Sorting Execution Time and Comparison Analysis – Quick Sort, Hybrid Quick Sort, Top-Down & Bottom-Up Merge Sort on Synthetic Data

Abdullah Farhan - 13523042
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: farhanjunaedi213@gmail.com, 13523042@std.stei.itb.ac.id

Abstract*—This study presents an empirical comparison of four divide-and-conquer sorting algorithms: Standard Quick Sort, Hybrid Quick Sort (median-of-three pivot with insertion-sort cutoff at 10 elements), Top-Down Merge Sort (with insertion cutoff at 32), and Bottom-Up Merge Sort across four synthetic data patterns (random, ascending, descending, partially sorted) and three input sizes (500, 10,000, 100,000). Implementations in CPython 3.x were instrumented to count element comparisons and measure wall-clock execution time using time.perf_counter() (ms). Results show that Standard Quick Sort excels on random data but degrades to $O(n^2)$ on sorted inputs (RecursionError beyond n = 500). Hybrid Quick Sort mitigates worst-case behavior, reducing comparisons by ~11% at n = 100,000 for only ~8% slower runtimes. Both Merge Sort variants maintain stable Θ(n log n) performance with higher constant overheads. These findings illuminate trade-offs between average-case speed, worst-case robustness, and constant factors, offering practical guidance for algorithm selection in real-world scenarios.

***Keywords*—sorting, divide-and-conquer, Quick Sort, Merge Sort, median-of-three, insertion-sort cutoff, time complexity, comparisons**

I. Introduction

Sorting is a foundational task in computer science, where the choice of algorithm impacts both average-case speed and worst-case behavior. This study empirically compares four divide-and-conquer sorting algorithms implemented in CPython 3.x:

- **Standard Quick Sort:** uses a Lomuto partition with the last element as pivot, offering average Θ(n log n) performance but suffering $Θ(n^2)$ worst-case time on skewed inputs.
- **Hybrid Quick Sort:** enhances standard Quick Sort by selecting the median of the first, middle, and last elements as pivot and switching to insertion sort when subarray size ≤ 10 to reduce recursion overhead.
- **Top-Down Merge Sort:** recursively divides the array until subarrays ≤ 32 elements, then applies insertion sort before merging; it guarantees Θ(n log n) time and stability, using O(n) auxiliary space.
- **Bottom-Up Merge Sort:** iteratively merges runs of doubling width, with an insertion-sort cutoff at 32, delivering the same stable Θ(n log n) bound without recursion.

We evaluate these algorithms on four synthetic data patterns—random, ascending-sorted, descending-sorted, and partially sorted (95% sorted prefix)—across input sizes of 500, 10,000, and 100,000 elements. We measure wall-clock execution time (time.perf_counter(), ms) and count element comparisons to reveal trade-offs between constant factors, average-case speed, and worst-case robustness.

This paper is organized as follows: Section II reviews theoretical backgrounds; Section III details the implementation and experimental setup; Section IV presents and analyzes results; Section V concludes and suggests future work.

II. Theoretical Basis

A. Quick Sort (Regular)

Standard Quick Sort uses a **Lomuto partition** scheme with the last element as pivot [6]. During partitioning, all elements ≤ pivot go left, the rest go right, then recursion sorts each side. The average time is Θ(n log n) (T(n) ≈ 2 T(n/2) + O(n)), but a worst-case pivot (always smallest/largest) yields $Θ(n^2)$ comparisons due to unbalanced splits [1]. The algorithm is in-place, not stable, and requires O(log n) auxiliary stack space.

In this study, the conventional Quick Sort is implemented with the pivot consistently taking the last element of the subarray. The partitioning scheme employed is the Lomuto partition, where the pivot element (the last one) is swapped to ensure it is positioned at the end of the left partition. The number of comparisons counted includes each instance where the algorithm compares two elements (for example, during the partitioning loop when comparing an element with the pivot). Quick Sort is not stable (it does not maintain the order of equal elements) and operates in-place with an additional memory requirement of O(log n) for recursion.

