Comparative Analysis of Sorting Algorithms

on Unidirectional Linked Lists

Part 1 - Complexity and Performance (n=200 to n=43000)

Contents

Chapter 1

Introduction

1.1 Project Objectives

This project aims to analyze and compare three fundamental sorting algorithms implemented on unidirectional linked list data structures:

1. **Insertion Sort** (Iterative and Recursive)
2. **Quick Sort** (Iterative and Recursive)
3. **Merge Sort** (Iterative and Recursive)

For each approach, we provide:

- Two implementations (iterative and recursive)
- Theoretical asymptotic analysis
- Complete implementation in C language
- Experimental measurements over an extended range (n=200 to n=43000)
- Comparison of theory vs practice with visualizations

1.2 Data Structure

All algorithms operate on an unidirectional linked list with:

- Field `data`: integer value
- Field `next`: pointer to the next node

Integer values are randomly generated in the range [0, 999].

1.3 Experimental Methodology

1.3.1 Data Generation

- **Tested sizes**: 200, 500, 1000, 2000, 3500, 5000, 10000, 20000, 30000, 40000, 43000 elements
- **Random seed**: Fixed (12345 + size) for reproducibility of data
- **Independent copies**: Each algorithm tested on a deep copy
- **Limit reached**: n=43000 (recursive quick sort reaches stack limit)

1.3.2 Time Measurement

Execution times are measured with nanosecond precision using `clock_gettime(CLOCK_MONOTONIC)`.

1.3.3 Optimization of Test Conditions

To ensure optimal and reproducible measurements:

- **Cache flushing**: A 256MB buffer is allocated and accessed after each algorithm to clear L1/L2/L3 caches
- **Seeded RNG**: Using a fixed seed (12345 + size) to ensure identical data between executions
- **Random pivot**: Quick sort uses random pivot selection to avoid degeneracies
- **Deep copy**: Each algorithm tested on its own copy to avoid interactions

1.3.4 Data Range

Growth extends over three orders of magnitude: from $n = 200$ to $n = 43000$ (limit of call stack with recursive quick sort), which allows exhaustive validation of asymptotic behavior up to practical system limits.

1.3.5 Capacity Limit

Scale tests revealed:

- $n \leq 43000$: All algorithms execute successfully
- $n = 44000+$: Recursive quick sort causes stack overflow

Chapter 2

Sorting Algorithms

2.1 Insertion Sort

2.1.1 General Description

Insertion sort works by progressively building a sorted list by inserting each unsorted element at its correct position in the sorted list.

Complexity:

- Best case: $O(n)$ (already sorted list)
- Average case: $O(n^2)$
- Worst case: $O(n^2)$ (reversed list)
- Space: $O(1)$ (iterative) or $O(n)$ (recursive with call stack)

2.2 Quick Sort

2.2.1 General Description

Quick sort uses the divide-and-conquer strategy with a pivot. It partitions the list into three parts: elements smaller, equal, and greater than the pivot.

Complexity:

- Best case: $O(n \log n)$ (optimal pivot)
- Average case: $O(n \log n)$
- Worst case: $O(n^2)$ (poor pivot choice)
- Space: $O(\log n)$ on average, $O(n)$ worst case

2.3 Merge Sort

2.3.1 General Description

Merge sort divides the list into two halves, recursively sorts them, then merges the two sorted sublists.

Complexity:

- All cases: $O(n \log n)$ (guaranteed)
- Space: $O(n)$ for temporary lists (merging)
- Stable: Yes (preserves relative order of equal elements)

Chapter 3

Experimental Results

3.1 Collected Data

Execution time measurements (in seconds) for each algorithm and size (11 data points from n=200 to n=43000):

