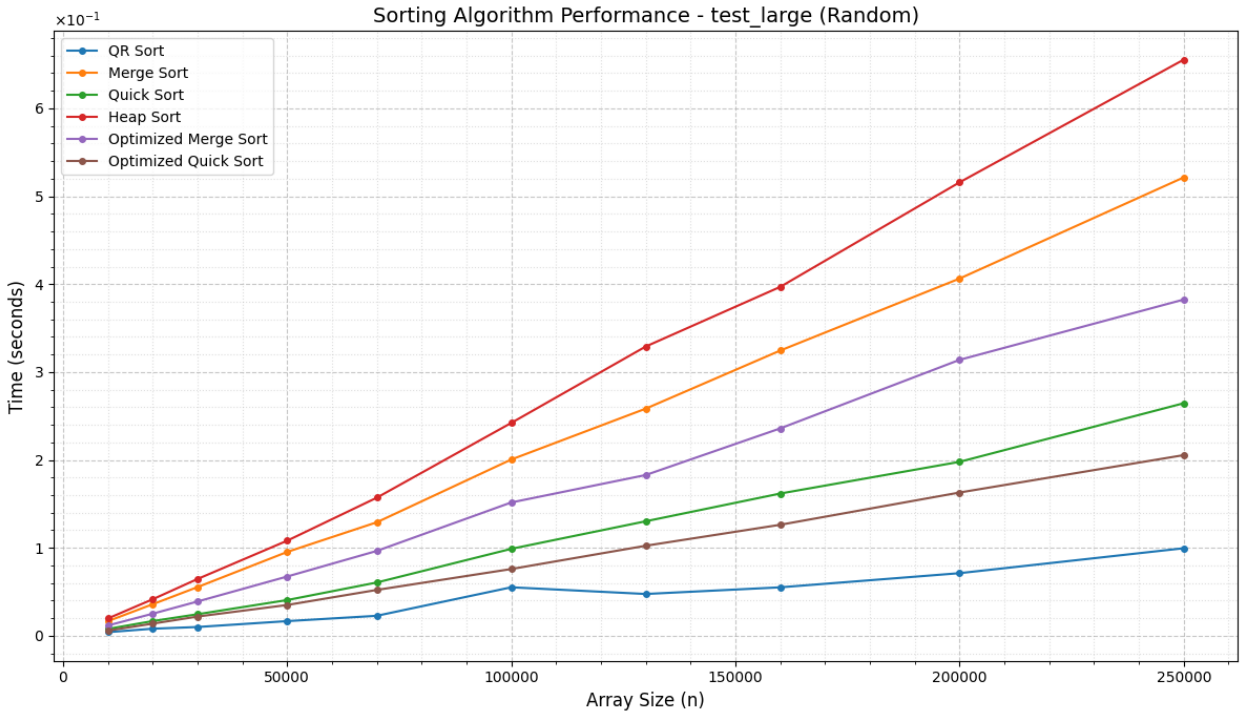


Figure 10 – results of the algorithms on test_large (Random)

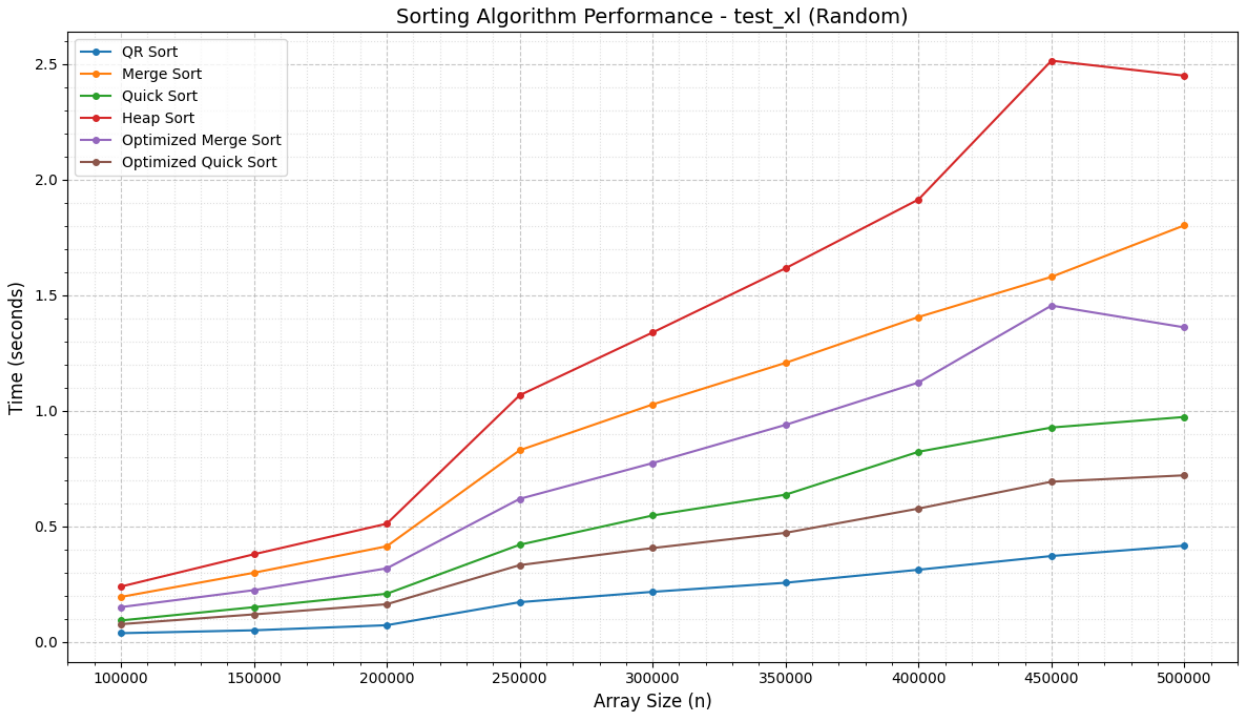


Figure 11 – results of the algorithms on test_xl (Random)

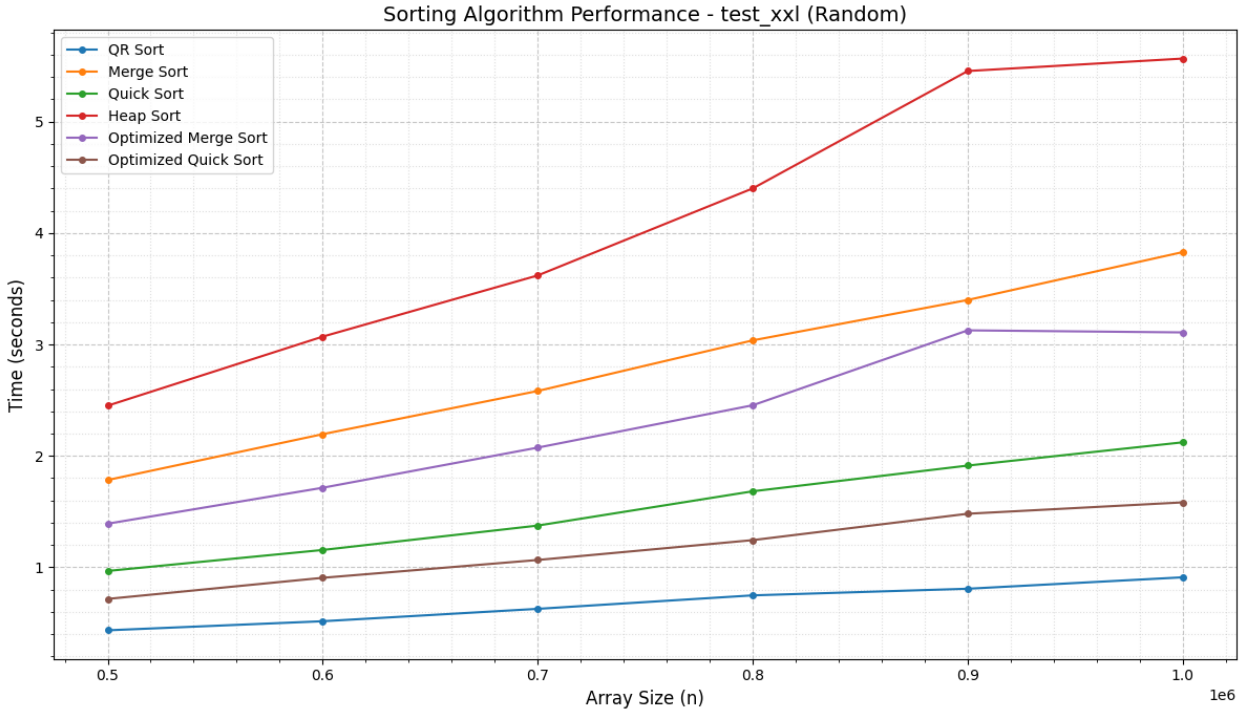


Figure 12 – results of the algorithms on test_xxl (Random)

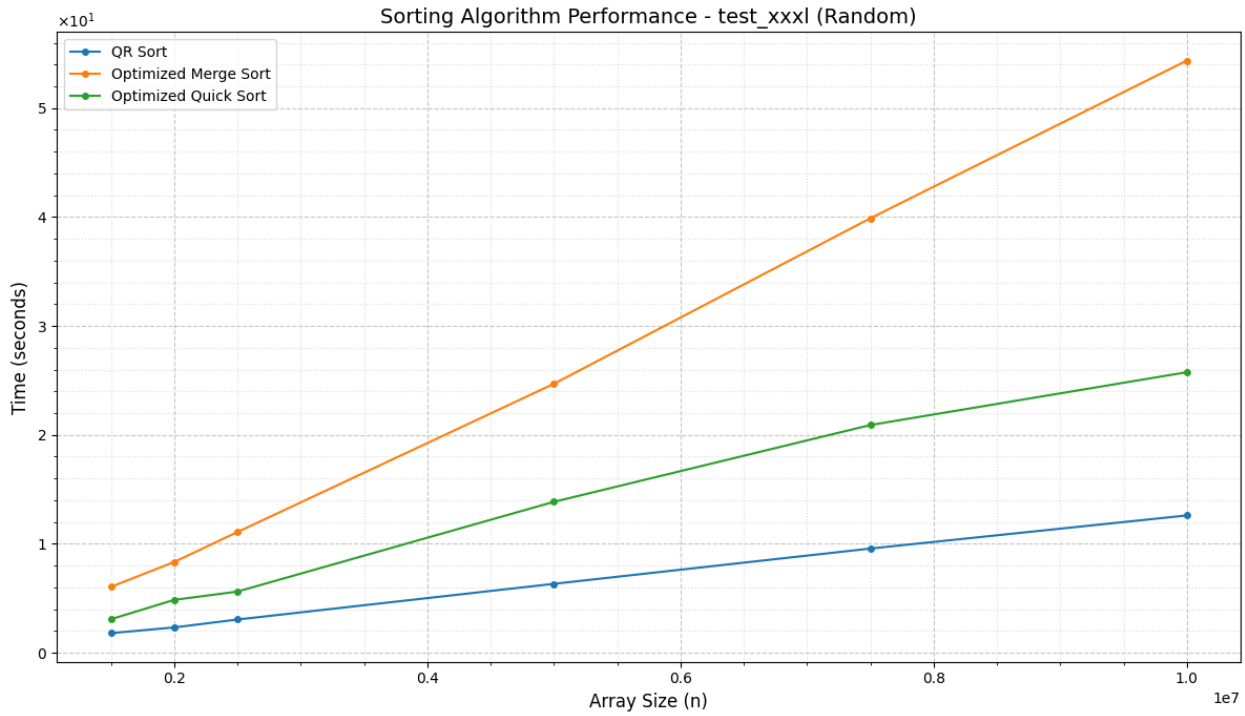


Figure 13 – results of the algorithms on test_xxxl (Random)

As we can see they all performed great algorithms with the increasing amount of data perform almost always as expected. We have a clear top: QR > Optimized Quick > Quick > Optimized Merge > Merge > Heap. Although Heap sometimes may perform better than merge, because at certain array sizes, the cost of Merge Sort's repeated memory allocation/copying can outweigh Heap Sort's poor cache locality, letting Heap Sort win occasionally. And later we can see that optimized version of Quick and Quick comes hand in hand although Optimized Merge performs a lot better than simple merge because merge Sort recurses all the way down to single elements, so the vast majority of its work happens at the bottom of the recursion tree, tiny subarrays of size 2–8. Replacing those with sorting networks eliminates a huge number of recursive calls and temporary array allocations, which has a massive cumulative effect. Quick Sort, by contrast, already handles small partitions cheaply, the Lomuto partition is just comparisons and in-place swaps with no memory allocation. The base case (size ≤ 1) is trivially fast. So sorting networks at the bottom help much less.