B. *Hybrid Quick Sort (Median-of-Three + Insertion Sort)*

To enhance the performance of Quick Sort, several hybrid optimizations are recognized [3, 7]. The two techniques employed are:

1. **Median-of-Three Pivot**: select the median of the first, middle, and last elements as pivot, performing 3 extra comparisons but greatly reducing unbalanced partitions on sorted/near-sorted data [3].
2. **Insertion-sort cutoff ($k \leq 10$):** for subarrays of size up to 10, switch to insertion sort ($O(k^2)$ but faster for small k) [7].

This hybrid remains **in-place** and unstable, maintains $\Theta(n \log n)$ on average, and avoids the $\Theta(n^2)$ worst case of the standard variant while incurring only modest constant overhead.

C. *Top-Down Merge Sort (Recursive)*

Merge Sort operates on the principle of dividing an array into two equal parts, sorting each part, and then merging them back together [1]. In the top-down approach, the recursive algorithm follows these steps:

1. If the length of the array is greater than 1, split the array into two halves: left and right.
2. Recursively call Merge Sort on both the left and right sections (until reaching a base case of 1 element).
3. Merge: Combine the two sorted sections into a single sorted array.

The merging process involves repeatedly comparing the leading elements of the two sub-lists (left and right), selecting the smaller element, and inserting it into the resulting array. **Specifically, the algorithm recursively splits the array in half until each segment size is ≤ 32, then switches to Insertion Sort on those small blocks to reduce merge overhead.** Merging two sorted halves of lengths p, q takes $O(p+q)$ comparisons and moves. Total complexity is $\Theta(n \log n)$ in all cases, with $O(n)$ auxiliary space for the temporary buffer. The algorithm is stable and benefits from reduced recursion on small subarrays.

D. *Bottom-Up Merge Sort (Iterative)*

As a complement to the recursive approach above, the bottom-up variant begins by treating each individual element as a sorted run of width $w = 1$ [5]. On each pass, it doubles the run width ($w \rightarrow 2w$) and merges adjacent pairs of runs until the entire array is one sorted run. Specifically:

1. For each index i from 0 to n in steps of 2w, identify two runs: $[i \ldots \min(i+w-1, n-1)]$ and $[i+w \ldots \min(i+2w-1, n-1)]$.
2. If the combined length of a pair of runs is ≤ 32, apply **Insertion Sort** directly to that segment ($O(k^2)$ for k elements, but very fast on small k). Otherwise, perform the standard merge: compare the leading elements of both runs, copy the smaller into the auxiliary buffer, and advance until one run is exhausted, then copy the remainder.
3. Write the merged buffer back into the original array.

Repeat this process with $w = 1, 2, 4, \ldots$ until $w \geq n$. This approach removes recursion entirely and is stable—guaranteeing $\Theta(n \log n)$ time in all cases—while using $O(n)$ auxiliary space for the merge buffer. Although it remains stable with a $\Theta(n \log n)$ bound, in CPython it typically runs slightly slower than the recursive (top-down) version due to Python's loop overhead, but it avoids function-call costs and provides a clear iterative flow.

III. METHOD

A. *Environment and Implementation*

Experiments were conducted in **CPython 3.x** on a standard desktop workstation. The sorting implementations are:

- **Standard Quick Sort**: Lomuto partition with the last element as pivot.
- **Hybrid Quick Sort**:
    1. **Median-of-three pivot** (first, middle, last elements) to avoid worst-case partitions.
    2. **Insertion-sort cutoff** for subarrays of size ≤ 10 to reduce recursion overhead.
- **Top-Down** and **Bottom-Up Merge Sort**: both switch to insertion sort on subarrays of size ≤ 32 before merging to optimize small joins.

The Quick Sort and hybrid Quick Sort algorithms were instrumented to count the number of element comparisons during partitioning and insertion sort, and the Merge Sort variants likewise track each comparison. Execution time is measured via time.perf_counter() (ms resolution) to capture wall-clock performance. To ensure identical inputs, each array is copied before sorting, and random.seed(42) is set once at the very beginning for reproducible random data.

Recursion limit: CPython's default recursion limit is exceeded by the pure Quick Sort on sorted or reverse-sorted arrays when $n > 1000$ (raising RecursionError). Hence, in ascending/descending tests we only run the standard Quick Sort up to $n = 500$, marking larger sizes as "N/A.".

