Calamiong, Yno Andrei
Hontiveros, Jan Aldo

I. INTRODUCTION

Sorting Algorithms are used to arrange elements into proper order, usually either ascending or descending. For this project, we are tasked to implement three (3) sorting algorithms on seven (7) data sets of various input sizes and order. Our group chose to write Python scripts for the Merge, Insertion, and Quick Sort algorithms which were implemented on data sets that were randomized, almost sorted, and totally reversed. The outline for this project is as follows:

A. Introduction
B. Sorting Algorithms
C. Process and Implementation
D. Data Analysis
E. Learnings and Insights
F. Contributions
G. References

II. SORTING ALGORITHMS

A. Merge Sort

Merge Sort is one of the most efficient sorting algorithms, it uses the strategy of divide-and-conquer. Generally, it continually splits a list into two until each element has only one item. As we see on the following code block, we first define the implementation of the merge sort algorithm. The first condition inspects if the size of the array is greater than 1 and is divided into two parts. If the condition says otherwise, the resulting array must mean it is empty or has only 1 item. We then define the left and right half of the array and sub-arrays and initialize the step count to zero.

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr)//2
        left_half = arr[:mid]
        right_half = arr[mid:]
        steps = 0
        steps += merge_sort(left_half)
        steps += 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
            steps += 1
        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1
            steps += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
            steps += 1
        return steps
    return 0
```

The ‘i’, ‘j’, and ‘k’ correspond to the indices of the left, right, and merged array, respectively. The while loop that follows represents the comparison, swapping of the elements, and merging of the sub-arrays that arrive at the resulting sorted array. Steps are counted for each comparison and movement of items to a sorted array.

B. Insertion Sort

Insertion Sort is a comparison-based sorting algorithm that divides a dataset into two parts: a sorted and an unsorted array. The algorithm initially takes the second element and compares it to the first element- which represents the sorted part of the array. Insertion sort then repeatedly takes one item from the unsorted array and compares and swaps it to each element on the sorted

array on the left. On the code block below, we see that it is particularly short as it only performs the sorting process one element at a time and increases the step count as it goes.

```
def insertion_sort(arr):
    steps = 0
    for i in range(1, len(arr)):
        steps += 1
        j = i
        while arr[j - 1] > arr[j] and j > 0:
            arr[j - 1], arr[j] = arr[j], arr[j - 1]
            j -= 1
            steps += 1
    return steps
```

We initially set the step count at 0 and enter a for loop where the comparison and swaps repeatedly occur. The [1] element is then compared to the [0] element to begin the insertion sort process. On the inner while loop, we can see the comparison of each element being compared to another element to its left. The ‘j -= 1 ‘ line dictates the leftward comparison of the [j] element to the [j-1] element until finally arriving at a sorted array.

C. Quick Sort

Quick Sort is a widely used sorting algorithm that employs a divide-and-conquer strategy to sort elements in an array. It operates by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. This process continues recursively for the sub-arrays until the entire array is sorted.

```
def partition(arr, low, high):
    steps = 0

    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]
            steps += 1

    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    steps += 1
    return i + 1, steps
```

1. The partition function takes an array (arr), the lower index (low), and the higher index (high) as input.
2. It selects the pivot element, here, the element at the high index of the array.
3. It initializes an index i as low - 1.
4. Using a loop, it compares each element in the array with the pivot: If the element is less than the pivot, it increments i and swaps the elements at indices i and j (the current iteration index).
5. Finally, it places the pivot element at its correct position in the array by swapping it with the element at index i + 1.
6. The function returns the index where the pivot is placed and the number of steps taken in the process.

```
def quick_sort(arr):
    size = len(arr)
    stack = [(0, size - 1)]

    total_steps = 0

    while stack:
        low, high = stack.pop()

        if low < high:
            p, steps = partition(arr, low, high)
            total_steps += steps

            stack.append((low, p - 1))
            stack.append((p + 1, high))

    return arr, total_steps
```

1. The quick_sort function takes an array (arr) as input.
2. It initializes a stack to keep track of the indices.
3. The function enters a while loop that runs as long as the stack is not empty.
4. Inside the loop:
    a. It pops the indices low and high from the stack.
    b. If low is less than high, it calls the partition function.
    c. The partition function rearranges the elements and returns the pivot index.
    d. It then pushes the indices related to the elements less than the pivot and the elements greater than the pivot onto the stack.
5. When the stack becomes empty, the array is completely sorted.

III. PROCESS AND IMPLEMENTATION

Our codebases are primarily composed of 3 parts:

1. Sorting Algorithm Code (Insertion, Merge, or Quick)
2. Code to measure the average Execution Time and Frequency Count
3. Code that reads/writes text files and iteratively applies the Sorting Algorithms on the Dataset.

