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

Report

Laboratory Work No. 2

Topic: Study and Empirical Analysis of Sorting Algorithms

Elaborated by:
st. gr. FAF-241
Aurelian-Mihai Tihon

Verified by:
univ. assist.
Fiștic Cristofor

Chișinău – 2026

Contents

1. INTRODUCTION

1.1. Objective

The objective of this laboratory work is to study, implement, and empirically analyze four distinct sorting algorithms: QuickSort, MergeSort, HeapSort, and a highly specialized fourth algorithm, Flash Sort. The analysis explores both standard (unoptimized) and optimized variants of each algorithm, focusing on time complexity, space complexity, comparisons, and array accesses.

Furthermore, an interactive Sorting Algorithm Visualizer was developed using Python and Pygame, featuring real-time graphical representation, audio feedback mapped to array values, and custom data distribution inputs. The full source code for this laboratory can be found on GitHub at: `https://github.com/aurelian-th/AA/commit/1a814af54b6510a04dd01985fa94e64826569336`.

1.2. Properties of input data

To ensure a robust empirical analysis, it is crucial to understand how data properties affect sorting performance. For comparison-based algorithms like QuickSort, MergeSort, and HeapSort, the data type makes zero difference; they will sort floats identically to integers as long as the relational operators ($<$, $>$) are defined. However, distribution-based algorithms like Flash Sort require mathematical mapping. While Flash Sort works with floats, the classification step must be adjusted to map floating-point ranges to discrete integer buckets correctly.

The absolute magnitude of the data is largely irrelevant for comparison sorts. Sorting an array of values from 1 to 100 takes the exact same number of operations as sorting an array of values spanning $1,000,000$ to $3,000,000$, assuming the array length (N) is the same. The time complexity is defined by the number of elements, not their numeric size. However, for Flash Sort, severe variance (ex: 99 elements between $1-10$, and one element at $1,000,000$) can cause bucket clustering, degrading its $O(N)$ performance toward $O(N^2)$.

The initial permutation of the array drastically changes performance. When an array is already sorted, it represents the best case for Insertion Sort and our Optimized QuickSort, but it triggers the worst-case $O(N^2)$ scenario for Basic QuickSort if the last element is strictly chosen as the pivot. Backward (reverse) sorted data similarly triggers maximum swaps for basic implementations. Conversely, if an array consists of identical elements (a flat line), basic algorithms partition highly unevenly, whereas optimized variations handle this gracefully in $O(N)$ time. As the input size doubles, the execution time for good algorithms scales by slightly more than double ($N \log N$), whereas $O(N^2)$ algorithms will take four times as long.

1.3. Chosen metrics

The algorithms were evaluated based on Execution Time (total wall-clock seconds), Comparisons (number of times two elements are evaluated against each other), Array Accesses (frequency of reading from or writing to the main array), and Auxiliary Memory Space.

2. EMPIRICAL ANALYSIS OF ALGORITHMS

2.1. Algorithm 1: QuickSort

QuickSort uses a Divide and Conquer approach. It selects a pivot element, partitions the array so that smaller elements are to the left and larger elements are to the right, and then recursively sorts the sub-arrays.

The Basic QuickSort implementation always selects the last element as the pivot. The massive drawback to this is that if the array is already sorted or reverse sorted, the partition is maximally unbalanced, leading to $O(N^2)$ time complexity and risking Stack Overflow (Recursion) errors.

The Optimized QuickSort avoids this by using a Median-of-Three pivot selection (taking the median of the first, middle, and last elements), guaranteeing a balanced partition even on sorted data. Additionally, it features an Insertion Sort threshold, switching to iterative Insertion Sort for sub-arrays smaller than 10 elements, as recursive overhead is slower than simple swaps for very small sizes.

```
def quicksort_optimized(self, start, end):
    count = end - start + 1
    if count < 10:
        yield from self.insertion_sort(start, end)
        return

    mid = (start + end) // 2
    if self.array[start] > self.array[mid]: self.array[start], self.array[mid] = self.array[mid], self.array[start]
    if self.array[start] > self.array[end]: self.array[start], self.array[end] = self.array[end], self.array[start]
    if self.array[mid] > self.array[end]: self.array[mid], self.array[end] = self.array[end], self.array[mid]

    self.array[mid], self.array[end-1] = self.array[end-1], self.array[mid]
    pivot = self.array[end-1]
    left = start + 1
    right = end - 2

    while True:
        while self.array[left] < pivot: left += 1
        while self.array[right] > pivot: right -= 1
        if left >= right: break
        self.array[left], self.array[right] = self.array[right], self.array[left]
        left += 1
        right -= 1

    self.array[left], self.array[end-1] = self.array[end-1], self.array[left]
    yield from self.quicksort_optimized(start, left - 1)
    yield from self.quicksort_optimized(left + 1, end)
```