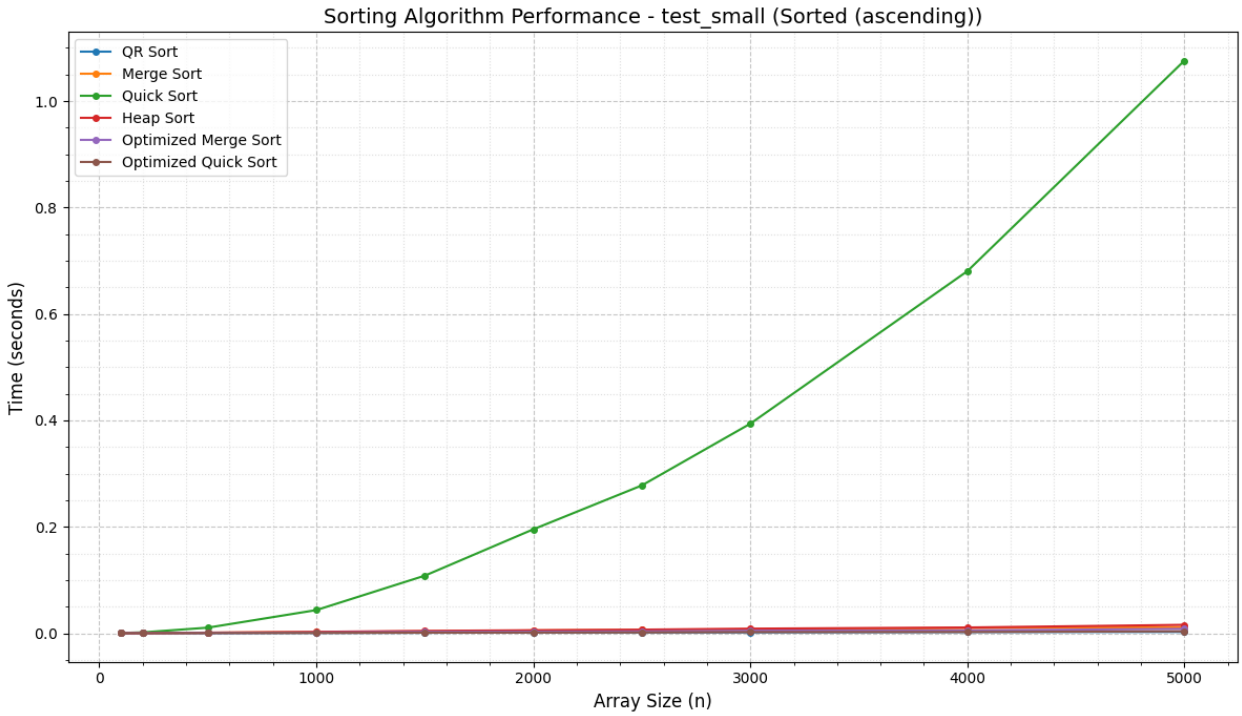


Figure 14 – results of the algorithms on test_small (Sorted - ascending)

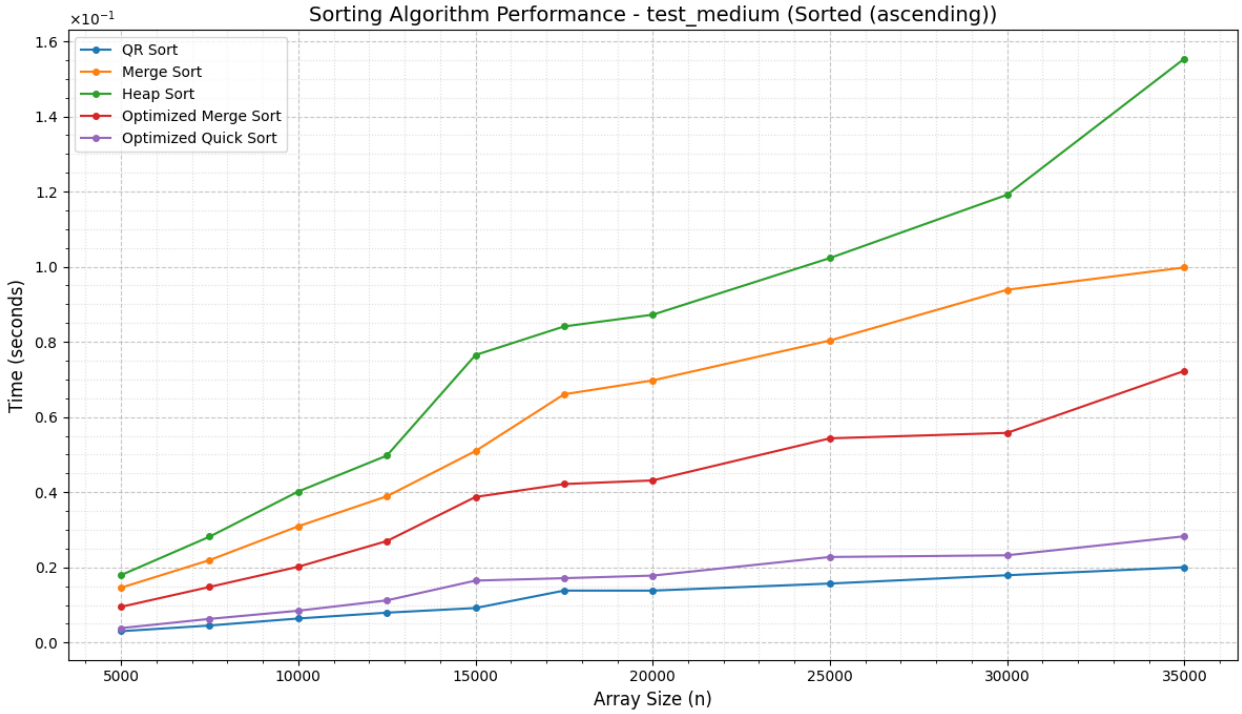


Figure 15 – results of the algorithms on test_medium (Sorted - ascending)

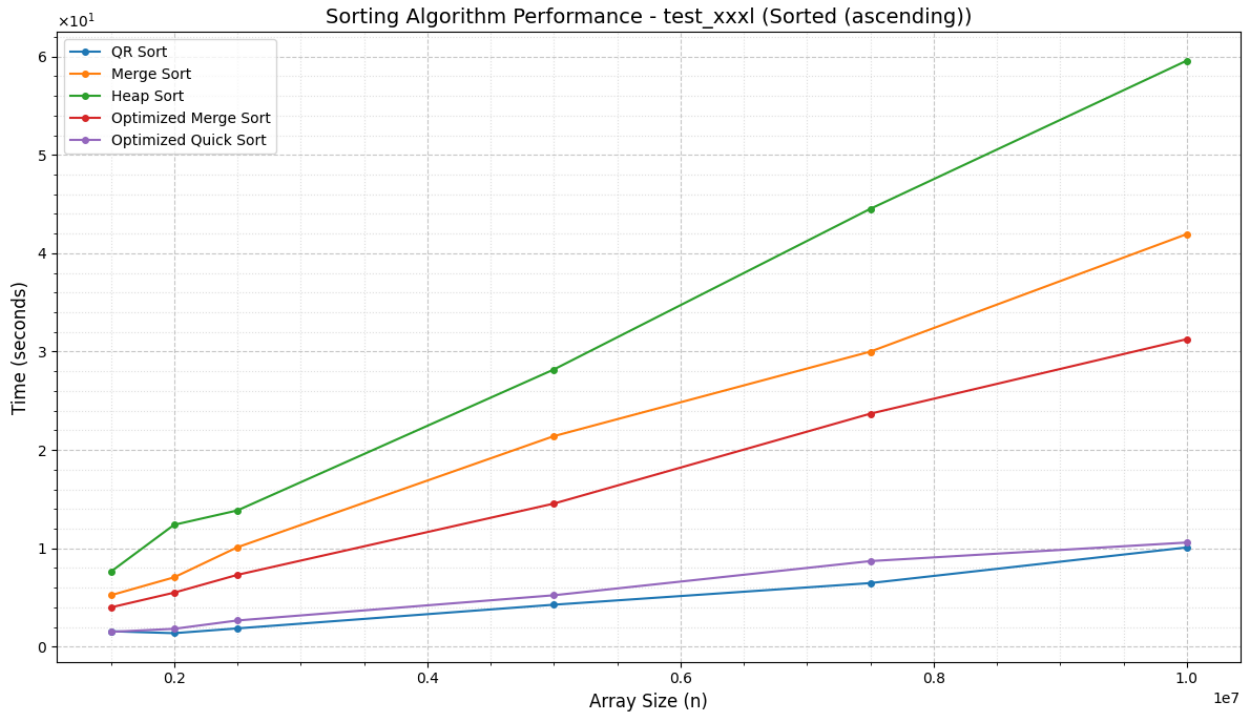


Figure 16 – results of the algorithms on test_xxxl (Sorted - ascending)

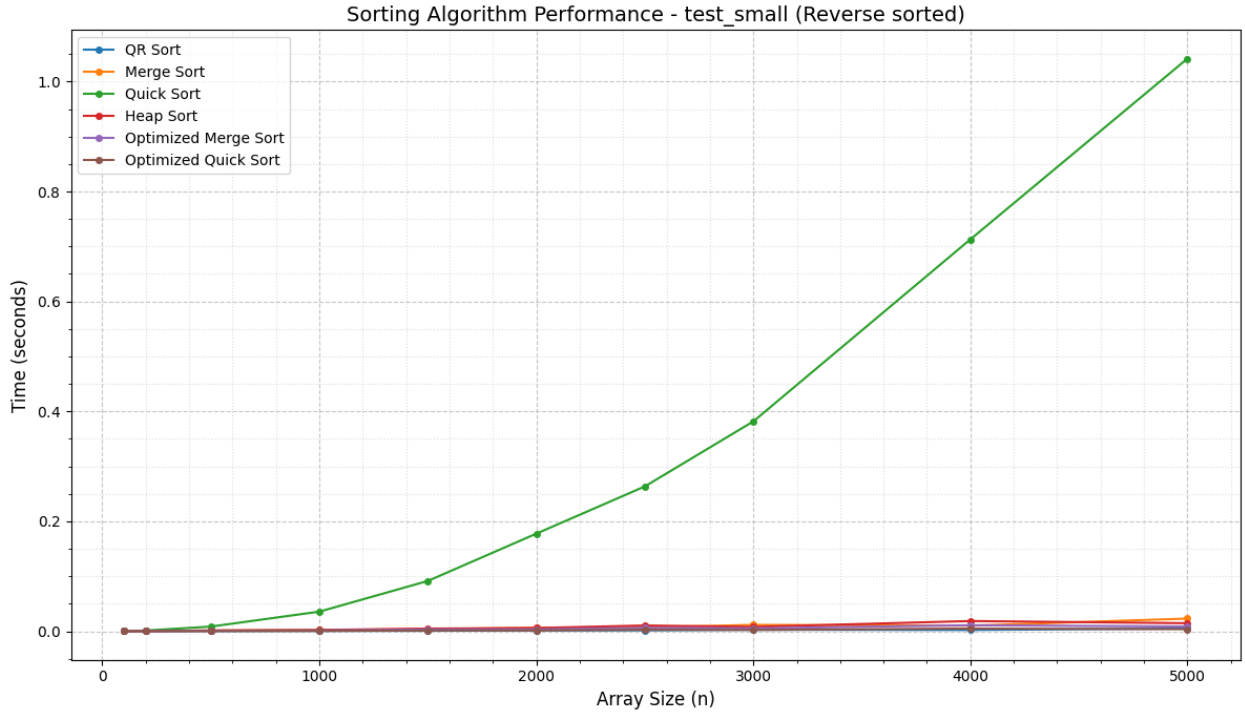


Figure 17 – results of the algorithms on test_small (Sorted - descending)

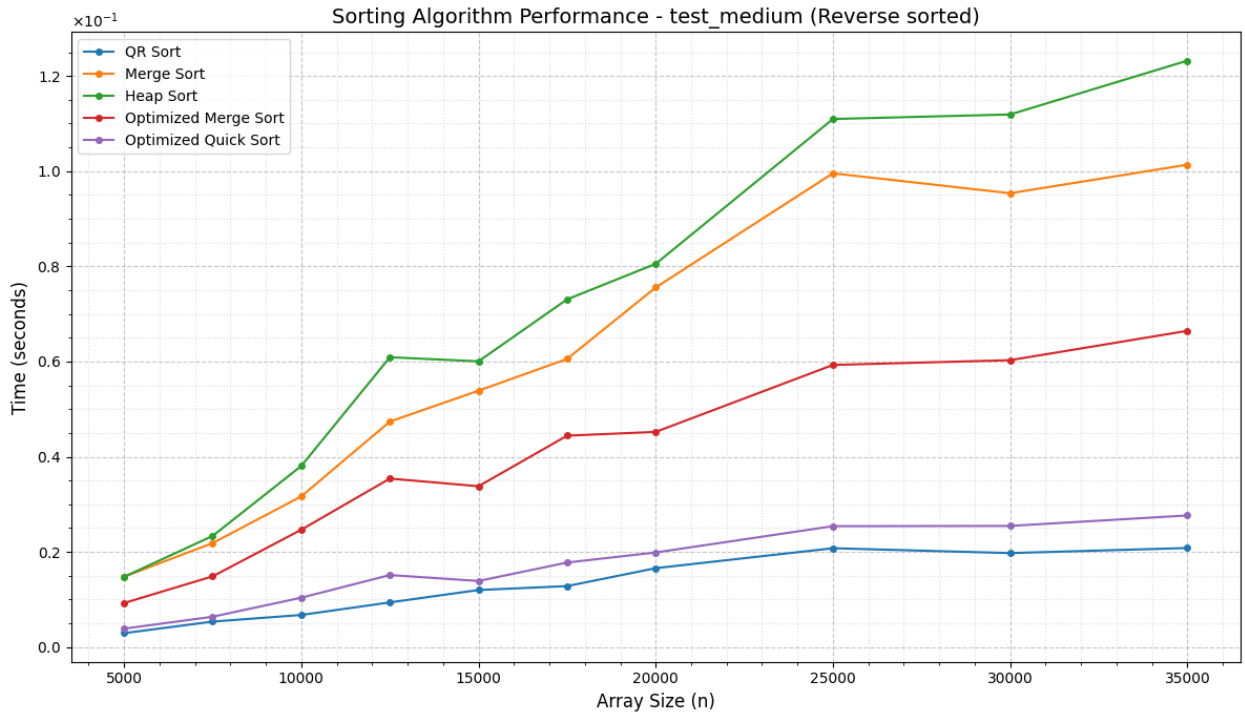


Figure 18 – results of the algorithms on test_medium (Sorted - descending)

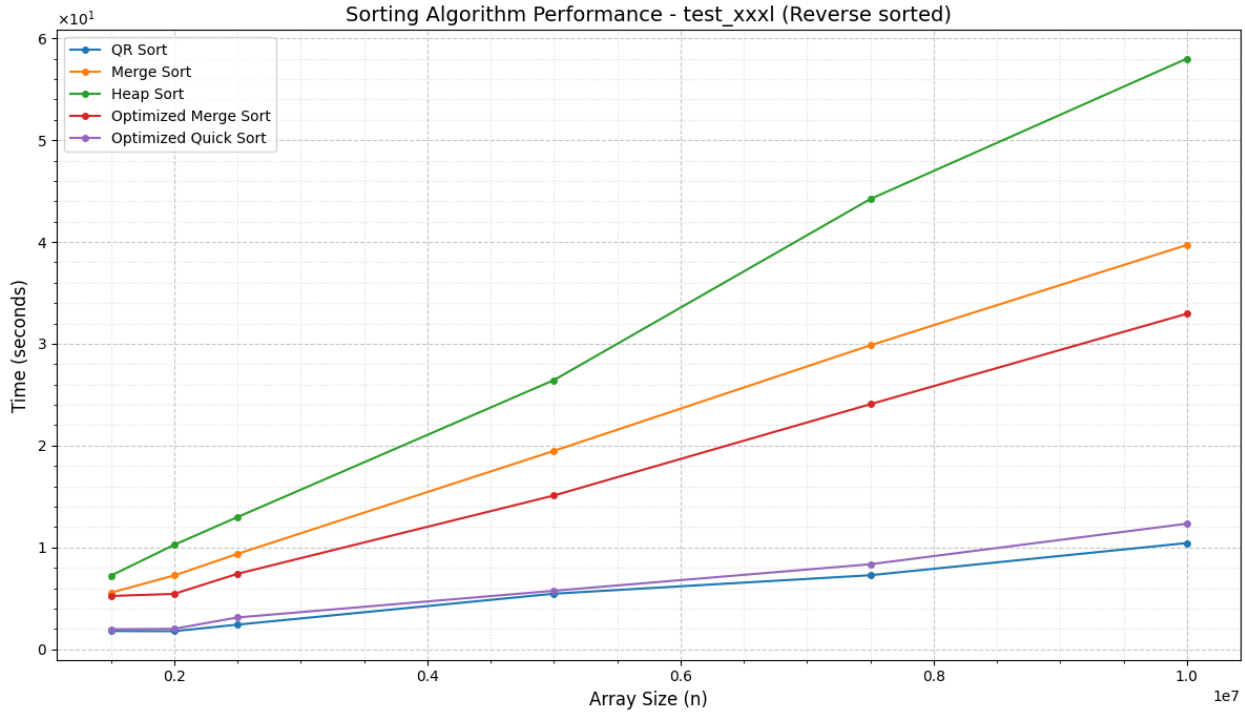


Figure 19 – results of the algorithms on test_xxxl (Sorted - descending)

When talking about ascending or descending sorted arrays clearly the loser is Quick Sort. Comparing the remained algorithms we can see same pattern QR > Optimized Quick > Optimized Merge > Merge > Heap. Altough now seems that the gap between first two are even smaller than in random arrays, the reason of that seems to be the fact that on sorted/reverse-sorted input, median-of-three pivot selection in Optimized Quick Sort consistently picks near-perfect pivots

(the median of three values from a sorted sequence is always in the middle region), producing nearly perfectly balanced partitions every time. This gives it close to ideal O(nlogn) behavior with minimal overhead. Meanwhile QR Sort's two counting-sort passes run in roughly the same time regardless of input ordering, so Optimized Quick Sort gets a relative boost on ordered data, narrowing the gap.

