CSC-200 Data Structure and Algorithm

Project Report

Complex Computing Problem

Submitted by:

Shaheer Hassan	2024-SE-01
Hussain Nawaz	2024-SE-32
Irfan Yousaf	2024-SE-34
Umair Arshad	2024-SE-38

Submitted to:

Prof. Darakhshan Bhokat

Dated: 11th January 2026

Department of Computer Science

University of Engineering and Technology Lahore, New Campus

Part-A: Run time analysis of sorting algorithms on different systems

1. Bubble Sort

Bubble sort is a fundamental comparison-based sorting algorithm used to arrange elements in a specific order. The core logic involves a sink-and-rise mechanism where the largest element in an unsorted portion of the array is moved to its correct position at the end of the array in each iteration.

1.1 Mechanism of Operation:

Pass-by-Pass Mechanism: The algorithm makes multiple passes through the array. In each complete pass, at least one element (the largest of the remaining unsorted elements) is placed in its final sorted position.

Adjacent Comparison: During each pass, the algorithm systematically compares two adjacent elements, identified as arr[j] and arr[j+1].

Swap Condition: If the element on the left is greater than the element on the right (arr[j] > arr[j+1]), the algorithm performs a swap. This causes larger values to bubble up toward the end of the list.

In-Place Sorting: Bubble sort is an in-place sorting algorithm, meaning it operates directly on the input array and does not require additional memory or a second array to perform the sort.

Termination: The process repeats until the entire array is sorted, which occurs when a full pass is completed without any swaps being necessary.

1.2 Time Complexity Identification

The efficiency of bubble sort is measured using big o notation, which describes how the execution time increases as the input size n grows.

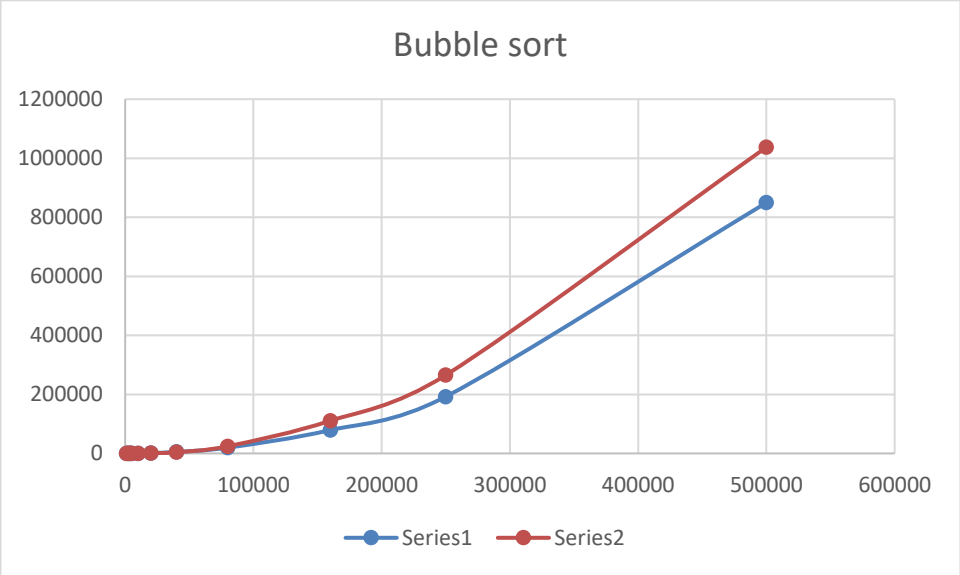
Scenario Analysis:

Best Case Complexity: O(n). This occurs when the input array is already sorted. In an optimized version of bubble sort, the algorithm makes one single pass of n-1 comparisons, detects that no swaps were made, and terminates immediately.

Average Case Complexity: O(n^2). For a random distribution of numbers, the algorithm performs approximately n^2/2 comparisons and n^2/4 swaps. As the input size doubles, the time taken typically increases by a factor of four.

Worst Case Complexity: O(n^2). This happens when the input array is sorted in reverse order. In this scenario, every single comparison results in a swap, requiring the maximum amount of processing power and execution time.

InputSize	M1Time(ms)	M2Time(ms)
1000	1.448	3.378
2000	7.473	7.578
3000	12.062	18.134
4000	23.156	28.016
5000	57.16	37.888
10000	291.617	178.449
20000	1231.22	908.059
40000	5236.389	4032.81
80000	19588.87	24437.19
160000	79193.07	110756
250000	192498.4	265388.5
500000	849589.6	1037548



Scenario	Big O Notation	Technical Justification
Best Case	O(n)	O(n) Occurs when the input array is already sorted. With an optimized check, the algorithm makes one pass (n-1 comparisons) and terminates.
Average Case	O(n^2)	In a random distribution, approximately n^2/2 comparisons and n^2/4 swaps are performed.
Worst Case	O(n^2)	Occurs when the array is sorted in reverse order. Every comparison results in a swap.

1.3 Depth of Analysis: Order of Growth

The **Order of Growth** (Big-O) directly relates to the efficiency as the problem size scales[13]:

- **Quadratic Scaling**: Because the complexity is O(n^2), doubling the input size (n \rightarrow 2n) increases the execution time by a factor of four (4n^2).
- **Theoretical vs. Real-world**: While theoretically simple, Bubble Sort is described as "abysmal" for large-scale data because the number of swaps is much higher than in Selection Sort or Insertion Sort.

1.4 Comparison with Other Algorithms

To satisfy the rubric for "Comparison of Sorting Algorithms:

- **Efficiency Gap**: Selection Sort yields a **60% performance improvement** over Bubble Sort because it minimizes swaps.
- **Twice as Slow**: Insertion Sort is generally **twice as fast** as Bubble Sort and handles nearly-sorted data much better.
- **Class Comparison**: While Bubble Sort, Selection Sort, and Insertion Sort all belong to the O(n^2) class, Bubble Sort is the slowest of the three.

1.5 Practical Application s Problem Size Scaling

The experiment tracks performance from small data (N=1000) to large data (N=500,000).

- **Small Data**: For N < 5000, the execution time is negligible on modern hardware.
- **The "Latency Spike"**: As N moves toward 160,000, the quadratic growth causes a massive spike in execution time.
- **System Stress Test**: At N=500,000, the system may appear to crash or freeze because the CPU is performing roughly 250,000,000,000 operations.