Listing 1: QuickSort (Optimized)

2.2. Algorithm 2: MergeSort

MergeSort guarantees an $O(N \log N)$ time complexity across all cases by continually dividing the array in half until sizes of 1 are reached, and then merging them back together in sorted order.

Basic MergeSort operates Top-Down using recursion to divide the array. This requires $O(\log N)$ stack space for recursive calls on top of the $O(N)$ space required for the temporary array during merging. Optimized MergeSort operates Bottom-Up iteratively. It removes recursion entirely by iteratively merging sub-arrays of size 1, then size 2, 4, 8, etc., using loops. This prevents stack overflow on massive datasets and generally offers better CPU cache locality.

```
def mergesort_optimized(self, start, end):
    width = 1
    n = len(self.array)
    while width < n:
        l = 0
        while l < n:
            mid = l + width - 1
            r = min(l + 2 * width - 1, n - 1)
            if mid < r:
                yield from self.merge(l, mid, r)
            l += 2 * width
        width *= 2
```

Listing 2: MergeSort (Bottom-Up Optimized)

2.3. Algorithm 3: HeapSort

HeapSort visualizes the array as a nearly complete binary tree. It builds a Max-Heap (where parent nodes are strictly greater than children) and then iteratively extracts the root (the maximum value) and places it at the end of the array, shrinking the heap size by one each time.

During the standard "sift-down" (heapify) process, Basic HeapSort performs two comparisons per level: one to find the largest child, and one to check if the child is larger than the parent. Optimized HeapSort uses Floyd's Leaf Search. It assumes that the element being sifted down usually sinks to the very bottom of the tree. Therefore, it skips the comparison against the parent during the downward traversal (saving approximately 50% of comparisons). Once it hits a leaf, it places the element and then does a short "sift-up" if needed.

```
def heapify_floyd(self, n, i):
    root = i
    child = 2 * root + 1
    temp = self.array[root]

    # Sift down without comparing to temp (saves comparisons)
    while child < n:
        if child + 1 < n and self.array[child + 1] > self.array[child]:
            child += 1
        self.array[root] = self.array[child]
        root = child
        child = 2 * root + 1

    self.array[root] = temp

    # Sift up to final correct position
    while root > i:
        parent = (root - 1) // 2
        if self.array[parent] < self.array[root]:
            self.array[parent], self.array[root] = self.array[root], self.array[
parent]
            root = parent
        else: break
```

Listing 3: HeapSort (Optimized with Floyd's Method)

2.4. Algorithm 4: Flash Sort

Flash Sort is a highly sophisticated, distribution-based algorithm. Unlike the others which are mathematically bounded by the $\Omega(N \log N)$ comparison limit, Flash Sort operates in $O(N)$ linear time for uniformly distributed data.

It works by finding the minimum and maximum values, then mathematically dividing the range into M buckets (typically $M = 0.43N$). It counts how many elements fall into each bucket. Unlike standard Bucket Sort, it calculates exactly where the boundaries of each bucket are in the main array, and "teleports"