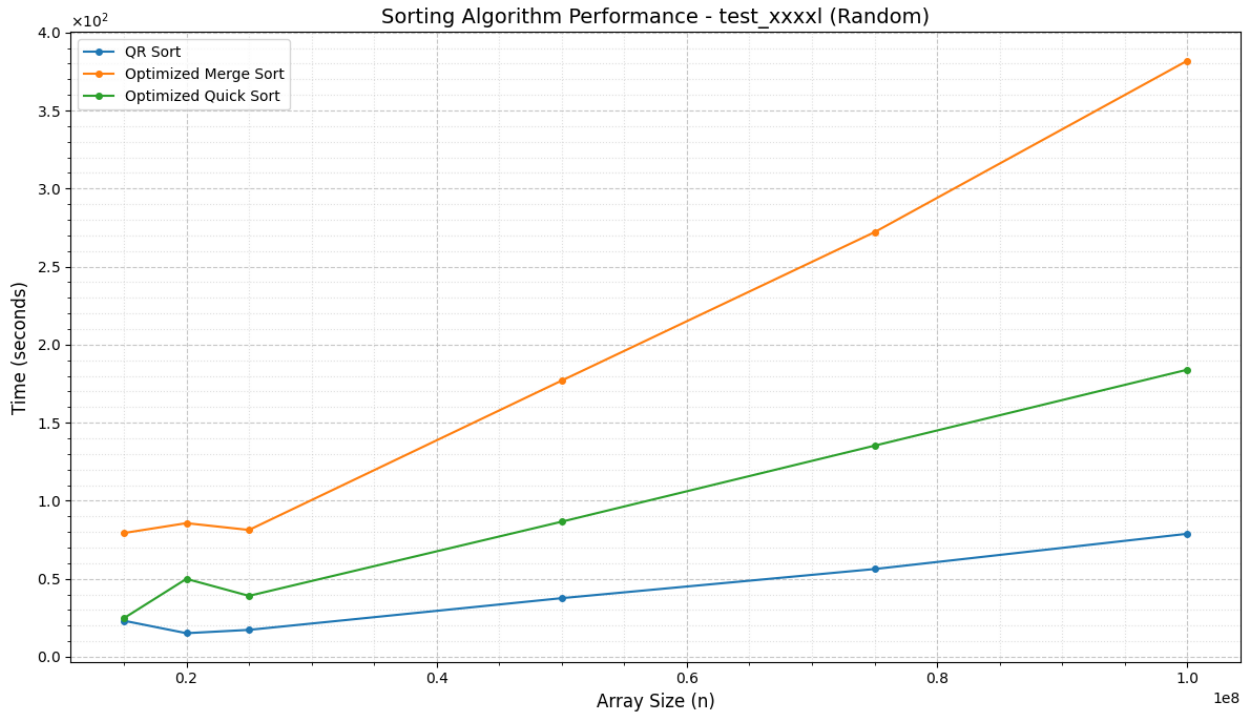


Figure 20 – results of the algorithms on test_xxxxl (Random)

Even when analyzing the extremely big arrays the pattern remains the same. I analyzed only the top 3 algorithms(QR and optimized Quick with Merge) and the pattern is the same. Just notice that the gap between QR and the rest gets wider.

I also experimented with arrays that are slightly and almost fully sorted but they have same pattern so I will not put them in report although they are in the notebook.

CONCLUSION

In this laboratory work, the sorting algorithms Quick Sort, Merge Sort, Heap Sort, and QR Sort were implemented and empirically benchmarked across multiple data distributions. The experiments were performed on random, sorted, reverse sorted, and partially sorted inputs, using array sizes ranging from 100 up to 10^8 elements. Two optimized variants were also evaluated as supplementary experiments, namely Quick Sort and Merge Sort enhanced with AlphaDev sorting networks.

The best performing algorithm across all input types and all tested size ranges was QR Sort. As a non comparative algorithm with near linear complexity O(n + $\sqrt{m}$), it consistently

outperformed all comparison based algorithms. Its advantage increased further at very large input sizes. While comparison based algorithms are limited by O(n log n), QR Sort surpasses this limitation for integer data, making it the most practical choice when the value range is not excessively large compared to n.

Among the comparison based algorithms, the performance ranking was as follows: Optimized Quick Sort, Quick Sort, Optimized Merge Sort, Merge Sort, and Heap Sort. Heap Sort occasionally outperformed Merge Sort at certain sizes because Merge Sort repeatedly allocates auxiliary arrays, which is costly in Python. In contrast, Heap Sort operates fully in place. The optimized Merge Sort variant showed a significantly larger speed improvement compared to its basic version than the optimized Quick Sort did compared to standard Quick Sort. This occurs because Merge Sort recursively splits down to single elements, so replacing small subarrays with sorting networks removes many recursive calls and temporary allocations. Quick Sort already processes small partitions efficiently using in place swaps.

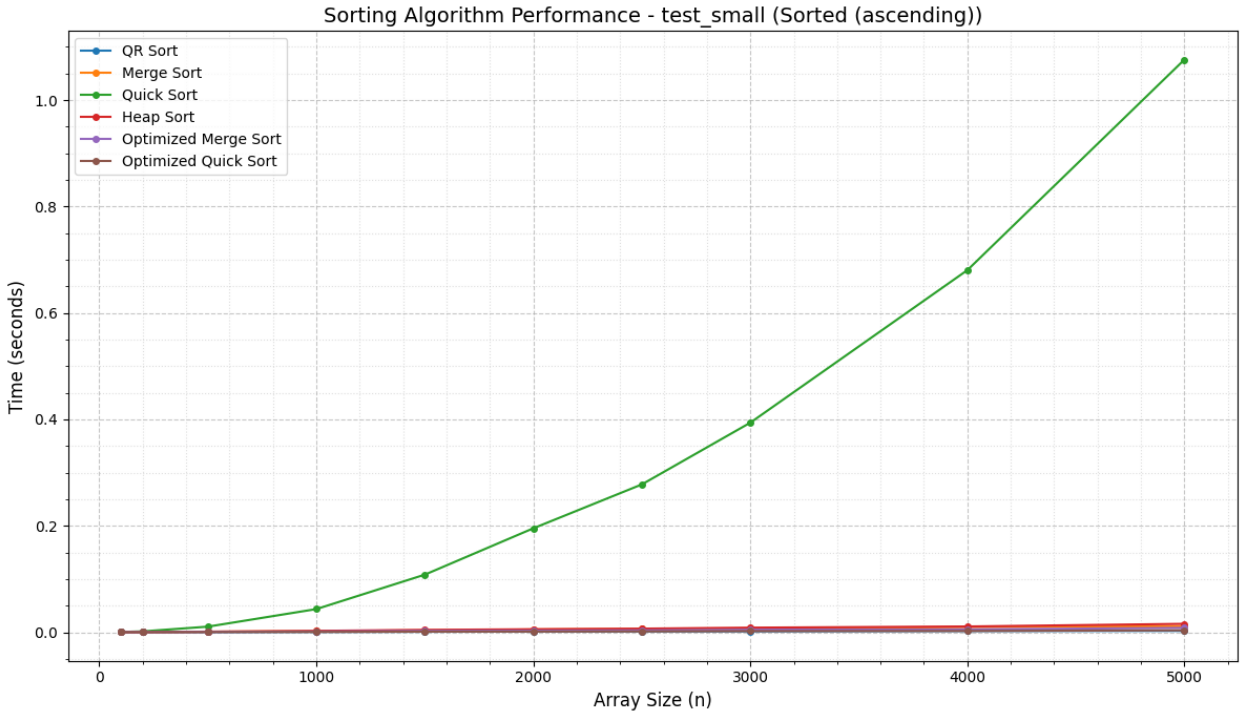
On sorted and reverse sorted inputs, basic Quick Sort degraded to $O(n^2)$, which is expected when using a last element pivot strategy. The other algorithms maintained O(n log n) behavior. The performance gap between QR Sort and Optimized Quick Sort became smaller on ordered data. Median of three pivot selection allows Optimized Quick Sort to produce nearly perfect partitions on sorted input, improving its performance. QR Sort performs two counting sort passes that run in approximately the same time regardless of input order. Partially sorted arrays with 30 to 70 percent pre sorted data followed the same performance ranking. These results are included in the notebook but were omitted from the report for brevity.

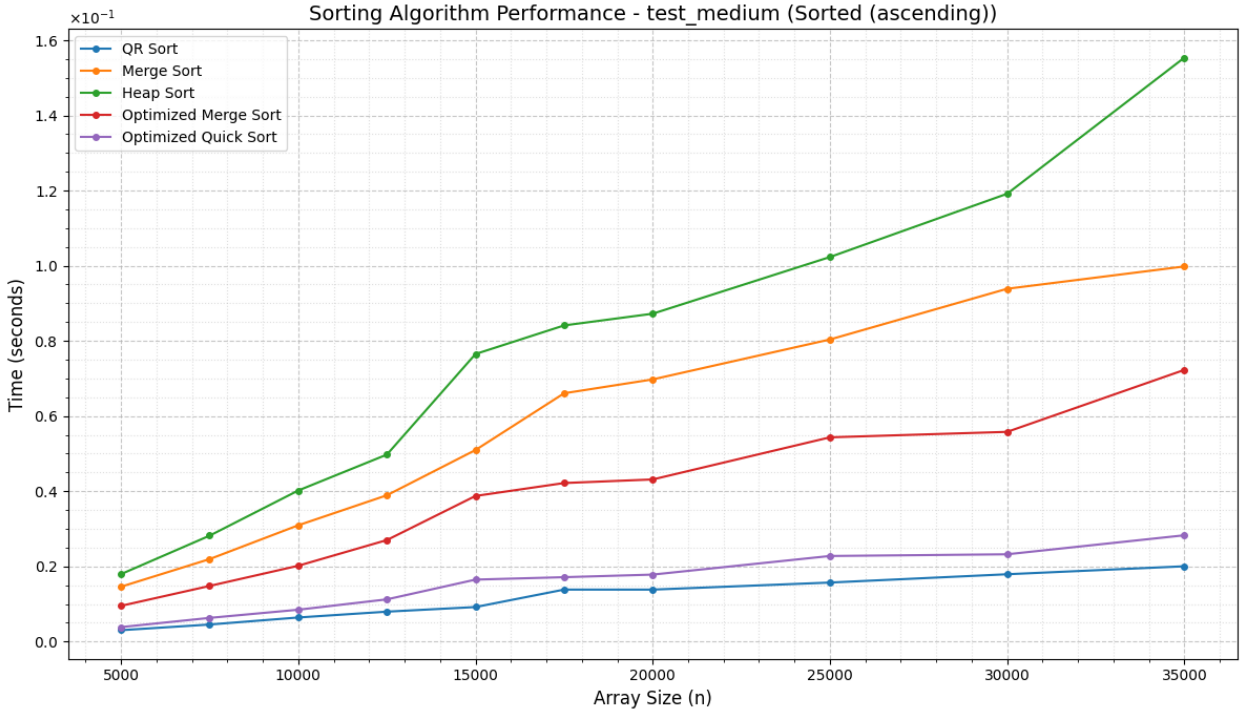
The experiments demonstrate that theoretical complexity analysis alone is not sufficient to predict real world performance. Algorithms with identical asymptotic complexity can behave very differently due to memory allocation patterns, cache efficiency, and recursion overhead. At the same time, QR Sort highlights the importance of selecting the correct algorithmic paradigm. Non comparative sorting for integer keys can achieve performance levels that comparison based algorithms cannot match at large scale.