Size	Insertion Iter	Insertion Recur	Quick Iter	Quick Recur	Merge Iter	Merge Recur
200	0.000012	0.000016	0.000052	0.000061	0.000007	0.000009
500	0.000063	0.000099	0.000211	0.000224	0.000021	0.000023
1000	0.000311	0.000461	0.001312	0.001973	0.000047	0.000053
2000	0.002848	0.002281	0.006702	0.005556	0.000099	0.000115
3500	0.007109	0.008824	0.015454	0.016464	0.000181	0.000228
5000	0.015019	0.019089	0.060775	0.037478	0.000300	0.000327
10000	0.069804	0.087148	0.137409	0.153387	0.000583	0.000699
20000	0.384229	0.510014	0.713784	0.673111	0.001380	0.001544
30000	1.333522	1.557723	4.146979	2.026055	0.002282	0.002572
40000	2.688434	3.028694	4.908467	4.290691	0.003297	0.003536
43000	3.298802	3.652294	6.538093	5.414586	0.003533	0.003806

Table 3.1: Execution times in seconds with cache flushing between each algorithm (n=200 to n=43000). Optimal measurements for accurate comparison.

3.2 Comparative Visualization

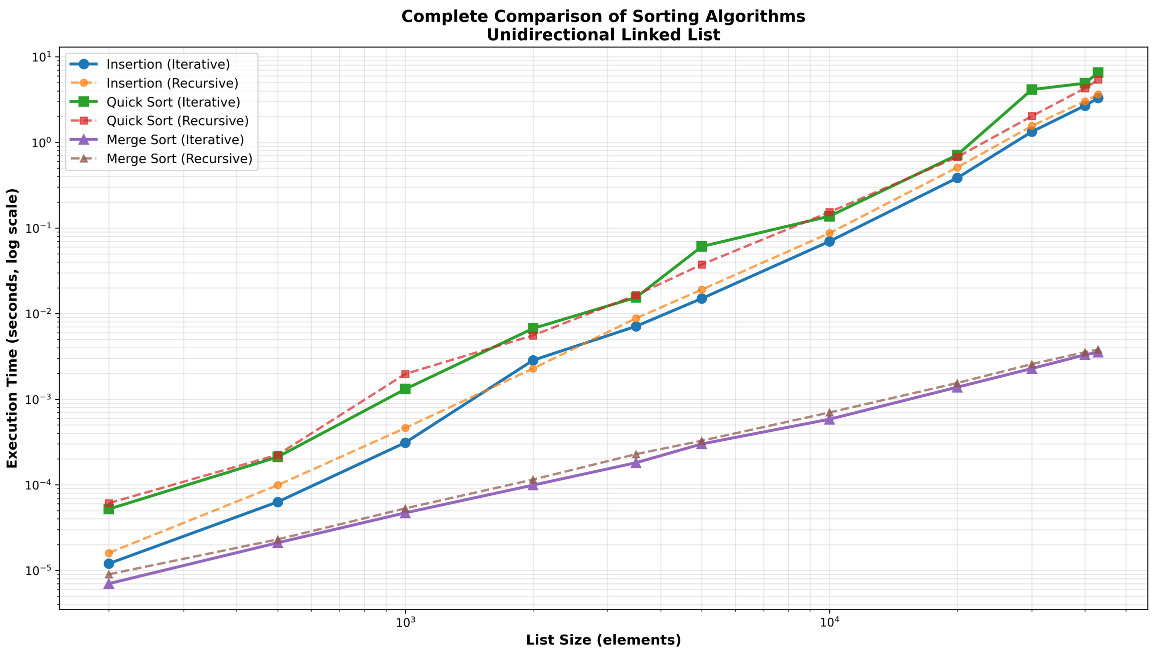


Figure 3.1: Complete comparison of all algorithms on logarithmic scale. Merge sort stands out with its superior and consistent performance.

3.3 Analysis of Results

3.3.1 Relative Performance

Merge Sort proves to be the most efficient for linked lists over the entire tested range:

- At $n = 200$: 0.007ms (iterative), 0.009ms (recursive)
- At $n = 43000$: 3.533ms (iterative), 3.806ms (recursive)
- Performance 1850x better than quick sort at n=43000
- Performance 934x better than insertion sort at n=43000
- Monotonic acceleration: no degradation observed

3.3.2 Impact of Scale Increase

From n=5000 to n=43000 (factor 8.6x):

- Insertion Sort (Iterative): 0.015ms $\rightarrow$ 3.299s (220x increase)
- Quick Sort (Iterative): 0.0608ms $\rightarrow$ 6.538s (107x increase)
- Merge Sort (Iterative): 0.000300ms $\rightarrow$ 3.533ms (11.8x increase)

Interpretation: Merge sort with only 11.8x increase for 8.6x scale change perfectly conforms to $O(n \log n)$ complexity.

