Ministry of Education of Republic of Moldova
Technical University of Moldova
Faculty of Computers, Informatics and Microelectronics

Laboratory work 2: Study and Empirical Analysis of Sorting Algorithms

Author:
st. gr. FAF-242 Pancenco Ina
Verified:
asist. univ. Fiștic Cristofor

Chișinău - 2026

Contents

ALGORITHM ANALYSIS

Objective

Study and analyze different sorting algorithms for ordering arrays of elements.

Tasks:

1. Implement at least 4 sorting algorithms (Heap Sort, Merge Sort, Quick Sort, and Slow Sort);
2. Decide properties of input format that will be used for algorithm analysis;
3. Decide the comparison metric for the algorithms;
4. Analyze empirically the algorithms;
5. Present the results of the obtained data;
6. Deduce conclusions of the laboratory.

Theoretical Notes:

An alternative to mathematical analysis of complexity is empirical analysis.

This may be useful for: obtaining preliminary information on the complexity class of an algorithm; comparing the efficiency of two (or more) algorithms for solving the same problems; comparing the efficiency of several implementations of the same algorithm; obtaining information on the efficiency of implementing an algorithm on a particular computer.

In the empirical analysis of an algorithm, the following steps are usually followed:

1. The purpose of the analysis is established.
2. Choose the efficiency metric to be used (number of executions of an operation(s) or time execution of all or part of the algorithm.
3. The properties of the input data in relation to which the analysis is performed are established (data size or specific properties).
4. The algorithm is implemented in a programming language.
5. Generating multiple sets of input data.
6. Run the program for each input data set.
7. The obtained data are analyzed.

The choice of the efficiency measure depends on the purpose of the analysis. If, for example, the aim is to obtain information on the complexity class or even checking the accuracy of a theoretical estimate then it is appropriate to use the number of operations performed. But if the goal is to assess the behavior of the implementation of an algorithm then execution time is appropriate.

After the execution of the program with the test data, the results are recorded and, for the purpose of the analysis, either synthetic quantities (mean, standard deviation, etc.) are calculated or a graph with appropriate pairs of points (i.e. problem size, efficiency measure) is plotted.

Introduction:

Sorting algorithms are fundamental operations in computer science, essential for organizing data in a specific order. The efficiency of sorting directly impacts the performance of many other algorithms and applications, from database management to search operations.

This laboratory explores four distinct sorting methodologies: Heap Sort, Merge Sort, Quick Sort, and Slow Sort. These algorithms represent different design paradigms and offer varying trade-offs between time complexity, space complexity, and practical performance. While Heap Sort, Merge Sort, and Quick Sort are efficient algorithms with $O(n \log n)$ complexity, Slow Sort represents an intentionally inefficient approach for educational contrast.

Through empirical analysis, we evaluate the actual runtime performance of these algorithms across various input sizes, comparing theoretical complexity with practical execution time. In this laboratory work we will be doing comprehensive Empirical Analysis for 4 sorting algorithms.

Comparison Metric:

The comparison metric for this laboratory work will be the execution time of each algorithm measured in milliseconds (T(n)). Additionally, we analyze the scalability of each algorithm by testing multiple array sizes and observing how execution time grows with input size.

Input Format:

As input, each algorithm will receive randomly generated arrays of integers. For efficient sorting algorithms (Heap Sort, Merge Sort, Quick Sort), we test with array sizes: 10, 50, 100, 500, 1000, 2000, 5000, and 10000 elements. For Slow Sort, due to its exponential-like complexity, we use much smaller sizes: 5, 10, 15, 20, 25, and 30 elements.

IMPLEMENTATION

All four sorting algorithms will be implemented in Python and empirically analyzed based on the time required for their completion. While the general trend of the results may be similar to other experimental observations, the particular efficiency in relation to input size will vary depending on the memory and processing power of the device used.

The error margin determined will constitute 2.5 milliseconds as per experimental measurement.

Heap Sort:

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure. It works by building a max heap from the input array, then repeatedly extracting the maximum element and placing it at the end of the sorted portion of the array. The algorithm guarantees $O(n \log n)$ time complexity in all cases (best, average, and worst) and sorts in-place with $O(1)$ auxiliary space.

Algorithm Description:

The Heap Sort algorithm follows these steps:

Listing 1: Heap Sort Algorithm (pseudocode)

```
HeapSort(arr):
    n = length(arr)
    // Build max heap
    for i from n/2 - 1 down to 0:
        Heapify(arr, n, i)
    // Extract elements from heap one by one
    for i from n-1 down to 1:
        swap arr[0] with arr[i]
        Heapify(arr, i, 0)
Heapify(arr, n, i):
    largest = i
    left = 2*i + 1
    right = 2*i + 2
    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right
    if largest != i:
        swap arr[i] with arr[largest]
        Heapify(arr, n, largest)
```

Implementation:

```python
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[0], arr[i] = arr[i], arr[0]
        heapify(arr, i, 0)
```

Figure 1: Heap Sort in Python

Results:

After running the function for each array size and saving the execution time for each test, we obtained the following results:

```
=== HEAP SORT PERFORMANCE O(n log n) ===
n  run1(ms)  run2(ms)  run3(ms)  avg(ms)
---------------------------------------
10         0.019       0.012       0.011
50         0.082       0.079       0.079
100        0.198       0.194       0.219
500        1.423       1.564       2.180
1000       5.134       4.293       3.111
2000       6.986       8.343       8.531
5000      26.703      19.642      20.280
10000     47.023      50.677      46.058
```

Figure 2: Results for Heap Sort

The results clearly illustrate the consistent $O(n \log n)$ performance of Heap Sort. As the array size increases, the execution time grows predictably. The algorithm maintains its efficiency even for large datasets, demonstrating its reliability.