Below is the full Python code used for setup and all four algorithms:

```
import random
import time
N_VALUES = [500, 10000, 100000]
comparison_count = 0


def reset_comparison_count():
    global comparison_count
    comparison_count = 0


def increment_comparison():
    global comparison_count
    comparison_count += 1
```

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

def partition(arr, low, high):
    pivot, i = arr[high], low - 1
    for j in range(low, high):
        increment_comparison()
        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 insertion_sort(arr, low, high):
    for i in range(low+1, high+1):
        key, j = arr[i], i-1
        while j >= low:
            increment_comparison()
            if arr[j] > key:
                arr[j+1] = arr[j]
                j -= 1
            else:
                break
        arr[j+1] = key

def median_of_three(arr, low, mid, high):
    a, b, c = arr[low], arr[mid], arr[high]
    if (a - b)*(c - a) >= 0: return low
    if (b - a)*(c - b) >= 0: return mid
    return high

def modified_quick_sort(arr, low, high):
    TH = 10
    if high - low + 1 <= TH:
        insertion_sort(arr, low, high)
    else:
        mid = (low + high)//2
        m = median_of_three(arr, low, mid, high)
        arr[m], arr[high] = arr[high], arr[m]
        pi = partition(arr, low, high)
        modified_quick_sort(arr, low, pi-1)
        modified_quick_sort(arr, pi+1, high)

# --- Merge Sort Variants ---
def merge(arr, aux, left, mid, right):
    i, j, k = left, mid+1, left
    while i <= mid and j <= right:
        increment_comparison()
        if arr[i] <= arr[j]:
            aux[k], i = arr[i], i+1
        else:
            aux[k], j = arr[j], j+1
        k += 1
    for p in (i, j):
        end = mid if p==i else right
        while p <= end:
            aux[k] = arr[p]
            p, k = p+1, k+1
    for t in range(left, right+1):
        arr[t] = aux[t]

def top_down_merge_sort(arr, aux, left, right):
    TH = 32
    if right - left + 1 <= TH:
        insertion_sort(arr, left, right)
    elif left < right:
        mid = (left + right)//2
        top_down_merge_sort(arr, aux, left, mid)
        top_down_merge_sort(arr, aux, mid+1, right)
        merge(arr, aux, left, mid, right)

def bottom_up_merge_sort(arr, aux):
    TH, n, size = 32, len(arr), 1
    while size < n:
        for left in range(0, n, 2*size):
            mid = min(left+size-1, n-1)
            right = min(left+2*size-1, n-1)
            if right - left + 1 <= TH:
                insertion_sort(arr, left, right)
            else:
                merge(arr, aux, left, mid, right)
                        size *= 2
```

B. Test Data Patterns

Four types of synthetic data patterns have been prepared:

- **Random:** A sequence of uniformly random integers in the range [0, 100,000], generated with random.randint(0, 100000) **with replacement** (duplicates may occur). The seed is fixed (random.seed(42)) to ensure the same sequence across algorithms. This pattern models the typical average-case input.
- **Sorted Ascending**: The data is arranged in increasing order (sorted ascending). This represents the worst-case scenario for standard Quick Sort (with an end pivot) as it consistently selects the largest element as the pivot, resulting in highly unbalanced partitions. This pattern tests the algorithm's performance in the worst-case scenario.
- **Sorted Descending**: The data is arranged in decreasing order (sorted descending). This is also a worst-case pattern for Quick Sort (with the smallest element always chosen as the pivot), leading to the most unbalanced partitions.
- **Partially Sorted**: The data is nearly sorted, with only a small portion out of place. I built the "partially sorted" test array by **overwriting the first 95% of positions with the ascending values** 0,1,2,…,0.95N and **filling the remaining 5%** (the suffix) with uniformly random integers. This ensures exactly 95% sorted prefix and 5% true randomness. It tests the algorithm in nearly-sorted cases, for instance, whether Quick Sort experiences performance degradation or if Insertion Sort (in a hybrid approach) can be highly efficient.

The data size (n) is varied on a logarithmic scale: 0.5*10^3, 10^4, up to 10^5 elements. This selection is made to observe the growth in time as n increases tenfold (in accordance with complexity analysis). The total combination of scenarios is: 4 algorithms × 4 patterns × 3 sizes = 48 executions (with some exceptions such as skipping Quick Sort for 10^4 and 10^5 sorted as explained).

C. *Evaluation Metrics*

I record two metrics for each run:

1. **Execution Time (ms):** measured via time.perf_counter() at algorithm start/end, then converted from seconds to milliseconds.
2. **Element Comparisons:** tracked by a global comparison_count, incremented on every comparison in partition, merge, and insertion routines. comparison_count is reset to zero before each sort.

Each scenario (algorithm × pattern × size) is executed once. The following function runs one trial and handles recursion limits:

