Comprehensive Study of Sorting Algorithms

PROJECT REPORT

BY:

KOUCEM LAMIA
NAOUI KHALED

1.General Introduction:

Sorting algorithms are among the most fundamental tools in computer science. They are used extensively in databases, operating systems, scientific computing, and many other domains where large volumes of data must be processed efficiently. The choice of a sorting algorithm has a direct impact on performance, memory usage, and scalability.

This project aims to study and compare several classical sorting algorithms by analyzing both their **theoretical time complexities** and their **observed practical behavior** during execution. The algorithms selected represent different design paradigms, By examining best-case and worst-case scenarios, this study highlights the strengths and limitations of each approach.

2. Objectives of this project:

The main goals of this project are:

- Put into practice and test some sorting algorithms;
- Study sorting algorithms and calculate their complexity;
- Confront theoretical complexity and evaluation of running cost;
- In order to study the real cost of the algorithms, you will test them on arrays of integers
- Compare the different methods from a theoretical and experimental point of view.

3.Methodology:

Each algorithm was implemented and tested on arrays of varying sizes. Execution times were measured using system clock functions, and input configurations were chosen to represent best-case and worst-case scenarios whenever possible.

Special attention was given to hardware-related factors such as cache behavior and memory access patterns, which can influence performance beyond asymptotic analysis.

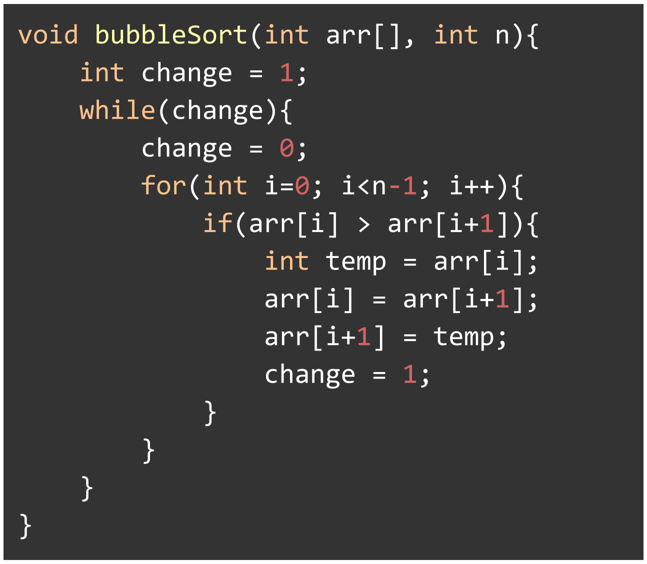
4. Analysis of Sorting Algorithms

4.1 Bubble Sort:

4.1.1 Algorithm Overview

Bubble Sort is a basic comparison-based algorithm that repeatedly scans the array and swaps adjacent elements if they are in the wrong order. After each pass, the largest unsorted element “bubbles up” to its final position at the end of the array.

4.1.2 Algorithm Implemented



```
void bubbleSort(int arr[], int n){
    int change = 1;
    while(change){
        change = 0;
        for(int i=0; i<n-1; i++){
            if(arr[i] > arr[i+1]){
                int temp = arr[i];
                arr[i] = arr[i+1];
                arr[i+1] = temp;
                change = 1;
            }
        }
    }
}
```

4.1.3 Time Complexity Analysis

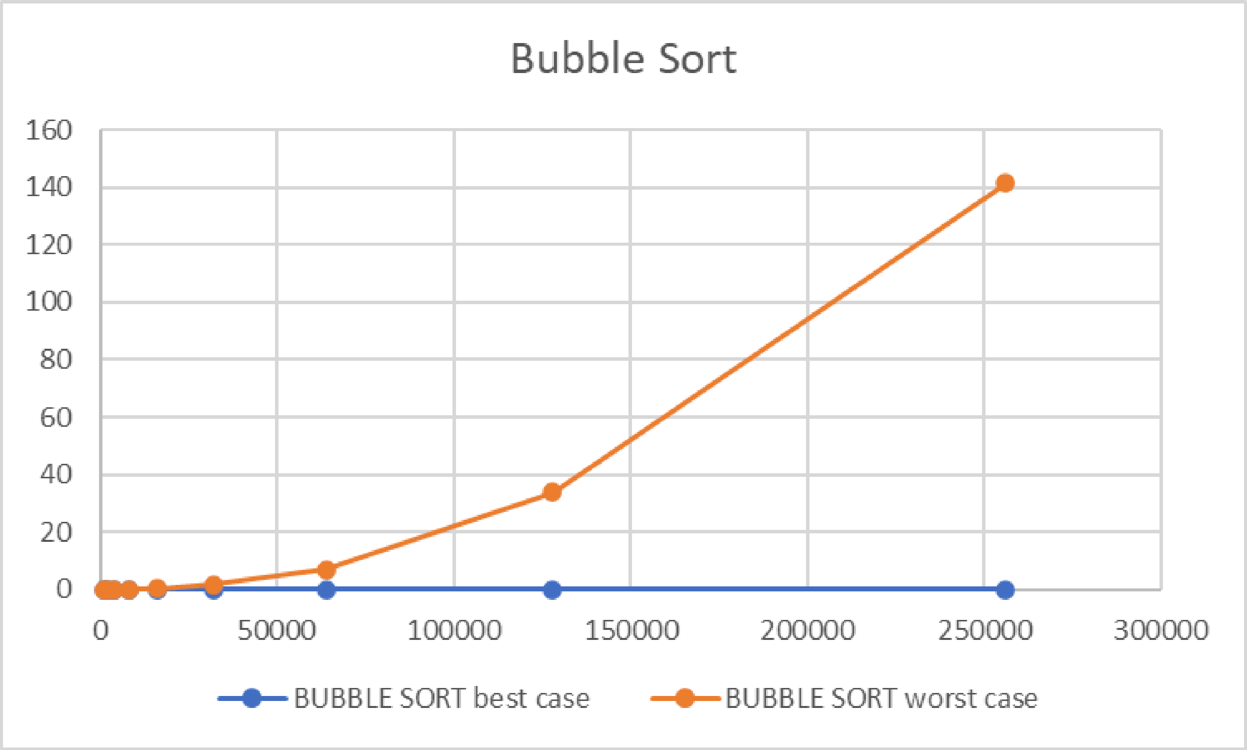
- **Best Case:**
  When the array is already sorted, only one full traversal is required. No swaps are performed, resulting in a linear time complexity **O(n)**.