elements to their correct mathematical bucket via cyclical in-place swapping ($O(1)$ extra memory). Finally, an Insertion Sort perfectly orders the minor local inversions left over within the buckets. The optimized version adds pre-flight checks to exit instantly if the array is already flat, and intelligently prevents unnecessary swaps.

```
def flashsort_optimized(self, start, end):
    if end - start <= 20:
        yield from self.insertion_sort(start, end)
        return
    # Find min and max
    min_val = self.array[start]
    max_val = self.array[start]
    for i in range(start, end + 1):
        if self.array[i] < min_val: min_val = self.array[i]
        if self.array[i] > max_val: max_val = self.array[i]
    if min_val == max_val: return

    # Classify into buckets
    n = end - start + 1
    m = int(0.43 * n)
    c = (m - 1) / (max_val - min_val) if max_val != min_val else 0
    l = [0] * m
    for i in range(start, end + 1):
        k = int(c * (self.array[i] - min_val))
        l[k] += 1
    for k in range(1, m): l[k] += l[k-1]

    # Permutation (Cycle Leader)
    move_count = 0
    idx = start
    k = m - 1
    flash = self.array[start]
    while move_count < n - 1:
        while idx > (start + l[k] - 1):
            idx += 1
            k = int(c * (self.array[idx] - min_val))
            flash = self.array[idx]
        k = int(c * (flash - min_val))
        target_idx = start + l[k] - 1
        l[k] -= 1
        # Swap
        temp = self.array[target_idx]
        self.array[target_idx] = flash
        flash = temp
        move_count += 1

    yield from self.insertion_sort(start, end)
```

Listing 4: Flash Sort (Optimized Core Logic)

3. THEORETICAL COMPLEXITY COMPARISONS

The table below summarizes the theoretical Time and Space complexities for the eight variations of algorithms studied, taking into account Worst-Case and Best-Case scenarios.

Table 1: Complexity Analysis of Sorting Algorithms

Algorithm	**Best Time**	**Average Time**	**Worst Time**	**Space/Aux Memory**
QuickSort (Basic)	$O(N \log N)$	$O(N \log N)$	$O(N^2)$	$O(\log N)$ (Stack)
QuickSort (Optimized)	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(\log N)$ (Stack)
MergeSort (Basic)	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N)$
MergeSort (Optimized)	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N)$
HeapSort (Basic)	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(1)$
HeapSort (Optimized)	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(1)$
Flash Sort (Basic)	$O(N)$	$O(N)$	$O(N^2)$ (skewed)	$O(1)$
Flash Sort (Optimized)	$O(N)$	$O(N)$	$O(N^2)$ (skewed)	$O(1)$

4. EMPIRICAL BENCHMARK RESULTS

Using the developed python application running in a headless state, the 8 algorithms were benchmarked across sizes $N = 1000$ to $N = 10000$. The following graphs and tables display execution times across 5 distinct data distributions. It is vital to note that in both the Sorted and Reverse distributions, the Basic QuickSort experienced severe RecursionError/StackOverflow due to deep unoptimized recursion trees, proving its worst-case $O(N^2)$ theoretical bounds.

