**Assignment-1**

**Design and Analysis of Algorithms**

**COMP 359**

**Date of Submission: 20 September 2024**

**Submitted to: Prof. Russell Campbell**

**Amarnoor Kaur, Karan Sharma, Vibhu Dixshit**

# Logical Reasoning of the Code:

## 1. Bubble Sort (O(n²))

Bubble Sort is a simple comparison-based sorting algorithm. It works by repeatedly swapping adjacent elements if they are in the wrong order. This process continues until the list is sorted. Its time complexity is O(n²), which makes it inefficient for large datasets.

**Pseudo code:**

function BubbleSort(arr):

n = length of arr

for i from 0 to n-1:

for j from 0 to n-i-2:

if arr[j] > arr[j+1]:

swap arr[j] and arr[j+1]

Here, "n" represents the length of the array. The outer loop runs "n" times, while the inner loop compares adjacent elements and swaps them if needed. The inner loop reduces in length after each iteration, since the largest element "bubbles" to the end after each pass.

**Time Complexity:**

**1. Worst-Case Time Complexity: O(n²)**

In the worst case, the array is in reverse order. Bubble Sort will compare and swap every adjacent pair for each element.

* The outer loop runs n times.
* The inner loop runs n-1 times in the first iteration, n-2 times in the second iteration, and so on.
* Therefore, the number of comparisons is n(n-1) /2 which results in a time complexity of O(n²).

**2. Best-Case Time Complexity: O(n)**

In the best case, the array is already sorted. Bubble Sort can be optimized by adding a flag to stop the algorithm when no swaps are made during a pass.

* The outer loop will still run n times.
* However, after one pass without any swaps, the algorithm will stop early, giving a time complexity of O(n).

**3. Average-Case Time Complexity: O(n²)**

In the average case, the array is in a random order. The algorithm will need to compare and swap elements in a similar manner as in the worst case, leading to an average time complexity of O(n²).

## 2. Merge Sort (O(n log n))

Merge Sort is a divide-and-conquer algorithm. It works on the following principals

* Divide: Split the array into two halves.
* Conquer: Recursively sort each half.
* Combine: Merge the two sorted halves back together to produce a single sorted array.

**Pseduo Code:**

Procedure MERGE\_SORT(arr):

if length of arr > 1:

mid = length of arr // 2

L = subarray of arr from index 0 to mid-1

R = subarray of arr from index mid to end

MERGE\_SORT(L)

MERGE\_SORT(R)

i = 0

j = 0

k = 0

while i < length of L and j < length of R:

if L[i] < R[j]:

arr[k] = L[i]

i = i + 1

else:

arr[k] = R[j]

j = j + 1

k = k + 1

while i < length of L:

arr[k] = L[i]

i = i + 1

k = k + 1

while j < length of R:

arr[k] = R[j]

j = j + 1

k = k + 1

In this code, the array is first divided into two halves (L and R). Both halves are then recursively sorted using the same merge\_sort function. After sorting, the two halves are merged together by comparing elements and adding them to the original array in the correct order. The merging process involves two pointers (i and j) to compare elements from L and R, respectively.

Merge Sort has a time complexity of O(n log n) because the array is divided in half log(n) times, and merging takes linear time in each division. This makes Merge Sort more efficient for larger datasets than Bubble Sort.

## 3. Counting Sort:

Counting Sort is a non-comparative sorting algorithm that is efficient for sorting integers or objects that can be mapped to integer keys. The algorithm works by counting the number of occurrences of each unique element in the array and then using this information to place each element in its correct position in the sorted array. The time complexity of Counting Sort is O(n + k), where n is the number of elements in the array and k is the range of input values.

**Pseudo Code:**

function CountingSort(arr):

max\_val = max(arr)

min\_val = min(arr)

range\_of\_elements = max\_val - min\_val + 1

count = [0] \* range\_of\_elements

for i from 0 to length of arr:

count[arr[i] - min\_val] += 1

for i from 1 to length of count:

count[i] += count[i - 1]

output = [0] \* length of arr

for i from length of arr - 1 to 0:

output[count[arr[i] - min\_val] - 1] = arr[i]

count[arr[i] - min\_val] -= 1

return output

**Time Complexity:**

**1. Worst-Case Time Complexity: O(n + k)**

In the worst case, Counting Sort needs to handle an array where the range of input values (k) is large.