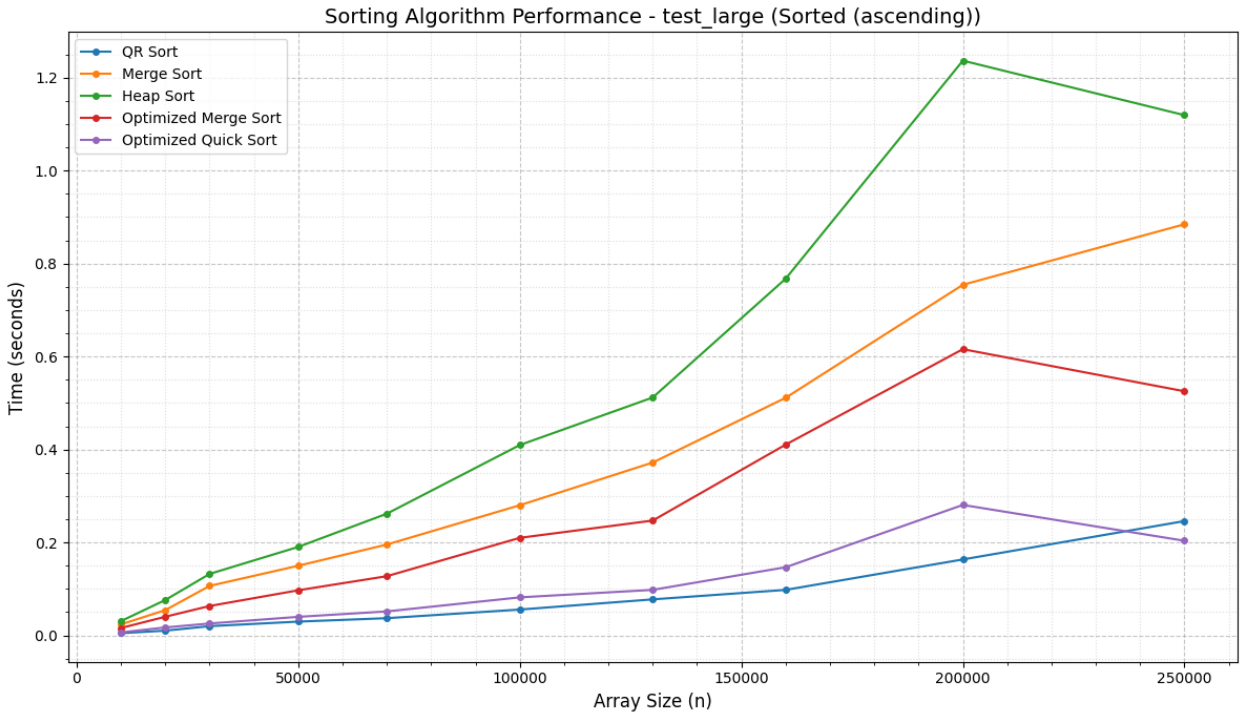
GitHub repository
https://github.com/Nichita111/AA_labs

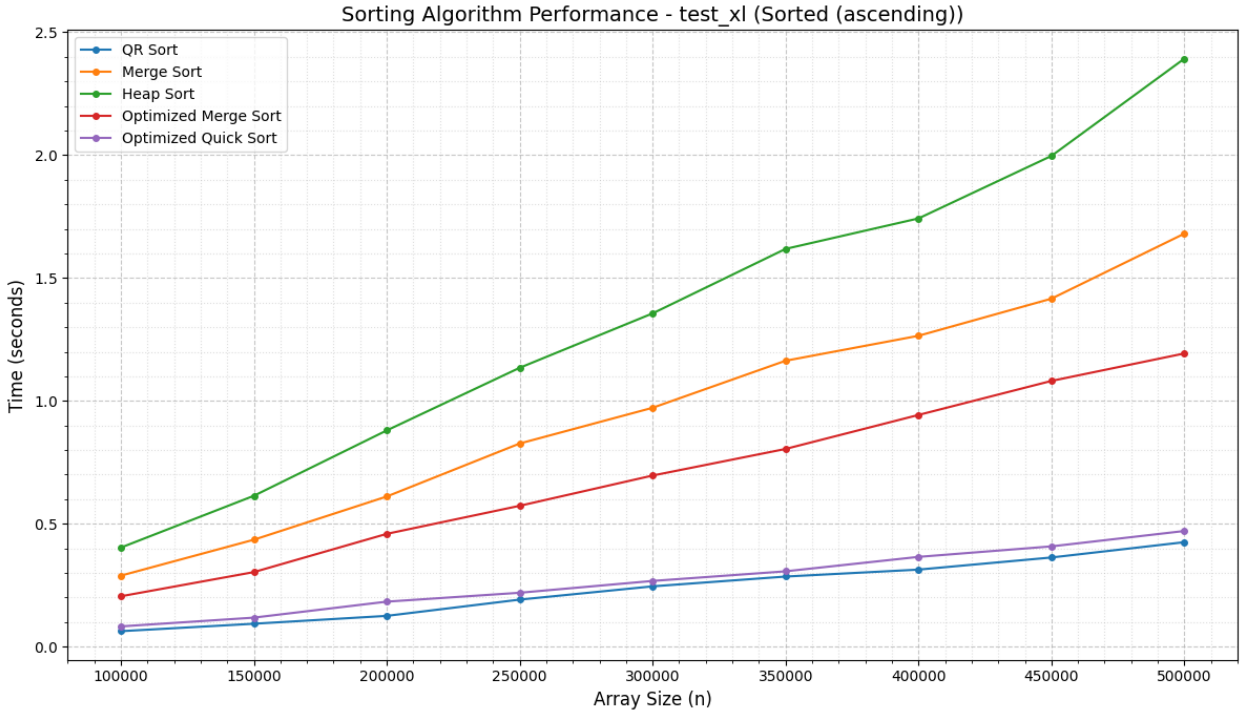
Annex A

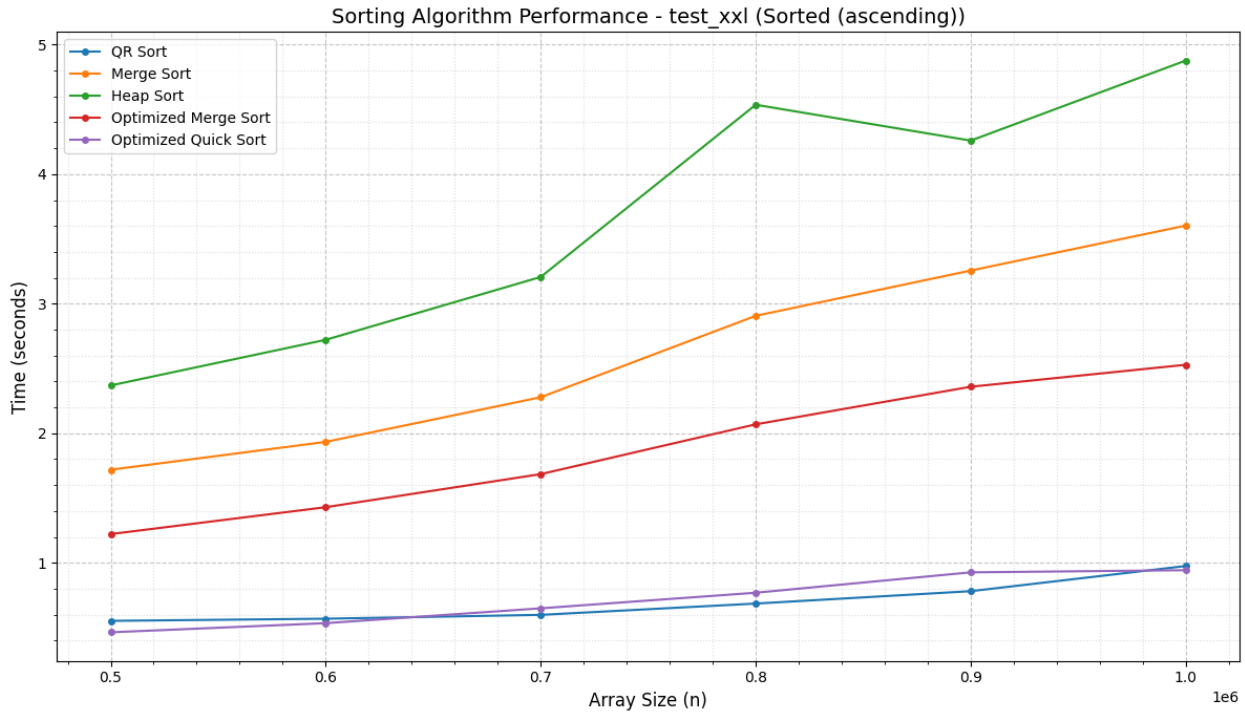
All figures from ascending sorted arrays and reverse sorted

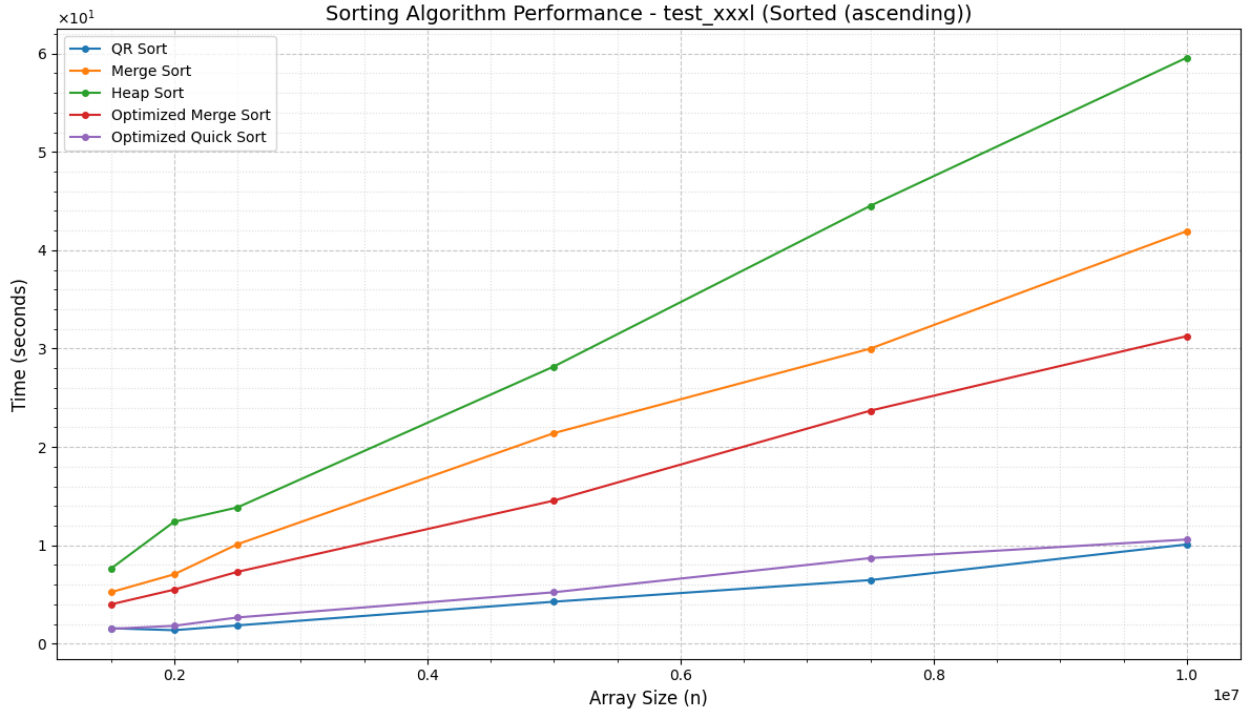


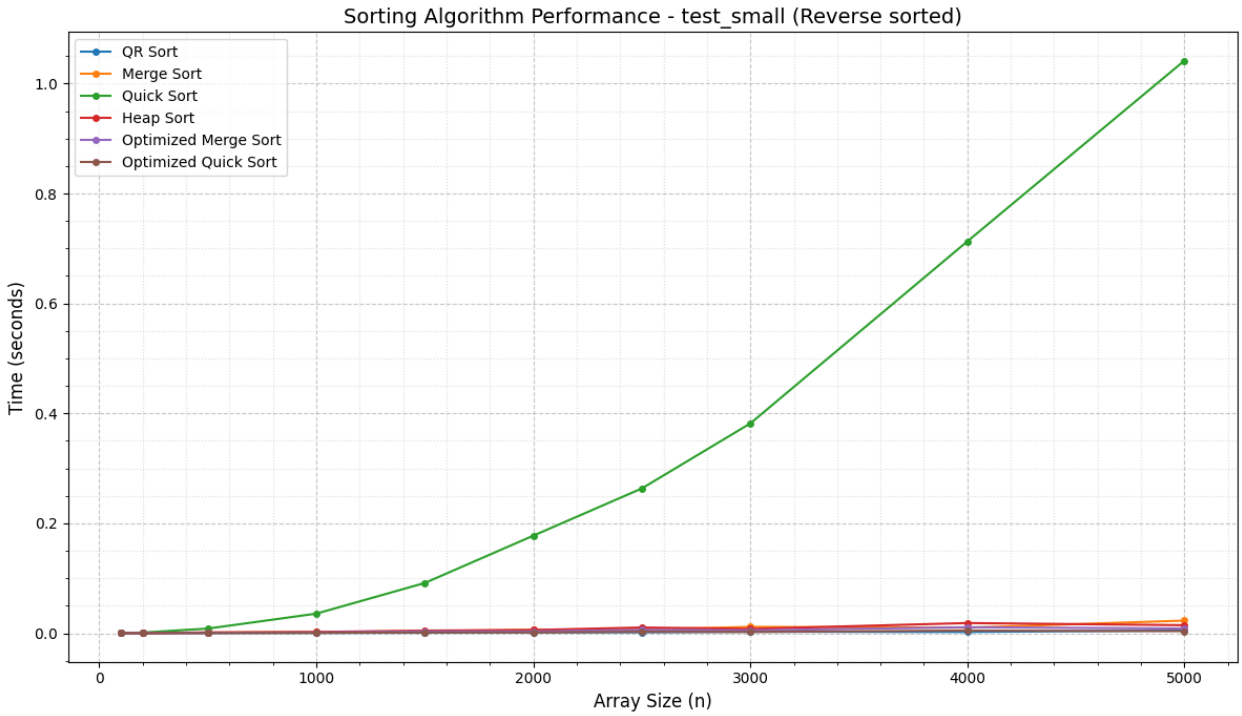


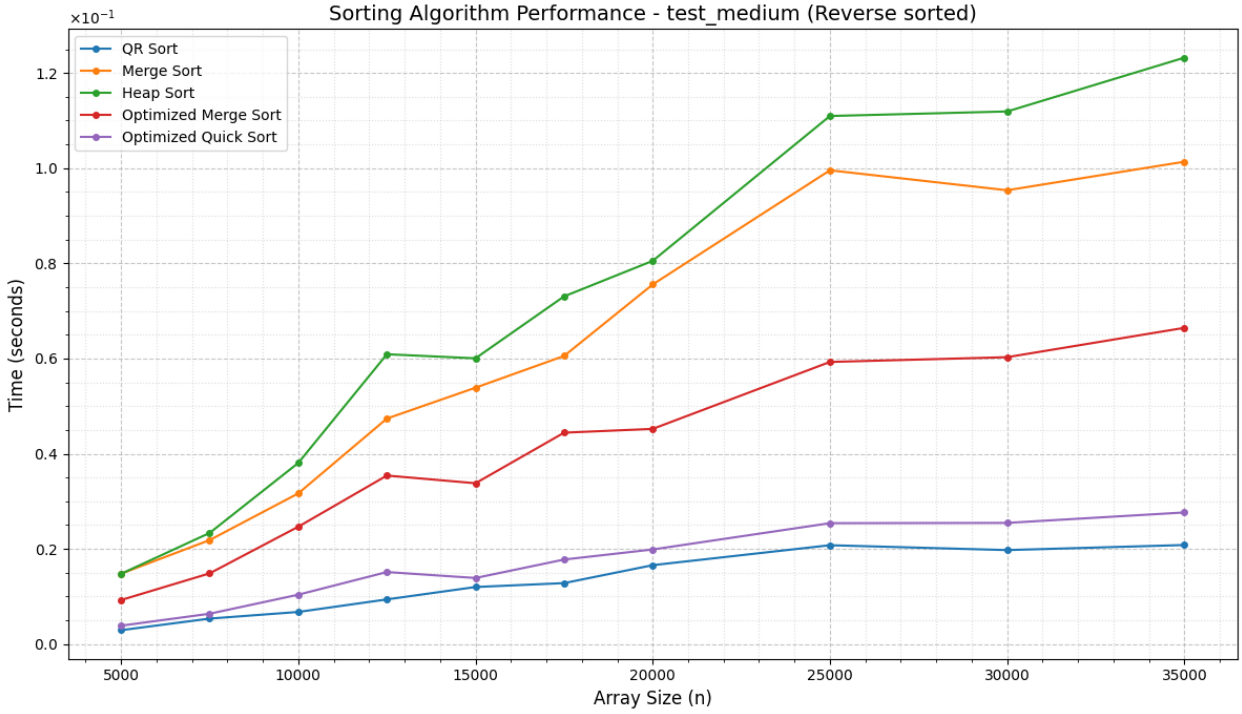

Sorting Algorithm Performance - test_large (Sorted (ascending))
QR Sort
Merge Sort
Heap Sort
Optimized Merge Sort
Optimized Quick Sort
Time (seconds)
Array Size (n)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
50000
100000
150000
200000
250000