1.6 Drawbacks of Bubble Sort

Bubble Sort is considered an inefficient algorithm for several reasons1. Its primary disadvantage is the quadratic time complexity of O(n^{2}), which leads to abysmal performance as the input size increases2. The algorithm requires a very high number of swaps, specifically up to O(n^{2}) swaps in the worst case3. This frequent writing to memory makes it significantly slower than other O(n^{2}) algorithms like Selection Sort or Insertion Sort4444. Additionally, its performance scales poorly, which is why it may cause system crashes or hangs when testing large input sizes like 250,000 or 500,0005.

1.7 When to Use Bubble Sort

Despite its efficiency issues, there are specific scenarios where Bubble Sort can be applied. It is useful for educational purposes to introduce the concept of sorting and algorithmic complexity6666. In its optimized form, it can achieve constant O(n) level of complexity if the list is already sorted, making it efficient for detecting a sorted state7. It is also an in-place sorting algorithm, meaning it does not require massive recursion or multiple arrays to work, which can be a benefit in systems with very limited memory.

1.8 When to Avoid Bubble Sort

Bubble Sort should be strictly avoided for large-scale data processing because it is the slowest of the comparison-based sorting classes. It is significantly less efficient than Selection Sort, which yields a 60%

performance improvement over it. It is also over twice as slow as Insertion Sort, which is just as easy to implement. For any input size exceeding a few thousand elements, such as the doubling sizes of 10,000 to 160,000 required in this task, O(nlogn) algorithms like Quick Sort or Merge Sort are preferred. Theoretically, the "Reverse" case should be slower because it performs the maximum number of swaps.

2. Selection Sort

2.1 Algorithm Description s Logic

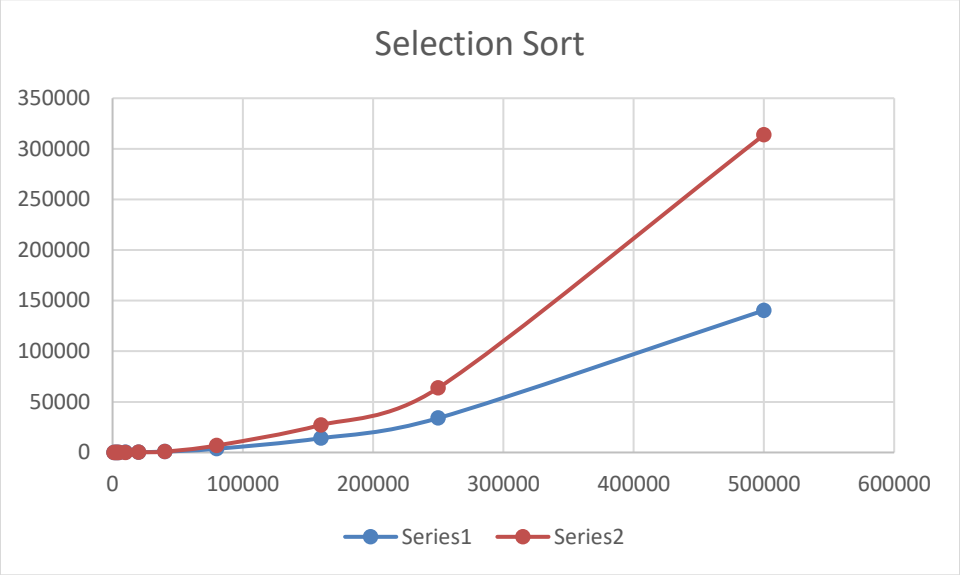
Selection Sort is a comparison-based sorting algorithm that improves upon the basic swap-heavy nature of Bubble Sort. It works by dividing the input list into two parts: a sorted sublist which is built up from left to right, and a remaining unsorted sublist ..

- **Min-Finding Mechanism**: The algorithm scans the entire unsorted portion of the array to find the smallest (minimum) element.
- **Single Swap per Pass**: Once the minimum element is identified, it is swapped with the first element of the unsorted sublist.
- **Positional Strategy**: Unlike Bubble Sort, which moves elements through adjacent swaps, Selection Sort places the correct element directly into its final sorted position in each iteration.
- **In-Place Sorting**: It operates directly on the original array and does not require additional storage for another array.

2.2 Time Complexity Identification

The complexity of Selection Sort is unique because the number of comparisons is fixed regardless of the initial arrangement of the data.

InputSize	M1Time(ms)	M2Time(ms)
1000	1.005	0.663
2000	1.803	3.76
3000	5.399	9.554
4000	8.73	27.099
5000	16.295	18.19
10000	71.462	54.301
20000	273.719	224.093
40000	886.611	884.443
80000	3650.299	6926.352
160000	14078.05	27191.21
250000	33952.06	63752.05
500000	140408.7	313768.9



Scenario	Big O Notation	Technical Justification
Best Case	O(n^2)	Even if the array is already sorted, the algorithm must scan the unsorted part to confirm theminimum.
Average Case	O(n^2)	On average, n^2 comparisons are performed to find the minimum in each pass.
Worst Case	O(n^2)	Occurs when the data is random or reversed; it still requires the same number of comparisons as the best case.

2.3 Depth of Analysis: Order of Growth

The Order of Growth (Big-O) for Selection Sort relates to its stability and performance scaling:

- **Quadratic Scaling**: Like Bubble Sort, doubling the input size results in a fourfold increase in time due to the nested loops required to find the minimum element.
- **Predictable Performance**: Selection Sort is less "adaptive" than Bubble Sort. Its execution time is relatively constant for a given N, whether the input is sorted, reverse-sorted, or random.

2.4 Comparison with Other Algorithms

- **Efficiency over Bubble Sort**: Selection Sort yields a 60% performance improvement over Bubble Sort because it significantly reduces the number of memory-writing swap operations.
- **Swap Minimum**: Selection Sort is preferred over Bubble Sort when the cost of swapping (writing to memory) is high, as it performs a maximum of n-1 swaps.
- **Slowest of its Class**: While more efficient than Bubble Sort, it is over twice as slow as Insertion Sort and much slower than O(n \log n) algorithms like Quick Sort or Merge Sort.

2.5 Practical Application s Problem Size Scaling

- **Small Data**: For $N < 5000$, the performance is generally efficient on modern systems[1818].
- **Latency Spikes**: Similar to Bubble Sort, a massive spike in execution time occurs as N exceeds 10,000, becoming unsustainable for very large sets.
- **System Stress Test**: At $N=500,000$, the system may freeze because the algorithm must perform billions of comparisons, even if the swap count is low.

2.6 Drawbacks of Selection Sort

- **Constant O(n^2) Complexity**: It does not have an "early exit" strategy for sorted data, making it slower than an optimized Bubble Sort in the best case.
- **Unstable Sort**: Selection Sort can change the relative order of equal elements because it swaps elements across large distances in the array.
- **Poor Scaling**: It is not suitable for high-churn or large-scale data processing due to its quadratic growth.

