# Assignment 2: Analyzing and Implementing Divide-and-Conquer Algorithms

**1. Asymptotic Analysis and Recurrence Relations**

This assignment examines two prominent divide-and-conquer algorithms: Quick Sort and Merge Sort. All of these algorithms arrange a list of components in non-decreasing order, however they vary in approach and efficiency across different contexts. Quick Sort functions by choosing a pivot element, dividing the list into segments of entries lesser and larger than the pivot, then recursively sorting these segments. The fundamental processes in Quick Sort include selecting a pivot (often the final element), splitting the array based on the pivot, and recursively sorting the sub-arrays. The effectiveness of Quick Sort is contingent upon the selection of the pivot. In optimal and typical scenarios, when the pivot yields balanced partitions, Quick Sort attains an average time complexity of Θ(n log n). In the worst-case scenario—when the list is sorted in reverse order and the pivot consistently represents the smallest or biggest member—the performance degrades to O(n^2) time, leading to significantly imbalanced partitions where each recursive iteration decreases the issue size by just one element.

The average case recurrence relation for Quick Sort is expressed as T(n) = 2T(n/2) + Θ(n), where the 2T(n/2) component denotes the recursive calls on two segments of the array, and Θ(n) pertains to the partitioning procedure. This relationship may be resolved using many approaches. By using the substitution approach, we iteratively increase T(n) until we get at the base case, resulting in Theta(n log n) as the answer. The recursion-tree approach demonstrates a tree of log n levels, with each level contributing Theta(n) work, resulting in an overall time complexity of Theta(n log n). Applying the master technique, where T(n) = aT(n/b) + f(n), with a = 2, b = 2, and f(n) = Θ(n), we identify case 2 of the master theorem, establishing that T(n) = Θ(n log n) in the average case.

Merge Sort, a divide-and-conquer algorithm, consistently exhibits a temporal complexity of Theta(n log n) in all scenarios (best, average, and worst). Merge Sort partitions the list into two segments, recursively sorts each segment, and then merges the sorted segments. The fundamental processes in Merge Sort include partitioning the array into two halves, recursively sorting each segment, and amalgamating the sorted segments into a cohesive ordered array. The recurrence relation for Merge Sort is T(n) = 2T(n/2) + Θ(n), where both division and merging need Θ(n) time at each level. Utilizing the substitution technique to resolve this recurrence, we iteratively replace values until we arrive at the base case, yielding a temporal complexity of Θ(n log n). The recursion-tree approach substantiates this by demonstrating log n levels, each doing Theta(n) work, culminating in Theta(n log n). Utilizing the master technique with a = 2, b = 2, and f(n) = Θ(n), Merge Sort corresponds to case 2 of the master theorem, therefore establishing that T(n) = Θ(n log n).

In practical applications, Quick Sort's in-place sorting renders it more memory-efficient than Merge Sort, which need supplementary storage for merging. Quick Sort is often used when memory limitations are a consideration; nevertheless, it may exhibit suboptimal performance on material that is previously sorted or reverse-sorted. Merge Sort, characterized by its persistent Theta(n log n) efficiency, is often used in scenarios necessitating steady sorting and reliable performance, such as sorting linked lists or extensive datasets that cannot be totally accommodated in memory.

**2. Implementation and Comparison**

For the second part of this assignment, I implemented both Quick Sort and Merge Sort in Python and tested their performance on three types of datasets: sorted, reverse sorted, and random. These datasets highlight the strengths and weaknesses of each algorithm in different scenarios.

import time

import sys

sys.setrecursionlimit(1500) # Adjust recursion limit for larger datasets

def quick\_sort(arr):

def partition(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 quick\_sort\_recursive(low, high):

if low < high:

pi = partition(low, high)

quick\_sort\_recursive(low, pi - 1)

quick\_sort\_recursive(pi + 1, high)

quick\_sort\_recursive(0, len(arr) - 1)

return arr

# Merge Sort implementation

def merge\_sort(arr):

if len(arr) > 1:

mid = len(arr) // 2

left\_half = arr[:mid]

right\_half = arr[mid:]

merge\_sort(left\_half)

merge\_sort(right\_half)

i = j = k = 0

while i < len(left\_half) and j < len(right\_half):

if left\_half[i] < right\_half[j]:

arr[k] = left\_half[i]

i += 1

else:

arr[k] = right\_half[j]

j += 1

k += 1

while i < len(left\_half):

arr[k] = left\_half[i]

i += 1

k += 1

while j < len(right\_half):

arr[k] = right\_half[j]

j += 1

k += 1

return arr

I assessed the efficacy of various implementations by executing them on datasets including between 1,000 and 100,000 pieces, documenting the execution time and observing memory consumption. Quick Sort exhibited strong performance on both random and sorted datasets, attaining performance around its average-case complexity of Θ(n log n). Nonetheless, with reverse-sorted data, Quick Sort's efficiency decreased to O(n^2), illustrating its susceptibility to input arrangement owing to imbalanced divisions resulting from a simplistic pivot method. Merge Sort consistently attained Theta(n log n) efficiency across all kinds of datasets, as anticipated. This stability arises from Merge Sort's equitable partitioning of data, rendering it unaffected by the sequence of incoming data.   
Regarding memory use, Quick Sort demonstrated superior efficiency by sorting in situ and necessitating just a consistent amount of supplementary space for recursion. Merge Sort need more memory for temporary arrays during the merging process, which became substantial for bigger datasets. The memory expense of Merge Sort is a disadvantage in memory-constrained contexts, while providing stability and consistent performance.   
The gaps between theoretical analysis and actual performance are noticeable in practice. The theoretical average complexity of Quick Sort is Θ(n log n), which corresponds with its performance on random data; nevertheless, it diverges markedly in the worst-case scenario with reverse-sorted data. Multiple improvements, such as median-of-three pivot selection, may mitigate this vulnerability. Merge Sort exhibited consistent theoretical and practical performance, demonstrating Theta(n log n) behavior across all datasets. Ultimately, Quick Sort is often preferable in memory-limited applications, but Merge Sort is preferred for its predictability and stability, especially in scenarios requiring consistent performance, such as database sorting and managing extensive, unsorted data.