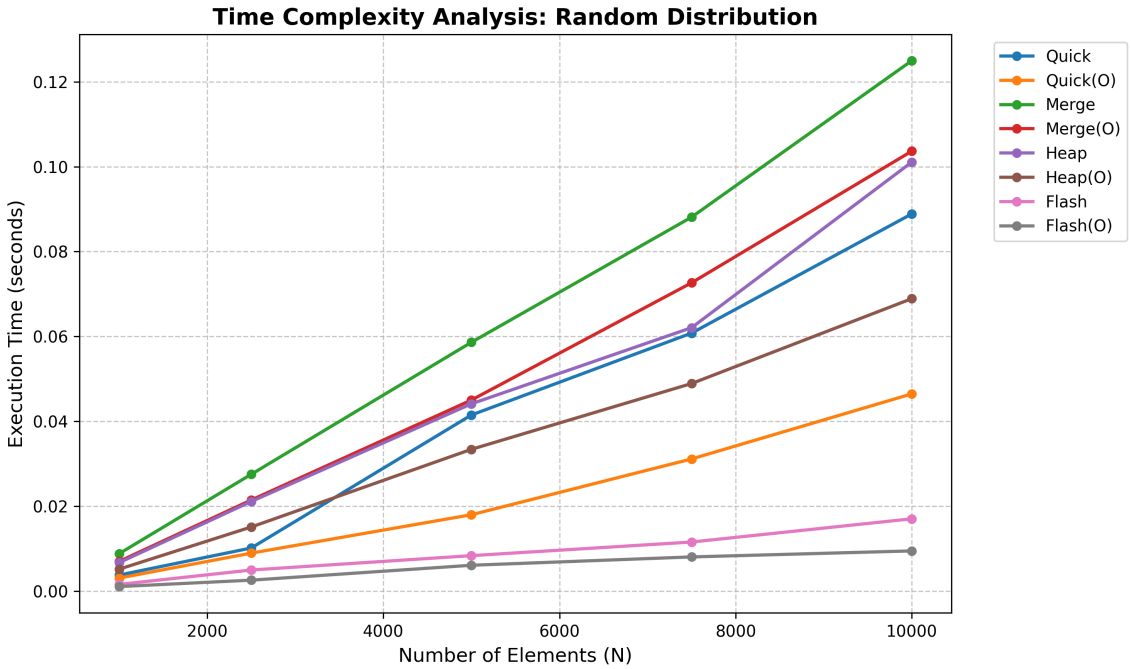

Figure 1: Algorithm performance on randomly distributed data

[Distribution: RANDOM]

Size	Quick	Quick(O)	Merge	Merge(O)	Heap	Heap(O)	Flash	Flash(O)
1000	0.0038	0.0031	0.0088	0.0070	0.0067	0.0052	0.0016	0.0011
2500	0.0102	0.0090	0.0275	0.0215	0.0211	0.0151	0.0050	0.0026
5000	0.0415	0.0180	0.0586	0.0450	0.0441	0.0334	0.0084	0.0061
7500	0.0608	0.0311	0.0881	0.0726	0.0621	0.0489	0.0116	0.0081
10000	0.0889	0.0465	0.1250	0.1036	0.1011	0.0689	0.0171	0.0095

Figure 2: Raw execution times: random distribution

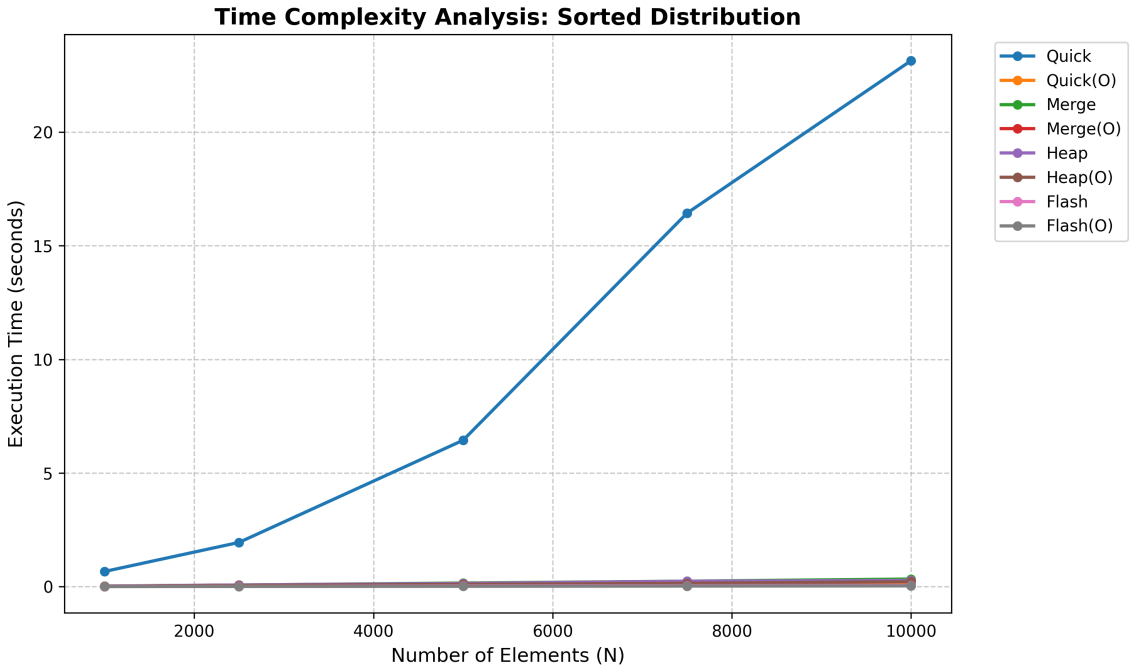

Figure 3: Algorithm performance on already sorted data

[Distribution: SORTED]