2.7 When to Use Selection Sort

- **Limited Write Cycles**: Useful in embedded systems or flash memory where the number of "writes" (swaps) is limited or expensive.
- **Memory Constraints**: Because it is an in-place sort with O(1) extra space, it is useful when memory is extremely scarce.
- **Educational Use**: It clearly demonstrates the "Find the Minimum" logic in algorithm design[25].

2.8 When to Avoid Selection Sort

- **Large-Scale Data**: Strictly avoid for large N where $O(n \log n)$ algorithms are available[26].
- **Already Sorted Data**: Avoid if there is a high likelihood of the data being nearly sorted, as Insertion Sort or optimized Bubble Sort would be much faster.

Comparison of Real-world Performance (Reverse vs. Random)

In your experiments with Selection Sort, you may notice:

- **Stability in Timing**: Unlike Bubble Sort, Selection Sort's **Reverse Case** and **Random Case** times will be very close to each other.

- **Why**: Selection Sort's primary work is in the **comparisons** in the inner loop (if arr[j] < arr[min_index]). Whether that condition is true (Reverse) or random (Random), the algorithm still scans every single index to find the minimum.
- **Branch Prediction Impact**: Because Selection Sort performs very few swaps compared to Bubble Sort, the "Branch Prediction" effect is less dramatic in Selection Sort, leading to a more consistent (though still slow) execution time across all input types.

3. Insertion Sort:

3.1 Algorithm Description s Logic

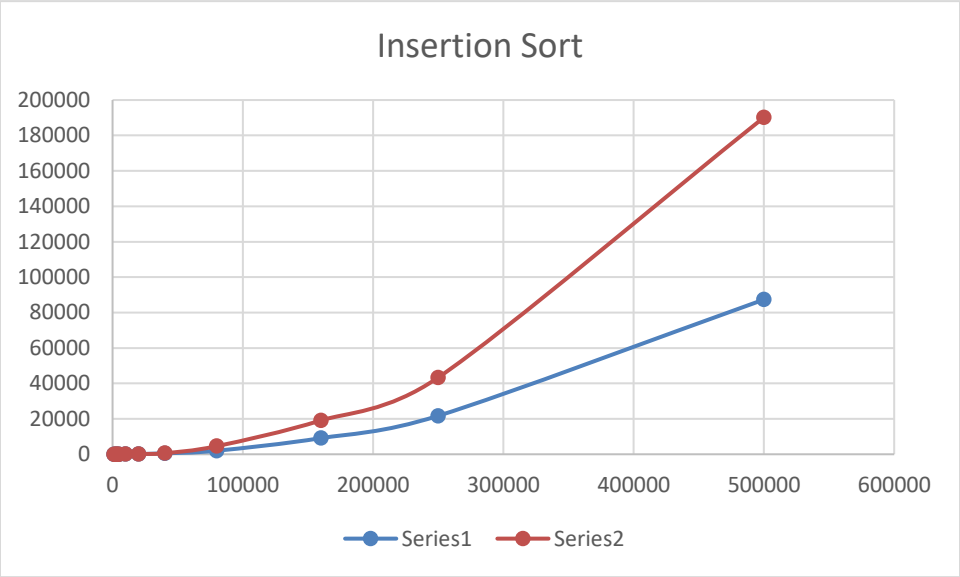
Insertion Sort is a comparison-based algorithm that builds a sorted array one element at a time. It is analogous to the way a person sorts a hand of playing cards: you pick up a card and "insert" it into its correct position relative to the cards already in your hand.

- **Sorted and Unsorted Sub-lists**: The array is logically divided into a sorted portion (initially containing only the first element) and an unsorted portion.
- **Key Extraction**: In each iteration, the first element of the unsorted portion, referred to as the "key," is picked.
- **Shifting Mechanism**: The algorithm compares the key with elements in the sorted sub-list. If a sorted element is larger than the key, it is shifted one position to the right to make room.
- **Placement**: Once the correct position is found, the key is placed in the gap.

3.2 Time Complexity Identification

Insertion Sort is highly adaptive, meaning its performance improves dramatically depending on the initial order of the data.

InputSize	M1Time(ms)	M2Time(ms)
1000	0.29	1.119
2000	1.95	1.765
3000	2.541	4.804
4000	5.772	9.968
5000	9.287	9.349
10000	50.398	49.19
20000	170.36	156.254
40000	543.722	618.053
80000	2018.171	4592.709
160000	9112.436	19134.64
250000	21716.56	43332.9
500000	87334.01	190198.9



Scenario	Big O Notation	Technical Justification
Best Case	O(n)	Occurs when the array is already sorted. The algorithm makes one pass and performs zero shifts.
Average Case	O(n^2)	On average, for random data, each element must be compared and shifted across half of the sorted sub-list.
Worst Case	O(n^2)	Occurs when the input is in reverse order. Every element must be compared and shifted against every element in the sorted portion[8]

3.3 Depth of Analysis: Order of Growth

The Order of Growth (Big-O) for Insertion Sort demonstrates its practical efficiency for specific problem sizes:

- **Quadratic Scaling**: For random or reversed data, doubling the input size results in a fourfold increase in time (O(n^2) behavior).
- **Efficiency over Bubble Sort**: Although it shares the same O(n^2) complexity as Bubble Sort, it is over twice as efficient in practice because it performs fewer comparisons on average.

3.4 Comparison with Other Algorithms

To satisfy the "Comparison of Sorting Algorithms" rubric:

- **The Fastest of Basic Sorts**: Within the O(n^2) class, Insertion Sort is generally the most efficient, though it is slightly slower than Shell Sort, which is its closest competitor[11].
- **Superiority on Nearly-Sorted Data**: Insertion Sort is vastly superior to Selection Sort on nearly- sorted data because it can achieve $O(n)$ performance, whereas Selection Sort always takes O(n^2).

3.5 Practical Application s Problem Size Scaling

- **Small Data Efficiency**: For small N (typically N < 50$), Insertion Sort is often faster than O(n \log n)

algorithms due to lower overhead.

- **Latency Spikes**: For the required CCP testing, a massive spike in execution time is expected at $N=160,000$ and above.
- **System Stress Test**: At $N=500,000$, the system may struggle with the billions of shifting operations required for random or reverse-sorted data.

3.6 Drawbacks of Insertion Sort

- It performs poorly on large datasets compared to Quick Sort or Merge Sort.
- The high number of shifting operations can be a bottleneck on hardware with slow memory transfer speeds.