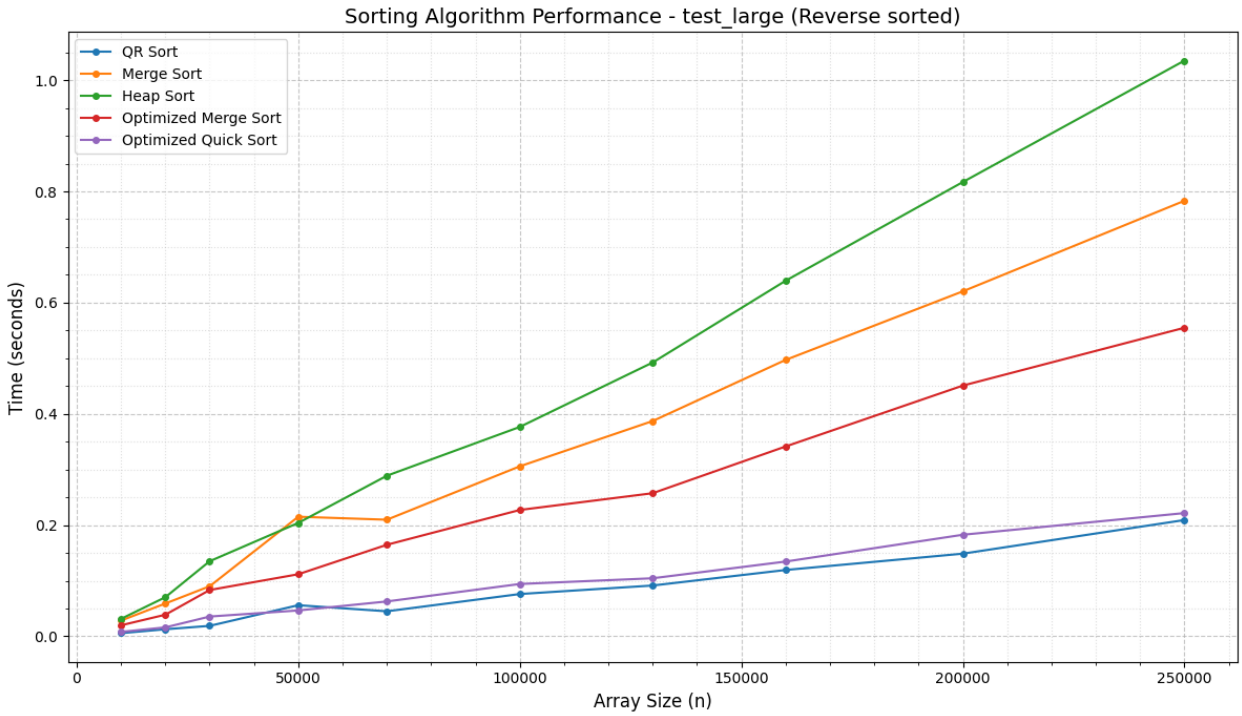

Sorting Algorithm Performance - test_xl (Sorted (ascending))
QR Sort
Merge Sort
Heap Sort
Optimized Merge Sort
Optimized Quick Sort
Time (seconds)
Array Size (n)
0.0
0.5
1.0
1.5
2.0
2.5
100000
150000
200000
250000
300000
350000
400000
450000
500000

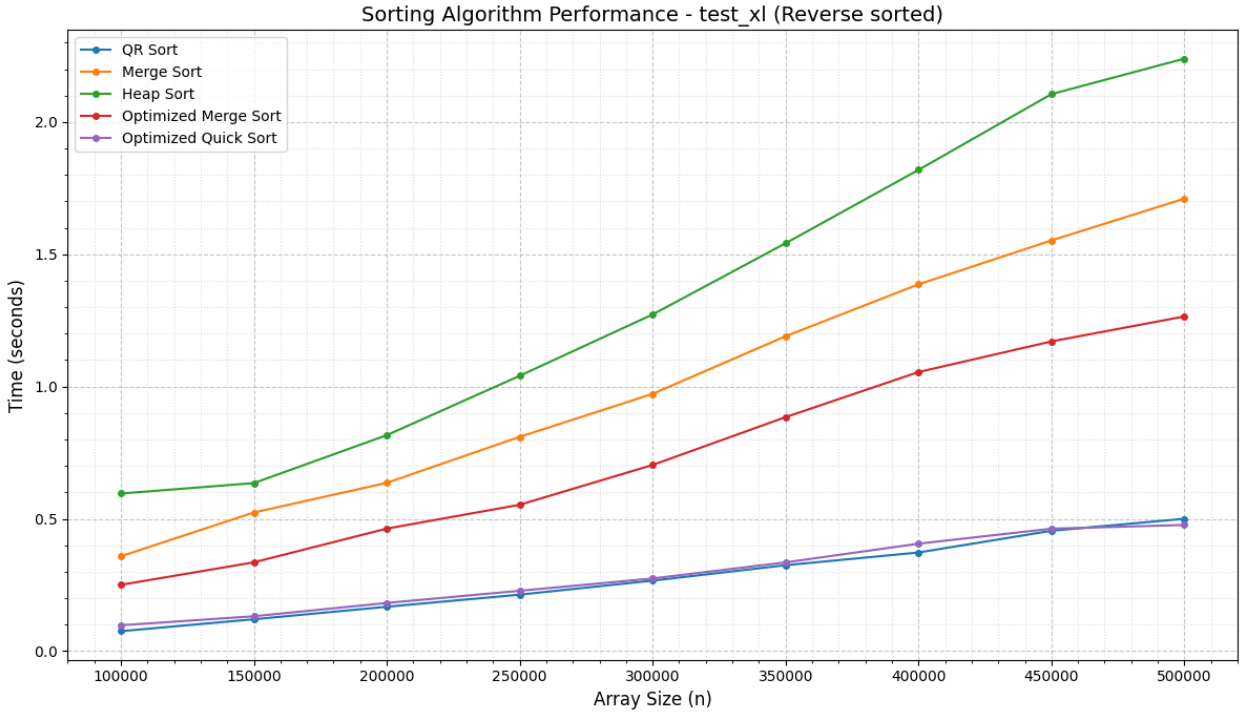

Sorting Algorithm Performance - test_xxl (Sorted (ascending))
QR Sort
Merge Sort
Heap Sort
Optimized Merge Sort
Optimized Quick Sort
Time (seconds)
Array Size (n)
1e6

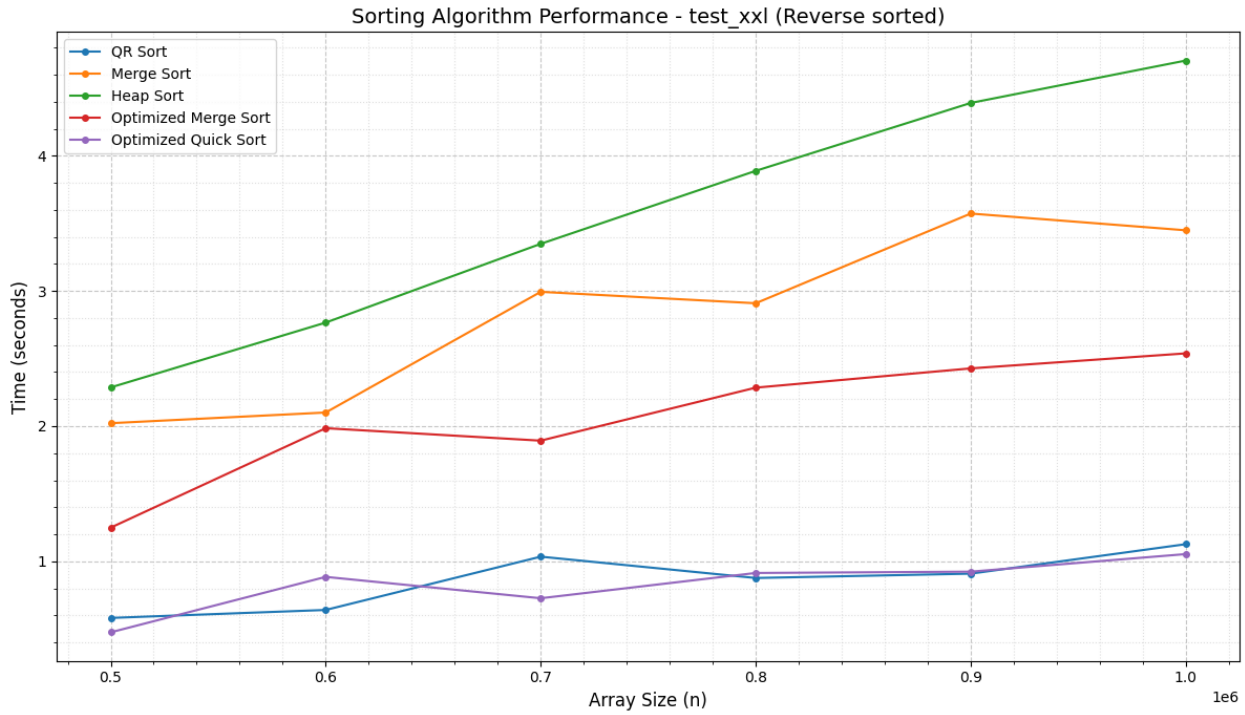

Sorting Algorithm Performance - test_xxxl (Sorted (ascending))
×10¹
QR Sort
Merge Sort
Heap Sort
Optimized Merge Sort
Optimized Quick Sort
Time (seconds)
Array Size (n)
1e7

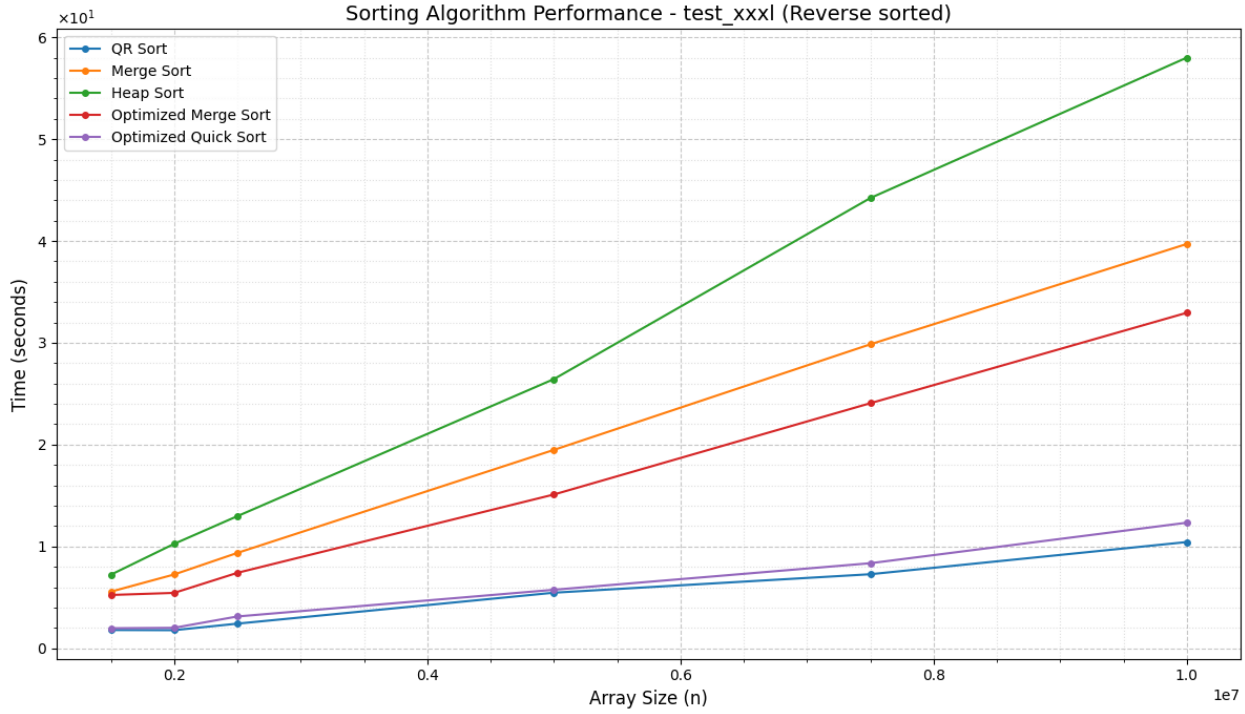

Sorting Algorithm Performance - test_small (Reverse sorted)
QR Sort
Merge Sort
Quick Sort
Heap Sort
Optimized Merge Sort
Optimized Quick Sort
Time (seconds)
Array Size (n)
0.0
0.2
0.4
0.6
0.8
1.0
0
1000
2000
3000
4000
5000


Sorting Algorithm Performance - test_medium (Reverse sorted)
×10⁻¹
QR Sort
Merge Sort
Heap Sort
Optimized Merge Sort
Optimized Quick Sort
Time (seconds)
Array Size (n)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
5000
10000
15000
20000
25000
30000
35000