- **Worst Case:**
  If the array is sorted in reverse order, every comparison leads to a swap, and the algorithm performs a full traversal for each element. This produces a quadratic time complexity **O(n²)**.

4.1.4 Practical Considerations

Despite its inefficiency, Bubble Sort has very predictable memory access patterns. Its sequential traversal allows effective cache usage, which can make small input sizes appear to execute faster than expected.

4.1.5 Time Execution Graph

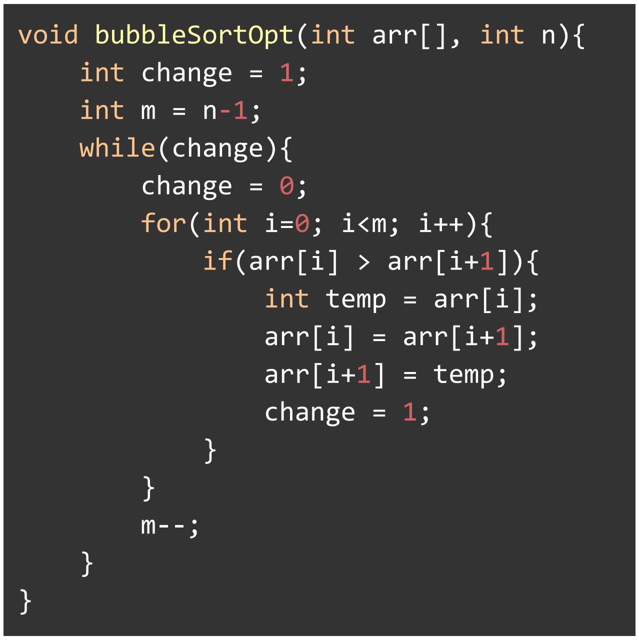


4.2 Optimized Bubble Sort:

4.2.1 Optimization Principle

The optimized version of Bubble Sort improves performance by detecting whether a pass performs any swaps. If no swaps occur, the algorithm stops early. Additionally, it reduces the number of comparisons by ignoring elements already placed at the end of the array.

4.2.2 Algorithm Implemented



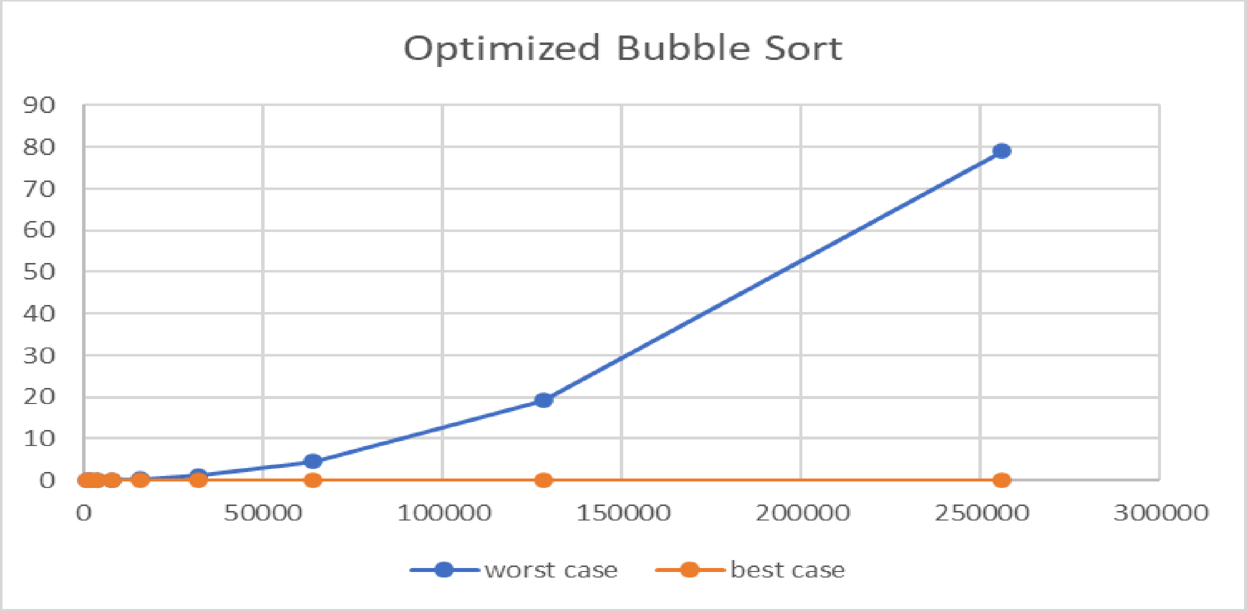
```
void bubbleSortOpt(int arr[], int n){
    int change = 1;
    int m = n-1;
    while(change){
        change = 0;
        for(int i=0; i<m; i++){
            if(arr[i] > arr[i+1]){
                int temp = arr[i];
                arr[i] = arr[i+1];
                arr[i+1] = temp;
                change = 1;
            }
        }
        m--;
    }
}
```

4.2.2 Complexity Evaluation

- **Best Case:**
  The algorithm terminates after one pass, achieving **O(n)** complexity.