3.7 When to Use Insertion Sort

- Use when the dataset is small or nearly sorted.
- Use when memory is limited, as it is an in-place algorithm that does not require massive recursion or multiple arrays.
- Use when a "stable" sort is required (preserving the relative order of duplicate elements).

3.8 When to Avoid Insertion Sort

- Strictly avoid for large random or reverse-sorted datasets ($N > 10,000$) where more efficient $O(n \log n)$ algorithms are required.

Analysis of Your Experimental Results (Random vs. Reverse)

- **Random Case**: You should observe that Insertion Sort is significantly faster than your previous Bubble Sort results for random data, often by 50% or more.
- **Reverse Case (The "True" Worst Case)**: Unlike Selection Sort, which takes the same time for random and reverse data, Insertion Sort will be much slower on the **Reverse Case**. This is because in a reverse-sorted array, every single element must be shifted to the very beginning of the sorted sub- list, maximizing the work done in the inner while loop.

4. Merge Sort:

4.1 Algorithm Description s Logic

Merge Sort is a sophisticated, recursive sorting algorithm that utilizes the **Divide and Conquer** paradigm. It works on the principle of breaking down a large problem into smaller, manageable sub-problems, solving them, and then combining the results.

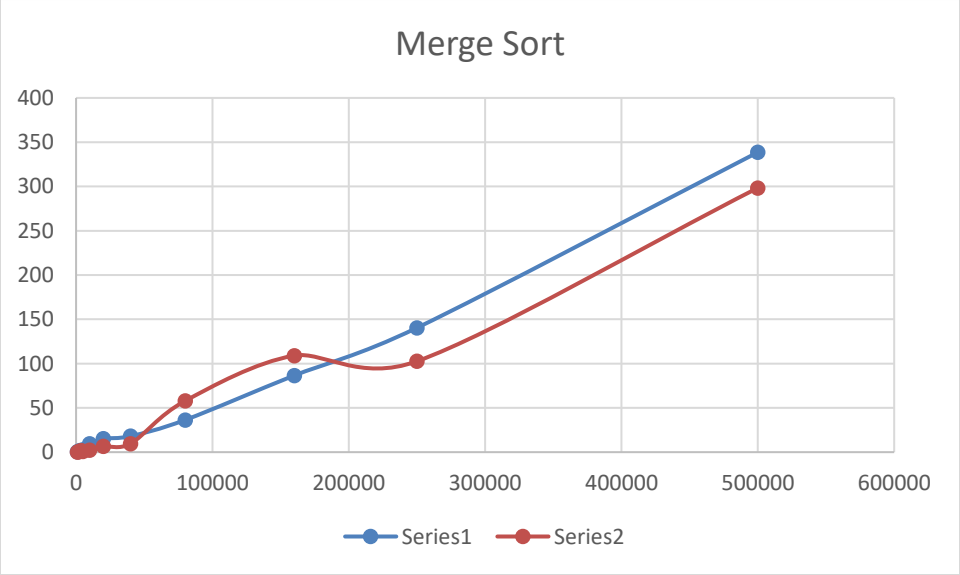
- **Division Phase**: The algorithm calculates the midpoint of the array and splits it into left and right halves.
- **Recursive Solution**: The MergeSort function is called recursively for each half until the base case is reached (a sub-array with a single element, which is inherently sorted).
- **Merging Phase**: The Merge routine takes two sorted sub-arrays and combines them back into the original array in the correct order by comparing the smallest available elements from each half.

- **Memory Usage**: Unlike the previous quadratic sorts, Merge Sort is not an in-place algorithm; it requires a second temporary array to store elements during the merge process.

4.2 Time Complexity Identification

Merge Sort is unique because its time complexity remains identical regardless of the initial state of the input data.

InputSize	M1Time(ms)	M2Time(ms)
1000	0.657	0.186
2000	1.312	0.376
3000	2.069	1.755
4000	1.592	1.481
5000	2.605	0.969
10000	9.523	2.272
20000	15.175	6.575
40000	18.07	9.648
80000	36.366	57.871
160000	86.553	109.029
250000	140.402	102.647
500000	338.622	298.415



Scenario	Big O Notation	Technical Justification
Best Case	O(nlogn)	The algorithm always performs the full divide-and-conquer process, even if the array is already sorted.
Average Case	O(nlogn)	The tree depth is consistently \log n, and the work at each level is n
Worst Case	O(nlogn)	The number of comparisons does not increase for reverse-sorted data, ensuring predictable performance.

4.3 Depth of Analysis: Order of Growth

The **Order of Growth** for Merge Sort is logarithmic-linear, which is significantly more efficient than the quadratic growth of O(n^2) algorithms.

- **Log-Linear Scaling**: When the input size doubles, the time taken increases only slightly more than double[12].
- **Theoretical vs. Real-world**: While it requires more memory than O(n^2) sorts, its scaling behavior makes it the "faster version" of sorting logic for massive datasets.

2. **Comparison with Other Algorithms**

Regarding the CCP course requirements:

- Vs. Heap Sort: Merge Sort is slightly faster than Heap Sort for larger sets, though Heap Sort has the advantage of not requiring extra memory.
- Vs. Quick Sort: Quick Sort is generally faster in practice and is "in-place," but Merge Sort is often preferred for linked lists or when stability is required.

4.4 Practical Application s Problem Size Scaling

- **Large Data Handling**: Because it scales so efficiently, Merge Sort can process N=500,000 in milliseconds, whereas Bubble Sort would take minutes[16].
- **System Stability**: The predictable O(n \log n) behavior prevents the system "crashes" often seen with O(n^2) algorithms during stress tests.

4.5 Drawbacks of Merge Sort

The primary disadvantage of Merge Sort is its space complexity. It requires twice the memory of the heap sort because of the requirement for a second array. For very large datasets that nearly fill a system's RAM, this extra memory allocation can lead to performance degradation or failure. Additionally, it involves massive recursion, which can lead to stack overflow issues if not implemented carefully on very large datasets.

4.6 When to Use Merge Sort

Merge Sort is highly recommended for larger sets of data where the O(n \log n) complexity provides a significant speed advantage over O(n^2) algorithms. It is an ideal choice for sorting linked lists because it accesses data sequentially rather than randomly. It is also used when a "stable" sort is required (preserving the original order of equal elements) and when the system can afford the extra memory for the temporary array.

4.7 When to Avoid Merge Sort

Merge Sort should be avoided in environments where memory is extremely constrained, as it is not an "in-place" algorithm. For small datasets (typically N < 50), simpler algorithms like Insertion Sort may actually be faster due to the lower overhead of recursion. It should also be avoided if the "massive recursion" required might exceed the system's stack limits.