```
for filename in os.listdir(input_directory):
    if filename.endswith(".txt"):
        file_path = os.path.join(input_directory, filename)
        n, records = read_file(file_path)
```

The code above is responsible for iterating through all the datasets in the data folder, such as almostsorted.txt, random100.txt, etc. We also wrote functions to read/write files as presented below:

```
def read_file(file_name):
    with open(file_name, 'r') as file:
        data = file.readlines()
        n = int(data[0])
        records = [[int(line.strip().split(' ')[0]), ' '.join(line.strip().split(' ')[1:])] for line in data[1:]]
        return n, records

def write_file(file_name, n, records):
    with open(file_name, 'w') as file:
        file.write(f"{n}\n")
        for record in records:
            file.write(f"{record[0]} {record[1]}\n")
```

The read_file function converts the datasets into an array containing paired lists of ID numbers and Names. Example: [[108, Galant einstein], [199, Wonderful Archimedes]...]. Note that the function also converts the ID numbers into integers so that the sorting algorithms can accurately compare the value of said numbers. We noticed that if the ID numbers are of string type, the algorithms couldn't sort the dataset accurately.

```
# Variables to store steps and time for each dataset
steps_list = []
time_list = []

for i in range(6):
    print(f"Processing {filename} - Sorting iteration {i+1}")

    start_time = time.time()
    sorted_records, step_count = quick_sort(records[:])
    end_time = time.time()

    execution_time = end_time - start_time
    exec_time_readable = execution_time

    steps_list.append(step_count)
    time_list.append(execution_time)

avg_execution_time = sum(time_list[1:]) / (len(time_list) - 1) if len(time_list) > 1 else 0
avg_steps = sum(steps_list[1:]) / (len(steps_list) - 1) if len(steps_list) > 1 else 0
```

Meanwhile, the code above applies the sorting algorithm (in this example, Quick Sort) at least 5x to a given data set. It also measures the average execution time and frequency count per each

dataset. We save the sorted dataset and the computed average frequency counts and execution time in their own respective text files through the code below:

```
        # Save results for each dataset
        results[filename] = {'avg_execution_time': avg_execution_time, 'avg_steps': avg_steps}

        sorted_file_name = filename.split('.')[0] + "_quick_sorted.txt"
        sorted_file_path = os.path.join(output_directory, sorted_file_name)
        write_file(sorted_file_path, n, sorted_records)
```

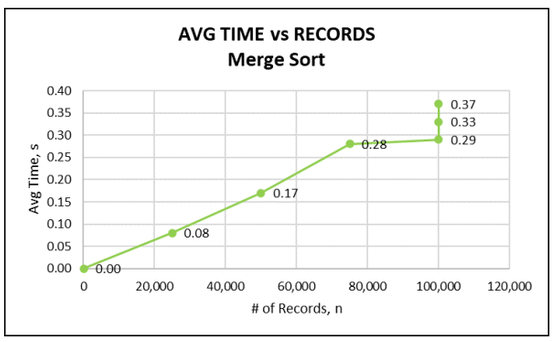
```
with open(results_log_file, 'w') as file:
    file.write("Algorithm Analysis Results (Quick Sort)\n\n")
    for dataset, values in results.items():
        avg_exec_time = values['avg_execution_time']
        formatted_time = str(datetime.timedelta(seconds=avg_exec_time)).split(".")[0]  # Convert seconds to HH:MM:SS
        file.write(f"Dataset: {dataset}\n")
        file.write(f"Avg Execution Time: {formatted_time} | Exact Seconds: {values['avg_execution_time']}\n")
        file.write(f"Avg Steps: {values['avg_steps']}\n\n")
```

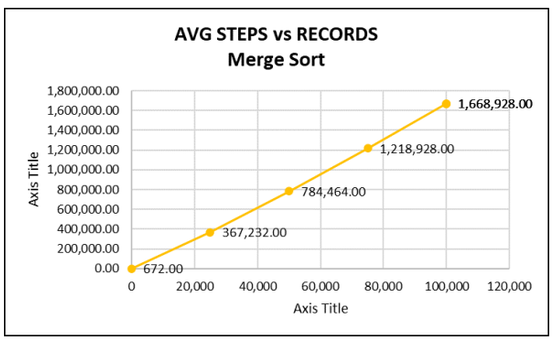
Finally, to verify if the files are sorted correctly, we wrote another script that reads through all the sorted datasets and confirms if the ID Numbers and Names are sorted in ascending order. The code is presented as follows:

```
import os

def is_sorted(file_path):
    with open(file_path, 'r') as file:
        lines = file.readlines()
        n = int(lines[0])
        data = [line.split() for line in lines[1:]]
        ids = [int(record[0]) for record in data]
        return ids == sorted(ids)

directory = 'Data'

for filename in os.listdir(directory):
    file_path = os.path.join(directory, filename)
    if os.path.isfile(file_path) and filename.endswith('.txt'):
        if is_sorted(file_path):
            print(f"{filename} is sorted in ascending order by ID number.")
        else:
            print(f"{filename} is NOT sorted in ascending order by ID number.")
```

IV. DATA ANALYSIS

A. Merge Sort



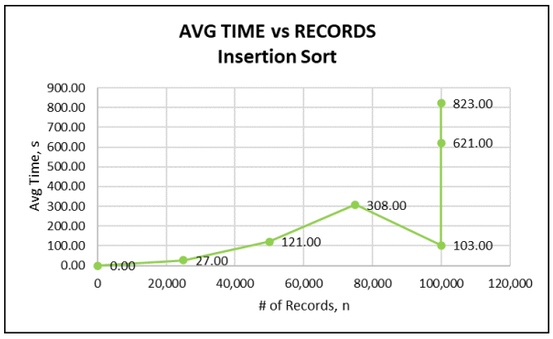


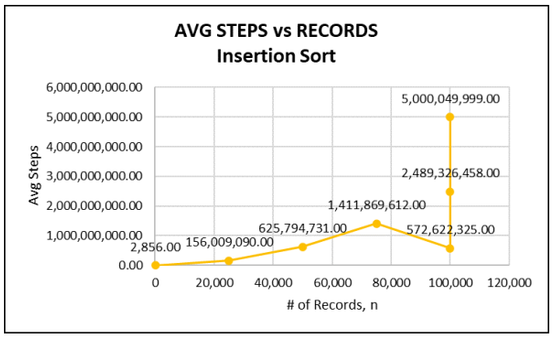
Data file	# of records, n	Avg Time, s	Avg Steps
Random 100	100	0.00	672.00
Random 25000	25,000	0.08	367,232.00
Random 50000	50,000	0.17	784,464.00
Random 75000	75,000	0.28	1,218,928.00
Totally Reversed	100,000	0.29	1,668,928.00
Almost Sorted	100,000	0.33	1,668,928.00
Random 100000	100,000	0.37	1,668,928.00

Merge sort is one of the most efficient sorting algorithms implemented for this project, the other being Quick Sort. Merge Sort has an average time complexity of O(n log n), meaning that it is consistently efficient regardless of the dataset. In our case, we noticed that the number of steps for the 'random 100000', 'totally reversed', and 'almost sorted' files are the same at 1.67M.

For execution time, we noticed that Merge Sort quickly sorts through all the datasets, having average times of less than a second, even if the input size reaches a value of 100,000. Simply put, Merge Sort consistently has very low execution times regardless of the input size and the initial sort order.

B. Insertion Sort

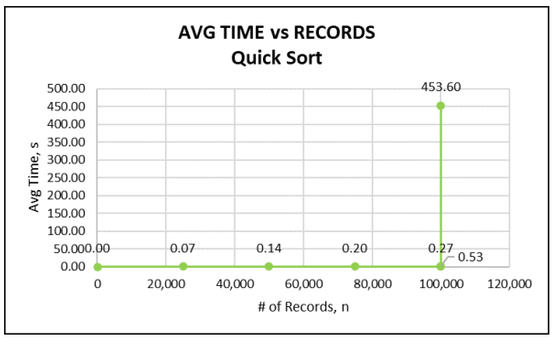


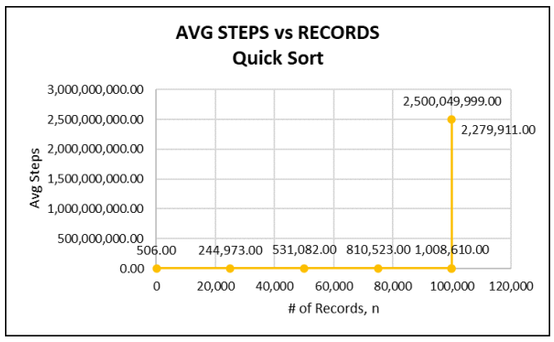


Data file	# of records, n	Avg Time, s	Avg Steps
Random 100	100	0.00	2,856.00
Random 25000	25,000	27.00	156,009,090.00
Random 50000	50,000	121.00	625,794,731.00
Random 75000	75,000	308.00	1,411,869,612.00
Almost Sorted	100,000	103.00	572,622,325.00
Random 100000	100,000	621.00	2,489,326,458.00
Totally Reversed	100,000	823.00	5,000,049,999.00

Insertion sort yielded the slowest results out of the three sorting algorithms implemented in this project. Upon observation of the results above, we can conclude that Insertion sort efficiently works on small and almost-sorted (best case: O(n)) datasets only. Insertion sort has a worst-case time complexity of O(n^2), proving it to be very inefficient for large and totally-reversed datasets. We see that the curve behavior of the average time is similar to the average step because the running time and step count are directly proportional to the input size of the data sets.

C. Quick Sort





Data file	# of records, n	Avg Time, s	Avg Steps
Random 100	100	0.00	506.00
Random 25000	25,000	0.07	244,973.00
Random 50000	50,000	0.14	531,082.00
Random 75000	75,000	0.20	810,523.00
Random 100000	100,000	0.27	1,008,610.00
Almost Sorted	100,000	0.53	2,279,911.00