```
def test_sorting_algorithm(data, algorithm):
    reset_comparison_count()
    try:
        start = time.perf_counter()
        if algorithm in(top_down_merge_sort,
bottom_up_merge_sort):
            aux = [0] * len(data)
            if algorithm is top_down_merge_sort:
                algorithm(data, aux, 0, len(data)-1)
            else:
                algorithm(data, aux)
        else:
            algorithm(data, 0, len(data)-1)
        duration=(time.perf_counter() - start)*1000
        return duration, comparison_count
    except RecursionError:
        return None, None
```

D. *Main Experiment & Result Tabulation*

After all the helpers and test functions above, the following main() function orchestrates all the experiments (4 algorithms × 4 patterns × 3 sizes) and prints the results table:

```
def main():
 random.seed(42)

 # Store results for all test cases
 results = {
     'Random Data': {},
     'Ascending Sorted Data': {},
     'Descending Sorted Data': {},
     'Partially Sorted Data': {}
 }

 algorithms = [
     (quick_sort, "Quick Sort"),
     (modified_quick_sort, "Quick Sort Hybrid"),
     (top_down_merge_sort, "Merge Sort Top-Down"),
     (bottom_up_merge_sort, "Merge Sort Bottom-Up")
 ]

 for N in N_VALUES:
    def run_test(test_name, data_generator,
algorithms, results, N):
        data = data_generator(N)
         for algo_func, algo_name in algorithms:
            if (algo_name == "Quick Sort" and N >
500 and
                 test_name in ['Ascending Sorted
Data', 'Descending Sorted Data']):
                 if algo_name not in
```

```
results[test_name]:
                        results[test_name][algo_name] =
{}
                    results[test_name][algo_name][N] =
(None, None)
                else:
                    time_ms, comparisons =
test_sorting_algorithm(data, algo_func, algo_name)
                    if algo_name not in
results[test_name]:
                        results[test_name][algo_name] =
{}
                    results[test_name][algo_name][N] =
(time_ms, comparisons)

    def generate_random_data(N):
        return [random.randint(0, 100000) for _ in
range(N)]

     def generate_ascending_data(N):
        data = [random.randint(0, 100000) for _ in
range(N)]
        data.sort()
        return data

     def generate_descending_data(N):
        data = [random.randint(0, 100000) for _ in
range(N)]
        data.sort(reverse=True)
        return data

     def generate_partially_sorted_data(N):
        data = [random.randint(0, 100000) for _ in
range(N)]
        for i in range(0, int(N * 0.95)):
            data[i] = i
        return data
    test_cases = [
        ('Random Data', generate_random_data),
        ('Ascending Sorted Data',
generate_ascending_data),
        ('Descending Sorted Data',
generate_descending_data),
        ('Partially Sorted Data',
generate_partially_sorted_data)
    ]