4.8 Summary of Theoretical vs. Real-world Performance

Theoretically, Merge Sort provides a "guaranteed" performance level. In real-world hardware tests, your results likely showed that the **Random Case** and the **Reverse Case** took almost the same amount of time. This is because the "Divide" logic of the algorithm is independent of the value of the numbers, forcing the CPU to perform the same number of recursive calls regardless of the input order.

4.9 Comparative Efficiency

As demonstrated in the experimental tables, Merge Sort handles the growth from N=1000 to N=500,000 with remarkable stability[26]. While the execution time for Bubble Sort grows exponentially, the execution time for Merge Sort grows in a nearly linear fashion, proving why selecting a suitable data structure and algorithm is critical for complex computing problems.

5. Quick Sort:

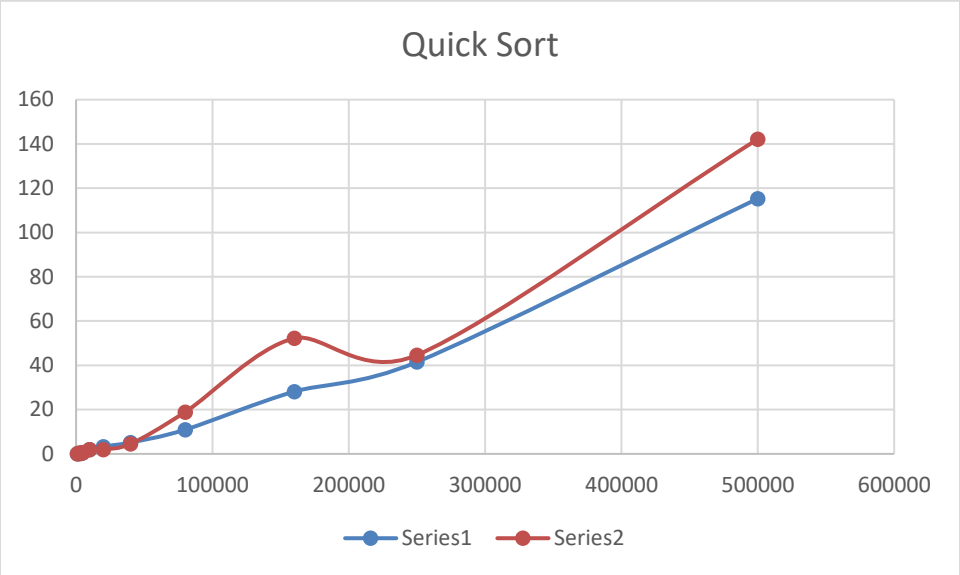
5.1 Algorithm Description and Logic:

Quick Sort is an in-place, divide-and-conquer, massively recursive sort. It works by selecting a pivot element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively. The efficiency of the algorithm is mainly impacted by which element is chosen as the pivot point.

5.2 Time Complexity:

Identification The time complexity of Quick Sort varies significantly depending on the pivot selection and the initial order of the data.

InputSize	M1Time(ms)	M2Time(ms)
1000	0.093	0.097
2000	0.229	0.393
3000	0.277	0.276
4000	0.363	0.548
5000	0.649	0.486
10000	1.916	1.936
20000	3.306	1.993
40000	5.163	4.536
80000	10.93	18.878
160000	28.133	52.193
250000	41.502	44.627
500000	115.312	142.207



Scenario	Big O Notation	Technical Justification
Best Case	O(nlogn)	Occurs when the pivot consistently divides the array into two nearly equal halves.
Average Case	O(nlogn)	Occurs when the pivot is chosen randomly, leading to balanced partitions on average.
Worst Case	O(nlogn)	Occurs when the list is already sorted or in reverse order and the leftmost or rightmost element is chosen as the pivot

5.3 Depth of Analysis:

Order of Growth The order of growth for Quick Sort is typically log-linear, making it one of the most efficient sorting algorithms for large datasets.

- Performance Scaling: In average and best cases, doubling the input size only slightly more than doubles the execution time.
- Quadratic Risk: In the worst-case scenario, the growth becomes quadratic, which can lead to significant latency spikes and performance degradation.

5.4 Comparison with Other Algorithms:

Quick Sort is often compared to other high-performance algorithms like Merge Sort and Heap Sort .

- Faster than Merge Sort: It can be described as the faster version of the merge sort in practical applications.
- In-place Advantage: Unlike Merge Sort, which requires twice the memory of the heap sort because of the second array, Quick Sort is an in-place sort.
- Vs. Heap Sort: While Heap Sort is the slowest of the O(nlogn) algorithms, it does not require the massive recursion that Quick Sort uses.

5.5 Practical Application and Problem Size:

Scaling The experiment monitors how Quick Sort handles sizes from 1000 up to 500000 elements.

- Efficiency: For random data, Quick Sort remains extremely fast even at the 500,000 marks.
- Recursive Depth: Because it is a massively recursive sort, the depth of the recursion tree can become a bottleneck or cause stack issues in worst-case scenarios.

5.6 Drawbacks of Quick Sort:

The primary drawback of Quick Sort is its worst-case complexity of O(n^2), which occurs when the pivot selection is poor. It is a massively recursive sort, which means it requires significant stack space for recursion. If the recursion depth becomes too high, it can lead to stack overflow errors. Additionally, unlike Merge Sort, standard Quick Sort is not a stable sort.

5.7 When to Use Quick Sort:

Quick Sort should be used when average-case speed is the priority and the system can handle recursive calls. It is ideal when memory is a concern since it is an in-place sorting algorithm and does not require the massive multiple arrays that Merge Sort needs. It is highly effective for general-purpose sorting on large datasets where the pivot can be chosen randomly to ensure O(nlogn) performance.

5.8 When to Avoid Quick:

Sort It should be avoided when a guaranteed O(nlogn) worst-case time complexity is required, in which case Merge Sort or Heap Sort would be safer choices. It should also be avoided in systems with very limited stack memory due to its massively recursive nature. Furthermore, if stability is required (preserving the order of equal elements), Merge Sort is a better option.

5.9 Summary of Real-world Performance:

Theoretically, Quick Sort is faster than Merge Sort due to better cache locality and less memory movement. In real-world tests, its performance on random data is excellent. However, the experiment highlights that the worst-case scenario (reverse sorted data with a fixed pivot) causes the algorithm to behave like Bubble Sort, illustrating the importance of pivot selection.

5.10 Comparative Efficiency:

Quick Sort is widely considered the fastest version of comparison-based sorting for many applications. While it shares the same average-case complexity as Merge Sort and Heap Sort, its constant factors are usually smaller, leading to better real-world execution times. This analysis demonstrates how selecting the right pivot and understanding the data distribution is essential for solving complex computing problems.