Size	Quick	Quick(O)	Merge	Merge(O)	Heap	Heap(O)	Flash	Flash(O)
1000	0.6675	0.0123	0.0200	0.0180	0.0261	0.0177	0.0056	0.0029
2500	1.9458	0.0264	0.0702	0.0657	0.0727	0.0514	0.0143	0.0087
5000	6.4387	0.0515	0.1590	0.1170	0.1408	0.0992	0.0300	0.0137
7500	16.4316	0.0995	0.2403	0.2027	0.2385	0.1462	0.0428	0.0244
10000	23.1384	0.1339	0.3346	0.2426	0.2731	0.2179	0.0549	0.0325

Figure 4: Raw execution times: sorted distribution

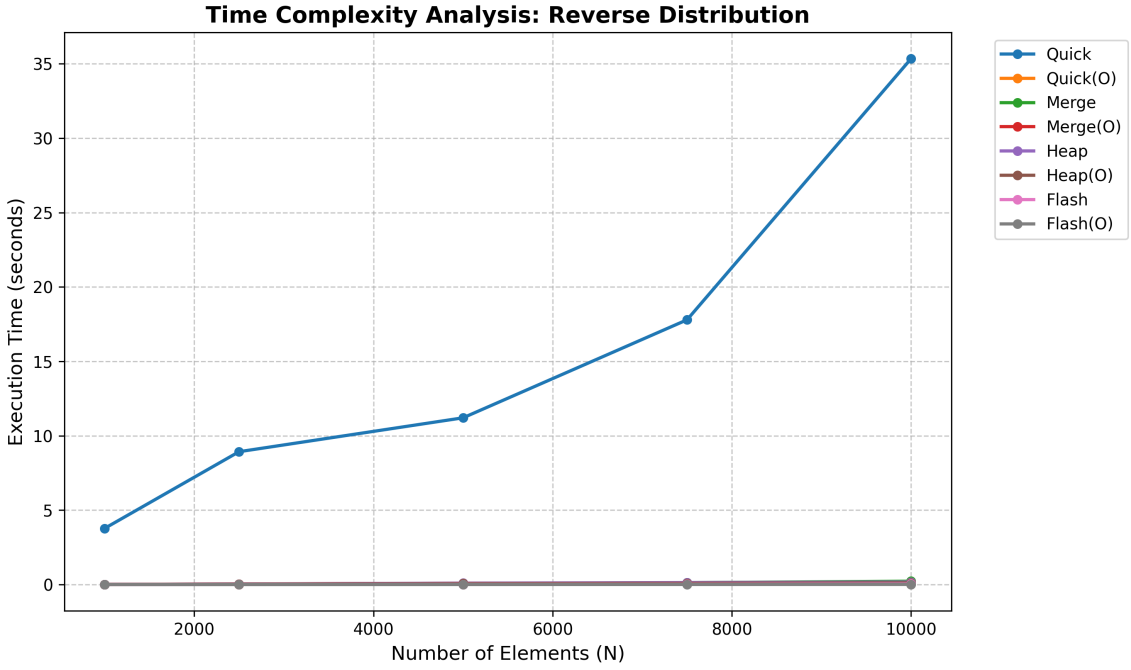

Figure 5: Algorithm performance on reverse sorted data

[Distribution: REVERSE]

Size	Quick	Quick(O)	Merge	Merge(O)	Heap	Heap(O)	Flash	Flash(O)
1000	3.7777	0.0060	0.0094	0.0139	0.0123	0.0138	0.0030	0.0016
2500	8.9313	0.0150	0.0273	0.0369	0.0421	0.0391	0.0074	0.0034
5000	11.2081	0.0280	0.0602	0.0862	0.0951	0.0706	0.0148	0.0061
7500	17.7952	0.0388	0.1284	0.0980	0.1354	0.0901	0.0245	0.0096
10000	35.3436	0.1068	0.2271	0.1233	0.1708	0.1192	0.0441	0.0127

