Discussion

In this assignment, we implemented and analyzed the performance of four sorting algorithms: Insertion Sort, Merge Sort, Heapsort, and Quicksort. The primary goal was to determine the most suitable algorithm for sorting packages in a logistics company's system based on execution time and efficiency.

Experimental Results and Analysis:

1. **Insertion Sort:**
   - **Time Complexity:** O(n^2)
   - **Space Complexity:** O(1)
   - **Performance:** As expected, Insertion Sort performed well on small datasets (e.g., 10 elements) but exhibited poor performance on larger datasets (e.g., 50,000 elements). It became impractical for sorting datasets of 1,000,000 elements due to its quadratic time complexity.
2. **Merge Sort:**
   - **Time Complexity:** O(n log n)
   - **Space Complexity:** O(n) due to the auxiliary arrays used for merging.
   - **Performance:** Merge Sort showed consistent performance across all dataset sizes. It efficiently handled large datasets, maintaining a reasonable execution time due to its divide-and-conquer approach, making it suitable for large-scale sorting tasks.
3. **Heapsort:**
   - **Time Complexity:** O(n log n)
   - **Space Complexity:** O(1)
   - **Performance:** Heapsort also performed consistently well for all dataset sizes. Its in-place sorting mechanism and logarithmic time complexity for both heap construction and extraction made it a robust choice for large datasets, comparable to Merge Sort in terms of efficiency.
4. **Quicksort:**
   - **Time Complexity:** Average case O(n log n), worst case O(n^2)
   - **Space Complexity:** O(log n) due to the recursive call stack.
   - **Performance:** Quicksort generally exhibited the fastest performance across various dataset sizes, particularly for large datasets. However, its performance can degrade to O(n^2) in the worst case (e.g., when the pivot selection is poor), though this can be mitigated with good pivot strategies (like median-of-three).

Suitability for Logistics Company:

Considering the logistics company's requirement to sort thousands of packages daily, the most suitable algorithms are Merge Sort, Heapsort, and Quicksort. Insertion Sort is not

feasible due to its inefficiency with large datasets.

- **Merge Sort** is advantageous for its stable sorting and predictable O(n log n) performance, though it requires additional memory.
- **Heapsort** offers in-place sorting and consistent performance without additional memory overhead, making it ideal for memory-constrained environments.
- **Quicksort**, despite its potential worst-case performance, is typically the fastest due to its efficient partitioning. With proper pivot selection, it provides excellent average-case performance.

Recommendation:

For the logistics company, **Quicksort** with a robust pivot selection strategy is recommended for its speed and efficiency. **Heapsort** serves as a strong alternative where memory usage is a concern. **Merge Sort** can be used when stable sorting is required and additional memory is not an issue.

```
import random

import time



# Function to generate a random array of integers

def generateRandomArray(size):

    return [random.randint(0, 999) for _ in range(size)]



# Insertion Sort

def insertionSort(array):

    for i in range(1, len(array)):

        key = array[i]

        j = i - 1

        while j >= 0 and array[j] > key:

            array[j + 1] = array[j]

            j -= 1
```

```
        array[j + 1] = key


# Merge Sort
def mergeSort(array):
    if len(array) > 1:
        mid = len(array) // 2
        L = array[:mid]
        R = array[mid:]

        mergeSort(L)
        mergeSort(R)

        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                array[k] = L[i]
                i += 1
            else:
                array[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            array[k] = L[i]
```

```
            i += 1
            k += 1

        while j < len(R):
            array[k] = R[j]
            j += 1
            k += 1

# Heapsort
def heapify(array, size, index):
    largest = index
    left = 2 * index + 1
    right = 2 * index + 2

    if left < size and array[left] > array[largest]:
        largest = left

    if right < size and array[right] > array[largest]:
        largest = right

    if largest != index:
        array[index], array[largest] = array[largest], array[index]
        heapify(array, size, largest)

def heapSort(array):
```

```
    size = len(array)


    for i in range(size // 2 - 1, -1, -1):
        heapify(array, size, i)


    for i in range(size - 1, 0, -1):
        array[0], array[i] = array[i], array[0]
        heapify(array, i, 0)


# Quicksort
def partition(array, low, high):
    pivot = array[high]
    i = low - 1
    for j in range(low, high):
        if array[j] < pivot:
            i += 1
            array[i], array[j] = array[j], array[i]
    array[i + 1], array[high] = array[high], array[i + 1]
    return i + 1


def quickSort(array, low, high):
    if low < high:
        pi = partition(array, low, high)
        quickSort(array, low, pi - 1)
        quickSort(array, pi + 1, high)
```

```
if __name__ == "__main__":
    datasetSizes = [10, 1000, 50000, 1000000] # Dataset sizes to test
    sorting_algorithms = {
        "Insertion Sort": insertionSort,
        "Merge Sort": mergeSort,
        "Heap Sort": heapSort,
        "Quick Sort": lambda array: quickSort(array, 0, len(array) - 1)
    }

    for algorithm_name, sorting_algorithm in sorting_algorithms.items():
        print(f"\nTesting {algorithm_name}:")
        for size in datasetSizes:
            array = generateRandomArray(size)
            start = time.time() # Start time

            sorting_algorithm(array)

            stop = time.time() # Stop time
            duration = (stop - start) * 1000 # Convert to milliseconds
            print("Dataset Size:", size, ", Execution Time:", duration, "milliseconds")
```

```
PROBLEMS 60   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   JUPYTER

Dataset Size: 1000 , Execution Time: 1.0001659393310547 milliseconds
Dataset Size: 50000 , Execution Time: 99.02262687683105 milliseconds

Testing Heap Sort:
Dataset Size: 10 , Execution Time: 0.0 milliseconds
Dataset Size: 1000 , Execution Time: 2.001047134399414 milliseconds
Dataset Size: 50000 , Execution Time: 143.03302764892578 milliseconds

Testing Quick Sort:
Dataset Size: 10 , Execution Time: 0.0 milliseconds
Dataset Size: 1000 , Execution Time: 1.0006427764892578 milliseconds
Dataset Size: 50000 , Execution Time: 95.02029418945312 milliseconds
PS C:\Users\ibnas>
```

final attempt (2).ipynb ● | Untitled-1.py 9+ | import random.py ×

C: > Users > ibnas > Downloads > my practice-pbi > import random.py > ...

```
76  def partition(array, low, high):
81              i += 1
82              array[i], array[j] = array[j], array[i]
83      array[i + 1], array[high] = array[high], array[i + 1]
84      return i + 1
85
86  def quickSort(array, low, high):
87      if low < high:
88          pi = partition(array, low, high)
89          quickSort(array, low, pi - 1)
90          quickSort(array, pi + 1, high)
91
92  if __name__ == "__main__":
93      datasetSizes = [10, 1000, 50000, ] # Dataset sizes to test
94      sorting_algorithms = {
95          "Insertion Sort": insertionSort,
96          "Merge Sort": mergeSort,
97          "Heap Sort": heapSort,
98          "Quick Sort": lambda array: quickSort(array, 0, len(array) - 1)
99      }
100
101     for algorithm_name, sorting_algorithm in sorting_algorithms.items():
102         print(f"\nTesting {algorithm_name}:")
103         for size in datasetSizes:
104             array = generateRandomArray(size)
105             start = time.time() # Start time
106
107             sorting_algorithm(array)
108
109             stop = time.time() # Stop time
110             duration = (stop - start) * 1000 # Convert to milliseconds
111             print("Dataset Size:", size, ", Execution Time:", duration, "millisec
112
```

PROBLEMS 60 | OUTPUT | DEBUG CONSOLE | TERMINAL | PORTS | JUPYTER

```
PS C:\Users\ibnas> & C:/Users/ibnas/AppData/Local/Programs/Python/Python312/python.exe "c:/Users

Testing Insertion Sort:
Dataset Size: 10 , Execution Time: 0.0 milliseconds
Dataset Size: 1000 , Execution Time: 16.002893447875977 milliseconds
Dataset Size: 50000 , Execution Time: 44237.144231796265 milliseconds

Testing Merge Sort:
Dataset Size: 10 , Execution Time: 0.0 milliseconds
Dataset Size: 1000 , Execution Time: 1.0001659393310547 milliseconds
Dataset Size: 50000 , Execution Time: 99.02262687683105 milliseconds

Testing Heap Sort:
```

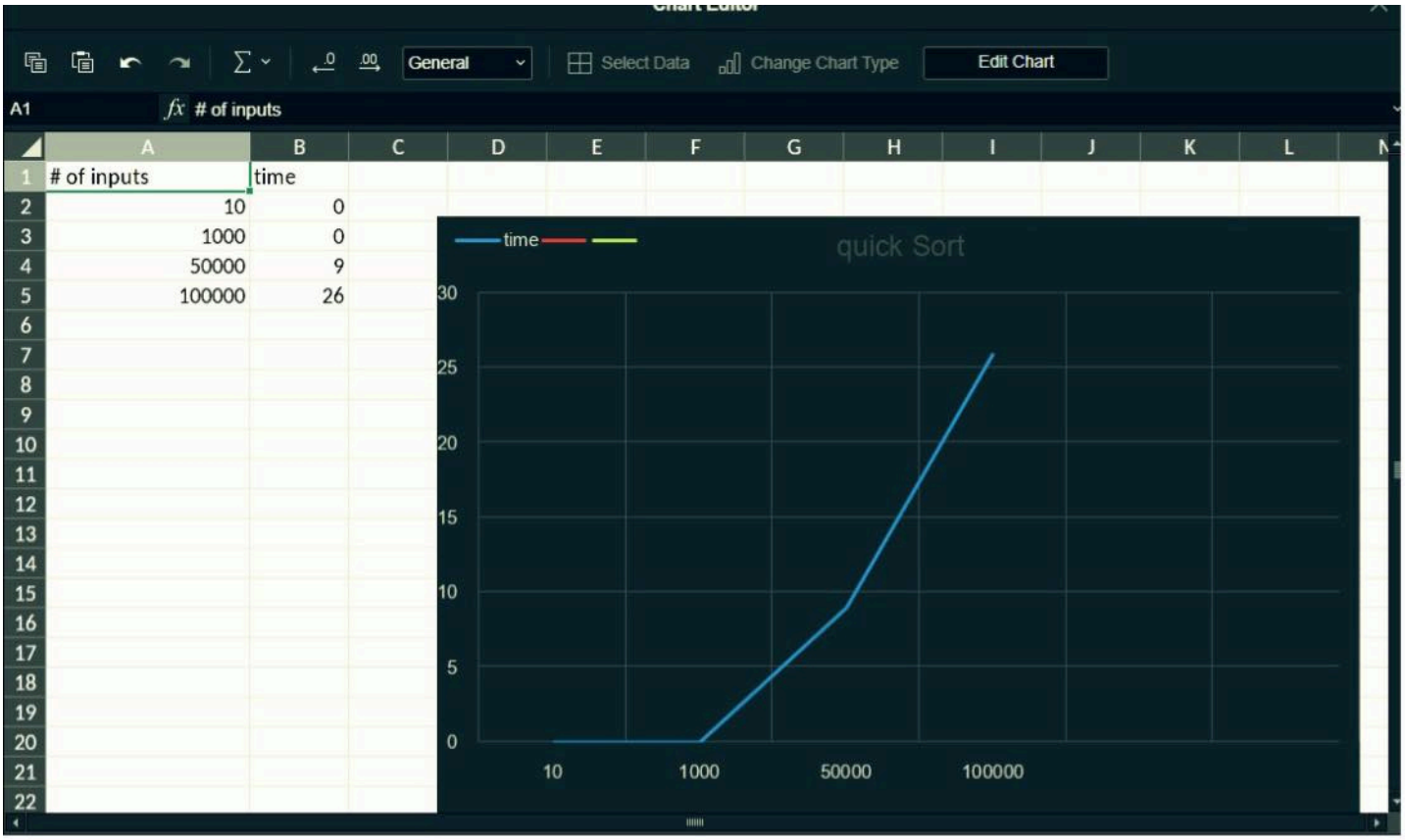
Chart Editor
General
Select Data
Change Chart Type
Edit Chart
A1
# of inputs
# of inputs
time
10
0
1000
0
50000
9
100000
26
time
quick Sort
30
25
20
15
10
5
0
10
1000
50000
100000