CSE 311 Algorithms Analysis
Empirical Study

Mahmut Selim YILBAŞ 20190702119

This report is prepared as part of the CSE 311 Algorithms Analysis course's Empirical Study assignment. The primary objective of this study is to evaluate the performance of various sorting algorithms by implementing and analyzing their execution times across different input sizes. The sorting algorithms examined in this study include Bubble Sort, Selection Sort, Quick Sort, Merge Sort, Improved Bubble Sort, and Improved Quick Sort.

To conduct this analysis, Python was chosen as the programming language due to its simplicity and powerful libraries. The development environment used for this project was Visual Studio Code (VSCode), which provided an efficient platform for coding, testing, and debugging. The performance of each algorithm was measured by running them on randomly generated datasets of varying sizes (100, 1000, 10000, 100000 elements) and recording the time taken for each run.

The results of these experiments were systematically recorded, averaged over multiple runs, and then visualized using Excel to create informative line charts. These charts offer a clear comparison of the efficiency of the different algorithms under study, highlighting their strengths and weaknesses in terms of time complexity and practical performance. The insights gained from this empirical study provide valuable knowledge on the behavior of these algorithms in real-world scenarios.

Bubble Sort

Bubble Sort repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The process is repeated until the list is sorted. It has a time complexity of O(n^2) in the worst and average case. It is primary practical usage is in educational contexts for teaching basic sorting concepts. It is not suitable for performance-critical applications.

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

Selection Sort

Selection Sort divides the list into two parts: the sorted part at the beginning and the unsorted part at the end. It repeatedly selects the smallest (or largest, depending on the order) element from the unsorted part and swaps it with the first unsorted element. The time complexity is O(n^2) in all cases. It is practical usage is limited to small datasets or educational purposes. It may be used in scenarios where memory writes are costly, as it performs fewer swaps than BubbleSort.

```python
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

Quick Sort

Quick Sort is a divide-and-conquer algorithm that selects a 'pivot' element from the array and partitions the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. It then recursively sorts the sub-arrays. Its average time complexity is O(n log n), but it can be O(n^2) in the worst case. QuickSort is suitable for a wide range of applications, including systems programming and large-scale data processing.

```python
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
```

Merge Sort

Merge Sort is another divide-and-conquer algorithm. It divides the array into two halves, recursively sorts them, and then merges the two sorted halves. It has a consistent time complexity of O(n log n) for all cases, making it reliable for large datasets. It is particularly useful for sorting linked lists and external sorting (sorting data that doesn't fit into memory). However, MergeSort requires additional memory for merging, which can be a drawback in memory-constrained environments. It is well-suited for applications where stable sorting (preserving the relative order of equal elements) is required.

```python
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)
        merge_sort(R)

        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

Improved Bubble Sort

Improved Bubble Sort is an optimization of Bubble Sort. It includes a flag that checks if any swaps were made during the iteration. If no swaps were made, the list is already sorted, and the algorithm can terminate early. This reduces the number of passes through the list. ImprovedBubbleSort enhances the basic BubbleSort by adding an early exit mechanism if the list becomes sorted before completing all passes. This optimization improves performance for nearly sorted datasets but does not significantly impact the worst-case time complexity of O(n^2). It can be used in educational contexts or for small datasets that are expected to be nearly sorted.

```python
def improved_bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if not swapped:
            break
```

Improved Quick Sort

Improved Quick Sort uses a hybrid approach. For small arrays (less than 20 elements), it uses Selection Sort, which is more efficient for small datasets. For larger arrays, it uses the traditional Quick Sort algorithm. This can help to optimize performance, leveraging the strengths of both algorithms. It maintains an average-case time complexity of O(n log n) and is suitable for a broad range of applications, including performance-critical systems and large-scale data processing.

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

def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] < pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i+1], arr[high] = arr[high], arr[i+1]
    return i+1