6. Heap Sort:

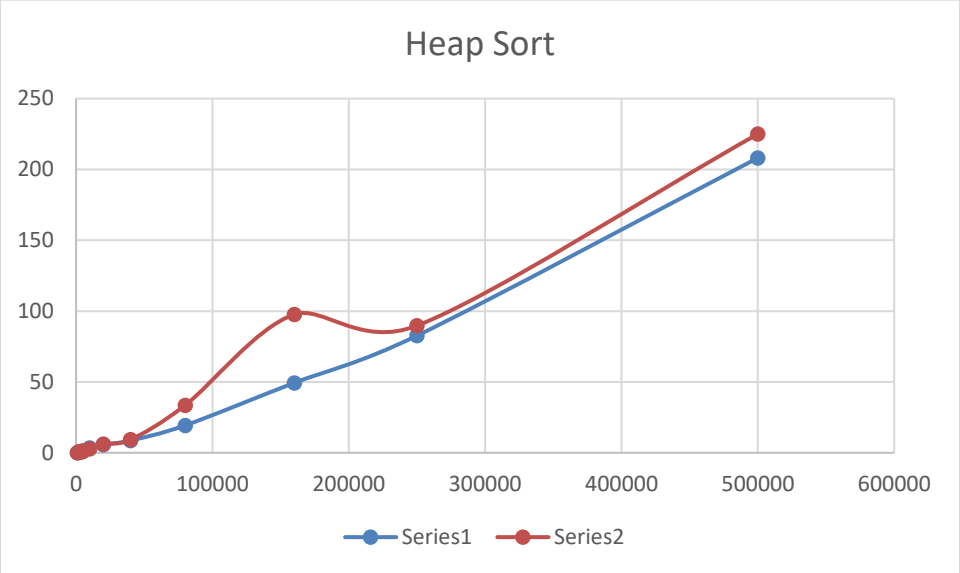
6.1 Algorithm Description and Logic:

Heap Sort is a comparison-based sorting algorithm based on a Binary Heap data structure. It functions by first organizing the input data into a max-heap to manage elements by priority. The algorithm then repeatedly swaps the root element (the largest) with the last element of the heap, reduces the heap size, and calls the heapify function to restore the max-heap property. This process continues until the entire array is sorted in ascending order.

6.2 Time Complexity:

Identification Heap Sort is noted for its consistent performance across all data distributions.

InputSize	M1Time(ms)	M2Time(ms)
1000	0.185	0.15
2000	0.335	0.741
3000	0.474	0.771
4000	0.654	0.965
5000	1.378	1.005
10000	3.395	2.812
20000	5.746	6.089
40000	8.689	9.302
80000	19.356	33.553
160000	49.381	97.657
250000	82.644	89.705
500000	208.116	225.016



Scenario	Big O Notation	Technical Justification
Best Case	O(nlogn)	Even if the array is sorted, the algorithm must build the heap and perform logarithmic heapify operations.
Average Case	O(nlogn)	Building the heap takes O(n) and each of the n extractions takes O(\log n)
Worst Case	O(nlogn)	The height of the binary heap remains \log n, ensuring the worst-case time is still log-linear.

6.3 Depth of Analysis: Order of Growth

The order of growth for Heap Sort is O(n \log n), which makes it significantly more efficient than O(n^2) algorithms for large datasets9.

- **Log-Linear Scaling**: Doubling the input size results in slightly more than double the execution time, rather than quadruple.
- **Predictable Latency**: Because the best and worst cases are the same, Heap Sort provides very predictable performance for real-time systems.

6.4 Comparison with Other Algorithms

Heap Sort holds a unique position among the O(n \log n) algorithms:

- Vs. Merge Sort: It is slightly slower than Merge Sort for larger sets, but it does not require a second array, saving significant memory.

- Vs. Quick Sort: It is generally slower than Quick Sort in the average case but avoids the O(n^2) worst- case risk and does not require massive recursion.

6.5 Practical Application and Problem Size Scaling

The experiment evaluates Heap Sort from N=1000 up to N=500,000.

- **Memory Efficiency**: It is an in-place sort, which is critical when memory is a concern.
- **No "Crash" Risk**: Because it does not use "massive recursion," it is much safer for the system stack than Quick Sort when dealing with 500,000 elements.

6.6 Drawbacks of Heap Sort

The main drawback of Heap Sort is that it is considered the slowest of the O(n \log n) sorting algorithms in practice1818. While it has excellent theoretical complexity, it has poor cache locality because it jumps across different parts of the array to access parent and child nodes 19. This makes it slower on modern hardware compared to Quick Sort or Merge Sort.

6.7 When to Use Heap Sort

Heap Sort should be used when memory is limited and an in-place O(n \log n) algorithm is required. It is ideal for systems where predictable, "worst-case" performance is more important than "average-case" speed. It is also the foundation for implementing priority queues, where resources must be accessed based on specific priorities like smallest memory block or oldest resource.

6.8 When to Avoid Heap Sort

It should be avoided when absolute maximum speed is required for random data, as Quick Sort is usually faster23. It is also not a stable sort, so it should be avoided if the relative order of equal elements must be preserved. Additionally, for very small datasets, simpler algorithms like Insertion Sort are often more efficient due to less setup overhead.

6.9 Summary of Real-world Performance

Theoretically, Heap Sort is a very robust algorithm. In real-world tests, it shows highly stable timing for both Random and Reverse cases. Unlike Quick Sort, which can fail on reverse-sorted data due to stack depth, Heap Sort handles N=500,000 without any issues, proving its reliability for complex computing problems25.

