**Analyzing and Implementing Divide-and-Conquer Algorithms**

Shyam Nath

University of Cumberland

Algorithms and Data Structures (MSCS-532-B01)

Student ID: 005032491

Assignment 2

July 13, 2025

**Analyzing and Implementing Divide-and-Conquer Algorithms**

**1. Asymptotic Analysis and Recurrence Relations**

**A. Quick Sort**

**a. Problem and Key Steps**

Quick Sort is a divide-and-conquer sorting algorithm that:

* Chooses a pivot element.
* Divides the array into two subarrays: items less than the pivot item, items bigger than the pivot item (Taiwo et al., 2020).
* Sorts recursively the subarrays.

**b. Time Complexity Analysis**

Optimal (evenly-divided partitions):

* The comparison count is in ratio with n log n.
* O(n log n), Ω(n log n), Θ(n log n).

Worst case (highly unbalanced partitions):

* Arises when the pivot selects the min/max element every time (e.g. sorted data).
* O(n²).

Average case:

* The number of comparisons that are expected to be made is still proportional to log n.
* Θ (n log n).

**c. Recurrence Relation**

General: Best / average: balanced partition

T(n)=2T(n/2)+Θ(n)

Worst case: unbalanced partition (e.g. pivot is divided into sizes 1 and n-1)

T(n)=T(n−1)+Θ(n)

Master Method: Solving (best/average):

T(n)=2T(n/2)+Θ(n)

a=2, b=2, f(n)= Θ (n)

Log\_b a = log\_2 2 = 1

f(n)= 0(n loga b) Since loga b is in n, so case 2 of master theorem holds:

T(n) = O (n log n)

Worst case recursion-tree/substitution:

T(n)=T(n−1)+Θ(n)=Θ(n2)

**d. Practical Implications**

Quick Sort is efficient in the sense it is efficient enough in practise because:

• In-place sorting (low space consumptions).

• Effective use of cache.

Disadvantage: in worst case it becomes O(n 2). However, the risk is mitigated by randomized Quick Sort.

**B. Merge Sort**

**a. Problem and Key Steps**

Merge Sort separates by:

* Partitioning the array in two parts.
* Sorting the two halves recursively.
* Combining the halves in a sorted array which is sorted (Izadkhah, 2022).

**b. Time Complexity Analysis**

Best, worst, and average cases:

* Never shrinks by more than a half; the merging time is linear.
* O(n log n), Ω(n log n), Θ(n log n).

**c. Recurrence Relation**

T(n)=2T(n/2)+Θ(n)

Master Method:

a=2 b=2 f(n)=Theta(n)

• log b a = 1

• f(n)=Θ(n^{log\_b a})

• Case 2:

T(n) = O (n log n)

The Recursion-tree and substitution are the same result.

**d. Practical Implications**

Merge Sort is O(n log n), no matter what the input is.

Cons: will need another O(n) space to merge.

**2. Implementation and Comparison**

**A. Python Implementation**

Merge Sort and Quick Sort were both implemented in Python. The algorithms were divide and conquer in nature and hence reflected in recursion. The time and tracemalloc have been used to measure execution time and memory usage respectively.

**B. Test Datasets**

**Sorted**: [1, 2, 3, ..., n]

**Reverse Sorted**: [n, n-1, ..., 1]

**Random**: Random integers from 1–10,000

**C. Performance Results**

Results saved in sorting\_results.csv.

**D. Graphs**

* Time Comparison
* Memory Usage Comparison

The results and the comparison graphs are in the GitHub repository

**Observations**

Quick Sort also shows weakness when applied to sorted and reverse-sorted data as poor pivot choices cause partitions to be unbalanced.

Merge Sort is consistent in performance and has additional memory usage because of arrays copying.

On random data, Quick Sort was a bit faster because in-place sorting has less large constants.

The practical performance is comparable to theoretical analysis particularly where the size of data is increased.

**GitHub Repository Link**

Here is the link where you can find the code implementation, results in a csv file and the comparison graphs.

https://github.com/Snath32491/MSCS\_532\_Assignment-2.git

**References**

Izadkhah, H. (2022). Divide and Conquer. In *Problems on Algorithms: A Comprehensive Exercise Book for Students in Software Engineering* (pp. 351-400). Cham: Springer International Publishing.