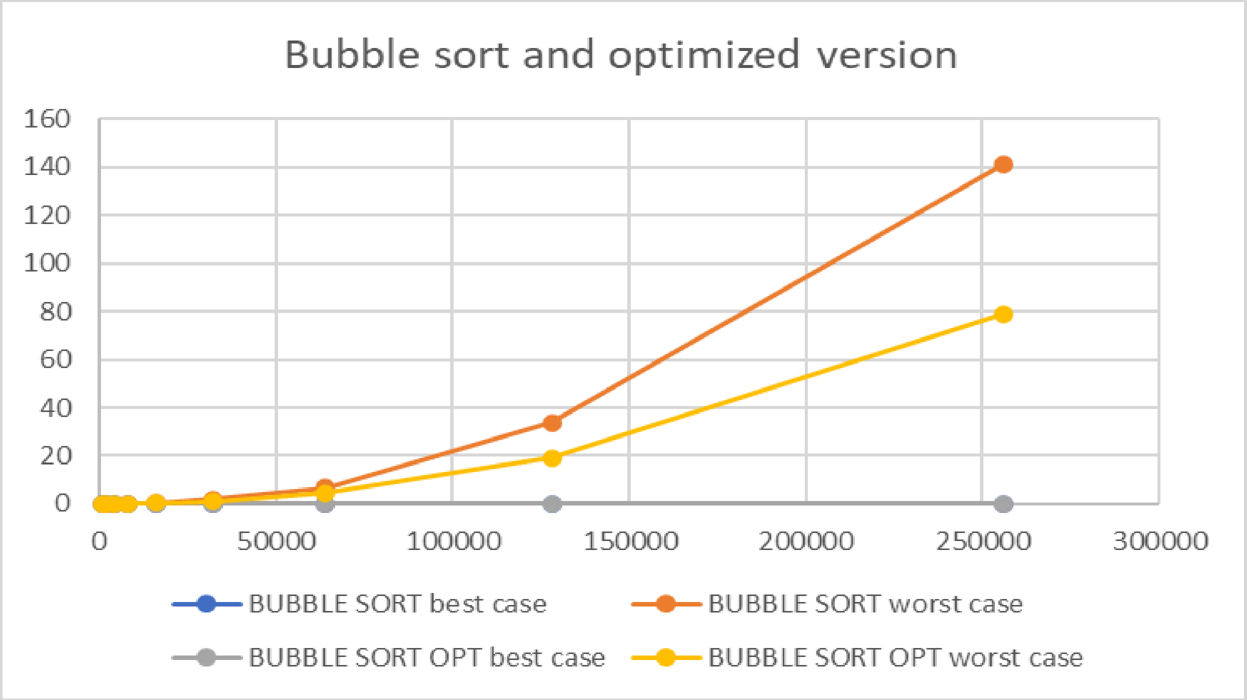
- **Worst Case:**
  For reverse-sorted data, all passes are still required, resulting in **O(n²)** complexity.

4.2.3 Time Execution Graph



4.2.4 Performance Discussion

Although this version reduces unnecessary operations, the asymptotic complexity remains unchanged. In practice, the optimized version performs noticeably better only when the data is nearly sorted as the graph shows below.



4.2.5 Conclusion

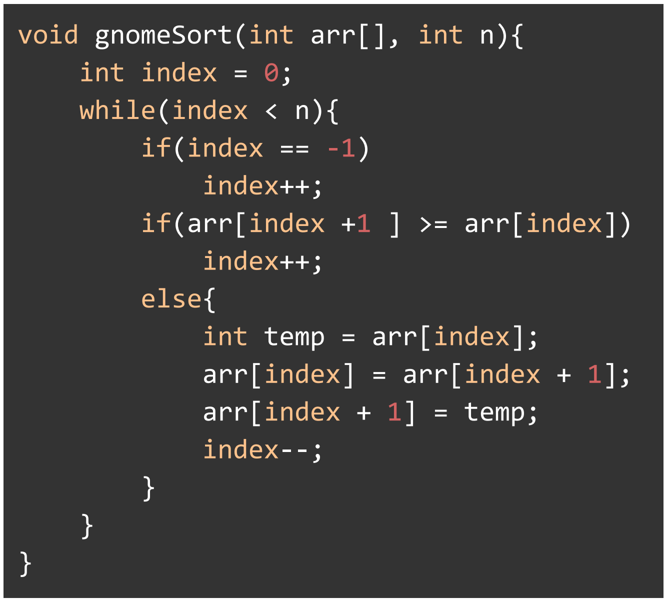
The optimized Bubble Sort significantly improves performance in the worst case by taking advantage that the last elements are always sorted in their correct positions. However, both algorithms retain a quadratic time complexity in the worst case O(n^2). In the best case, both algorithms run in linear time O(n), but the execution time appears constant due to the limited resolution of the timing function and the optimization of the hardware, like usage of cache since accessing the memory linearly is preferred by the CPU and lessens the memory calls.

4.3 Gnome Sort:

4.3.1 Algorithm Description

Gnome Sort compares adjacent elements and swaps them when necessary. After each swap, the algorithm moves one step backward to ensure previous elements remain in order.

4.3.2 Algorithm Implementation



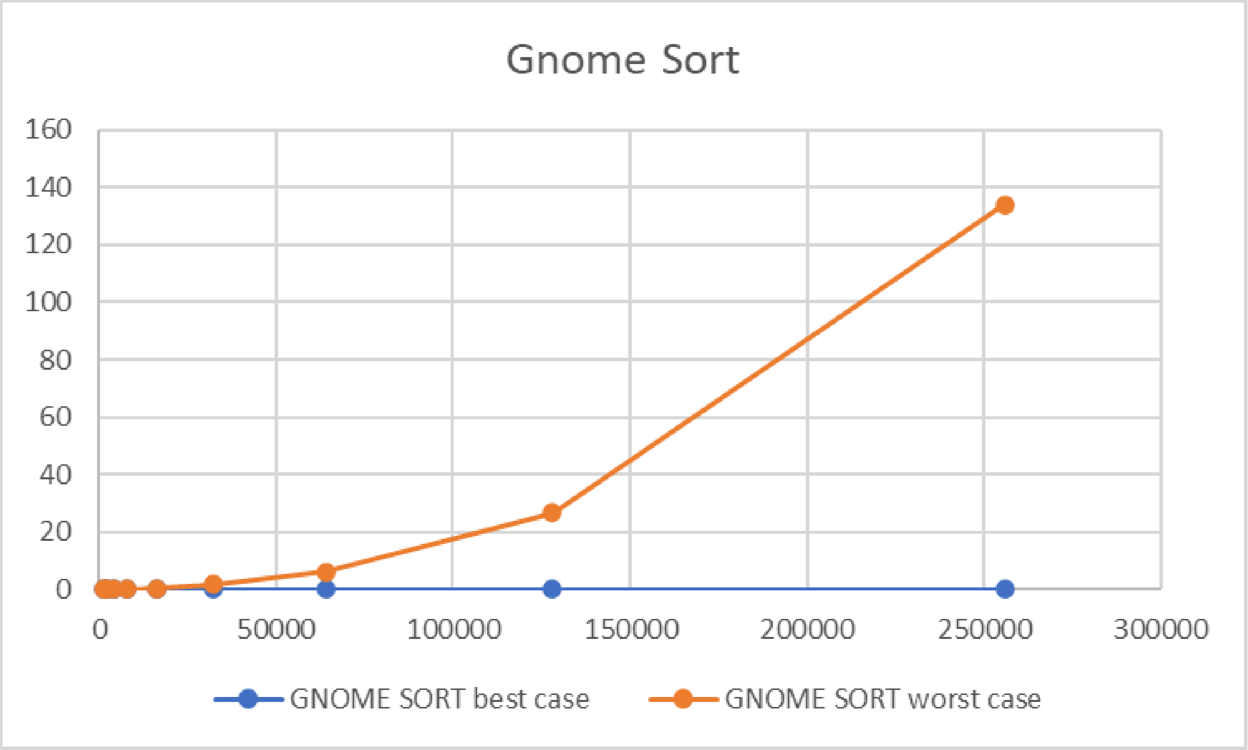
```
void gnomeSort(int arr[], int n){
    int index = 0;
    while(index < n){
        if(index == -1)
            index++;
        if(arr[index +1 ] >= arr[index])
            index++;
        else{
            int temp = arr[index];
            arr[index] = arr[index + 1];
            arr[index + 1] = temp;
            index--;
        }
    }
}
```