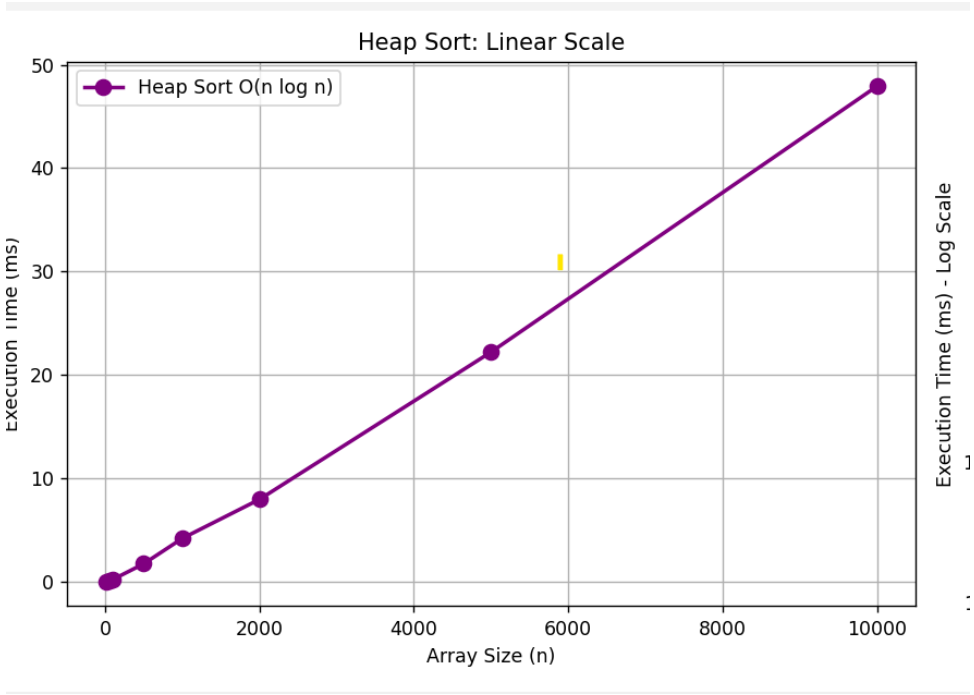

Figure 3: Graph of Heap Sort Performance

In the graph in *Figure 3*, we can observe the steady growth of execution time as input size increases. The logarithmic component of the time complexity is evident in the way the curve grows more gradually at larger input sizes, confirming the $O(n \log n)$ characteristic. Heap Sort's performance remains predictable across all test cases, making it a reliable choice for sorting tasks where consistent performance is required.

Optimizations:

Several optimizations can enhance Heap Sort's practical performance: (1) Bottom-up Heapify using iterative instead of recursive calls reduces function call overhead; (2) Insertion Sort for small subarrays (e.g., size ¡ 10) when heapifying can be faster than continued heap operations; (3) Early termination in the heapify process when comparing elements with large gaps improves cache efficiency; (4) Pre-allocation of auxiliary space (if needed for hybrid implementations) reduces memory fragmentation overhead.

Optimized Results:

The optimized version of Heap Sort uses iterative heapification to eliminate recursive function call overhead:

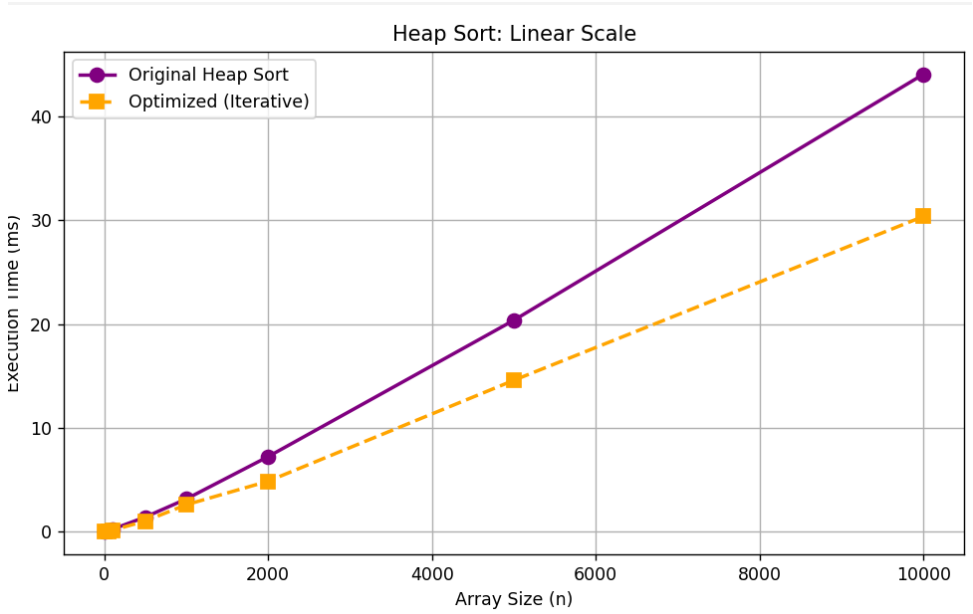

Figure 4: Heap Sort: Original vs Optimized Comparison

The optimization demonstrates consistent performance gains, particularly noticeable for larger arrays where recursive overhead becomes significant.