Taiwo, O. E., Christianah, A. O., Oluwatobi, A. N., & Aderonke, K. A. (2020). Comparative study of two divide and conquer sorting algorithms: quicksort and mergesort. *Procedia Computer Science*, *171*, 2532-2540.

**Appendices**

**Python Code**

import random

import time

import tracemalloc

import sys

import matplotlib.pyplot as plt

import pandas as pd

sys.setrecursionlimit(10000)

# ---------------- SORTING ALGORITHMS ----------------

def merge\_sort(arr):

    if len(arr) > 1:

        mid = len(arr) // 2

        left = arr[:mid]

        right = arr[mid:]

        merge\_sort(left)

        merge\_sort(right)

        i = j = k = 0

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

            if left[i] < right[j]:

                arr[k] = left[i]

                i += 1

            else:

                arr[k] = right[j]

                j += 1

            k += 1

        while i < len(left):

            arr[k] = left[i]

            i += 1

            k += 1

        while j < len(right):

            arr[k] = right[j]

            j += 1

            k += 1

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 quick\_sort(arr, low, high):

    if low < high:

        pi = partition(arr, low, high)

        quick\_sort(arr, low, pi - 1)

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

# ---------------- TESTING & MEASUREMENT ----------------

results = []

def run\_and\_measure(sort\_func, arr, algo\_name, dataset\_name):

    arr\_copy = arr.copy()

    tracemalloc.start()

    start\_time = time.perf\_counter()

    if algo\_name == "Quick Sort":

        sort\_func(arr\_copy, 0, len(arr\_copy) - 1)

    else:

        sort\_func(arr\_copy)

    end\_time = time.perf\_counter()

    current, peak = tracemalloc.get\_traced\_memory()

    tracemalloc.stop()

    exec\_time = round((end\_time - start\_time) \* 1000, 3)  # in ms

    peak\_mem = round(peak / 1024, 3)  # in KB

    print(f"{algo\_name} on {dataset\_name} data:")

    print(f"  Time taken      : {exec\_time} ms")

    print(f"  Peak memory     : {peak\_mem} KB")

    print("-" \* 50)

    results.append({

        "Algorithm": algo\_name,

        "Dataset": dataset\_name,

        "Time (ms)": exec\_time,

        "Memory (KB)": peak\_mem

    })

# ---------------- MAIN ----------------

def main():

    size = 1000  # Feel free to increase for larger testing

    sorted\_data = list(range(size))

    reverse\_data = list(range(size, 0, -1))

    random\_data = [random.randint(1, 10000) for \_ in range(size)]

    datasets = [

        ("Sorted", sorted\_data),

        ("Reverse Sorted", reverse\_data),

        ("Random", random\_data)

    ]

    for name, data in datasets:

        run\_and\_measure(merge\_sort, data, "Merge Sort", name)

        run\_and\_measure(quick\_sort, data, "Quick Sort", name)

    # Convert results to DataFrame

    df = pd.DataFrame(results)

    # Save to CSV

    df.to\_csv("sorting\_results.csv", index=False)

    print("\n Results saved to 'sorting\_results.csv'")

    # Plot graphs

    plot\_results(df)

# ---------------- PLOTTING ----------------

def plot\_results(df):

    plt.figure(figsize=(10, 6))

    for metric in ["Time (ms)", "Memory (KB)"]:

        plt.clf()

        pivot = df.pivot(index="Dataset", columns="Algorithm", values=metric)

        pivot.plot(kind='bar', figsize=(10, 6))

        plt.title(f"{metric} Comparison")

        plt.ylabel(metric)

        plt.xlabel("Dataset Type")

        plt.xticks(rotation=0)

        plt.tight\_layout()

        filename = f"{metric.replace(' ', '\_').replace('(', '').replace(')', '')}\_comparison.png"

        plt.savefig(filename)