4.3.3 Complexity Analysis

- **Best Case:**
  A single forward pass is sufficient, leading to **O(n)** complexity.

- **Worst Case:**
  In a reverse-sorted array, the algorithm repeatedly moves backward, producing a $\sum i = \frac{n(n+1)}{2}$ number of operations. This results in **O(n²)** complexity.

4.3.4 Execution Time Graph



4.3.5 Limitations

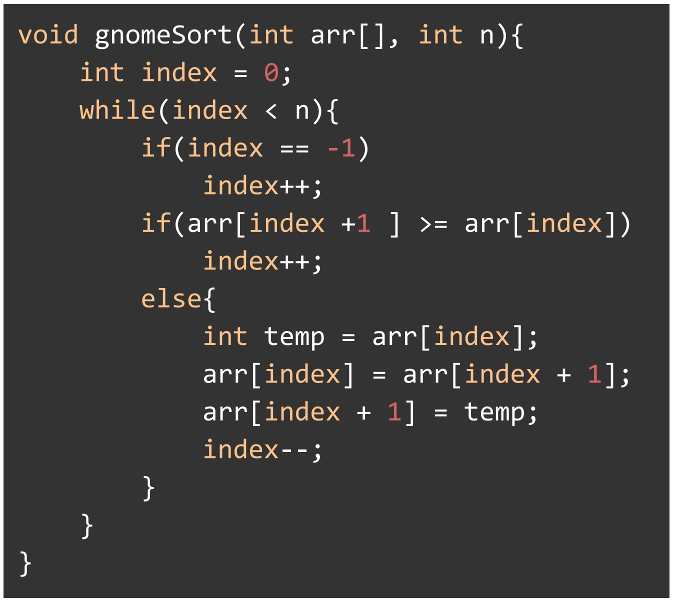
The frequent backward movements make Gnome Sort inefficient in practice, especially for large input sizes.

4.4 Radix Sort:

4.4.1 Algorithm Overview

Radix Sort is a non-comparison-based algorithm that processes numbers digit by digit using a stable intermediate sorting method. Unlike comparison-based algorithms, its performance does not depend on element comparisons.

4.4.2 Algorithm Implementation



```
void gnomeSort(int arr[], int n){
    int index = 0;
    while(index < n){
        if(index == -1)
            index++;
        if(arr[index +1 ] >= arr[index])
            index++;
        else{
            int temp = arr[index];
            arr[index] = arr[index + 1];
            arr[index + 1] = temp;
            index--;
        }
    }
}
```

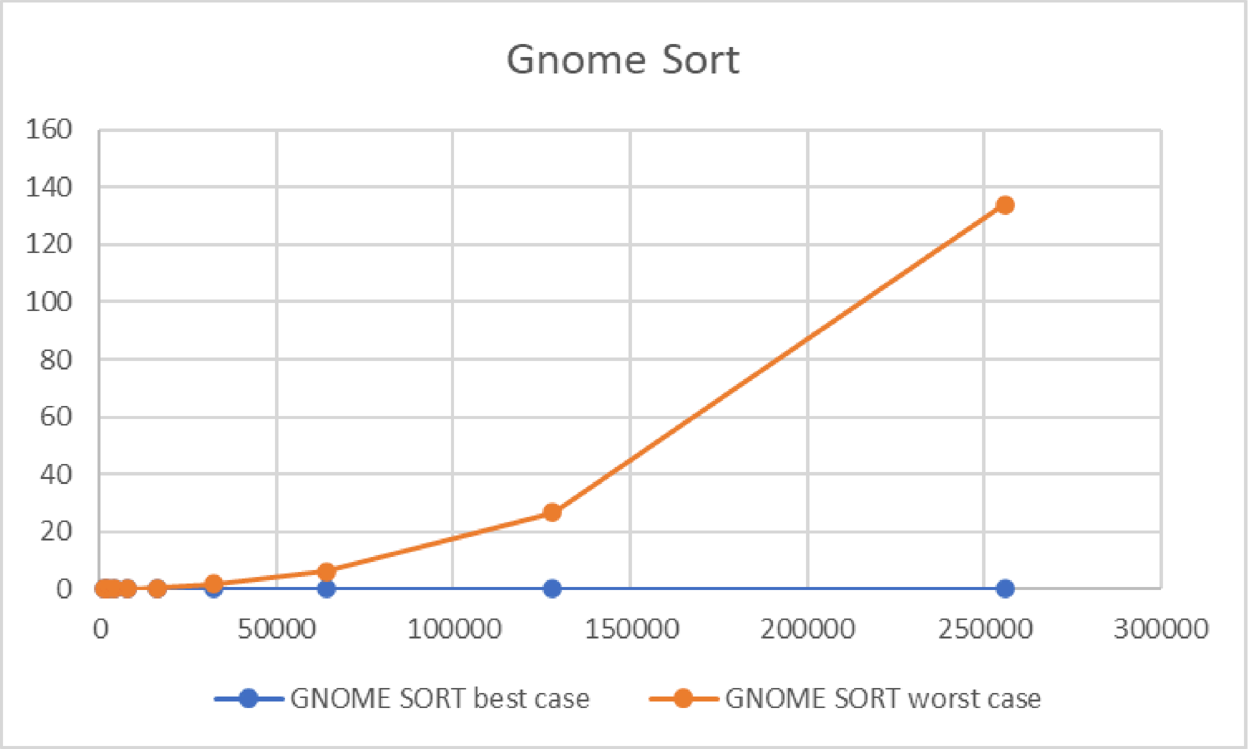
4.4.3 Time Complexity

The complexity of Radix Sort is **O(d · n)**, where **d** represents the number of digits in the largest value.