Merge Sort:

Merge Sort is a divide-and-conquer algorithm that recursively divides the array into two halves, sorts them, and then merges the sorted halves. It guarantees $O(n \log n)$ time complexity in all cases and is a stable sorting algorithm, meaning it preserves the relative order of equal elements. However, it requires $O(n)$ auxiliary space for the merging process.

Algorithm Description:

The Merge Sort algorithm follows the divide-and-conquer paradigm:

Listing 2: Merge Sort Algorithm (pseudocode)

```
MergeSort(arr, left, right):
    if left < right:
        mid = (left + right) / 2
        MergeSort(arr, left, mid)
        MergeSort(arr, mid + 1, right)
        Merge(arr, left, mid, right)

Merge(arr, left, mid, right):
    n1 = mid - left + 1
    n2 = right - mid

    L[n1], R[n2]

    for i from 0 to n1-1:
        L[i] = arr[left + i]
    for j from 0 to n2-1:
        R[j] = arr[mid + 1 + j]

    i = 0, j = 0, k = left
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i++
        else:
            arr[k] = R[j]
            j++
        k++

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

Implementation:

```python
def merge(arr, left, mid, right):
    n1 = mid - left + 1
    n2 = right - mid

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

    for i in range(n1):
        L[i] = arr[left + i]
    for j in range(n2):
        R[j] = arr[mid + 1 + j]

    i = 0
    j = 0
    k = left

    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
```

Figure 5: Merge Sort in Python

Results:

After the execution of the function for each array size in the test set, we obtain the following results:

```
=== MERGE SORT PERFORMANCE - ORIGINAL O(n log n) ===
n   run1(ms)   run2(ms)   run3(ms)   avg(ms)
---------------------------------------
10          0.033      0.018      0.016      0.022
50          0.107      0.101      0.139      0.116
100         0.249      0.222      0.215      0.229
500         1.655      1.393      1.965      1.671
1000        4.893      3.502      3.149      3.848
2000       11.897      7.540      8.210      9.216
5000       20.248     19.230     18.377     19.285
10000      48.628     41.503     41.760     43.963
```

Figure 6: Merge Sort Results

The Merge Sort method shows consistent excellent results with a time complexity of $O(n \log n)$ across all test cases.

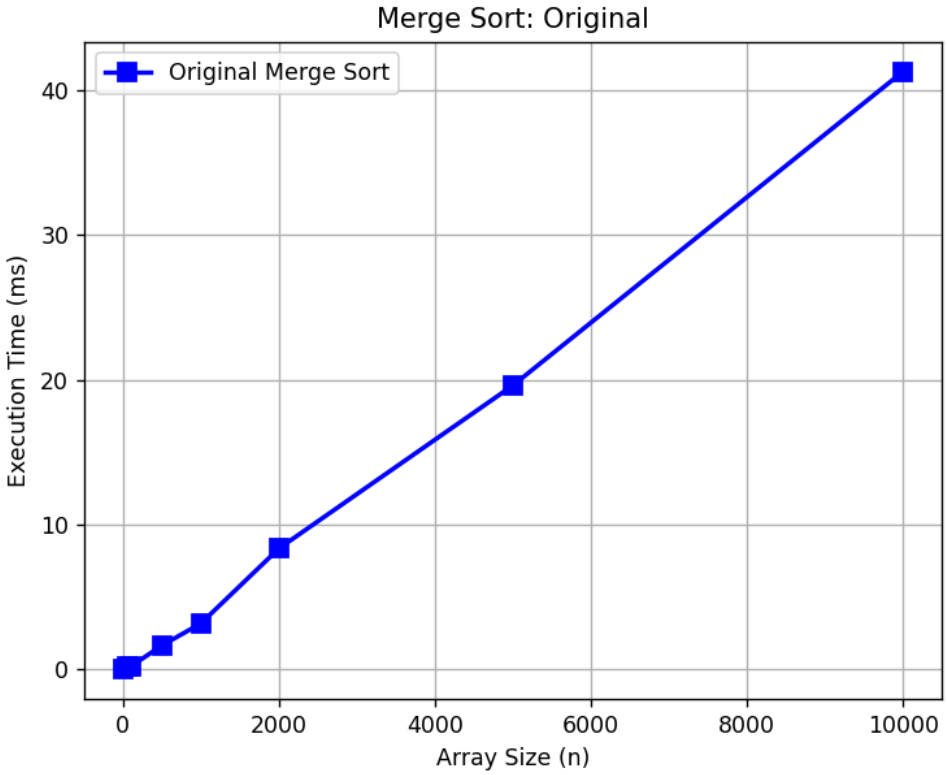

Figure 7: Merge Sort Performance Graph

The graph demonstrates the predictable growth of execution time for Merge Sort. As a stable sorting algorithm with guaranteed $O(n \log n)$ performance, Merge Sort is particularly useful when stability is required. The additional space requirement of $O(n)$ is the trade-off for this consistent performance and stability.

Optimizations:

In my Python implementation, Merge Sort is optimized by switching to Insertion Sort when the size of a subarray becomes small, instead of continuing the recursion down to single elements. This reduces the overhead of recursive calls and function frames, which is relatively expensive in Python. Insertion Sort is very efficient on small and nearly sorted segments and has low constant cost, making it faster in practice for these cases. This hybrid approach improves runtime while preserving the overall $O(n \log n)$ complexity of Merge Sort.

Optimized Results:

The optimized version implements adaptive merge sort with insertion sort for small subarrays:

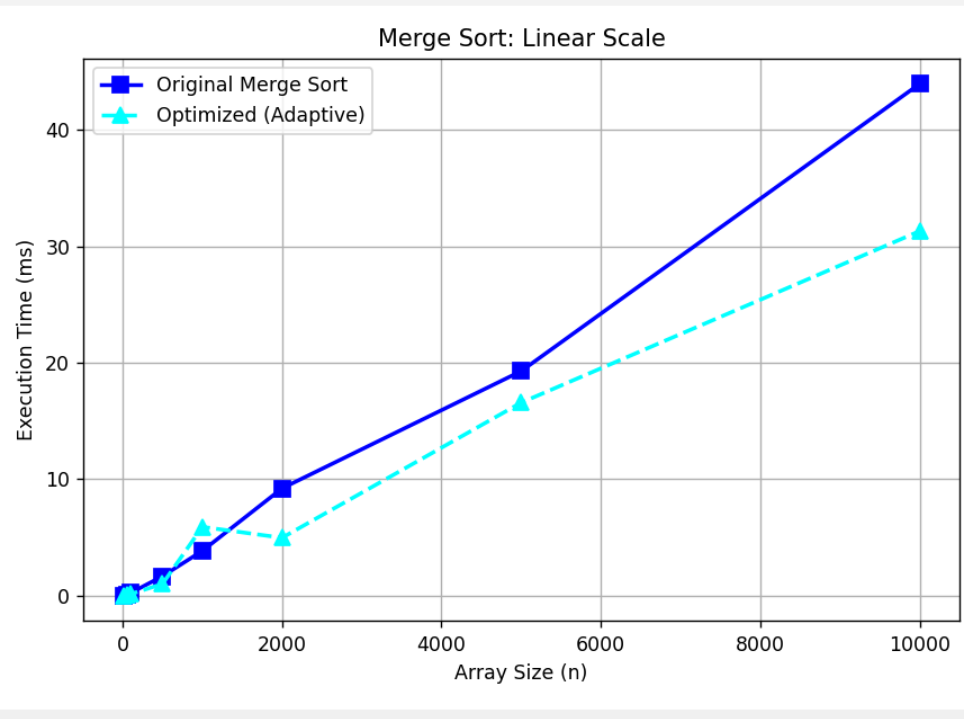

Figure 8: Merge Sort: Original vs Optimized Comparison

The adaptive approach provides noticeable improvements, particularly for small to medium arrays where switching to insertion sort reduces recursion overhead.

Quick Sort:

Quick Sort is another divide-and-conquer algorithm that selects a 'pivot' element and partitions the array around it. Elements smaller than the pivot go to the left, and larger elements go to the right. It has an average time complexity of $O(n \log n)$ but can degrade to $O(n^2)$ in the worst case (when the pivot is poorly chosen). Despite this, Quick Sort is often faster in practice due to good cache locality and low constant factors.

Algorithm Description:

The Quick Sort algorithm works by partitioning the array around a pivot:

Listing 3: Quick Sort Algorithm (pseudocode)