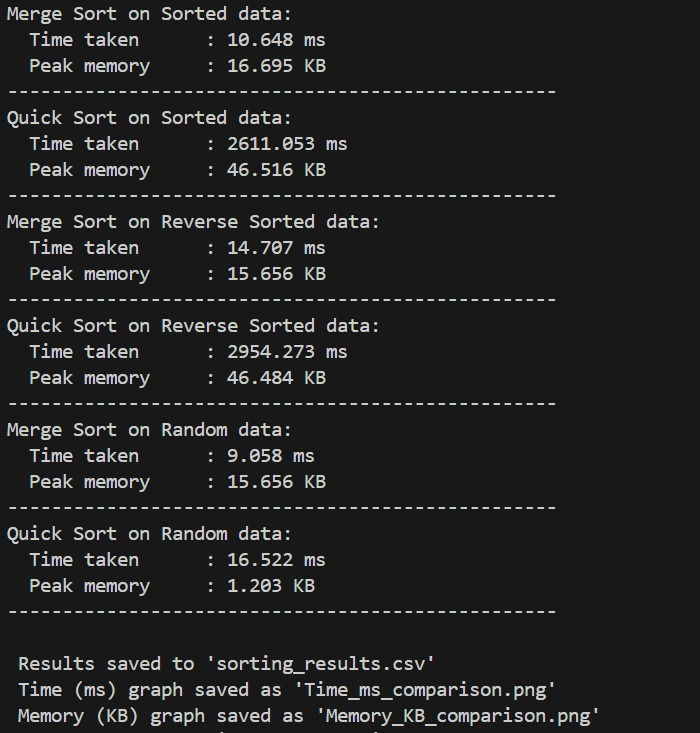
        print(f" {metric} graph saved as '{filename}'")

# ---------------- RUN ----------------

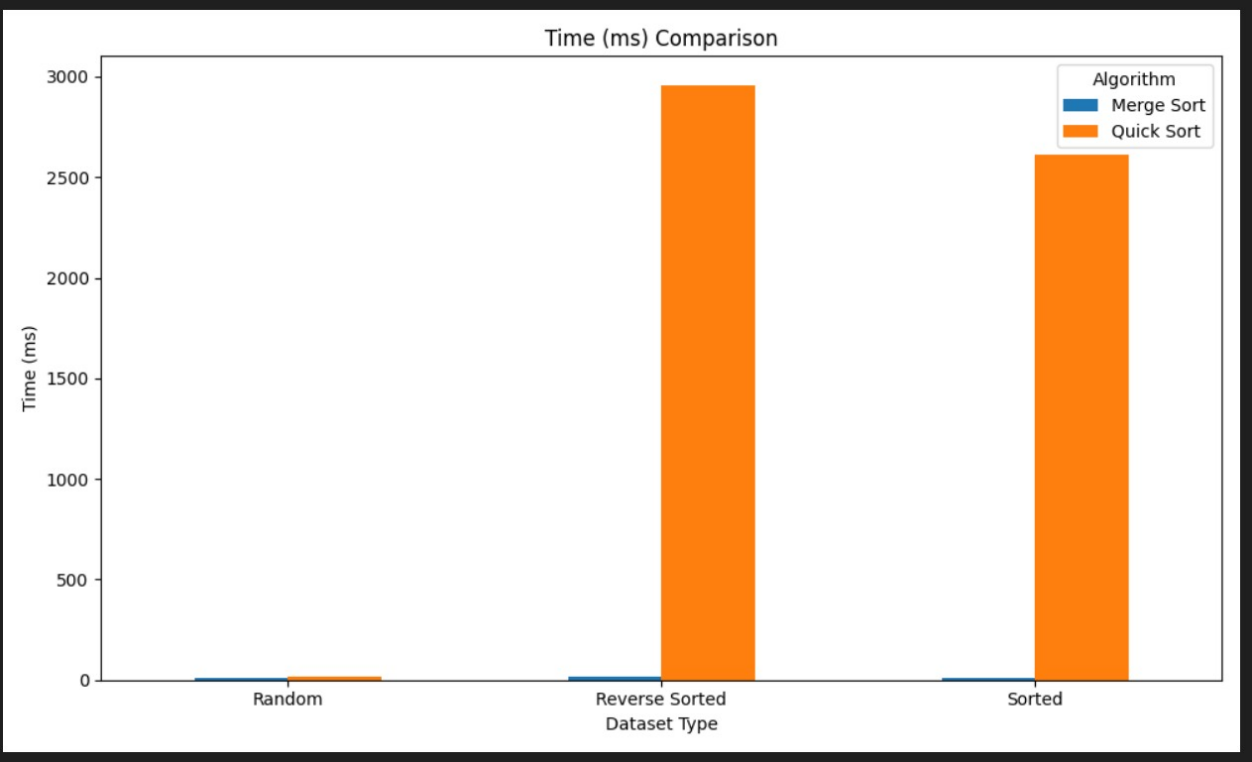
if \_\_name\_\_ == "\_\_main\_\_":