def improved_quick_sort(arr):
    if len(arr) < 20:
        selection_sort(arr)
    else:
        quick_sort_helper(arr, 0, len(arr) - 1)
```

The Outputs of Each Algorithm (Execution Time)

Input Size	BubbleSort	SelectionSort	QuickSort	MergeSort	ImprovedBubl	ImprovedQuickSort	
100	0,000186	0,000088	0,000078	0,000073	0,00019	0,000034	
1000	0,023872	0,010322	0,000765	0,000857	0,022926	0,000545	1s run
10000	2,622928	1,053325	0,007953	0,01095	2,602136	0,007334	
100000	262,20189	306,06036	0,072654	0,13829	272,85953	0,089052	

Input Size	BubbleSort	SelectionSort	QuickSort	MergeSort	ImprovedBubl	ImprovedQuickSort
100	0,000184	0,000092	0,00007	0,000061	0,000192	0,00003
1000	0,023489	0,010371	0,000736	0,000804	0,023034	0,000538
10000	2,647879	1,057422	0,007818	0,010959	2,606662	0,007339
100000	262,40426	105,67247	0,072035	0,138437	275,53243	0,088996

2nd run

Input Size	BubbleSort	SelectionSort	QuickSort	MergeSort	ImprovedBubl	ImprovedQuickSort
100	0,000184	0,000087	0,000071	0,000061	0,000212	0,000035
1000	0,022548	0,010467	0,000755	0,000806	0,023018	0,000553
10000	2,575431	1,054517	0,007825	0,010889	2,630054	0,007336
100000	265,29354	106,54342	0,071765	0,13906	278,2795	0,088842

3rd run

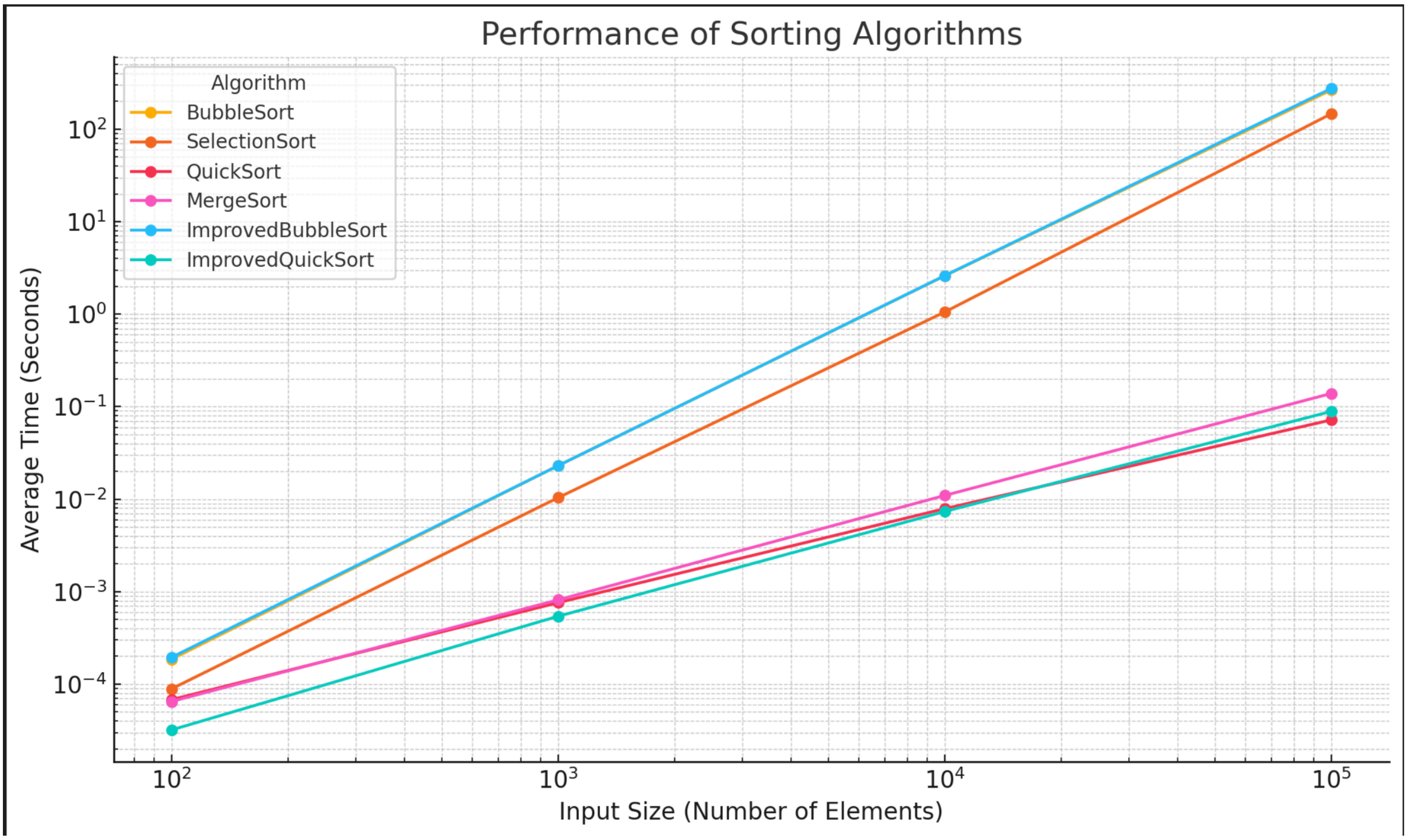
Input Size	BubbleSort	SelectionSort	QuickSort	MergeSort	ImprovedBubl	ImprovedQuickSort