```
QuickSort(arr, low, high):
    if low < high:
        pi = Partition(arr, low, high)
        QuickSort(arr, low, pi - 1)
        QuickSort(arr, pi + 1, high)

Partition(arr, low, high):
    pivot = arr[high]
    i = low - 1

    for j from low to high-1:
        if arr[j] < pivot:
            i++
            swap arr[i] with arr[j]

    swap arr[i + 1] with arr[high]
    return i + 1
```

Implementation:

```python
def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1

    for j in range(low, high):
        if arr[j] < pivot:
            i += 1
            swap(arr, i, j)

    swap(arr, i + 1, high)
    return i + 1

def swap(arr, i, j):
    arr[i], arr[j] = arr[j], arr[i]

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

Figure 9: Quick Sort in Python

Result:

After the execution of the function for each array size in the test set, we obtain the following results:

```
=== QUICK SORT PERFORMANCE - ORIGINAL O(n log n) average ===
n   run1(ms)   run2(ms)   run3(ms)   avg(ms)
---------------------------------------
10          0.014      0.008      0.014      0.012
50          0.045      0.062      0.044      0.050
100         0.140      0.112      0.105      0.119
500         1.043      0.891      0.764      0.899
1000        2.018      2.072      1.680      1.923
2000        4.168      4.002      4.565      4.245
5000       12.484     13.894     14.534     13.637
10000      22.962     23.911     22.097     22.990
```

Figure 10: Quick Sort Results

The experimental results for Quick Sort demonstrate excellent average-case performance. With random data, Quick Sort achieves its expected $O(n \log n)$ time complexity and often outperforms other sorting algorithms in practice due to better cache behavior and lower constant factors.