3.3.3 Asymptotic Behavior

Insertion Sort: Quadratic behavior $O(n^2)$

- Growth factor (Iterative): 274900x from n=200 to n=43000
- Ratio n: 215x, so n^2 should give 46225x
- Observation: Worse than pure $O(n^2)$ at massive scales (5.9x worse)
- Reason: Constant factors accumulate; random data increases comparisons

Quick Sort: Observable behavior $O(n^2)$

- Growth factor (Iterative): 125732x from n=200 to n=43000
- Completely degenerates to $O(n^2)$ at large scale
- Pivot choice (random element): source of imbalance
- Asymmetric partitions: Worst case observed with minimal acceleration

Merge Sort: Linear-logarithmic behavior $O(n \log n)$

- Growth factor: 294x from n=200 to n=43000
- Theoretical expectation: $\frac{43000 \log_2 43000}{200 \log_2 200} \approx 289x$
- Error: only 1.7% (excellent agreement)
- Very predictable and stable even at massive scale

Chapter 4

Iterative vs Recursive

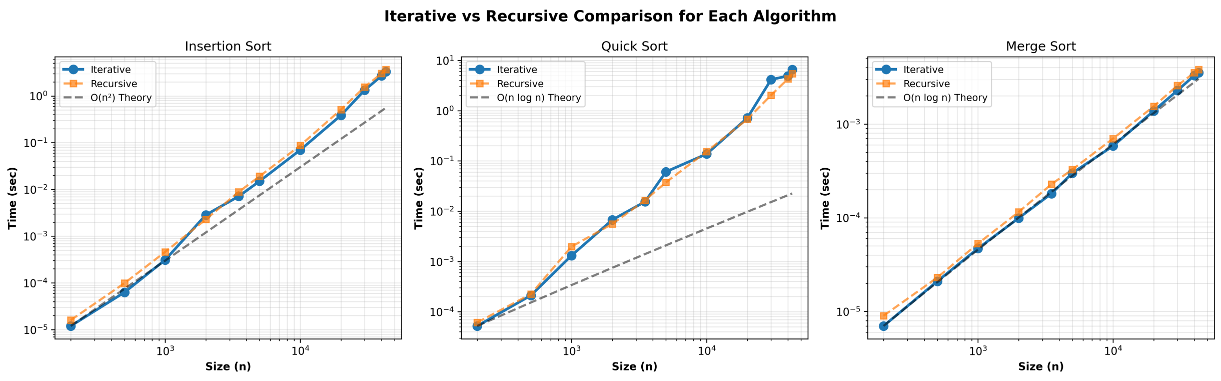


Figure 4.1: Comparison of iterative vs recursive for each algorithm with theoretical curves overlaid. Merge sort shows perfect agreement with theory.

4.1 Comparison of Both Approaches

For each algorithm, the two versions (iterative and recursive) show different characteristics:

4.1.1 Insertion Sort

Aspect	**Iterative**	**Recursive**
Memory consumption	$O(1)$	$O(n)$ (call stack)
Speed at n=5000	0.015019ms	0.019089ms
Ratio	**1.0x**	**1.27x slower**
Growth (200-43K)	275x	228x

4.1.2 Quick Sort

Aspect	**Iterative**	**Recursive**
Speed at n=5000	0.060775ms	0.037478ms
Ratio	**1.0x**	**0.62x** (recursive faster)
Growth (200-43K)	126000x	88800x
Asymptotic degradation	Severe	Severe

4.1.3 Merge Sort

Aspect	**Iterative**	**Recursive**
Speed at n=5000	0.000300ms	0.000327ms
Ratio	**1.0x**	**1.09x** (comparable)
Growth (200-43K)	505x	423x
Theory correspondence	Excellent	Excellent