* Input array size: n
* Range of elements: k

The algorithm takes:

* O(n) time to populate the count array with the occurrences of each element.
* O(k) time to initialize and modify the count array.
* O(n) time to build the sorted output array.

Therefore, the worst-case time complexity is O(n + k).

**2. Best-Case Time Complexity: O(n + k)**

In the best case, even when the input is already sorted, Counting Sort still has to count each element and build the output array.

* The operations remain the same regardless of whether the array is sorted or unsorted.

Thus, the best-case time complexity is also O(n + k).

**3. Average-Case Time Complexity: O(n + k)**

The average-case time complexity occurs when the input array is in random order. In this case:

* The count array still needs to be populated based on the occurrences of each element.
* The time to create and use the count array remains proportional to O(n + k), as the range of input values affects the time required to process the array.

Therefore, the average-case time complexity is O(n + k).

## 4. Selection Sort

Selection sort is a comparison-based sorting algorithm. It works by repeatedly selecting the smallest element from the unsorted portion of the list and swapping it with the first unsorted element. The process continues until the entire list is sorted. Its time complexity is O(n^2), making it inefficient for large datasets.

**Pseudo Code:**

function SelectionSort(arr):

n = length of arr

for i from 0 to n-1:

min\_index = i

for j from i+1 to n:

if arr[j] < arr[min\_index]:

min\_index = j

swap arr[i] and arr[min\_index]

Here, "n" is the length of the array. The outer loop iterates through the list, and the inner loop finds the smallest element in the unsorted portion. This element is then swapped with the first unsorted element.

**Time Complexity:**

**Worst-Case Time Complexity: O(n^2)**

* In the worst case, Selection Sort will still perform n(n−1)/2 comparisons and swaps for each element.
* The outer loop runs n times, and the inner loop scans the unsorted portion each time, making it inefficient for large datasets.

**Best-Case Time Complexity: O(n^2)**

* Even if the array is already sorted, Selection Sort will still perform the same number of comparisons. Unlike Bubble Sort, Selection Sort doesn't optimize for already sorted data.
* As a result, the best-case time complexity remains O(n^2)

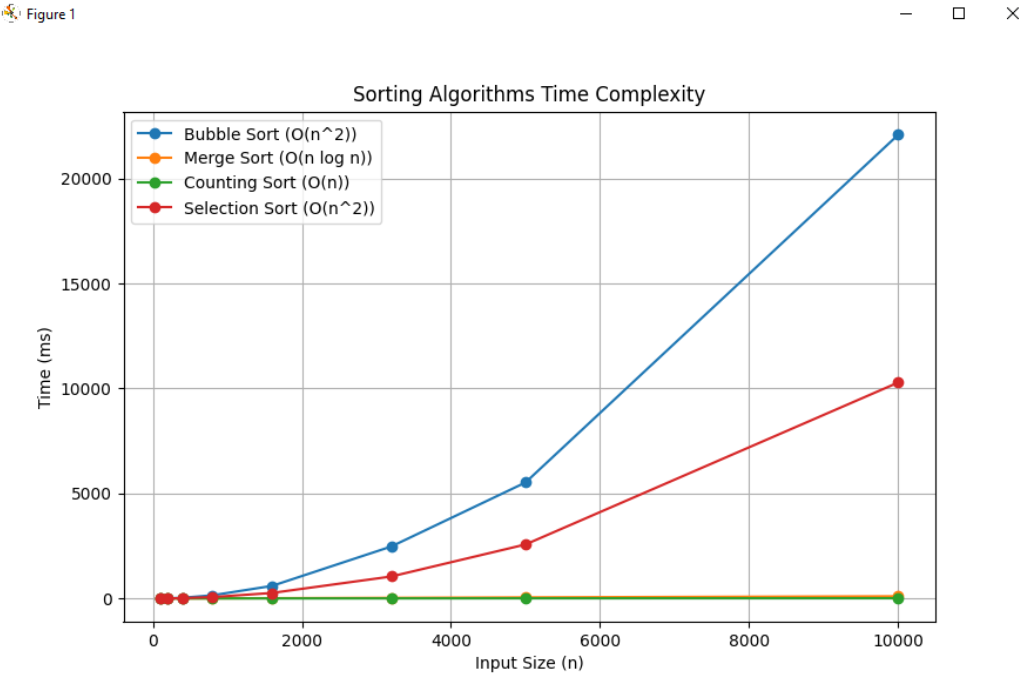
**Average-Case Time Complexity: O(n^2)**

* In the average case, the array is in a random order. The algorithm will still perform a similar number of comparisons and swaps as in the worst case, leading to an average time complexity of O(n^2)

**Results Visualization from the graph:**

The results are plotted using Matplotlib to visually compare the runtimes of each algorithm across different input sizes. The x-axis represents the array size, and the y-axis represents the time taken (in ms).

