**Avijit Saha**

**Student ID #005023281**

**Algorithms and Data Structures (MSCS-532-B01)**

**Assignment 2**

**November 1, 2024**

**Asymptotic Analysis and Recurrence Relations**

**Divide and Conquer:** Divide and Conquer Algorithm is a problem-solving technique used to solve problems by dividing the main problem into subproblems, solving them individually, and then merging them to find a solution to the original problem. It involves breaking a more significant problem into smaller subproblems, solving them independently, and then combining their solutions to solve the original problem. The basic idea is to recursively divide the problem into smaller subproblems until they become simple enough to be solved directly. Quick sort and merge sort are both sorting techniques that use the divide and conquer algorithm.

**Quick Sort**

**Problems:** Quick Sort is a sorting algorithm based on Divide and Conquer. It picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array. It sorts a list of items in ascending or descending order.

**Key Steps:** The critical steps of quick sort are pivot selection, partitioning array, and recursive call.

* **Pivot Selection:** First, a pivot needs to be selected. The pivot can vary; it can be the first element of the array or the last element of the array, or it may be the median.
* **Partition the Array:** After the pivot selection, rearrange elements so that the element less than the pivot comes before it and the greater than it comes after the pivot.
* **Recursion:** Then, recursively apply the same process to the two partitioned sub-arrays.

**Time Complexity:** Quick-sort time complexity depends on the pivot selection.

* **Best Case:** It occurs when the pivot element divides the array equally, and the time complexity is (Ω(n log n)).
* **Average Case:** The average case happens when the pivot element divides the array into two parts but is not necessarily equal. The time complexity is (θ(n log n)).
* **Worst Case:** It happens when the most minor or significant element is consistently chosen as a pivot. The time complexity is (O(n2))

**Recurrence Relation:** It typically reflects the runtime of recursive algorithms. The recurrence relation for quick sort can be expressed as T(n)=T(k)+T(n−k−1)+Θ(n), where n is the total number of elements in an array and k is the number of elements in the subarray. This represents the two recursive element calls plus the linear time Θ(n) for partitioning.

* **Substitution Method:** Assuming T(n)=Θ(n log n)T(n)=Θ(n log n), we expand the recurrence relation through substitution. By replacing T(n)T(n) at each level, we observe that the recursive depth is roughly log n, and each level costs O(n), validating T(n)=Θ(n log n)T(n)=Θ(n log n).
* **Recursion-Tree Method:** Visualizing each level of recursion as nodes in a tree, with partition operations at each level costing O(n)O(n), results in log n levels and O(n log n) complexity.
* **Master Method:** This method is less applicable to Quick Sort due to non-fixed partition sizes, but for average-case scenarios, it confirms Θ(n log n).

**Practical Implications:** Quick sort operates in place. It requires little additional memory, which is better for a system with limited memory resources. The in-place nature improves cache utilization, which makes it efficient for large arrays. However, it can perform poorly for the sorted or nearly sorted data. Techniques like hybrid approaches, such as switching to insertion sort for small subarray or random pivot selection, are often taken to mitigate this.

**Merge Sort**

**Problems:** Merge sort is a sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller subarrays, sorting those subarrays, and then merging them back together to obtain the sorted array. Therefore, the process of merge sort is to divide the array into two halves, sort each half, and merge the sorted half. This process happens recursively until the array is completely sorted.

**Critical Step:** This sorting algorithm has three key steps: divide, conquer, and merge.

* **Divide:** Divide the list or array recursively into two halves until it is dividable.
* **Conquer:** Each subarray is sorted individually using the merge sort algorithm.
* **Merge:** Merge the sorted subarrays back together in sorted order. The process continues until all elements from both subarrays have been merged.

**Time Complexity:** Merge sort has a consistent time complexity because it always splits the array into halves and merges them, regardless of the initial order of elements. The time complexity for Best, Average, and Worst case is O(n log n).

**Recurrence Relation:** Merge Sort’s recurrence relation is T(n)=2T(n/2)+Θ(n), representing two recursive calls on halves of the array, plus O(n) where T(n) Represents the total time taken by the algorithm to sort an array of size n. 2T(n/2) represents the time the algorithm takes to sort the two halves of the array recursively. Since each half has n/2 elements, we have two recursive calls with input size as (n/2), and O(n) represents the time taken to merge the two sorted halves.

Substitution Method: Assuming T(n) = Θ(n log n), we can prove this by substitution, confirming that dividing the array down and merging each level yields log n levels with O(n) at each, resulting in Θ(n log n) complexity.

Recursion-Tree Method: Each recursive level costs O(n) for merging, and with log n levels, the overall complexity is Θ(n log n).

Master Method: Using the Master Theorem, with a = 2, b = 2, and f(n) = Θ(n) (Case 2), we find that T(n) = Θ(n log n), confirming the time complexity.

**Practical Implications:** Merge sorts to preserve the relative order of equal elements. It is valuable in applications that need stable sorting, such as sorting records with multiple keys. It is well suited for sorting large files on external due to its predictable divide-and-conquer approach. It requires O(n) auxiliary memory to merge, which makes it less ideal for in-place sorting but suitable for linked lists, where data is scattered and cannot be easily divided.

Both algorithms are powerful for different contexts: Quick Sort for in-memory and in-place sorting tasks and Merge Sort for scenarios requiring stability and consistent performance across varying data types.

**References:**

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms* (4th ed.).

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). "*The Analysis of Algorithms.*" In Introduction to Algorithms.

GeeksforGeeks. (n.d.). *Quick Sort vs. Merge Sort*. GeeksforGeeks. Retrieved November 3, 2024, from <https://www.geeksforgeeks.org/quick-sort-vs-merge-sort/>

GeeksforGeeks. (n.d.). *Quick Sort Algorithm*. GeeksforGeeks. Retrieved November 3, 2024, from <https://www.geeksforgeeks.org/quick-sort-algorithm/>

GeeksforGeeks. (n.d.). *Merge Sort*. GeeksforGeeks. Retrieved November 3, 2024, from <https://www.geeksforgeeks.org/merge-sort/>

Knuth, D. E. (1998). The Art of Computer Programming, *Volume 3: Sorting and Searching. Addison-Wesley*.

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.

**Implementation and Comparison**

I found some interesting things after implementing the quick sort and merge sort method.

A screenshot of a computer program

Description automatically generated**Figure 1**: Implementation Result of Quick Sort and Merge Sort for Various Data

Merge sort is always taking more space than quick sort. However, for reverse order, merge sort takes less time; other than that, quick sort takes less time than merge sort. I have tasted the different datasets. I have ranged up to 1000 elements of an array for a random dataset. Merger sorting always needs more time and space than a quick sorting process.

**A computer code with green text

Description automatically generatedFigure 2**: Quick sort implementation

I have used the median method to choose the pivot of the array; it might happen because of that. I am getting either the best or average case scenario. For the merge sort, it is always the same time complexity for the best, worst, and average.

According to the theoretical analysis, quick sorting is better for systems with limited memory resources. The in-place nature improves cache utilization, which makes it efficient for large arrays. Which I have found from the implementation. Also, it needs less time and space than merge sort. Merge sort is consistent in its performance, and quick sort is suitable for large datasets where space complexity is a concern.

**GitHub Link: https://github.com/ovi-saha/MSCS532\_Assignment2**