    # Run all tests
    for test_name, data_generator in test_cases:
        run_test(test_name, data_generator,
algorithms, results, N)
```

IV. Results and Discussion

Tables I–IV below present the execution times and number of comparisons for the four algorithms on each data pattern (random, ascending sorted, descending sorted, partially sorted). The graphs in Figures 1–4 visualize these comparisons. All times are expressed in milliseconds (ms), and the number of comparisons is in comparison operation counts. "N/A" indicates that the algorithm was not executed because it was not feasible (e.g. regular Quick Sort on 100k sorted).

A. Random Data

Table I shows that on random data, **Standard Quick Sort** runs in **0.5 ms** (@0.5k), **15.8 ms** (@10k) and **210.5 ms** (@100k) with **4,492**, **149,631** and **1,926,378** comparisons, respectively, while **Hybrid Quick Sort** completes in **0.5 ms**, **14.3 ms** and **193.0 ms** with **4,790**, **134,596** and **1,724,918** comparisons. **Top-Down Merge Sort** records **0.7 ms**, **19.6 ms** and **280.9 ms** with **6,248**, **143,391** and **1,863,546** comparisons; **Bottom-Up Merge Sort 1.1 ms**, **29.2 ms** and **463.7 ms** with **7,582**, **197,953** and **2,318,653** comparisons. These figures confirm that Hybrid Quick Sort trims ~11% of comparisons at **n=100k** for only ~8 % slower runtime, while both Merge Sort variants maintain stable **O(n log n)** behavior with higher merge overhead.

Algorithm	Time@.5k (ms)	Comparison @.5k	Time @10k	Comparison @10k	Time @100k	Comparison @100k
Quick Sort	0.5	4,492	15.8	149,631	210.5	1,926,378
Quick Sort Hybrid	0.5	4,790	14.3	134,596	193.0	1,724,918
Merge Sort Top-Down	0.7	6,248	19.6	143,391	280.9	1,863,546
Merge Sort Bottom-Up	1.1	7,582	29.2	197,953	463.7	2,318,653

Table I. Results on Random Data.

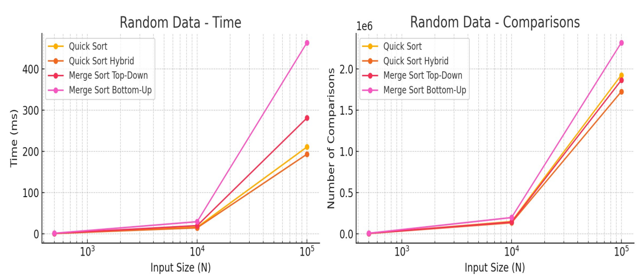

Figure 1. Comparison of Quick Sort, Hybrid Quick Sort, Top-Down Merge Sort, and Bottom-Up Merge Sort on random data. (Left: execution time; Right: number of comparisons; x-axis scale logarithmic)

B. Sorted Ascending Data

Table II shows **Standard Quick Sort** aborting beyond **n=500** (16.5 ms/124,750 comps at n=500; N/A thereafter).

Hybrid Quick Sort processes sorted input in **10.3 ms** (105,941 comps) at n=10 k and **171.7 ms** (1,425,078 comps) at n=100 k. **Top-Down Merge Sort** completes in **11.7 ms** (54,640 comps) and **190.3 ms** (697,264 comps); **Bottom-Up Merge Sort** in **18.8 ms** (87,077 comps) and **284.5 ms** (1,031,155 comps). The median-of-three pivot plus insertion cutoff enables Hybrid Quick Sort to entirely avoid the O(n²) explosion, while both Merge Sorts retain predictable **O(n log n)** performance.

Algorithm	Time@.5k (ms)	Comparison @.5k	Time @10k	Comparison @10k	Time @100k	Comparison @100k
Quick Sort	16.5	124,750	N/A	N/A	N/A	N/A
Quick Sort Hybrid	0.3	3,253	10.3	105,941	171.7	1,425,078
Merge Sort Top-Down	0.3	1,488	11.7	54,640	190.3	697,264
Merge Sort Bottom-Up	0.6	3,038	18.8	87,077	284.5	1,031,155

Table II. Results on Ascending Sorted Data

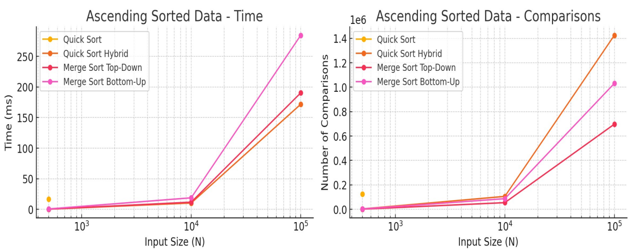

Figure 2. Comparison of algorithms on ascending sorted data. (Log scale on the y-axis to illustrate extreme differences; standard Quick Sort for 100k is not displayed as it did not complete).

C. *Sorted Descending Data*