* Bubble Sort: Displays a steep increase in time as the array size grows due to its O(n²) complexity.
* Merge Sort: This shows a much more gradual increase compared to Bubble Sort due to its O(n log n) complexity.
* Counting Sort: Demonstrates almost linear performance, growing at a slower rate as the input size increases, especially since its time complexity is O(n).
* Selection Sort: Similar to Bubble Sort, shows a steep rise in time with larger array sizes, as it also has a O(n^2) complexity.



**Expected Results:**

* **Bubble Sort** was expected to be slow, especially for larger input sizes. Its time complexity, O(n²), means that the time taken increases dramatically with larger datasets.
* **Merge Sort** will perform significantly better, with an O(n log n) complexity, making it much faster for larger input sizes.
* **Counting Sort** should have the best performance as it operates in O(n) time for a suitable range of input values.
* **Selection Sort** is expected to show a similar slow performance as Bubble Sort due to its O(n^2) complexity, making it inefficient for large input sizes.

The expected results for the sorting algorithms suggest that Bubble Sort and Selection Sort, both with O(n^2) complexity, should take significantly longer on larger datasets. Meanwhile, Merge Sort O(n log n) and Counting Sort O(n) should perform faster. However, in actual results, the performance gap may not be as wide for smaller input sizes due to overhead from recursion in Merge Sort or the setup of auxiliary structures in Counting Sort. These factors can cause the actual runtimes to deviate from the expected theoretical complexity, especially for moderately sized arrays, where Counting Sort might not always outperform Merge Sort or even Selection Sort due to its overhead.

**Observations:**

**Bubble Sort**

* Inefficient for Large Datasets: Time complexity of O(n^2), leading to poor performance with many elements.
* HighNumber of Operations: Performs many redundant comparisons and swaps.
* Not Adaptive: Does not optimize for nearly sorted lists.
* Poor Performance Compared to Other Sorts: More advanced algorithms generally perform better.

**Merge Sort**

* Space Complexity: Requires O(n) additional space for merging, which can be a drawback.
* Merge Overhead: The merging process can be time-consuming.
* Not In-Place: Needs extra space, unlike in-place algorithms.
* Complex Implementation: More complex to implement compared to simpler algorithms.
* Unnecessary Overhead for Small Lists: Can be less efficient than simpler algorithms for small datasets.

**Counting Sort**

* Limited Applicability:Only effective for small integer ranges; impractical for large ranges.
* High Space Complexity: Requires O(k) extra space, where k is the range of input values, leading to high memory usage.
* Not Suitable for Large Ranges: Becomes inefficient when sorting data with large integer ranges.
* Limited to Integer-Like Data: Not a comparison-based sort, making it less flexible for sorting complex objects.
* Inefficient for Sparse Data: Wastes space when the data is sparse across a large range.

**Selection Sort**

* Inefficient for Large Datasets: The time complexity of O(n^2) makes it slow for large lists.
* High Comparison Count: Performs a large number of comparisons.
* Unchanged Performance Regardless of Order: Does not adapt to partially sorted data.
* Not Stable: May change the relative order of equal elements.
* Poor Performance Compared to Other Sorts: Generally outperformed by more advanced sorting algorithms.

References:

GeeksforGeeks. (2016, September 24). *Selection Sort | GeeksforGeeks* [Video]. YouTube. <https://www.youtube.com/watch?v=xWBP4lzkoyM>

GeeksforGeeks. (2024a, August 6). *Bubble Sort Algorithm*. GeeksforGeeks. <https://www.geeksforgeeks.org/bubble-sort-algorithm/>

GeeksforGeeks. (2024b, August 6). *Counting Sort data Structures and Algorithms tutorials*. GeeksforGeeks. <https://www.geeksforgeeks.org/counting-sort/>

GeeksforGeeks. (2024c, September 5). *MatPlotLib Tutorial*. GeeksforGeeks. <https://www.geeksforgeeks.org/matplotlib-tutorial/>

GeeksforGeeks. (2024d, September 5). *MatPlotLib Tutorial*. GeeksforGeeks. <https://www.geeksforgeeks.org/matplotlib-tutorial/>