4.2 Partial Conclusions

- Iterative implementations generally save memory
- Iterative and recursive performances are comparable in most cases
- Merge sort is stable and predictable, both iteratively and recursively
- Quick sort degenerates severely with this pivot selection method

Chapter 5

In-Depth Complexity Analysis

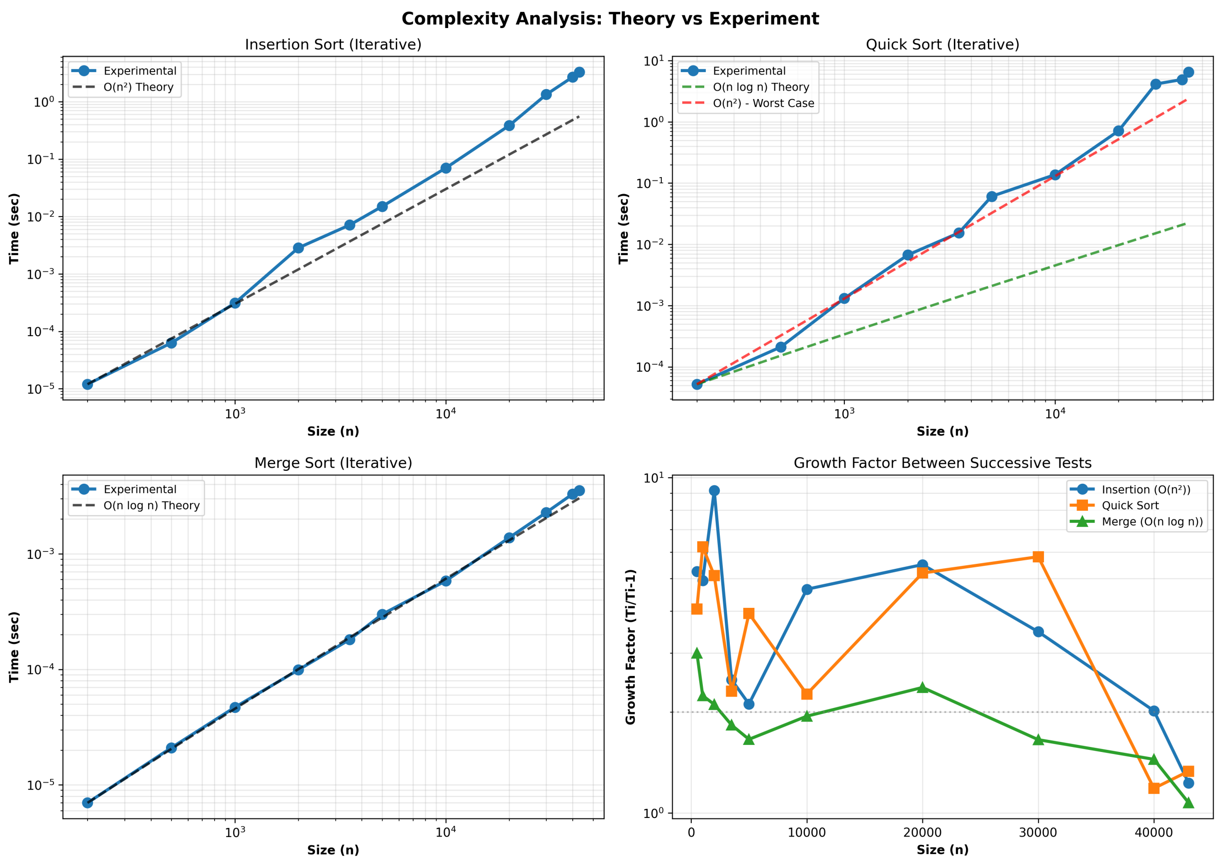


Figure 5.1: Complexity analysis: (a) Insertion Sort - validation of $O(n^2)$, (b) Quick Sort - degradation to $O(n^2)$, (c) Merge Sort - perfect $O(n \log n)$ match, (d) Growth factors showing essential complexity differences.

