CMP2003 – DATA STRUCTURES TERM PROJECT REPORT

Group Name: Group_CGHN

Group Members:

- **Ronay Turap Üneşi 2367146**
- **Şiyar Tekdemir 2001605**

Project Name: City RideShare Trip Analyzer

The main objective of our project was to process large datasets (containing more than 5 million rows) within a limited time. To achieve this, the code we would write needed to be as optimized as we can. As expected in our project, our code had to O(1) mean time complexity and the least possible memory overhead. In our team, we considered and make strategies that how our code is work faster.

1. Design Justification

- **Choice of Data Structure (`std::unordered_map`):** We used (std::unordered_map) instead of (std::map for data collection). While (std::map) sorts the data using a Red-Black Tree (O(\log n)), we needed to develop a faster approach due to the large size of the data in this project. And we added unordered_map, which is based on a Hash Table, and provided an average time complexity of O(1) for searching, allowing our program to process this massive data as quickly as possible.
- **Composite Key Strategy: For the object of observe the most active hour slots, we considered a strategy of linking the Time and Zone ID of our data. As requested in our project, we avoided using inefficient nested maps (e.g., map<string, map<int, int>>), instead implementing a Composite Key by combining the Zone ID and Time into a single string. This straightforward structure significantly reduced memory allocation calls and pointer indirection.**
- **High-Speed Parsing:** Instead of using heavy and slow libraries like stringstream or regex, we performed manual string parsing using (std::string::find) and (std::string::substr). This approach allowed the program to directly jump to the necessary commas in the CSV line and extract the ZoneID and PickupTime columns with minimal CPU cycles. This optimization resulted in a 30ms execution time on the computer where we ran the program.

2. Complexity Analysis

The efficiency of our implementation is summarized in the Big-O table below:

Operation	Time Complexity	Space Complexity	Reason
Data Ingestion	O(N)	O(U)	N is the total number of rows. Each row is processed exactly once. U is the number of unique keys stored in memory.
Search/Lookup	O(1) (avg)	-	Hash map lookup provides constant time access on average.
Sorting	O(M \log M)	O(M)	M is the number of unique regions or slots. We used the std::sort code because it is suitable for our program to run in the most efficient and understandable way possible.

3. Tie-Breaking Strategy

To ensure deterministic and consistent results across different platforms (Test Category B), we implemented a multi-level tie-breaking logic using custom lambda comparators:

- **For Top Zones:**
  - Primary: **Trip Count** (Descending).
  - Secondary (Tie-breaker): **Zone ID** (Lexicographical ascending order).
- **For Top Busy Slots:**
  - Primary: **Trip Count** (Descending).
  - Secondary: **Zone ID** (Lexicographical ascending order).
  - Tertiary: **Hour** (Numerical ascending order).

The logic we used ensures that when two different regions have the same number of occurrences, the region that came first in alphabetical order always ranks higher, thus preventing ambiguity in the program's output and providing a regular sequence.

4. Robustness and Error Handling

Our implementation is designed to handle "Dirty Data" gracefully:

- Empty lines are skipped using `line.empty()`.
- Lines with missing columns are caught by checking `string::npos` during comma detection.
- Time formats are validated using `isdigit()` to ensure that only valid integers are converted into hours, and any value outside the 0-23 range is ignored.