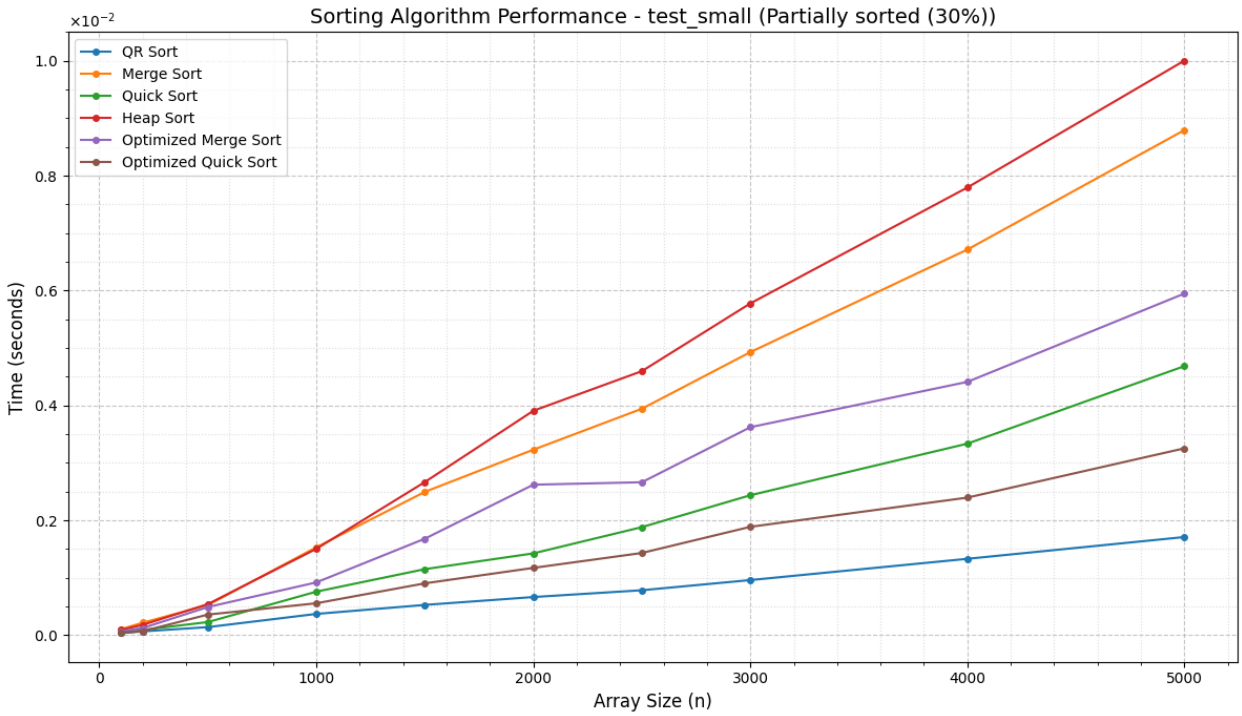

Sorting Algorithm Performance - test_large (Reverse sorted)
QR Sort
Merge Sort
Heap Sort
Optimized Merge Sort
Optimized Quick Sort
Time (seconds)
Array Size (n)
0.0
0.2
0.4
0.6
0.8
1.0
0
50000
100000
150000
200000
250000

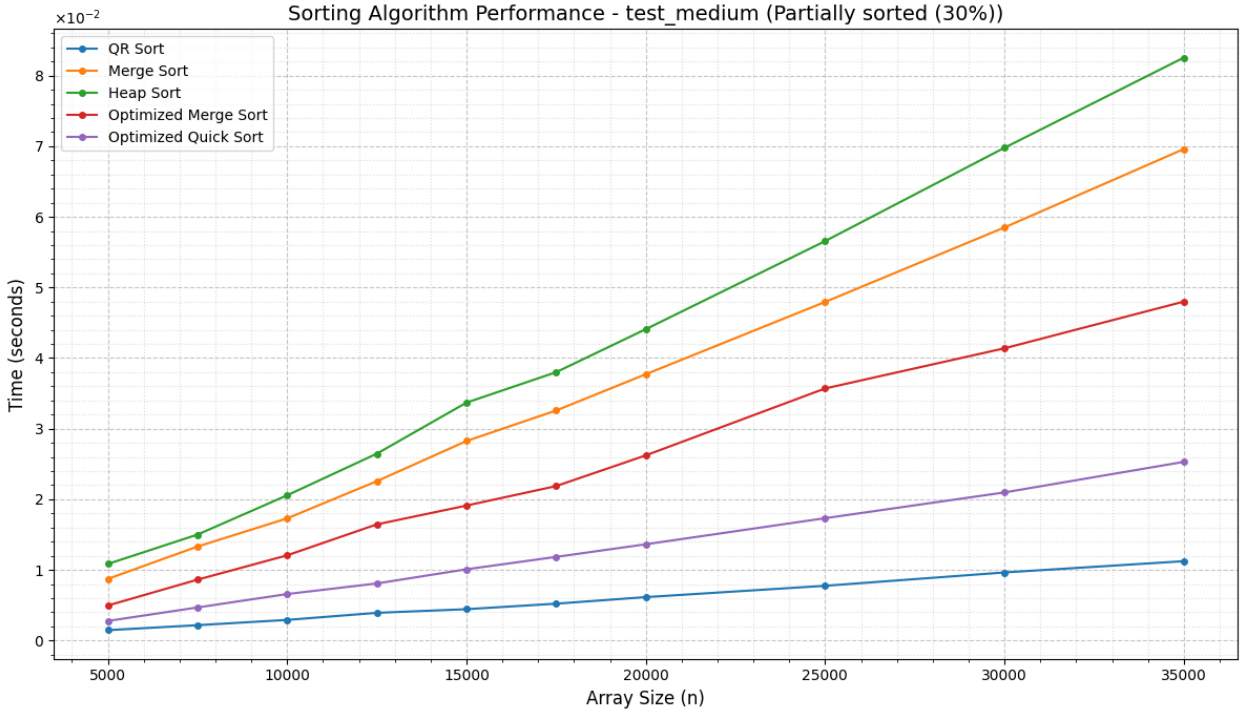

Sorting Algorithm Performance - test_xl (Reverse sorted)
QR Sort
Merge Sort
Heap Sort
Optimized Merge Sort
Optimized Quick Sort
Time (seconds)
Array Size (n)
0.0
0.5
1.0
1.5
2.0
100000
150000
200000
250000
300000
350000
400000
450000
500000



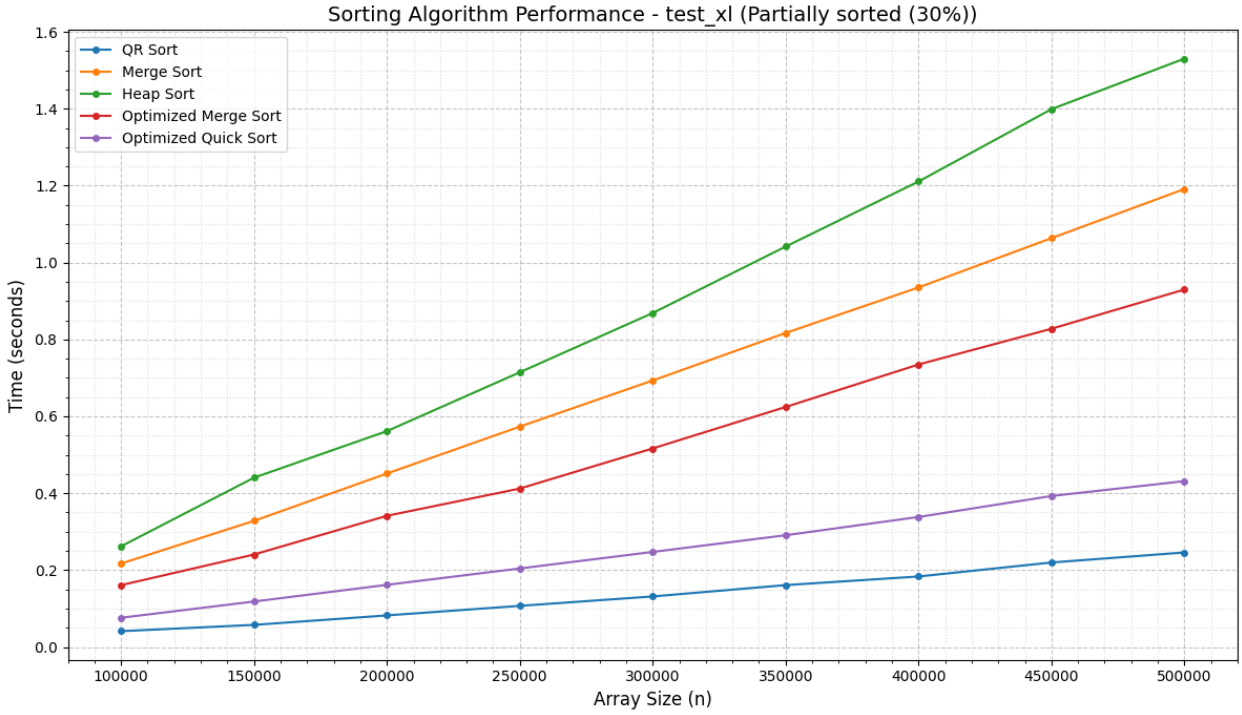


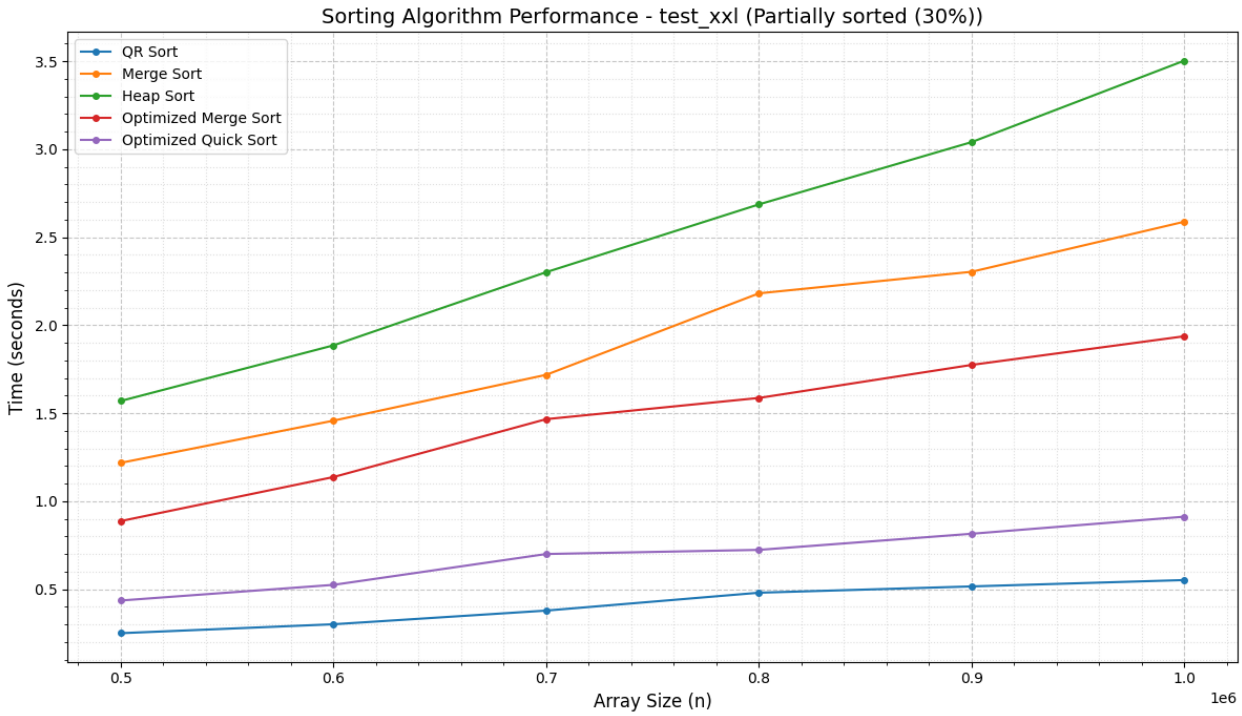
Also the partially sorted arrays

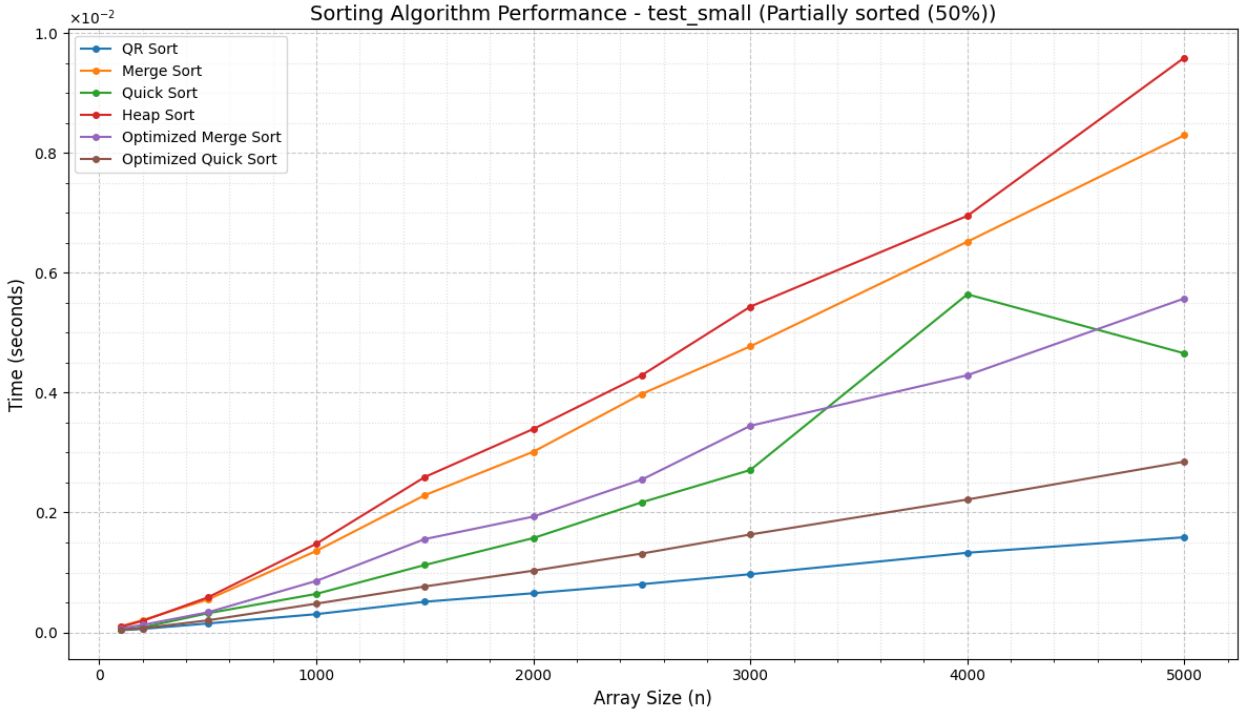

Sorting Algorithm Performance - test_small (Partially sorted (30%))
×10⁻²
QR Sort
Merge Sort
Quick Sort
Heap Sort
Optimized Merge Sort
Optimized Quick Sort
Time (seconds)
Array Size (n)
0.0
0.2
0.4
0.6
0.8
1.0
0
1000
2000
3000
4000
5000