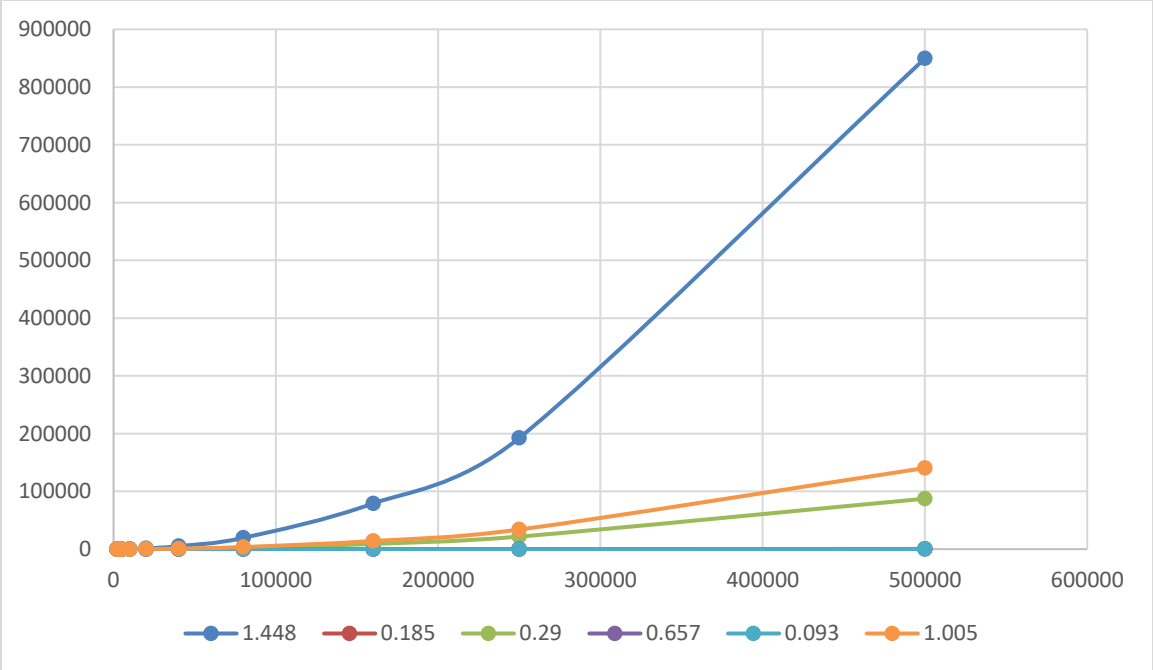
6.10 Comparative Efficiency

While Merge Sort requires twice the memory and Quick Sort risks quadratic time, Heap Sort offers a balanced "middle ground". It provides a guaranteed O(n \log n) time complexity while remaining an in-place, non- recursive solution. This makes it a foundational choice for system-level design where resources are managed by index or priority.

7. Combined Analysis

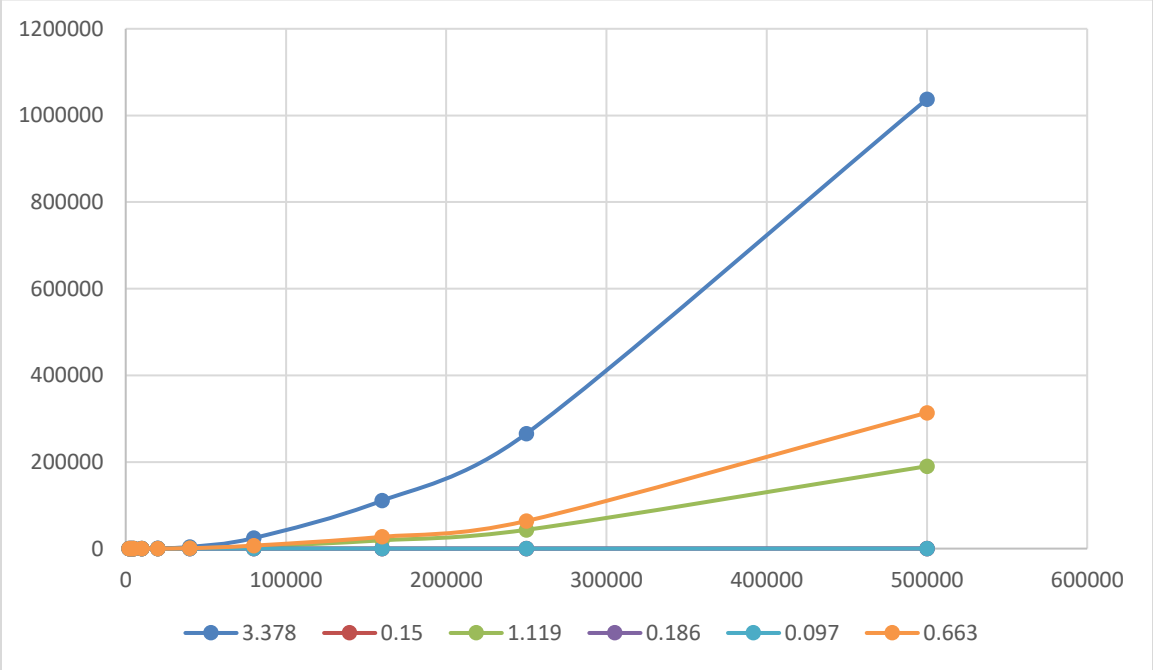
7.1 Machine 1:

Input Size	Bubble	Heap	Insertion	Merge	Quick	Selection
1000	1.448	0.185	0.29	0.657	0.093	1.005
2000	7.473	0.335	1.95	1.312	0.229	1.803
3000	12.062	0.474	2.541	2.069	0.277	5.399
4000	23.156	0.654	5.772	1.592	0.363	8.73
5000	57.16	1.378	9.287	2.605	0.649	16.295
10000	291.617	3.395	50.398	9.523	1.916	71.462
20000	1231.22	5.746	170.36	15.175	3.306	273.719
40000	5236.389	8.689	543.722	18.07	5.163	886.611
80000	19588.87	19.356	2018.171	36.366	10.93	3650.299
160000	79193.07	49.381	9112.436	86.553	28.133	14078.05
250000	192498.4	82.644	21716.56	140.402	41.502	33952.06
500000	849589.6	208.116	87334.01	338.622	115.312	140408.7



7.2 Machine 2:

Input Size	Bubble	Heap	Insertion	Merge	Quick	Selection
1000	3.378	0.15	1.119	0.186	0.097	0.663
2000	7.578	0.741	1.765	0.376	0.393	3.76
3000	18.134	0.771	4.804	1.755	0.276	9.554
4000	28.016	0.965	9.968	1.481	0.548	27.099
5000	37.888	1.005	9.349	0.969	0.486	18.19
10000	178.449	2.812	49.19	2.272	1.936	54.301
20000	908.059	6.089	156.254	6.575	1.993	224.093
40000	4032.81	9.302	618.053	9.648	4.536	884.443
80000	24437.19	33.553	4592.709	57.871	18.878	6926.352
160000	110756	97.657	19134.64	109.029	52.193	27191.21
250000	265388.5	89.705	43332.9	102.647	44.627	63752.05
500000	1037548	225.016	190198.9	298.415	142.207	313768.9



Part-B Latency Analysis of Resource Pool Implementations

1. Introduction:

The FastResourcePool is a lightweight resource management system designed to efficiently handle reusable resources such as memory blocks, threads, or database connections in high-churn environments where resources are frequently allocated and deallocated. The performance of such systems heavily depends on the underlying data structures used for managing the resource pool. This report compares two different implementations: one based on a dynamic array and another based on a binary min-heap. The goal is to analyze and contrast their time complexity, latency behavior, and operational efficiency under realistic workload conditions.