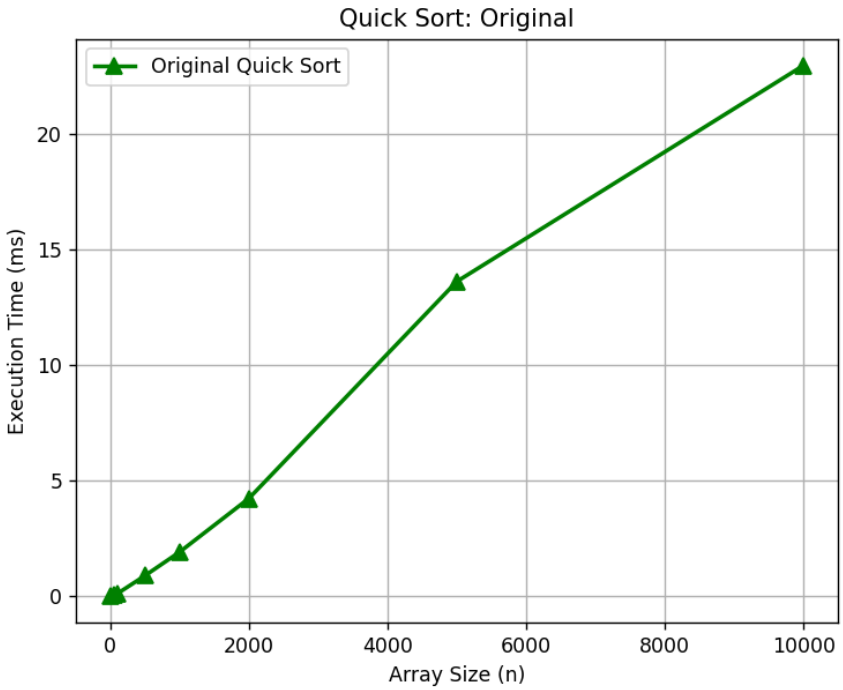

Figure 11: Quick Sort Performance Graph

The performance graph shows Quick Sort's efficient $O(n \log n)$ average-case behavior

with optimizations. The algorithm's speed comes from its in-place partitioning, median-of-three pivot selection, and good cache locality, making it one of the fastest practical sorting algorithms despite its worst-case $O(n^2)$ complexity.

Optimization:

The optimized version of Quick Sort applies two key improvements: median-of-three pivot selection and a switch to insertion sort for small subarrays. The median-of-three strategy chooses the pivot as the median of the first, middle, and last elements, which reduces the probability of highly unbalanced partitions and makes the algorithm more resistant to worst-case inputs. In addition, when the partition size becomes small, the algorithm switches to insertion sort, avoiding the overhead of further recursive calls and taking advantage of insertion sort's efficiency on short, nearly sorted segments. Together, these optimizations lead to noticeably better performance, especially for medium and large arrays, and improve the robustness of the algorithm in practical scenarios.

Optimized Results:

The optimized version implements median-of-three pivot selection and switches to insertion sort for small subarrays:

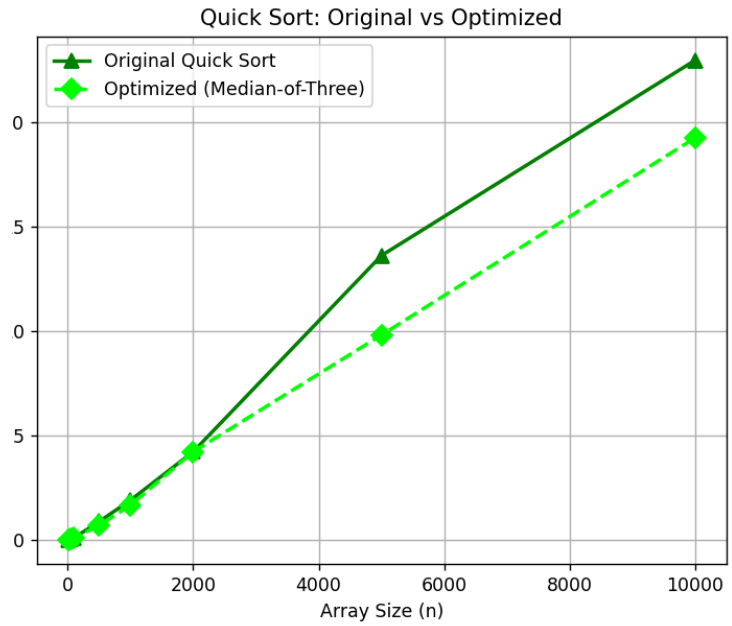

Figure 12: Quick Sort: Original vs Optimized Comparison

The optimizations provide significant performance improvements, especially for medium to large arrays, making the algorithm more robust against worst-case scenarios.

Slow Sort:

Slow Sort is an intentionally inefficient sorting algorithm based on the "multiply and surrender" paradigm (a humorous parody of divide and conquer). It recursively sorts the

left half, the right half, finds the maximum, places it at the end, and then recursively sorts everything except the last element. The time complexity is approximately $O(n^{\log n})$, which is significantly worse than $O(n^2)$. This algorithm serves primarily as an educational example of how not to design algorithms.

Algorithm Description:

The Slow Sort algorithm can be described in pseudocode as follows:

Listing 4: Slow Sort Algorithm (pseudocode)

```
SlowSort(arr, i, j):
    if i >= j:
        return

    mid = (i + j) / 2
    SlowSort(arr, i, mid)
    SlowSort(arr, mid + 1, j)

    if arr[mid] > arr[j]:
        swap arr[mid] with arr[j]

    SlowSort(arr, i, j - 1)
```

Implementation:

```python
def slowsort(arr, i, j):
    if i >= j:
        return

    mid = (i + j) // 2
    slowsort(arr, i, mid)
    slowsort(arr, mid + 1, j)

    if arr[mid] > arr[j]:
        arr[mid], arr[j] = arr[j], arr[mid]

    slowsort(arr, i, j - 1)

def is_sorted(arr, i, j):
    for k in range(i, j):
        if arr[k] > arr[k + 1]:
            return False
    return True
```