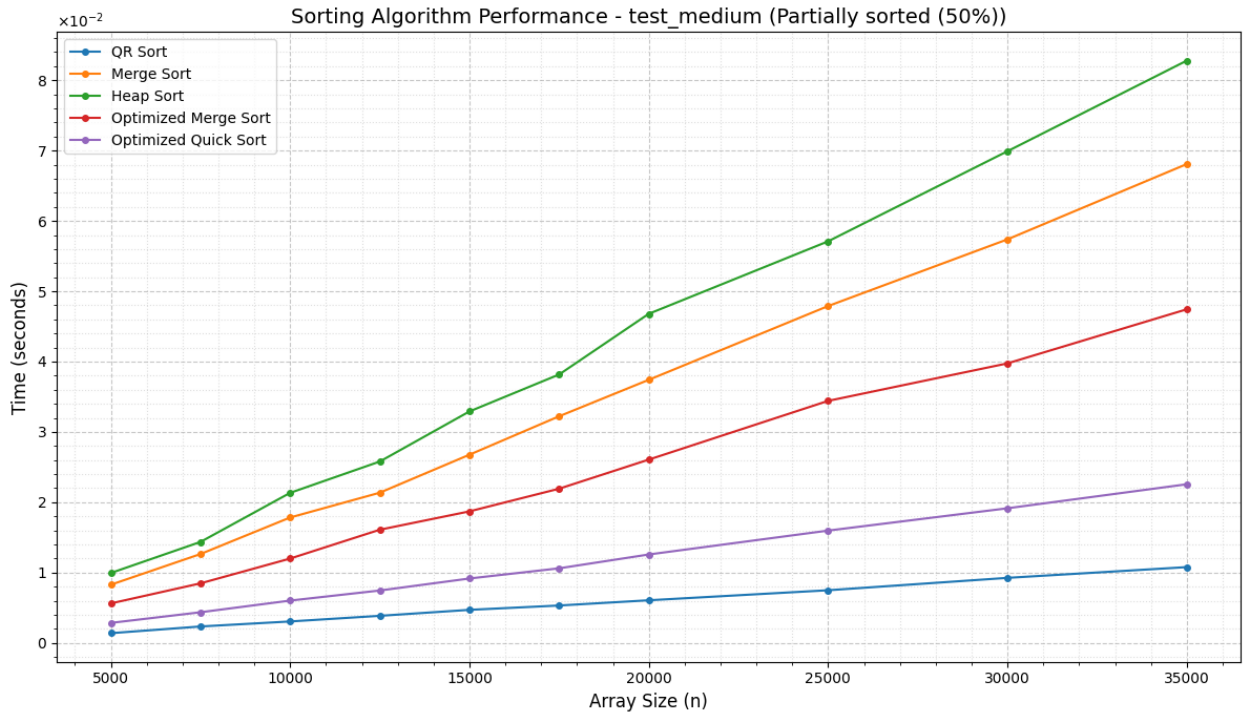

Sorting Algorithm Performance - test_medium (Partially sorted (30%))
×10⁻²
QR Sort
Merge Sort
Heap Sort
Optimized Merge Sort
Optimized Quick Sort
Time (seconds)
Array Size (n)
0
1
2
3
4
5
6
7
8
5000
10000
15000
20000
25000
30000
35000

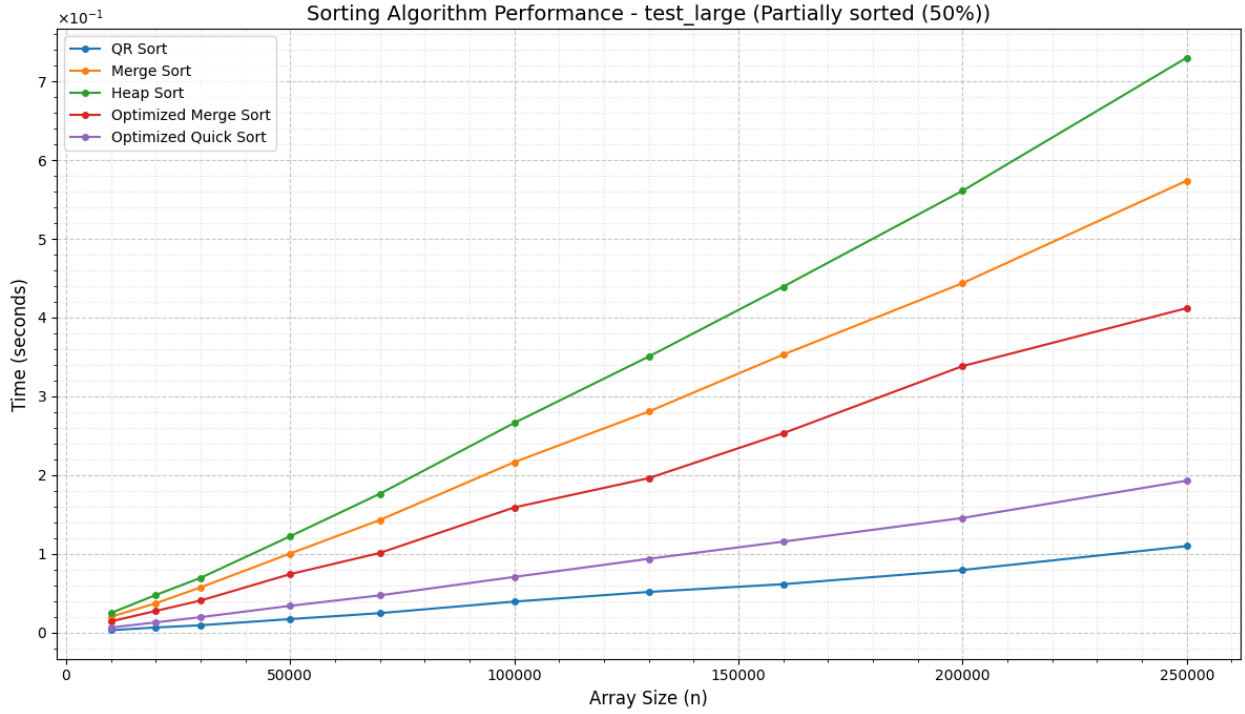

Sorting Algorithm Performance - test_large (Partially sorted (30%))
×10⁻¹
QR Sort
Merge Sort
Heap Sort
Optimized Merge Sort
Optimized Quick Sort
Time (seconds)
Array Size (n)
0
1
2
3
4
5
6
7
0
50000
100000
150000
200000
250000

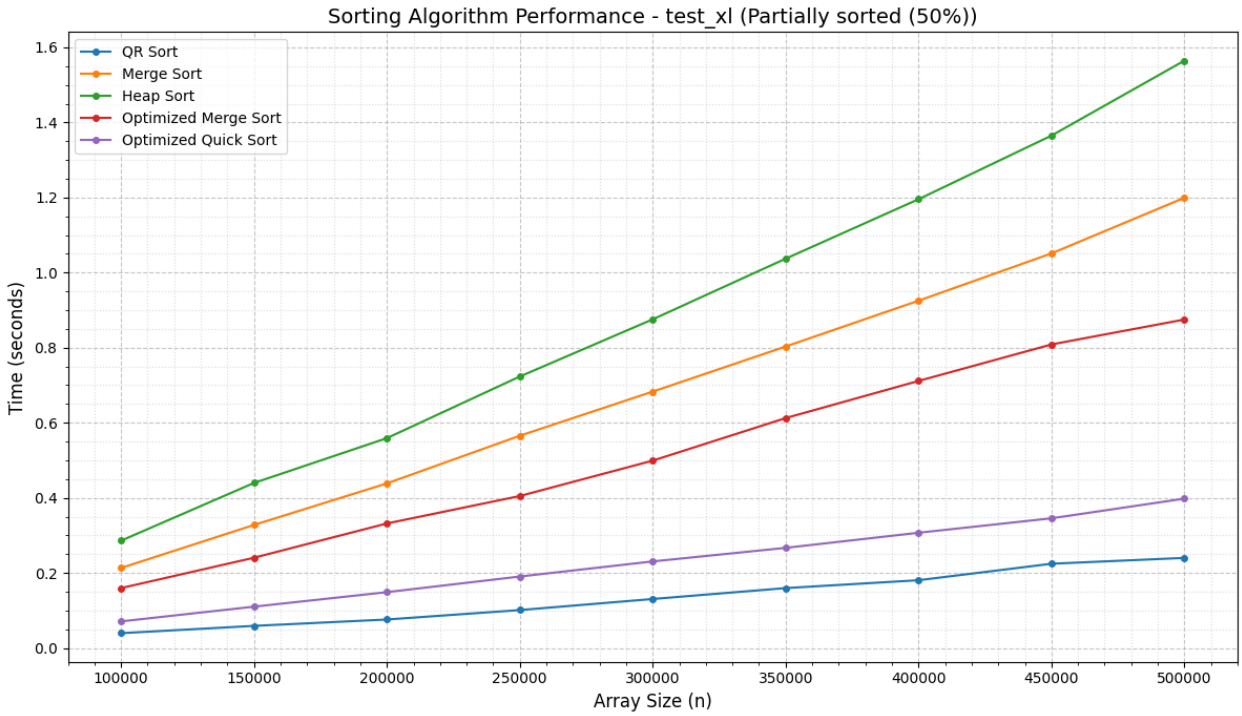

Sorting Algorithm Performance - test_xl (Partially sorted (30%))
QR Sort
Merge Sort
Heap Sort
Optimized Merge Sort
Optimized Quick Sort
Time (seconds)
Array Size (n)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
100000
150000
200000
250000
300000
350000
400000
450000
500000

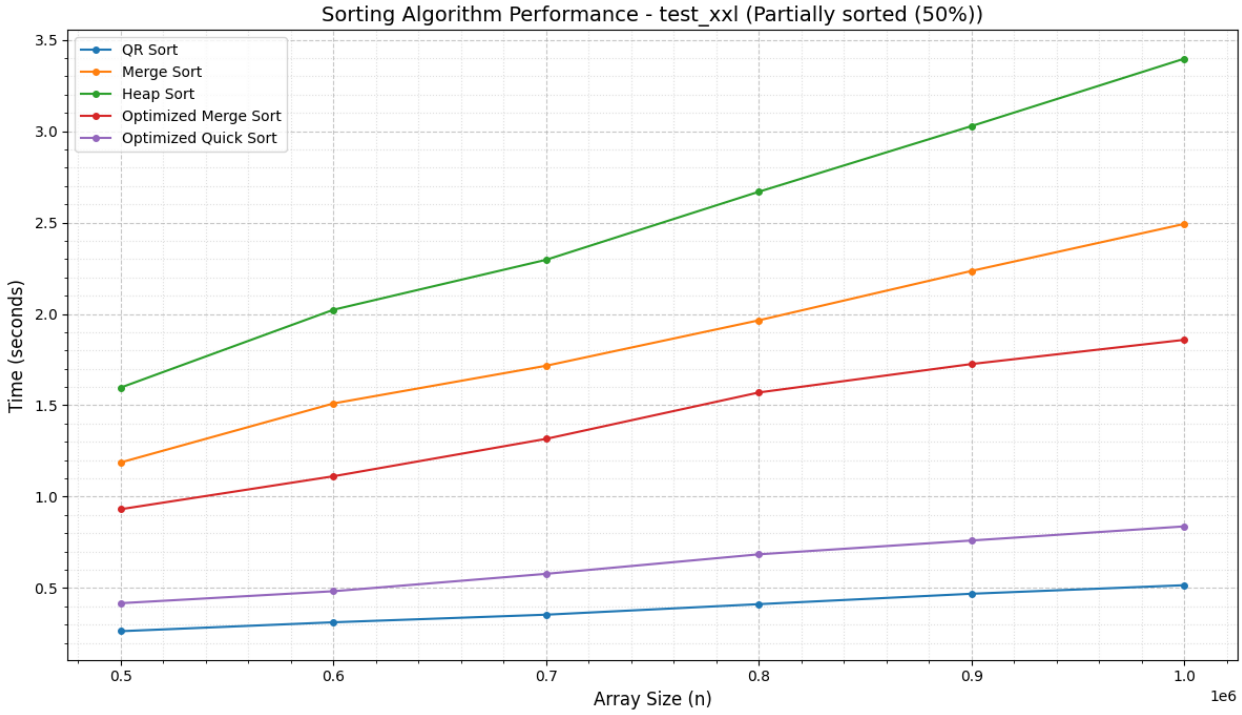

Sorting Algorithm Performance - test_xxl (Partially sorted (30%))
QR Sort
Merge Sort
Heap Sort
Optimized Merge Sort
Optimized Quick Sort
Time (seconds)
Array Size (n)
1e6

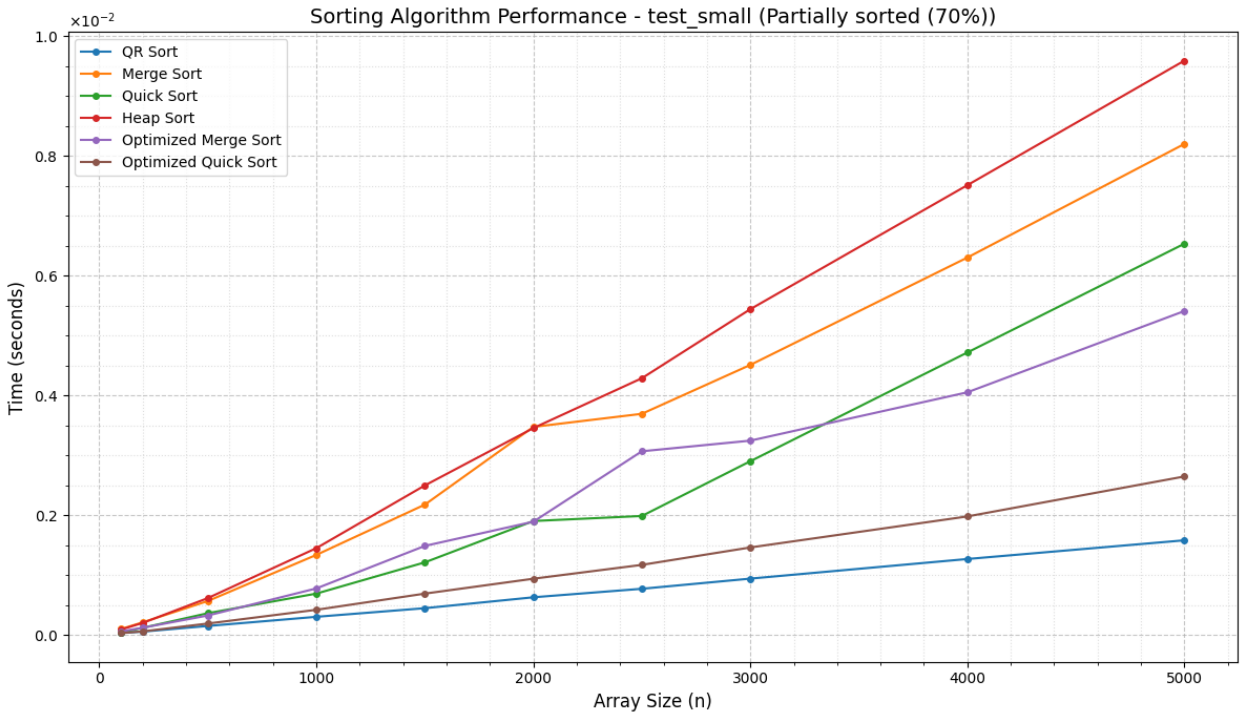

Sorting Algorithm Performance - test_small (Partially sorted (50%))
×10⁻²
QR Sort
Merge Sort
Quick Sort
Heap Sort
Optimized Merge Sort
Optimized Quick Sort
Time (seconds)
Array Size (n)