    main()

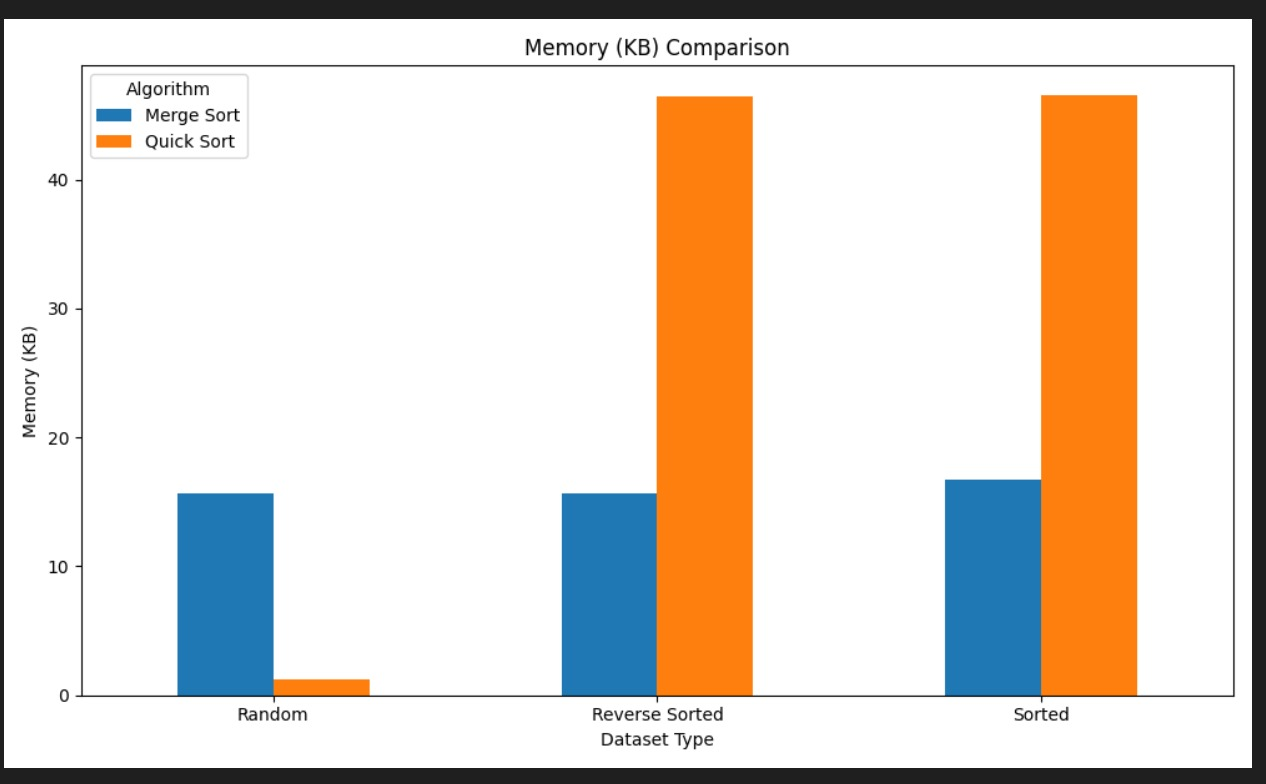
**Results from the Implemented Code**



**Time Comparison Graph**

****

**Memory Usage Comparison Graph**

****