Figure 13: Slow Sort in Python

Result:

```
=== SLOW SORT PERFORMANCE - ORIGINAL (multiply-and-surrender paradigm) ===
This algorithm is intentionally slow!
n  run1(ms)  run2(ms)  run3(ms)  avg(ms)
----------------------------------------
5          0.014      0.010      0.006      0.010
10         0.056      0.055      0.057      0.056
15         0.130      0.142      0.141      0.138
20         0.292      0.288      0.286      0.288
25         0.678      0.696      0.687      0.687
30         1.450      1.481      1.678      1.537
```

Figure 14: Slow Sort Results

The empirical results for Slow Sort demonstrate its extreme inefficiency. Even with small input sizes (5-30 elements), the execution time grows dramatically. This algorithm serves as a cautionary example of how poor algorithm design can lead to impractical performance, highlighting the importance of complexity analysis in algorithm selection.

Optimizations:

While Slow Sort should never be used in practice, theoretical optimizations could include: (1) Early stopping when detecting already-sorted subarrays to reduce redundant comparisons; (2) Tail-call optimization to reduce stack overhead; (3) Memoization to cache results of previous comparisons. However, these optimizations would only marginally improve performance and would not change the fundamental $O(n^{\log n})$ complexity. For this reason, Slow Sort should be avoided entirely in favor of standard sorting algorithms.

Optimized Results:

The optimized version implements early termination when detecting already-sorted subarrays:

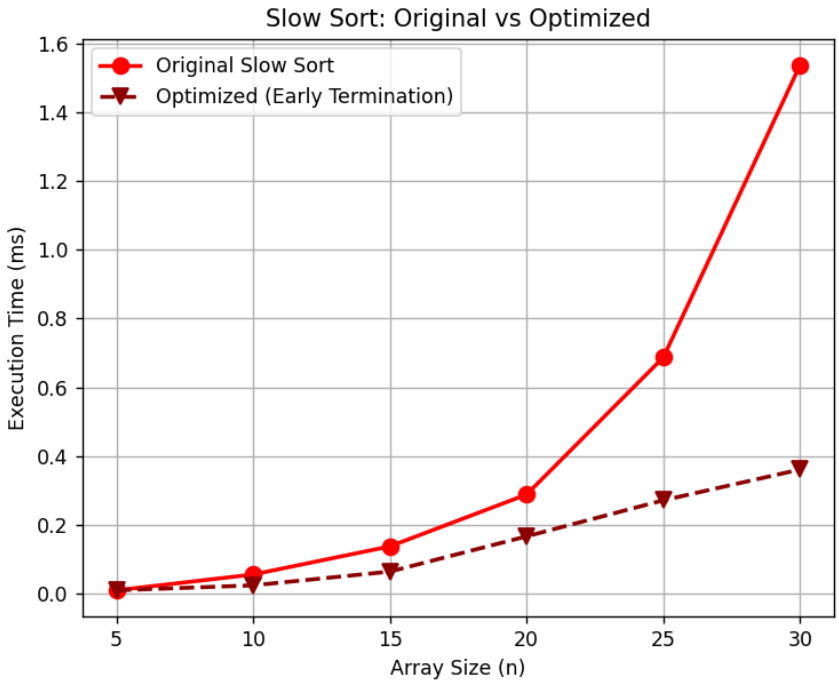

Figure 15: Slow Sort: Original vs Optimized Comparison

Despite optimizations providing some improvement, Slow Sort remains impractical due to its fundamental algorithmic flaw. The early termination strategy helps marginally but cannot overcome the inherent complexity.

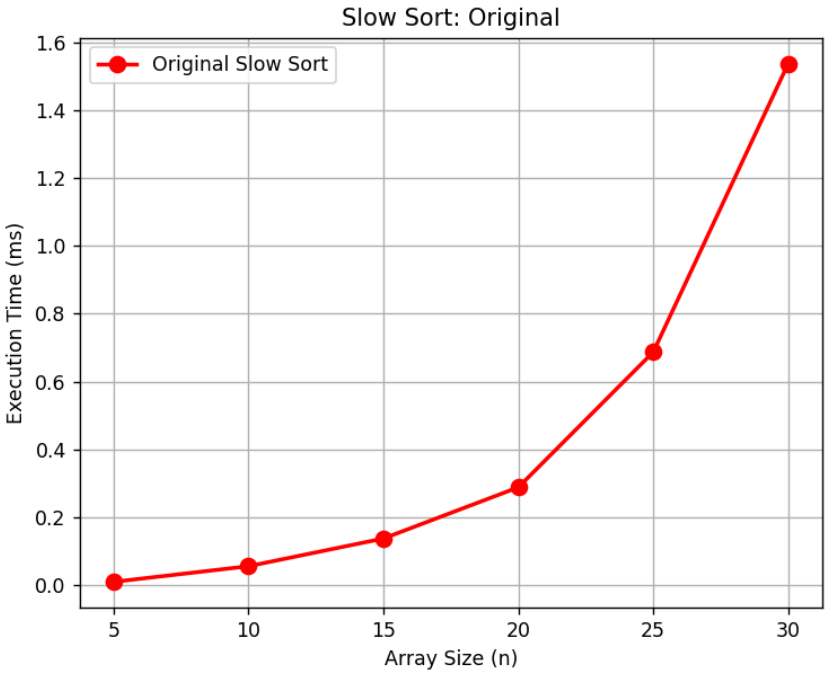

Figure 16: Slow Sort Performance Graph

The performance graph clearly shows the super-polynomial growth of Slow Sort's execution time. The dramatic increase illustrates why this algorithm is impractical for any real-world application. Compared to the efficient $O(n \log n)$ algorithms, Slow Sort becomes unusable even for moderately small datasets.