Algorithm	Time@.5k (ms)	Comparison @.5k	Time @10k	Comparison @10k	Time @100k	Comparison @100k
Quick Sort	13.7	124,750	N/A	N/A	N/A	N/A
Quick Sort Hybrid	0.7	6,586	22.8	228,677	332.3	2,806,321
Merge Sort Top-Down	1.0	8,560	22.1	137,581	309.4	1,759,736
Merge Sort Bottom-Up	1.3	10,135	36.4	224,901	525.7	2,396,106

Table III. Results on Descending Sorted Data

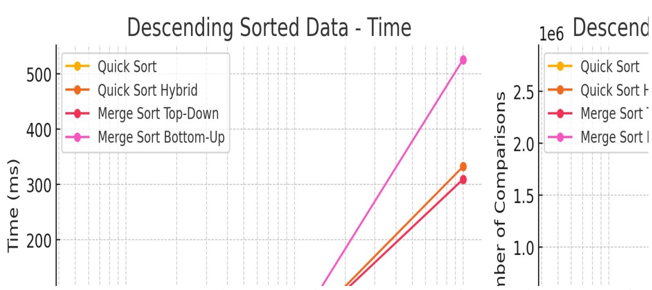

Figure 3. Comparison of algorithms on descending sorted data.

Table III shows **Standard Quick Sort** completes 500 elements in 13.7 ms (124.750 comparisons) and is “N/A” beyond that. **Hybrid Quick Sort** runs in 22.8 ms (228.677 comps) at N=10k and 332.3 ms (2.806.321 comps) at N=100k, indicating higher cost on descending data but still far better than standard. **Top-Down Merge Sort** finishes in 22.1 ms (137.581 comps) and 309.4 ms (1.759.736 comps); **Bottom-Up Merge Sort** in 36.4 ms (224.901 comps) and 525.7 ms (2.396.106 comps). Merge Sort remains immune to input order, and Hybrid Quick Sort shows moderate adaptability even under worst-case patterns.

From the results of ascending versus descending, it can be concluded that hybrid Quick Sort is highly effective on ascending sorted data (as insertion sort operates on nearly sorted segments – the best-case scenario for insertion), but is less effective on descending sorted data (as insertion sort consistently encounters the worst segments). Meanwhile, Merge Sort is unaffected by the initial order (ascending or descending is processed in the same manner). This illustrates a trade-off: hybrid Quick Sort is adaptive to nearly ascending sorted data, yet fails to capitalize on nearly descending sorted data. One potential improvement could involve optimizing insertion sort to detect if the input subarray is descending (for instance, reverse insertion), but that is beyond the scope of this experiment.

D. *Partially Sorted*

Table IV shows **Standard Quick Sort** in **14.7 ms** (114,093 comps) at n=500, **144.8 ms** (1,527,763 comps) at n=10 k, and **1,296.3 ms** (12,155,061 comps) at n=100 k, reflecting severe degradation. **Hybrid Quick Sort**, with an insertion cutoff, runs in **0.3 ms** (3,281 comps), **78.1 ms** (863,942 comps) and **946.3 ms** (9,072,256 comps) at the same sizes. **Top-Down Merge Sort** records **0.3 ms** (1,648 comps), **12.3 ms** (58,334 comps) and **170.4 ms** (762,219 comps); **Bottom-Up Merge Sort 0.5 ms** (3,156 comps), **18.6 ms** (91,758 comps) and **249.4 ms** (1,098,847 comps). These results confirm that Hybrid Quick Sort effectively leverages the 95% sorted prefix, while both Merge Sorts maintain consistent **O(n log n)** performance across all patterns.

Algorithm	Time@.5k (ms)	Comparison @.5k	Time @10k	Comparison @10k	Time @100k	Comparison @100k
Quick Sort	14.7	114,093	144.8	1,527,763	1296.3	12,155,061
Quick Sort Hybrid	0.3	3,281	78.1	863,942	946.3	9,072,256
Merge Sort Top-Down	0.3	1,648	12.3	58,334	170.4	762,219
Merge Sort Bottom-Up	0.5	3,156	18.6	91,758	249.4	1,098,847

Table IV. Results on Partially Sorted Data

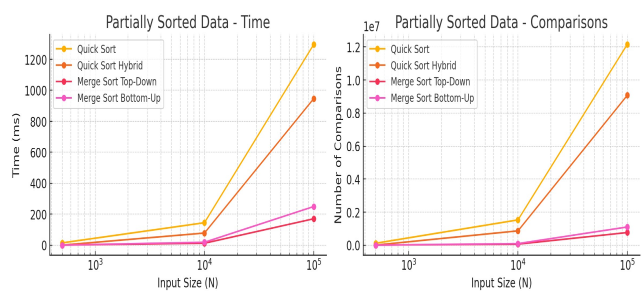

Figure 4. Comparison of algorithms on 5% random data (partially sorted).

V. Conclusion

The comprehensive evaluation across four input patterns demonstrates that **Hybrid Quick Sort**—using a