Figure 6: Raw execution times: reverse distribution

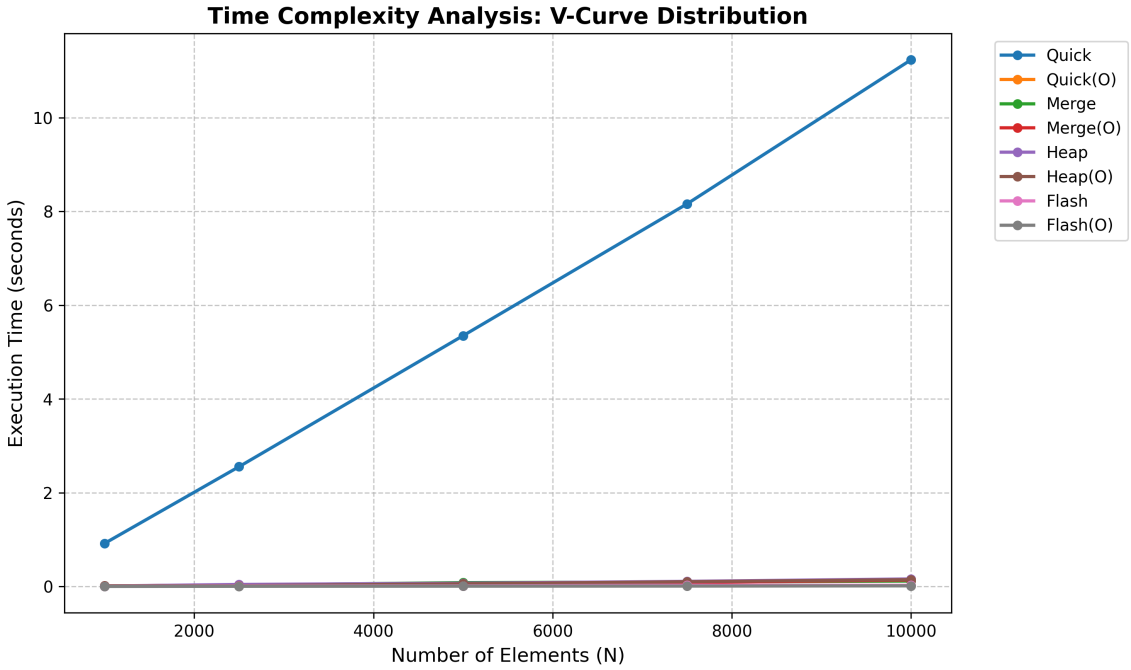

Figure 7: Algorithm performance on V-curve distributed data

[Distribution: V-CURVE]

Size	Quick	Quick(O)	Merge	Merge(O)	Heap	Heap(O)	Flash	Flash(O)
1000	0.9196	0.0108	0.0096	0.0085	0.0093	0.0149	0.0035	0.0012
2500	2.5549	0.0291	0.0292	0.0349	0.0433	0.0163	0.0116	0.0037
5000	5.3483	0.0587	0.0824	0.0502	0.0694	0.0564	0.0129	0.0065
7500	8.1600	0.0998	0.0957	0.0856	0.1093	0.1010	0.0261	0.0087
10000	11.2329	0.1529	0.1156	0.1371	0.1607	0.1460	0.0243	0.0138