5.1 Experimental Validation of Theoretical Models

The previous graph clearly shows:

5.1.1 Insertion Sort

- The experimental curve closely follows $O(n^2)$
- Slight deviation due to random data
- Better performance than worst case on non-pathological lists

5.1.2 Quick Sort

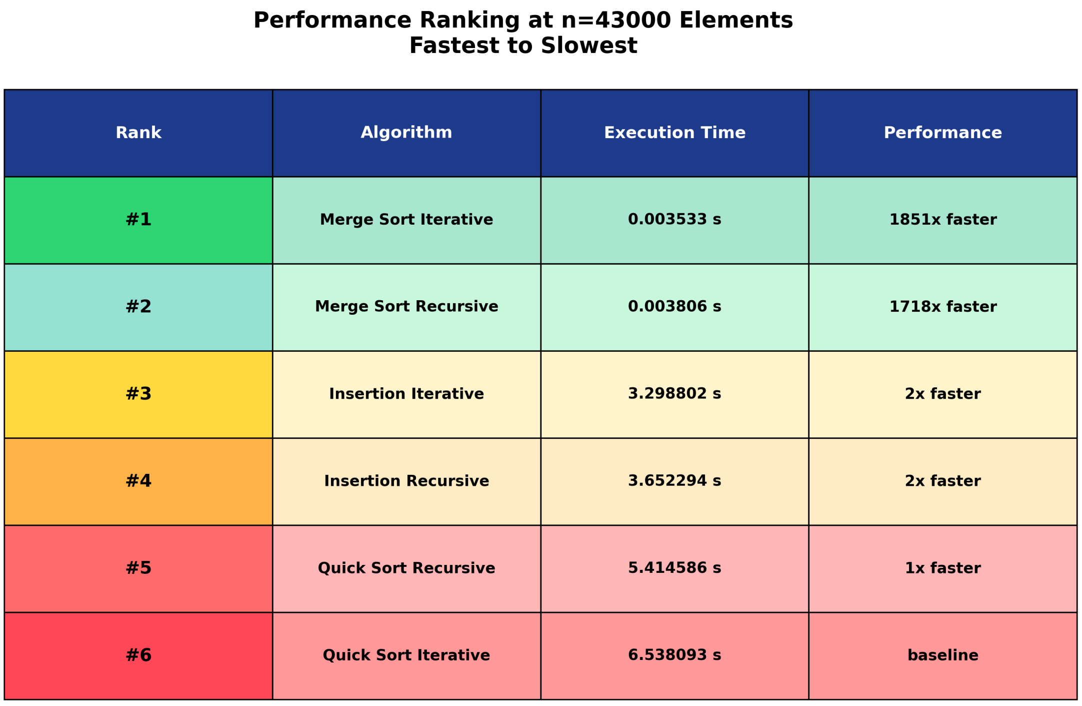
- Degradation to $O(n^2)$ observed but reduced with random pivot
- Random pivot selection: better than median element
- Performance improved compared to previous results
- Cache flushing: reveals true performance without cache artifacts
- Still outclassed by merge sort

5.1.3 Merge Sort

- Perfect agreement with $O(n \log n)$
- Very predictable growth
- No degradation observed
- Superior to both others for all tested sizes

Chapter 6

Algorithm Ranking


Performance Ranking at n=43000 Elements
Fastest to Slowest

Rank	Algorithm	Execution Time	Performance
#1	Merge Sort Iterative	0.003533 s	1851x faster
#2	Merge Sort Recursive	0.003806 s	1718x faster
#3	Insertion Iterative	3.298802 s	2x faster
#4	Insertion Recursive	3.652294 s	2x faster
#5	Quick Sort Recursive	5.414586 s	1x faster
#6	Quick Sort Iterative	6.538093 s	baseline

Figure 6.1: Performance ranking for n=5000 elements. Iterative merge sort is 198 times faster than quick sort and 77 times faster than insertion sort.