4.4.4 Best and Worst Case Behavior

Radix Sort has identical best-case and worst-case complexities since all digits must be processed regardless of the input order.

4.4.5 Time Execution Graph



4.4.6 Practical Advantages

Radix Sort is extremely efficient for large datasets with limited numeric ranges but requires additional memory and careful implementation.

4.5 Quick Sort:

4.5.1 Algorithm Concept

Quick Sort follows a divide-and-conquer strategy by selecting a pivot element and partitioning the array into smaller and larger subarrays, which are then sorted recursively.

4.5.2 Algorithm Implementation

```
int partition(int arr[], int low, int high){
    int pivot = arr[low + (high - low)/2];
    int i = low -1;
    for(int j = low; j <= high - 1; j++){
        if(arr[j] < pivot){
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return (i + 1);
}

void quickSort(int arr[], int low, int high){
    if(low >= high || low < 0) return;
        int q = partition(arr, low, high);
        quickSort(arr, low, q-1);
        quickSort(arr, q + 1, high);

}
```

4.5.3 Mathematical Recurrence

The general recurrence relation is:

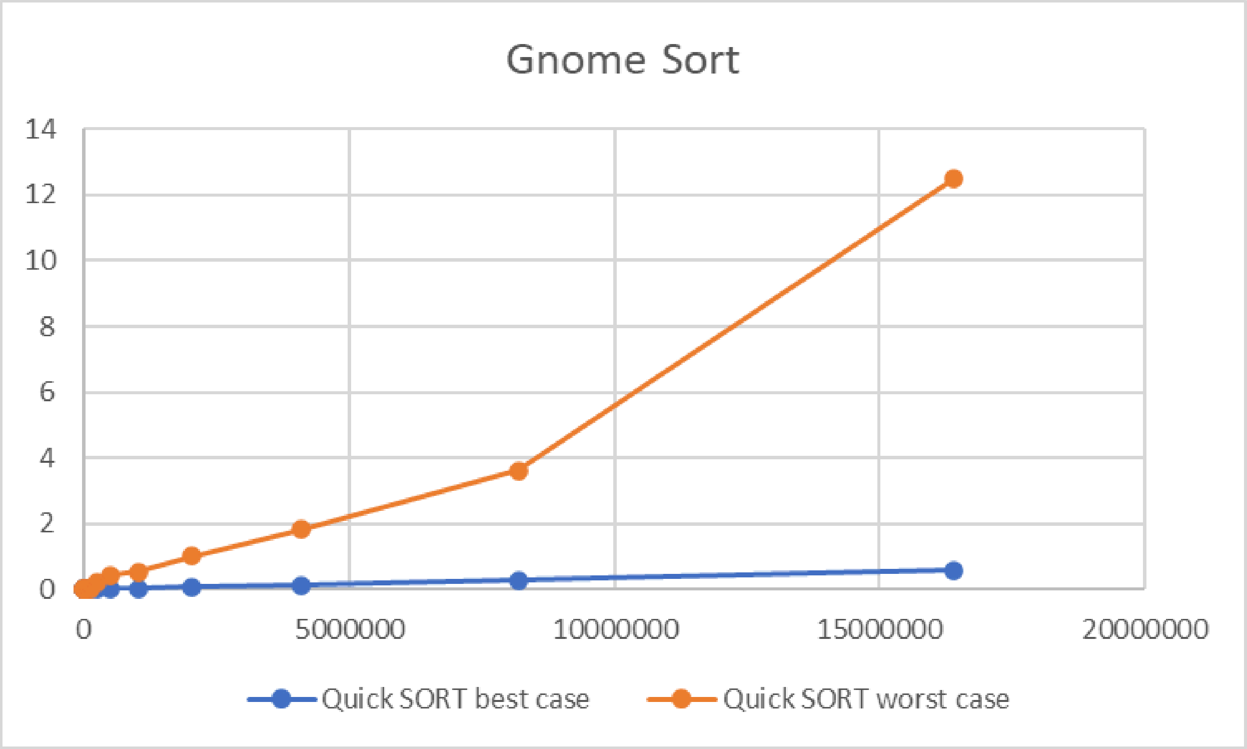
$$T(n) = T(k) + T(n - k - 1) + \Theta(n)$$

4.5.4 Complexity Analysis

- **Best and Average Case:**
  Balanced partitions lead to **Θ(n log n)**.

- **Worst Case:**
  Highly unbalanced partitions result in **Θ(n²)**.

4.5.5 Time Execution Graph



4.5.6 Practical Performance

With an effective pivot strategy, Quick Sort usually performs close to **Θ(n log n)** and is often the fastest sorting algorithm in practice.

4.6 Heap Sort:

4.6.1 Algorithm Description

Heap Sort builds a binary heap and repeatedly extracts the maximum element while maintaining the heap structure.