100	0,00019	0,000086	0,000062	0,000068	0,000188	0,00003
1000	0,022688	0,010366	0,000725	0,000808	0,023105	0,000533
10000	2,577956	1,056542	0,00786	0,011085	2,606986	0,007315
100000	266,24013	107,335	0,071628	0,139082	276,34176	0,088774

4th run

Input Size	BubbleSort	SelectionSort	QuickSort	MergeSort	ImprovedBubl	ImprovedQuickSort
100	0,000184	0,000092	0,000058	0,000061	0,000192	0,000029
1000	0,02251	0,010378	0,000817	0,0008	0,023063	0,000539
10000	2,578971	1,054718	0,007982	0,010936	2,60675	0,007458
100000	270,34139	108,44779	0,071471	0,139921	275,65652	0,088913

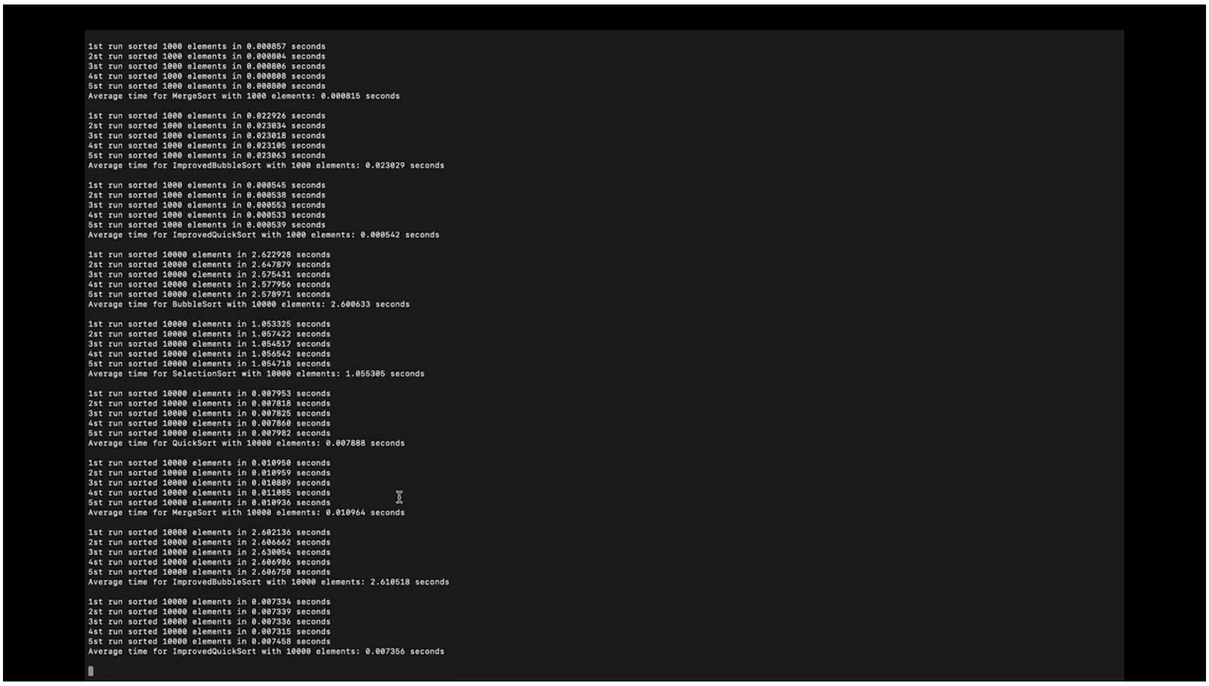
5th run

Input Size	BubbleSort	SelectionSort	QuickSort	MergeSort	ImprovedBubbleSort	ImprovedQuickSort	
100	0,000186	0,000089	0,000068	0,000065	0,000195	0,000032	
1000	0,023021	0,010381	0,00076	0,000815	0,023029	0,000542	Average of
10000	2,600633	1,055305	0,007888	0,010964	2,610518	0,007356	5 runs
100000	265,296244	146,811805	0,071911	0,138958	275,733947	0,088915	



What is the biggest problem size you can run in two seconds?

- BubbleSort **1000** elements
- SelectionSort **1000** elements
- QuickSort **100000** elements
- MergeSort **100000** elements
- ImprovedBubbleSort **10000** elements
- ImprovedQuickSort **100000** elements



www.youtube.com/watch?v=pmqzkHfuml0