Totally Reversed	100,000	453.60	2,500,049,999.00

On Average, Quick Sort performed better than Merge Sort, with slightly faster execution times and a lower number of steps. However, as Quick Sort's worst-case time complexity is O(n^2), it performed significantly worse in the totally reversed dataset (at around 7 minutes average execution time) as against Merge Sort (at only 0.29 seconds). This can be attributed to the fact that Merge Sort's worst, best, and average time complexity is consistent at O(n log n), regardless of the dataset. And while Quick Sort is generally faster on average, with a time complexity of around O(n log n), it has the potential to degrade to O(n^2) given some datasets.

V. LEARNINGS AND INSIGHTS

- Not all algorithms are created equal. Some are more efficient than others, especially given specific datasets. Thus it is essential to have at least a baseline understanding of the dataset we are going to process before we choose an algorithm. For example, given our experiments, Quick Sort performed faster than Merge Sort for most of the datasets, but it performed significantly worse in the Totally Reversed dataset. With this info, we can surmise that it would always be better to choose Merge Sort if we already know beforehand that the dataset is totally reversed.
- When dealing with very big datasets, it's important to double-check results, as the sorted dataset may seem correct in the first n items but may be wrong in the middle or near the end of the dataset. In our experiments, we noticed that when we didn't explicitly convert the ID numbers into integers, this would have led to wrong sorting orders in some places within the dataset that are not readily apparent if we hadn't verified the results.
- The step count, which is an integral criterion of this project, proved to be challenging as it is also coded deep within the respective sorting algorithm's block. During the initial test runs, it keeps returning illogical values. That's why it is essential to understand which part of the algorithm makes comparisons and swaps, so we can insert our step counter more accurately.
- Sorting algorithms are widely used in different fields; one practical use is Inventory Management. With a properly sorted database, we can quickly retrieve specific product/item data by their name, number, size, color, and other attributes.

VI. CONTRIBUTIONS

A. Yno
   a. Quick Sort Algorithm and Implementation
   b. Code for:
      i. Iteratively Reading/Writing Files
      ii. Iterative application of Sorting Algorithm per Dataset
      iii. Measuring Average Execution Time and Frequency Count
      iv. Script to Verify Correctness of Sorted Files
   c. Written Report

B. Aldo
   a. Merge Sort Algorithm and Implementation
   b. Insertion Sort Algorithm and Implementation
   c. Written Report

VII. REFERENCES

GeeksforGeeks. (n.d.). Quick Sort. Retrieved from https://www.geeksforgeeks.org/quick-sort/

DataCamp. (n.d.). Data Structures and Algorithms in Python: Sorting Algorithms. Retrieved from https://campus.datacamp.com/courses/data-structures-and-algorithms-in-python/sorting-algorithms