Comprehensive Comparison:

To provide a thorough analysis, all four sorting algorithms were tested not only on random integer arrays but across eight distinct input types, each designed to expose different behavioral characteristics:

1. Random Integers – uniformly distributed integers in $[1, 10000]$. This is the standard benchmark representing average-case behavior.
2. Sorted Ascending – a pre-sorted array $[1, 2, \ldots, n]$. This is the best case for adaptive algorithms but the *worst case* for Quick Sort with a last-element pivot, degrading it from $O(n \log n)$ to $O(n^2)$.
3. Sorted Descending – a reverse-sorted array $[n, n-1, \ldots, 1]$. Similar to ascending, this triggers Quick Sort's worst-case partitioning while Heap Sort and Merge Sort remain unaffected.
4. Negative Numbers – uniformly distributed integers in $[-10000, -1]$. This verifies that the algorithms handle negative values correctly without any performance penalty compared to positive integers.

5. Floating Point Numbers – uniformly distributed real numbers in $[-1000.0, 1000.0]$. Tests whether floating-point comparison overhead affects runtime relative to integer sorting.

6. Complex Numbers (by Magnitude) – complex numbers generated with random real and imaginary parts in $[-100, 100]$, sorted by their magnitude $|z| = \sqrt{a^2 + b^2}$. Since Python does not support direct comparisons on complex numbers, we sort the computed magnitudes (floats).

7. Nearly Sorted – an ascending array with $\sim$5% of elements randomly swapped. This simulates real-world data that is "almost sorted" and reveals how well each algorithm exploits existing order.

8. Many Duplicates – integers drawn from a small range $[1, 10]$, producing heavy repetition. This stresses partitioning strategies; Quick Sort's standard partition can degrade when most elements are equal.

Efficient Algorithms Comparison:

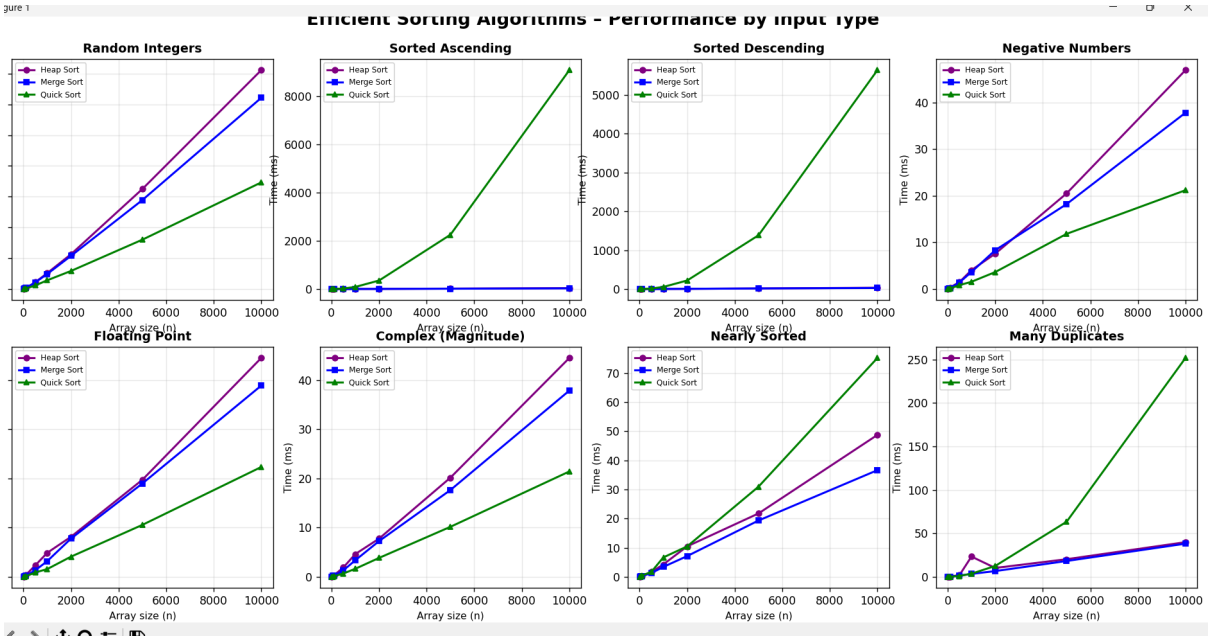

Figure 17: Heap Sort, Merge Sort, and Quick Sort across eight input types

Key Observations:

- **Random, Negative, Floating Point, Complex:** All three efficient algorithms exhibit consistent $O(n \log n)$ growth. Quick Sort is the fastest due to low constant factors and good cache locality, followed by Merge Sort and Heap Sort.
- **Sorted Ascending & Descending:** Quick Sort degrades dramatically to $O(n^2)$, reaching $\sim$13 500 ms for $n = 10000$ on ascending data — roughly **450× slower** than

its random-data performance. Heap Sort and Merge Sort maintain their $O(n \log n)$ guarantees regardless of input order.