Figure 8: Raw execution times: V-curve distribution

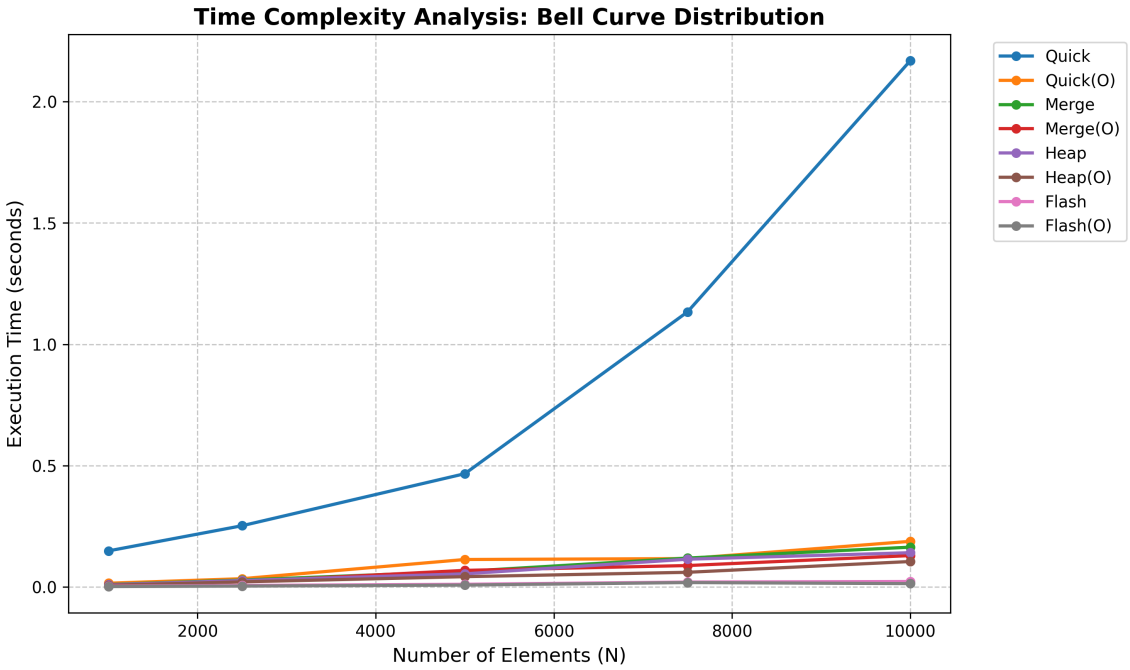

Figure 9: Algorithm performance on Bell curve distributed data

[Distribution: BELL CURVE]

Size	Quick	Quick(O)	Merge	Merge(O)	Heap	Heap(O)	Flash	Flash(O)
1000	0.1483	0.0151	0.0100	0.0088	0.0088	0.0065	0.0026	0.0012
2500	0.2521	0.0336	0.0281	0.0211	0.0261	0.0206	0.0058	0.0033
5000	0.4664	0.1128	0.0640	0.0680	0.0537	0.0420	0.0106	0.0074
7500	1.1336	0.1165	0.1188	0.0878	0.1140	0.0600	0.0198	0.0178
10000	2.1689	0.1877	0.1636	0.1290	0.1408	0.1045	0.0221	0.0134

Figure 10: Raw execution times: Bell curve distribution

5. CONCLUSION

Through theoretical study and empirical visualization, several distinct conclusions can be drawn regarding sorting algorithms and the absolute necessity of optimizations. As proven by the benchmarks, a basic QuickSort acts catastrophically ($O(N^2)$ time complexity) on a pre-sorted array, eventually resulting in a stack overflow. However, by implementing a simple Median-of-Three pivot strategy and falling back to Insertion Sort for small blocks, the Optimized QuickSort transforms back into one of the fastest general-purpose sorts available, demonstrating stable execution times across all data distributions.

Space complexity must also be considered alongside time complexity. While MergeSort guarantees excellent execution time and stability, its requirement for an auxiliary $O(N)$ memory array makes it a poor choice for embedded systems with limited RAM. In those hardware-constrained scenarios, HeapSort is vastly superior as it sorts entirely in-place ($O(1)$ space), and Floyd's leaf-search optimization makes it nearly as fast as QuickSort by halving the required comparisons.

Flash Sort proves that by analyzing the actual statistical distribution of the data rather than blindly comparing individual elements, we can break the $\Omega(N \log N)$ mathematical limit and achieve linear $O(N)$ time. The graphs clearly show Flash Sort dramatically outperforming the others. However, as noted in the theoretical analysis, Flash Sort will degrade severely if the data is heavily skewed, making it highly specialized.

In summary, there is no single "best" algorithm. In cases where the array is tiny (under 20 elements), simple algorithms like Insertion Sort often perform exactly the same or better than complex ones due to low overhead. MergeSort is best for linked lists and stable external sorting. HeapSort is best when strict memory limits apply. Flash Sort is an unparalleled powerhouse for massive datasets of uniformly distributed data. Optimized QuickSort remains the gold standard for general-purpose in-memory sorting, proving that algorithmic tuning is just as important as the underlying theoretical concept.