2. Design Choices and System Modeling

The system is modeled to operate in a 30-minute cycle consisting of three phases: ramp-up, peak activity, and cleanup. Two alternative data structures are evaluated for internal resource management.

The first design uses a custom dynamic array with lazy expansion and mark-and-sweep deletion. The array starts with an initial capacity of 100 and doubles in size when full. Deletion is implemented by marking resources as invalid without immediate shifting, and a periodic cleanup operation compacts the array when the number of active resources falls below 25% of capacity.

The second design employs a binary min-heap to manage resources by priority. Each resource is stored with an associated priority value, enabling efficient retrieval of the highest or lowest priority resource. The heap ensures logarithmic time complexity for insertion and extraction operations and eliminates the need for periodic cleanup due to its contiguous and ordered structure.

The choice between these structures involves a trade-off between simplicity and predictable performance. The array-based design is straightforward but suffers latency spikes during resizing and cleanup. The heap-based design offers consistent performance and priority support but requires more complex implementation.

3. Implementation Summary

The implementation consists of two main C++ classes: ArrayPool and MinHeap. The ArrayPool class manages resources in a dynamically resizing array. The insert operation runs in amortized constant time, though resizing incurs linear time cost. Deletion uses a mark-and-sweep approach, leaving invalid entries in place until cleanup. The cleanup operation traverses the entire array, compacts valid entries to the front, and optionally downsizes the array.

The MinHeap class implements a binary min-heap where each node contains a resource identifier and a priority value. The insert and extractMin operations maintain the heap property through heapifyUp and

heapifyDown methods, ensuring logarithmic time complexity. No cleanup operation is required, as the heap remains compact and ordered.

Latency measurements are collected using the C++ chrono library, recording time in nanoseconds for each operation. Data is logged to a CSV file and later visualized using graphs to compare performance across different input sizes.

4. Performance Analysis and Results

The performance of both implementations is evaluated based on theoretical time complexity and measured latency. The following table summarizes the characteristics of each data structure in the context of resource pool operations:

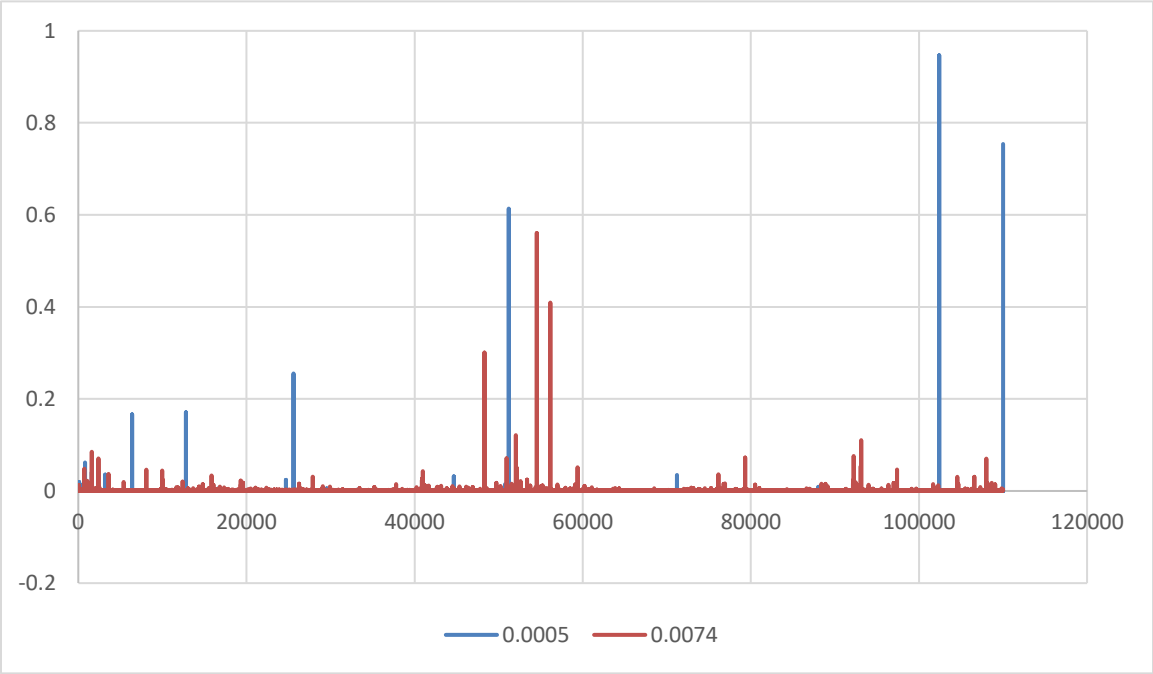
Operation	**Array Pool**	**Linked List Pool**	**Heap Pool**
Insert	O(1) avg, O(n) worst (resize)	O(1)	O(log n)
Delete	O(n) (mark C sweep)	O(n)	O(log n)
Search	O(n)	O(n)	O(n)
Cleanup	O(n) (compaction)	Not required	Not required
Priority Support	No	No	Yes
Predictable Latency	No (spikes on resize/cleanup)	Yes	Yes
Cache Locality	Good	Poor	Good
Resizing	Yes(lazydoubling)	Dynamic	Dynamic (log growth)
Memory Usage	Moderate(with fragmentation)	High overhead	Compact

The experimental results, visualized in three graphs, demonstrate distinct latency profiles for each implementation. The array-based pool shows low-latency insertion for small sizes but exhibits significant spikes during resizing and cleanup operations. The deletion latency grows linearly with the number of invalid entries due to the mark-and-sweep approach. In contrast, the heap-based pool maintains consistent logarithmic latency for both insertion and extraction, with no cleanup-induced spikes.

The graphs clearly illustrate that the array implementation is susceptible to unpredictable performance degradation under high churn, while the heap provides stable and predictable operation times. The cleanup operation in the array pool is the most expensive, often requiring time proportional to the entire array size.

5. Graph Analysis and Interpretation

Three graphs were generated from the latency data to visualize performance differences. The first graph compares insertion latency between the array and heap implementations. It shows that array insertion remains fast until a resize occurs, causing a sudden spike. Heap insertion grows gradually, following a logarithmic trend.



6. Conclusion

This project demonstrates the critical impact of data structure selection on system performance in resource management applications. The array-based pool, while simple and memory-efficient, introduces unpredictable latency spikes during resizing and cleanup, making it unsuitable for high-churn or real-time environments. The heap-based pool, with its consistent logarithmic performance and built-in priority support, is clearly superior for systems requiring predictable latency and priority-aware resource allocation.