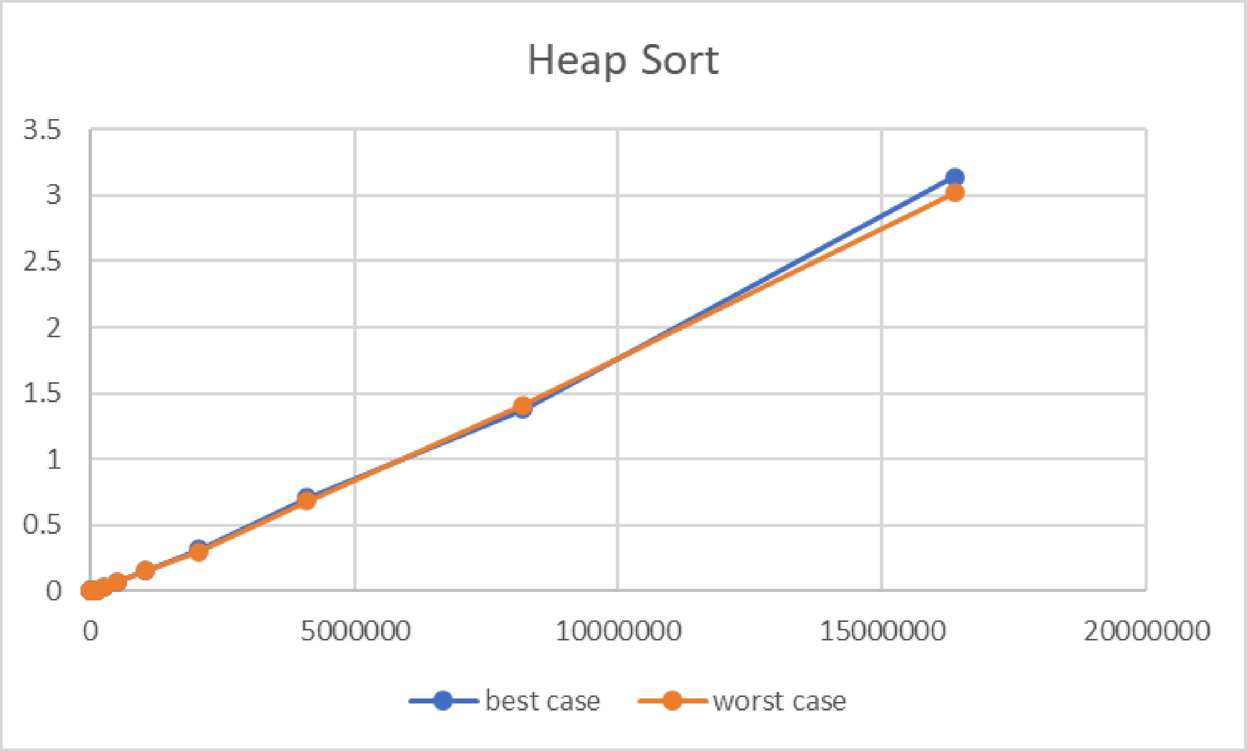
4.6.2 Algorithm Implementation

```
void heapify(int arr[], int n, int i){
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;

    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;
        heapify(arr, n, largest);
    }
}
void HeapSort(int arr[], int n){
    for(int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for(int i = n - 1; i >= 0; i--){
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        heapify(arr, i, 0);
    }

}
```

4.6.3 Complexity Analysis

- Heap construction: **O(n)**
- Heapify operation: **O(log n)**
- Total complexity: **O(n log n)**

4.6.4 Time Execution Graph



4.6.5 Characteristics

Heap Sort offers guaranteed performance and does not rely on input distribution but may suffer from cache inefficiencies.

4.7 Bucket Sort:

4.7.1 Algorithm Overview

Bucket Sort is a distribution-based sorting algorithm that operates by dividing the input data into a set of smaller groups called *buckets*. Each element from the original array is assigned to a bucket based on a predefined mapping function, usually derived from the value range of the data. Once all elements have been distributed, each bucket is sorted individually, and the final sorted array is obtained by concatenating the contents of all buckets in order.

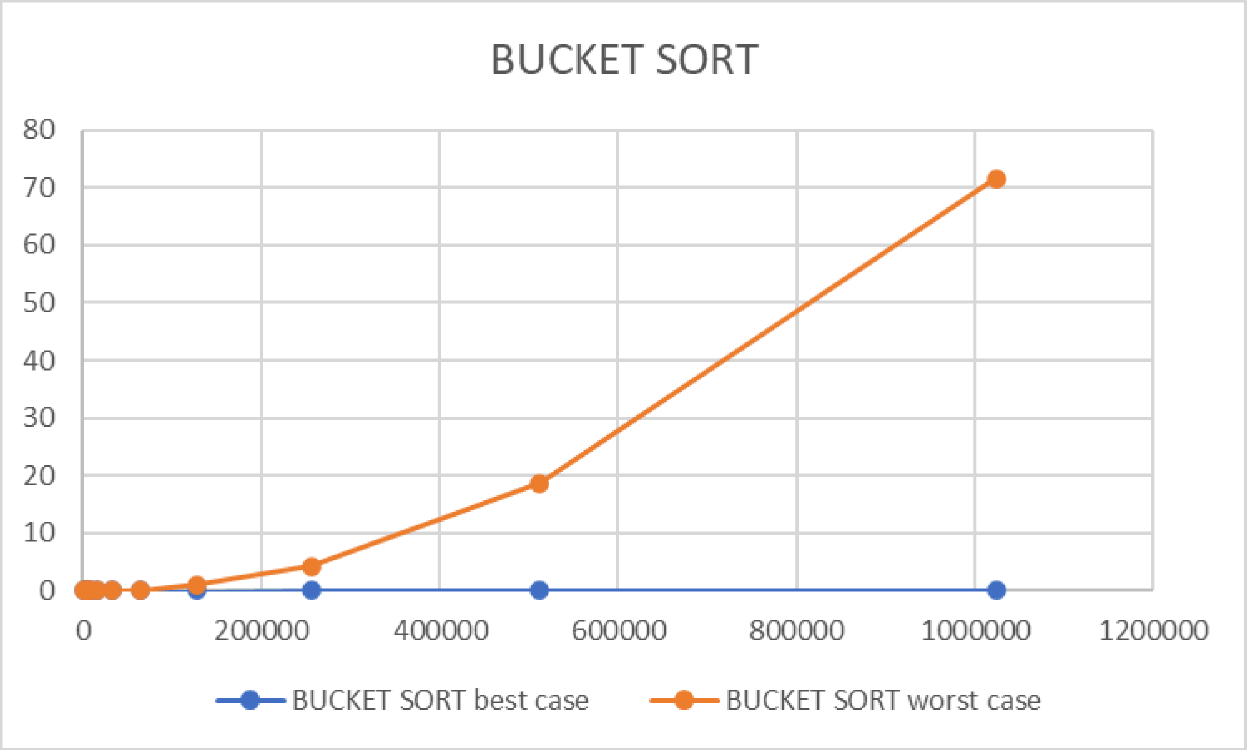
4.7.2 Algorithm Implementation

```
void insertionSort(int* bucket, int size) {
    for (int i = 1; i < size; ++i) {
        int key = bucket[i];
        int j = i - 1;
        while (j >= 0 && bucket[j] > key) {
            bucket[j + 1] = bucket[j];
            j--;
        }
        bucket[j + 1] = key;
    }
}

void BucketSort(int arr[], int n){
    int bucketCount = 10;
    int** buckets = (int**)malloc(bucketCount * sizeof(int*));
    int* bucketSizes = (int*)calloc(bucketCount, sizeof(int));
    int* bucketCapacities = (int*)malloc(bucketCount * sizeof(int));

    for (int i = 0; i < bucketCount; i++) {
        bucketCapacities[i] = n / bucketCount + 1;
        buckets[i] = (int*)malloc(bucketCapacities[i] * sizeof(int));
    }

    for (int i = 0; i < n; i++) {
        int index = arr[i] * bucketCount / (n + 1);
        if (bucketSizes[index] == bucketCapacities[index]) {
            bucketCapacities[index] *= 2;
            buckets[index] = (int*)realloc(buckets[index],
bucketCapacities[index] * sizeof(int));
        }
        buckets[index][bucketSizes[index]++] = arr[i];
    }
    int idx = 0;
    for (int i = 0; i < bucketCount; i++) {
        insertionSort(buckets[i], bucketSizes[i]);
        for (int j = 0; j < bucketSizes[i]; j++) {
            arr[idx++] = buckets[i][j];
        }
        free(buckets[i]);
    }

    free(buckets);
    free(bucketSizes);
    free(bucketCapacities);

}
```