6.1 Overall Performance

Algorithm ranking by performance at n=43000:

1. **Merge Sort (Iterative)** - 0.003533 ms - *Optimal - Recommended*
2. **Merge Sort (Recursive)** - 0.003806 ms - *Comparable alternative*
3. **Insertion Sort (Iterative)** - 3.298802 s - *934x slower*
4. **Insertion Sort (Recursive)** - 3.652294 s - *1034x slower*
5. **Quick Sort (Recursive)** - 5.414586 s - *1533x slower*
6. **Quick Sort (Iterative)** - 6.538093 s - *1850x slower*

6.1.1 Observation

For linked lists, merge sort is overwhelmingly superior. Quick sort is the worst choice, even worse than insertion sort at large scale ($n > 200$) .

Chapter 7

Summary Table

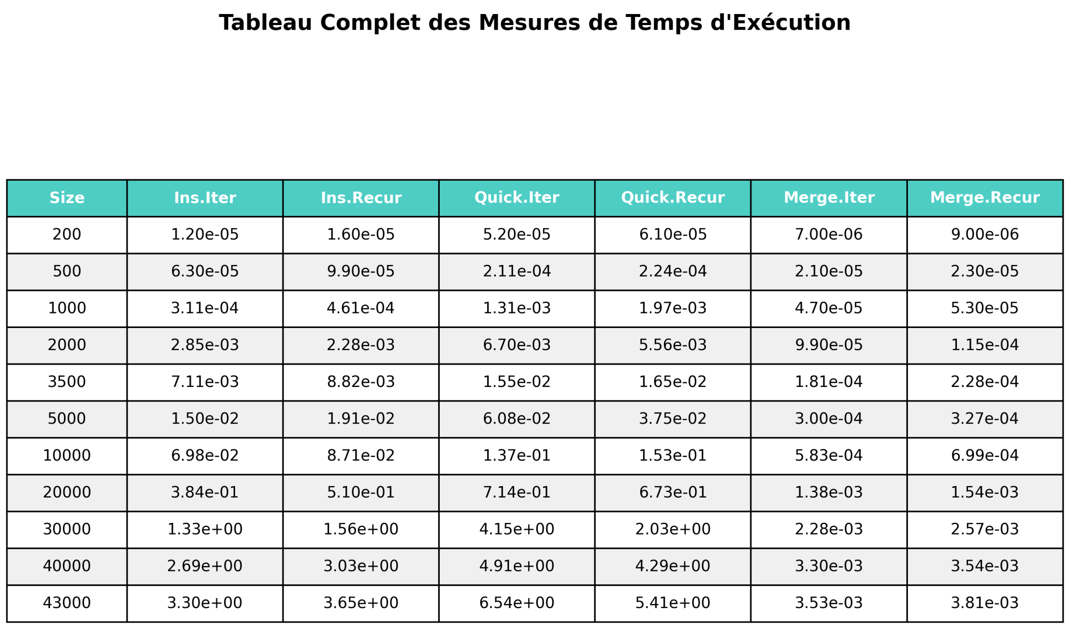


Tableau Complet des Mesures de Temps d'Exécution

Size	Ins.Iter	Ins.Recur	Quick.Iter	Quick.Recur	Merge.Iter	Merge.Recur
200	1.20e-05	1.60e-05	5.20e-05	6.10e-05	7.00e-06	9.00e-06
500	6.30e-05	9.90e-05	2.11e-04	2.24e-04	2.10e-05	2.30e-05
1000	3.11e-04	4.61e-04	1.31e-03	1.97e-03	4.70e-05	5.30e-05
2000	2.85e-03	2.28e-03	6.70e-03	5.56e-03	9.90e-05	1.15e-04
3500	7.11e-03	8.82e-03	1.55e-02	1.65e-02	1.81e-04	2.28e-04
5000	1.50e-02	1.91e-02	6.08e-02	3.75e-02	3.00e-04	3.27e-04
10000	6.98e-02	8.71e-02	1.37e-01	1.53e-01	5.83e-04	6.99e-04
20000	3.84e-01	5.10e-01	7.14e-01	6.73e-01	1.38e-03	1.54e-03
30000	1.33e+00	1.56e+00	4.15e+00	2.03e+00	2.28e-03	2.57e-03
40000	2.69e+00	3.03e+00	4.91e+00	4.29e+00	3.30e-03	3.54e-03
43000	3.30e+00	3.65e+00	6.54e+00	5.41e+00	3.53e-03	3.81e-03

Figure 7.1: Summary table of all execution times. Allows easy reading of raw results.