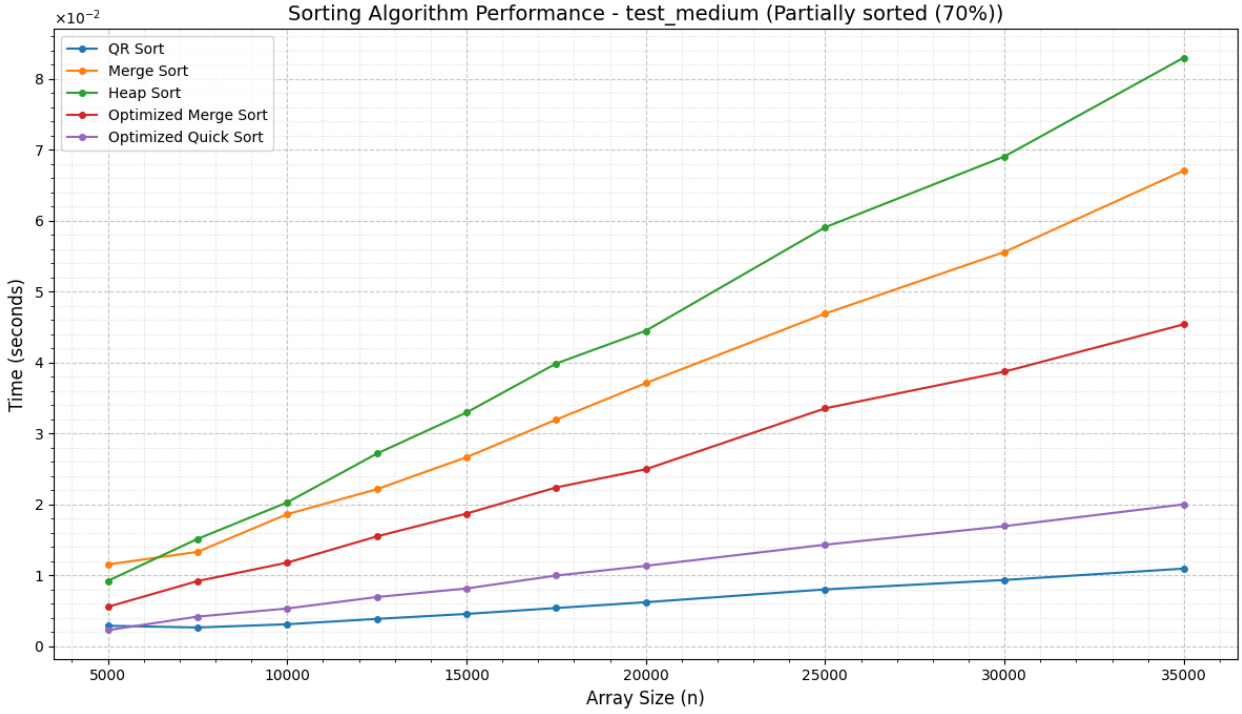

Sorting Algorithm Performance - test_medium (Partially sorted (50%))
×10⁻²
QR Sort
Merge Sort
Heap Sort
Optimized Merge Sort
Optimized Quick Sort
Time (seconds)
Array Size (n)
0
1
2
3
4
5
6
7
8
5000
10000
15000
20000
25000
30000
35000

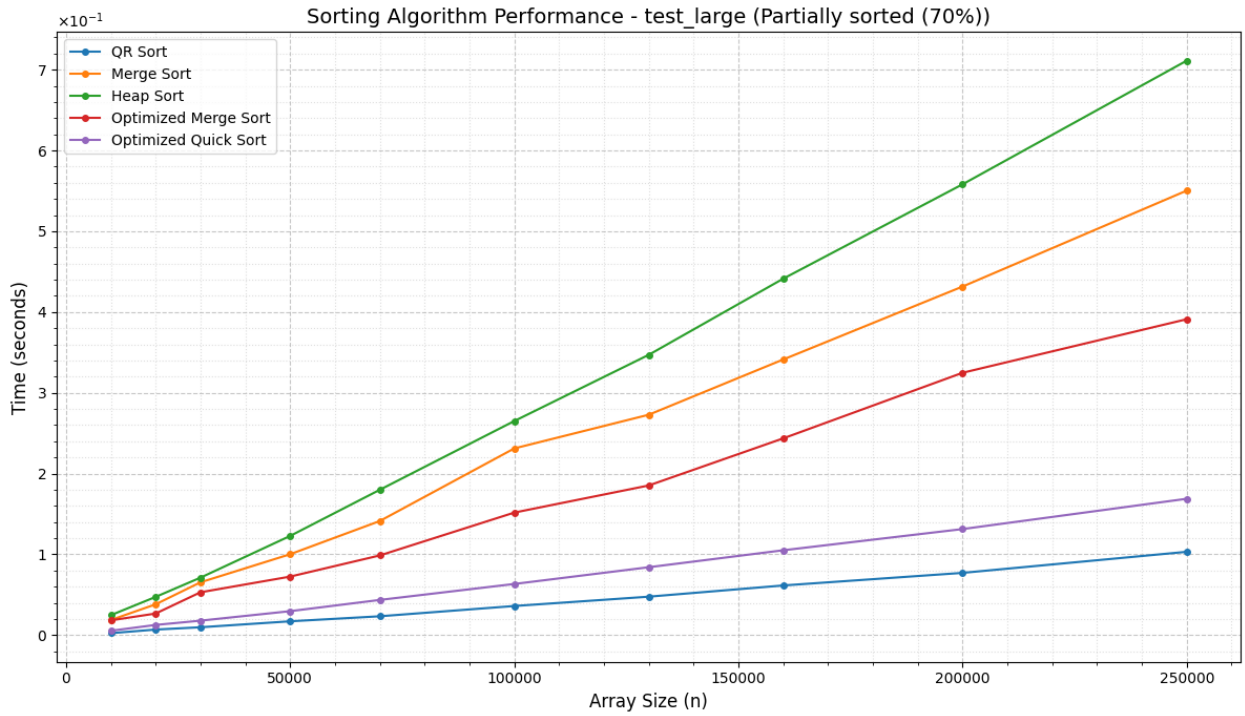

Sorting Algorithm Performance - test_large (Partially sorted (50%))
×10⁻¹
QR Sort
Merge Sort
Heap Sort
Optimized Merge Sort
Optimized Quick Sort
Time (seconds)
Array Size (n)
0
1
2
3
4
5
6
7
0
50000
100000
150000
200000
250000

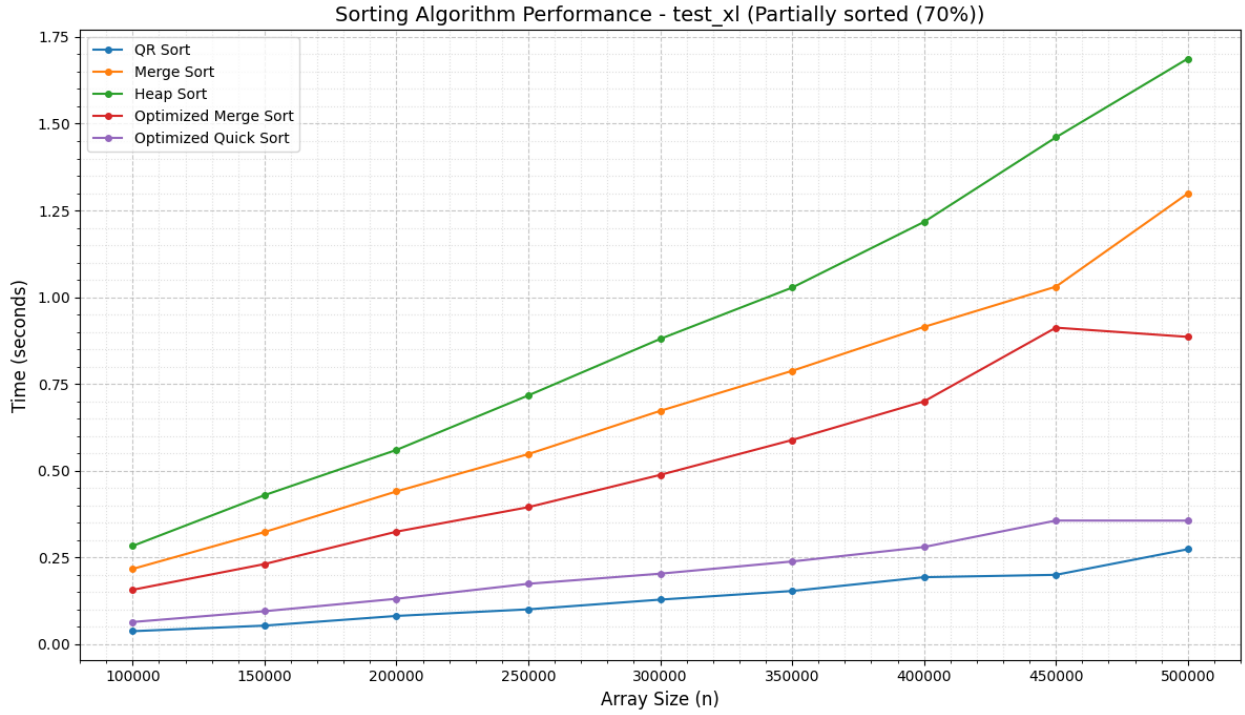

Sorting Algorithm Performance - test_xl (Partially sorted (50%))
QR Sort
Merge Sort
Heap Sort
Optimized Merge Sort
Optimized Quick Sort
Time (seconds)
Array Size (n)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
100000
150000
200000
250000
300000
350000
400000
450000
500000

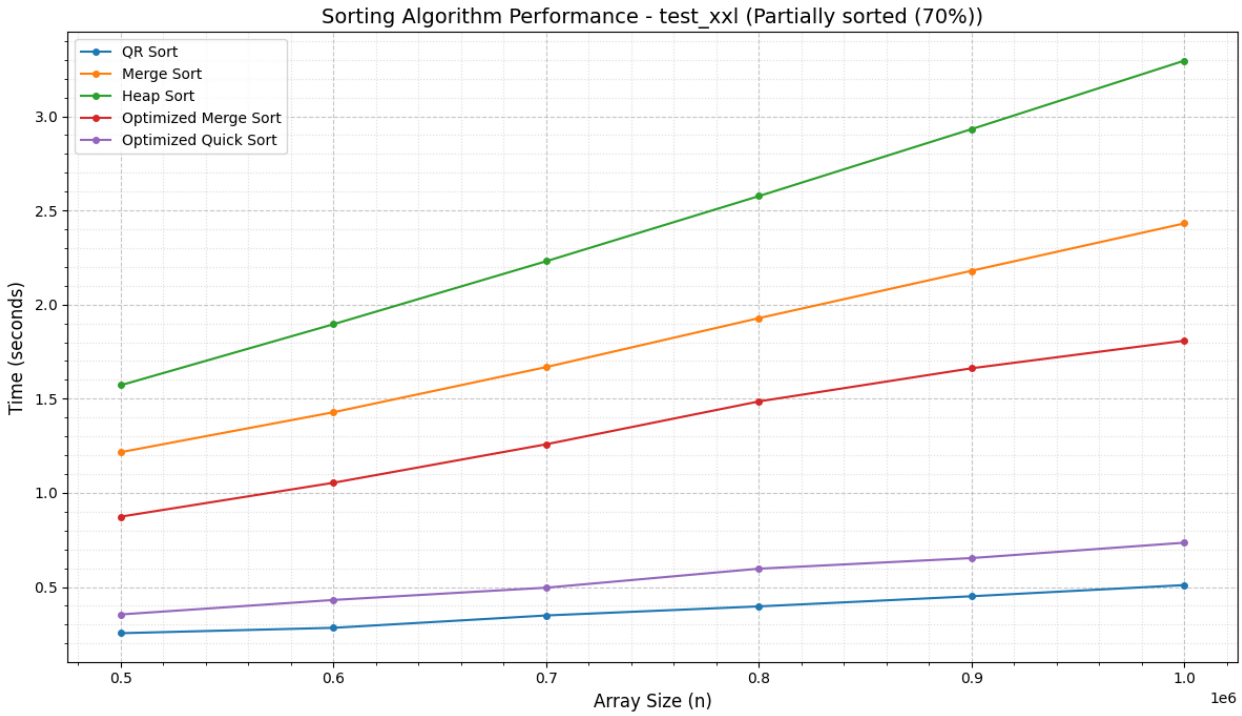

Sorting Algorithm Performance - test_xxl (Partially sorted (50%))
QR Sort
Merge Sort
Heap Sort
Optimized Merge Sort
Optimized Quick Sort
Time (seconds)
Array Size (n)
1e6
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0.5
0.6
0.7
0.8
0.9
1.0


Sorting Algorithm Performance - test_small (Partially sorted (70%))
×10⁻²
QR Sort
Merge Sort
Quick Sort
Heap Sort
Optimized Merge Sort
Optimized Quick Sort
Time (seconds)
Array Size (n)
0.0
0.2
0.4
0.6
0.8
1.0
0
1000
2000
3000
4000
5000


Sorting Algorithm Performance - test_medium (Partially sorted (70%))
×10⁻²
QR Sort
Merge Sort
Heap Sort
Optimized Merge Sort
Optimized Quick Sort
Time (seconds)
Array Size (n)
0
1
2
3
4
5
6
7
8
5000
10000
15000
20000
25000
30000
35000


Sorting Algorithm Performance - test_large (Partially sorted (70%))
×10⁻¹
QR Sort
Merge Sort
Heap Sort
Optimized Merge Sort
Optimized Quick Sort
Time (seconds)
Array Size (n)
0
1
2
3
4
5
6
7
0
50000
100000
150000
200000
250000


Sorting Algorithm Performance - test_xl (Partially sorted (70%))
QR Sort
Merge Sort
Heap Sort
Optimized Merge Sort
Optimized Quick Sort
Time (seconds)
Array Size (n)
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
100000
150000
200000
250000
300000
350000
400000
450000
500000


Sorting Algorithm Performance - test_xxl (Partially sorted (70%))
QR Sort
Merge Sort
Heap Sort
Optimized Merge Sort
Optimized Quick Sort
Time (seconds)
3.0
2.5
2.0
1.5
1.0
0.5
0.5
0.6
0.7
0.8
0.9
1.0
1e6
Array Size (n)