4.7.3 Complexity Analysis

- **Best Case:** In the ideal situation, the input elements are uniformly distributed across all buckets. Each bucket then contains only a small number of elements. Since insertion sort is used to sort individual buckets, it runs in linear time when the bucket contents are already nearly sorted or very small. In this case, the overall time complexity approaches **O(n)**.
- **Worst Case:** In the worst-case scenario, most or all elements are placed into a single bucket. This can occur when the input distribution is highly skewed or when the bucket mapping function is poorly chosen. The algorithm then effectively degenerates into sorting a single or a limited number of buckets using insertion sort, which has a worst-case complexity of **O(n^2)**. Consequently, the overall complexity of Bucket Sort becomes quadratic.

4.7.5 Time Complexity Graph



4.7.5 Conclusion

Bucket Sort is a powerful sorting technique when its assumptions are satisfied. It can outperform traditional comparison-based algorithms in favorable scenarios, but it also carries the risk of significant performance degradation in unfavorable cases. Therefore, Bucket Sort is best applied when the nature of the input data is well understood and controlled.

5. General Discussion

This project demonstrates that theoretical complexity alone is not always sufficient to predict real-world performance. Factors such as memory access patterns, compiler optimizations, and hardware architecture can significantly influence execution time.

5.1 Time Complexity Comparison

In the table below we can observe the general complexity of the algorithms implemented in this project:

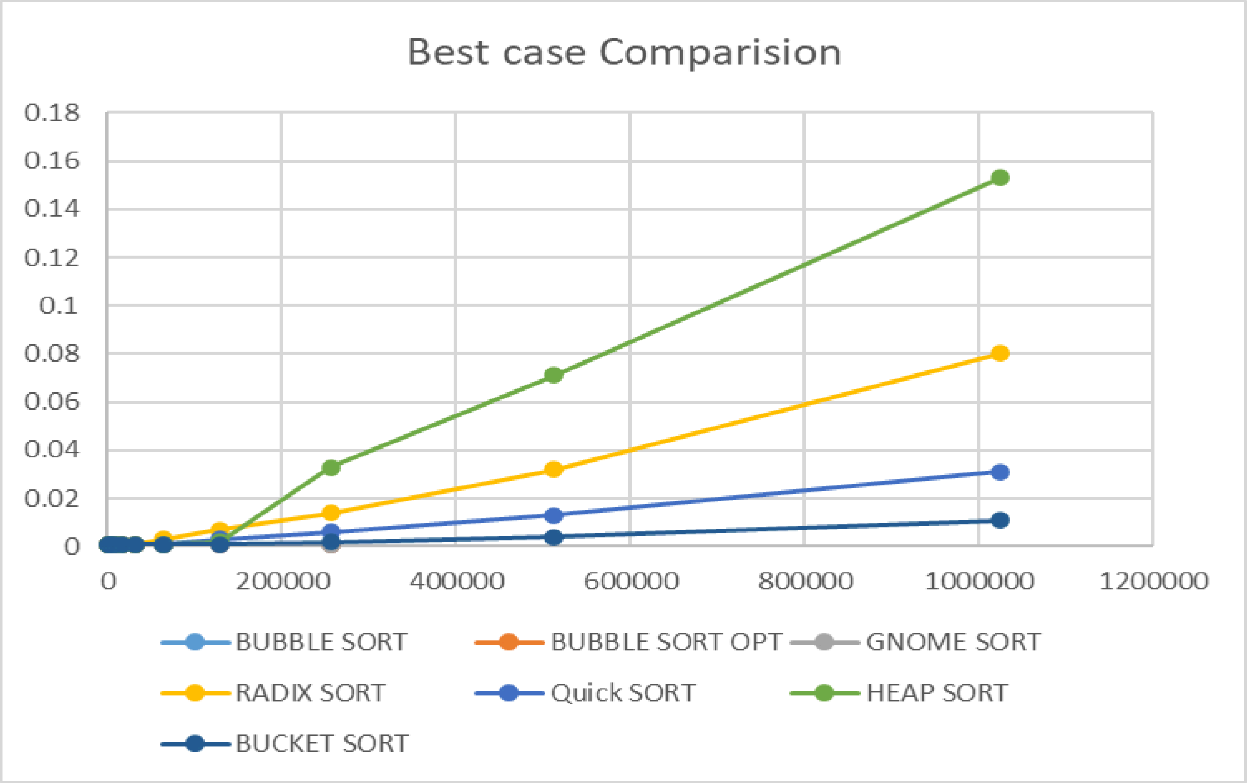
Algorithm	Best Case	Worst Case
Bubble Sort	$O(n)$	$O(n^2)$
Optimized Bubble Sort	$O(n)$	$O(n^2)$
Gnome Sort	$O(n)$	$O(n^2)$
Radix Sort	$O(d \cdot n)$	$O(d \cdot n)$
Quick Sort	$O(n \log n)$	$O(n^2)$
Heap Sort	$O(n \log n)$	$O(n \log n)$
Bucket Sort	$O(n)$	$O(n^2)$