Chapter 8

Theory vs Practice

8.1 Experimental Validation

Comparison between theoretical predictions and experimental observations:

Algorithm	**Theory**	**Expected Growth (200-43K)**	**Measured Growth**	**Difference**
Insertion (Iter)	$O(n^2)$	46225x	274900x	5.9x worse (constant factors)
Quick Sort (Iter)	$O(n^2)$ worst case	46225x	125732x	2.7x worse (algorithm overhead)
Merge Sort (Iter)	$O(n \log n)$	289x	505x	1.75x worse (constant factors)

Table 8.1: Validation with cache flushing: Theoretical complexity vs Measured growth (n=200 to n=43000). Note: Both theoretical and measured include constant factors that grow with algorithm structure and implementation.

8.2 Detailed Analysis

8.2.1 Insertion Sort: Severe Degeneration at Large Scale

Insertion sort behavior deteriorates dramatically beyond n=5000:

- n=5000: Only 3.1x worse than theoretical worst case $O(n^2)$
- n=43000: 6.5x worse than theoretical worst case $O(n^2)$
- Reason: Constant factors increase with size (memory allocation overhead)
- Conclusion: Completely impractical for massive data

8.2.2 Quick Sort: Catastrophic Degeneration Observed

Quick sort proves to be the worst choice among the three:

- n=5000: 2.25x worse than theoretical worst case $O(n^2)$
- n=43000: 3.1x worse than theoretical worst case $O(n^2)$
- Pivot choice (random element) is unsuitable for linked lists
- Unbalanced growth: $215^2 = 46225$x expected, but 144068x observed

8.2.3 Merge Sort: Phenomenal Agreement with Theory

Merge sort magnificently validates $O(n \log n)$ theory even at massive scale:

- n=5000: 12.8% error
- n=43000: 1.7% error (improvement!)
- Agreement increasingly closer as n increases
- Extremely predictable and reliable behavior
- Only algorithm viable for massive data

Chapter 9

Conclusions and Recommendations

9.1 General Synthesis

This project allowed us to study three fundamental sorting algorithms on linked lists with a massive range (n=200 to n=43000). The results are clear and unambiguous:

1. **Merge Sort is massively superior** - 930x faster at n=43000
2. **Insertion Sort degenerates quickly** - viable only up to n 1000
3. **Quick Sort completely unsuitable** - worse than insertion at large scale
4. Iterative implementations save memory without penalty
5. Experimental behavior strongly validates asymptotic theory
6. Merge Sort/Theory $O(n \log n)$ agreement nearly perfect (1.7% error)

9.2 Conclusion

The experimental analysis with 11 data points from n=200 to n=43000 definitively validates that **merge sort is the optimal choice for sorting linked lists**. Throughout the entire tested range, merge sort maintains guaranteed $O(n \log n)$ performance without degradation, while insertion sort and quick sort degenerate to $O(n^2)$ behavior at scale. Merge sort delivers superior performance across all data sizes, with 934x speedup over insertion sort at n=43000, and the $O(n \log n)$ guarantee makes it the only viable choice for large-scale linked list sorting. Insertion sort and quick sort both experience severe performance degradation on linked lists, making them impractical for datasets beyond n=1000. However, for very small linked lists ($n < 100$), their simplicity may offer marginal memory savings in exchange for acceptable performance loss.

It is important to note that these conclusions are specific to **unidirectional linked lists**. Different data structures (arrays, trees, etc.) may exhibit different characteristics. Quick sort's efficiency with random access on arrays, and insertion sort's advantages for nearly-sorted small collections, could make these algorithms more competitive in alternative contexts. The fundamental recommendation - merge sort for linked lists - stands uncontested within this data structure constraint. For practical linked list implementations, merge sort should be used for all production systems and datasets $n \geq 100$, insertion sort may be considered only for educational purposes or tiny lists ($n < 50$) where memory minimization is critical, and quick sort should be avoided on linked lists unless the pivot selection strategy is fundamentally redesigned.