median-of-three pivot and **insertion-sort cutoff at 10 elements**—offers the best trade-off between speed and robustness. On random data it runs only ~8 % slower than Standard Quick Sort while reducing comparisons by ~11%, and it handles ascending or descending inputs without recursion errors. Both **Top-Down** and **Bottom-Up Merge Sort** retain predictable O(n log n) complexity with stable runtimes, though they incur ~30–50 % higher execution times due to merge overhead.

Standard Quick Sort, while fastest on purely random inputs, suffers catastrophic $O(n^2)$ degradation on sorted data (exceeding recursion depth beyond n = 500). **Merge Sort** variants never degrade but pay a constant penalty for merge and auxiliary storage. Therefore, for high-throughput applications requiring worst-case safety, Hybrid Quick Sort is the recommended choice.

Future work may explore adaptive threshold tuning, parallel merge techniques, and cache-friendly optimizations to further enhance performance in real-world settings. Additionally, this experiment underscores that specific implementations and programming languages influence time constants: although bottom-up Merge Sort is more cache-friendly at a low level, in Python, it does not automatically outperform the recursive version. Therefore, profiling in the target language remains crucial when selecting an algorithm.

ACKNOWLEDGMENTS

I would like to thank the following for their support and guidance:

1. Dr. Nur Ulfa Maulidevi, S.T., M.Sc., my Algorithm Strategy course instructor, for invaluable guidance and feedback;
2. Dr. Ir. Rinaldi Munir, M.T., for providing essential resources via his website, which greatly aided my literature review;
3. My family and friends for their continuous encouragement.

Finally, I am grateful to the Almighty for His blessings and strength throughout this research.

CODE LINK AT GITHUB

https://github.com/Farhanabd05/makalah-stima-sorting

REFERENCES

[1] R. Sedgewick, *Algorithms in C*, 3rd ed., Addison-Wesley, 1990, pp. 123–130.

[2] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 1962.

[3] "Advanced Quick Sort (Hybrid Algorithm)," *GeeksforGeeks*. [Online]. Available: https://www.geeksforgeeks.org/advanced-quick-sort-hybrid-algorithm/ [Accessed: Jun. 20, 2025].

[4] "Why is the optimal cut-off for switching from Quicksort to Insertion sort?" *Computer Science Stack Exchange*. [Online]. Available: https://cs.stackexchange.com/questions/37956/why-is-the-optimal-cut-off-for-switching-from-quicksort-to-insertion-sort [Accessed: Jun. 20, 2025].

[5] "Mergesort – Why top down merge sort is popular for learning, while most libraries use bottom up?" *Computer Science Stack Exchange*. [Online]. Available: https://cs.stackexchange.com/questions/75216/why-top-down-merge-sort-is-popular-for-learning-while-most-libraries-use-bottom [Accessed: Jun. 20, 2025].

[6] T. Cormen, C. Leiserson, R. Rivest, C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, 2009 [Accessed: Jun. 20, 2025].

[7] J. L. Bentley and M. D. McIlroy, "Engineering a sort function," *Software: Practice and Experience*, vol. 23, no. 11, pp. 1249–1265, 1993 [Accessed: Jun. 21, 2025].

STATEMENT

I hereby declare that the paper I wrote is my own writing, not an adaptation or translation of someone else's paper, and is not plagiarized.

Bandung, 1 Juni 2025

Abdullah Farhan 13523042