Algorithm Assignment #1

23013445 Jinhyeop Hong, 21010730 MinsuPark

Q 1 – Sorting Student Records (CSV Processing)

1. Objective

The goal of this task was to implement a Python program that reads a list of student records from a CSV file named student_records.csv and sorts them based on the students' marks
using two different algorithms (Insertion Sort and Merge Sort).

After sorting, the results were saved into two separate CSV files for verification.

2. Implementation Overview

The program used the built-in csv module in Python to handle file reading and writing.

Each record contained the following fields:

- Name
- Age
- Marks

Summary (Q1)

This task demonstrated how sorting algorithms differ in performance when applied to real data such as CSV files.

Both Insertion Sort and Merge Sort successfully arranged student records by marks, but Merge Sort proved to be the more efficient algorithm for handling larger datasets.

Q 2 – Selection Sort Analysis

1. Pseudocode

```
SelectionSort(A)
for i = 0 to n - 2
    min_index = i
    for j = i + 1 to n - 1
        if A[j] < A[min_index]
            min_index = j
    swap A[i] and A[min_index]
```

This shows the two nested loops

→ The outer loop selects a position, and the inner loop finds the minimum element to place there.

2. Why the algorithm runs until (n − 1)

The algorithm only needs to run for the first (n−1) elements because after the (n−1)th iteration, the remaining last element will automatically be the largest or smallest depending on the sorting order.
Therefore, no further comparisons are necessary for the (n)th element ← It is already in the correct position.

3. Running Time Analysis …Using Θ-notation

Best and worst case analysis

Case	Comparisons	Swaps	Overall Time Complexity
Best Case	$n(n-1)/2$	$n-1$	$\Theta(n^2)$
Worst Case	$n(n-1)/2$	$n-1$	$\Theta(n^2)$

The number of comparisons is always the same regardless of input order.
Therefore, both best and worst cases have the same $\Theta(n^2)$ complexity.
However, the number of swaps may vary slightly → at most (n−1) swaps will occur, but this doesn't change the asymptotic behavior.

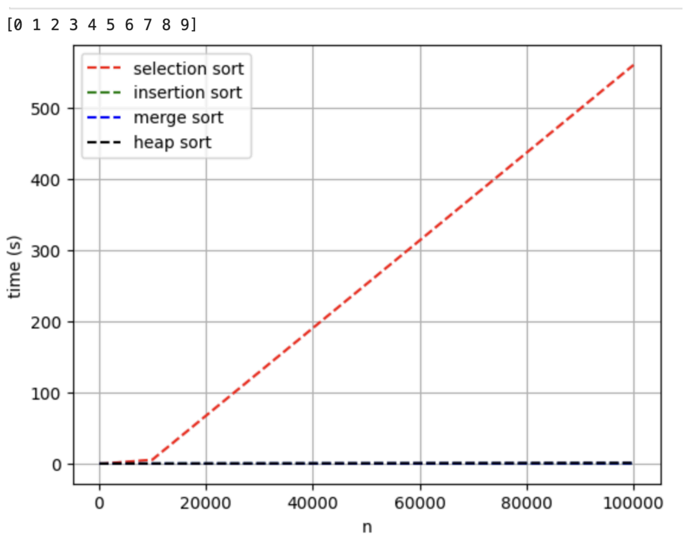
Summary (Q2)

Selection Sort is not adaptive, meaning it does not take advantage of existing order in the input data.
It is also not stable, because swapping elements can change the relative order of equal keys.

Q3 – Sorting Comparison and Time Analysis

In this experiment, four sorting algorithms (Selection Sort, Insertion Sort, Merge Sort, Heap Sort) were implemented and tested to compare their performance.

The input array size (n) was increased gradually, and the total execution time for each algorithm was measured using the time library in Python. The results were then visualized using matplotlib, as shown in the figure below. (It is the result of the Q3 code.)



```
[0 1 2 3 4 5 6 7 8 9]
```

As the graph shows, Selection Sort (red line) demonstrates a sharp increase in execution time as *n* grows. This is because it performs approximately $n^2/2$ comparisons, resulting in $O(n^2)$ time complexity.

Insertion Sort (green line) also shows quadratic growth, though slightly more efficient for partially sorted data.

In contrast, Merge Sort (blue line) and Heap Sort (black line) exhibit almost flat curves even for large *n*, confirming their O(n log n) performance.

These results empirically verify the theoretical time complexities:

Algorithm	Time Complexity	Performance
Selection Sort	$O(n^2)$	Slowest
Insertion Sort	$O(n^2)$	Moderate
Merge Sort	O(n log n)	Fast
Heap Sort	O(n log n)	Fastest / Stable

Summary (Q3)

The experiment confirms that for large datasets ($n \geq 10^4$), Merge Sort and Heap Sort are significantly faster than Selection Sort and Insertion Sort, both in theoretical and practical runtime performance.

Q4 – Priority Queue Real-World Applications

A priority queue is an abstract data type similar to a regular queue or stack structure, but each element is associated with a priority value. Elements with higher priority are processed before elements with lower priority. Priority queues are usually implemented using heaps, allowing insertion and deletion of the highest or lowest priority element in O(log n) time.

1. Task Scheduling (Operating Systems)

One of the most common uses of a priority queue is in CPU process scheduling.

Operating systems often manage multiple tasks simultaneously, where each process has a priority level.

A priority queue ensures that processes with higher priority are executed first.

For example, real-time tasks such as handling user input or system interrupts are given higher priority than background jobs.

Algorithms like Preemptive Priority Scheduling and Shortest Job Next (SJN) use this concept directly.

2. Graph Algorithms (Dijkstra and Prim)

In graph theory, priority queues are essential in algorithms like Dijkstra’s Shortest Path and Prim’s Minimum Spanning Tree.

In Dijkstra’s algorithm, a priority queue is used to repeatedly extract the vertex with the smallest tentative distance, allowing efficient pathfinding with a time complexity of O(E log V) when implemented with a binary heap.

Similarly, Prim’s algorithm uses a priority queue to select the edge with the minimum weight that connects a new vertex to the growing spanning tree.

3. Event-Driven Simulations

Priority queues are also used in discrete event simulations, such as modeling a banking system, airport operations, or network packet scheduling.
Each event (e.g., arrival, departure, completion) is scheduled with a specific time, and the event with the earliest timestamp is processed next.
The priority queue ensures that events occur in the correct chronological order without scanning the entire list of pending events.

4. Data Streams and Real-Time Systems

In real-time data systems or load balancers, priority queues help manage incoming requests efficiently.

For example, search engines or social media platforms often prioritize urgent or trending queries using priority queues.

Similarly, in network routers, packets with higher priority (like video streaming or emergency signals) are processed before next important packets, ensuring Quality of Service.

Summary (Q4)

Priority queues play a fundamental role in optimizing decision-making where order and urgency matter.

Their ability to efficiently manage elements based on priority makes them essential in operating systems, networking, simulations, and graph algorithms.

By implementing priority queues using efficient data structures like heaps, systems can achieve both speed and fairness when managing multiple competing tasks.