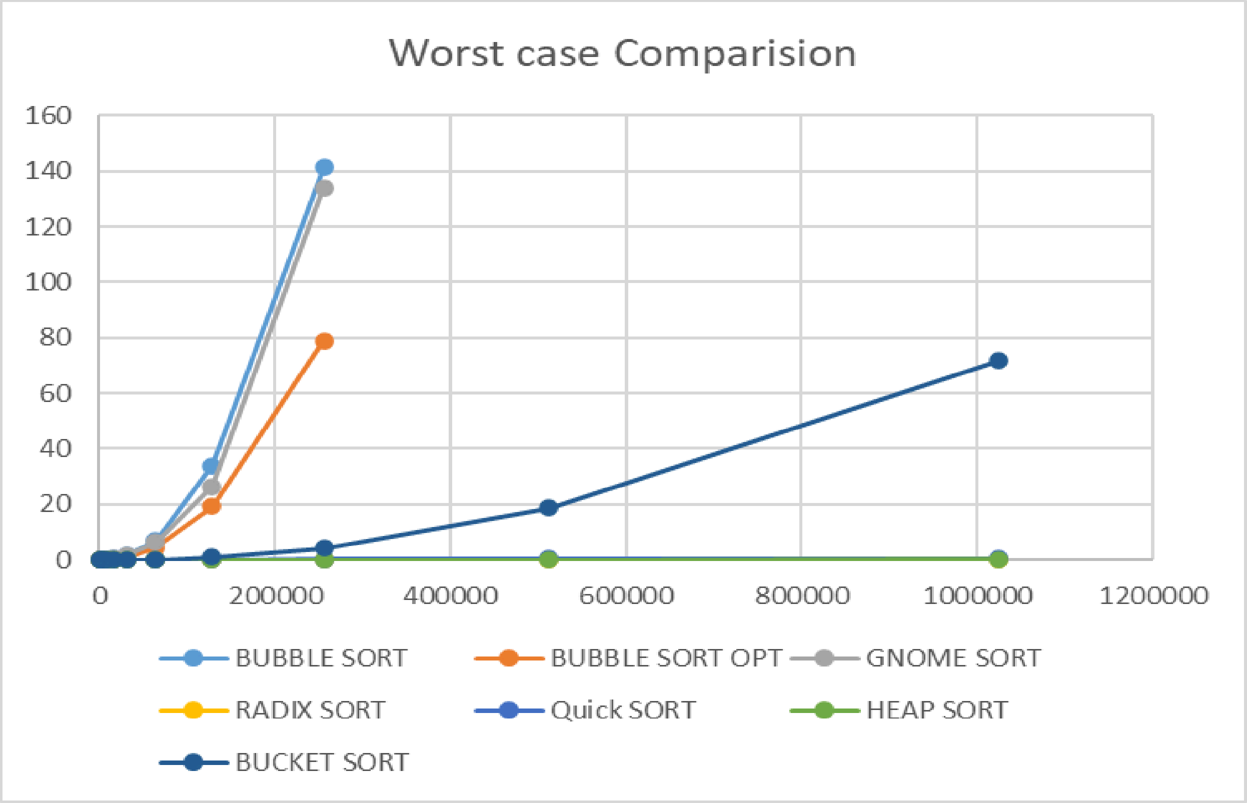
5.2 Practical Performance Characteristics

This table highlights qualitative performance aspects observed during experiments.

Algorithm	Stability	Extra Memory	Cache Friendly	Practical Speed
Bubble Sort	Yes	Low	High	Very Slow
Gnome Sort	Yes	Low	Medium	Slow
Quick Sort	No	Low	High	Very Fast
Heap Sort	No	Low	Medium	Fast
Radix Sort	Yes	High	Medium	Very Fast
Bucket Sort	Yes	Medium	Medium	Data-Dependent

5.3 Time Execution Graph of different algorithms





6. General Conclusion

The results clearly demonstrate that no single sorting algorithm is optimal for all situations; instead, performance depends heavily on input size, data distribution, and implementation details.

Simple algorithms such as Bubble Sort, Optimized Bubble Sort, and Gnome Sort are useful for very small datasets. However, their quadratic time complexity in the worst case severely limits their scalability. Even with optimizations, these algorithms quickly become impractical as the input size grows.

More advanced comparison-based algorithms, such as Quick Sort and Heap Sort, provide significantly better performance for large datasets. Quick Sort offers excellent average case behavior and often outperforms other algorithms in practice due to efficient memory access patterns and low constant factors. However, its worst-case quadratic complexity highlights the importance of pivot selection strategies. Heap Sort, while slightly slower in practice, guarantees a time complexity of O(n log n) regardless of input order, making it a reliable choice when worst-case performance must be controlled.

Non-comparison-based algorithms, such as Radix Sort and Bucket Sort, show that linear-time sorting is achievable under specific assumptions. Radix Sort performs consistently in both best and worst cases and is highly effective when sorting integers with a limited number of digits. Bucket Sort can achieve linear time complexity when the input data is uniformly distributed, but its performance degrades rapidly when this assumption does not hold. These algorithms trade generality for speed and require careful consideration of input characteristics.

An important outcome of this project is the observation that theoretical improvements do not always translate directly into faster execution times. Factors such as memory access patterns and cache efficiency affect real-world performance. As a result, algorithms with fewer theoretical operations may still run slower than simpler alternatives.

The experimental results confirm complexity analysis while also highlighting the importance of practical considerations. Algorithm selection should therefore be guided not only by asymptotic complexity but also by the nature of the data, available memory, and performance requirements of the application.