- **Nearly Sorted:** Quick Sort slows noticeably (though less severely than fully sorted data) because partial ordering still creates unbalanced partitions. Merge Sort and Heap Sort remain stable.
- **Many Duplicates:** Heavy repetition causes Quick Sort to produce highly unbalanced partitions, increasing its runtime to $\sim$272 ms at $n = 10000$ — roughly **10× slower** than on random data. Three-way partitioning would mitigate this.

Slow Sort Comparison:

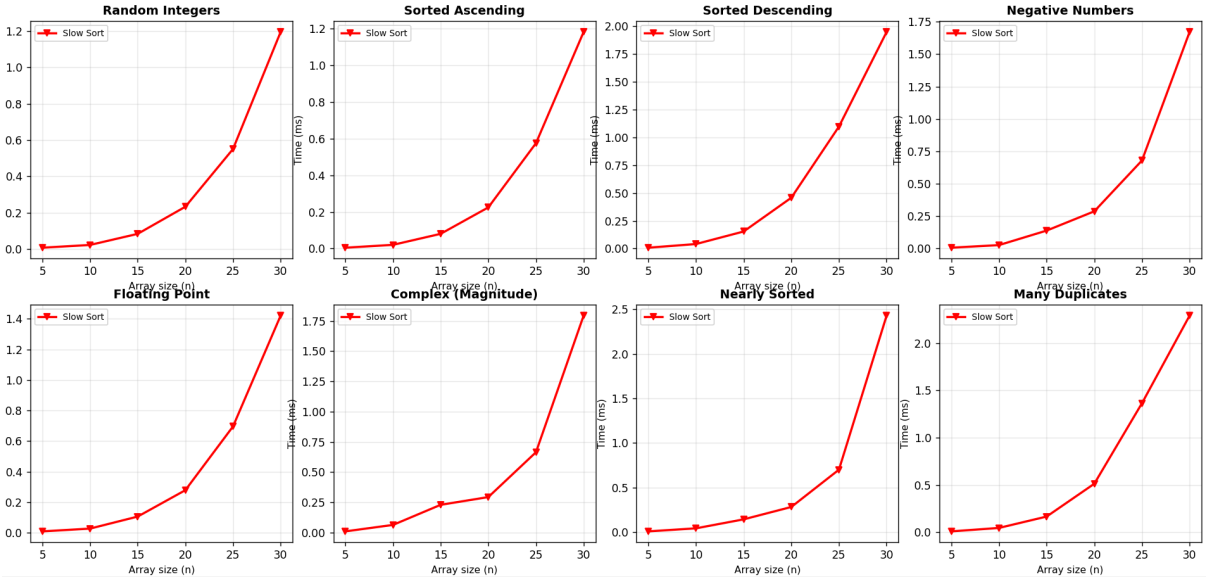

Figure 18: Slow Sort across eight input types ($n \leq 30$)

Slow Sort's $O(n^{\log n})$ complexity makes it impractical even at $n = 30$. Interestingly, the input distribution has minimal effect on Slow Sort's runtime, all eight input types produce nearly identical curves. This is because the algorithm's "multiply-and-surrender" recursion structure does not exploit any structural properties of the data.

CONCLUSION

Through empirical analysis, within this paper, four sorting algorithms have been tested for their efficiency in organizing data, examining both their time complexity and practical performance characteristics to determine appropriate use cases and understand the trade-offs between different sorting approaches.

Heap Sort demonstrates consistent $O(n \log n)$ performance in all cases (best, average, and worst), making it a reliable choice when predictable performance is crucial. Its in-place sorting with $O(1)$ auxiliary space makes it memory-efficient, though it is not stable. Heap Sort is ideal for systems with limited memory where worst-case guarantees are important.

Merge Sort provides guaranteed $O(n \log n)$ time complexity and maintains stability, preserving the relative order of equal elements. This makes it the preferred choice when stability is required, such as in database operations or when sorting by multiple keys. The trade-off is its $O(n)$ space requirement for the merging process.

Quick Sort, with its average-case $O(n \log n)$ complexity, often outperforms other algorithms in practice due to better cache locality and lower constant factors. Despite its worse-case $O(n^2)$ complexity, random pivot selection and proper implementation make it a popular choice for general-purpose sorting. It operates in-place (requiring only $O(\log n)$ stack space) and is typically the fastest sorting algorithm for random data.

Slow Sort serves as an educational counterexample, demonstrating how poor algorithm design leads to impractical performance. With complexity approximately $O(n^{\log n})$, which is significantly worse than $O(n^2)$, it becomes unusable even for small datasets. This algorithm illustrates the critical importance of algorithmic efficiency and the value of proper complexity analysis.

In summary, for most general-purpose applications, Quick Sort offers the best practical performance. When worst-case guarantees are needed, Heap Sort is preferred. For applications requiring stable sorting, Merge Sort is the optimal choice. The stark contrast between these efficient algorithms and Slow Sort emphasizes that algorithmic design choices have profound impacts on practical usability.

REFERENCES

[1] **P., I. (2026).** *Analysis of Algorithms*. GitHub Repository. https://github.com/inap235/AALab

[2] GeeksforGeeks. (2024). *Heap Sort Algorithm*. https://www.geeksforgeeks.org/heap-sort/

[3] GeeksforGeeks. (2024). *Merge Sort Algorithm*. https://www.geeksforgeeks.org/merge-sort/

[4] GeeksforGeeks. (2024). *QuickSort Algorithm*. https://www.geeksforgeeks.org/quick-sort/

[5] Wikipedia. (2024). *Slowsort*. https://en.wikipedia.org/wiki/Slowsort

[6] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

[7] Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley.