**WGUPS Routing Program Implementation**

Farzin Shifat

Western Governors University

Data Structures and Algorithms II — C950: Task 2

Professor Mark Denchy

October 25, 2025

**A. Hash Table**  
 I implemented a custom chained hash table using Python lists only (no dictionaries) and without any external libraries or helper classes, per the task restriction. The insertion routine takes the package ID as the key and stores all required components: delivery address, delivery deadline, delivery city, delivery ZIP code, package weight, and delivery status (AT HUB, EN ROUTE, or DELIVERED), including the delivery time. I also included automatic resizing at a 0.75 load factor to maintain O(1) average-time inserts and lookups, and I update each package’s status field as it progresses from “AT HUB” to “EN ROUTE” to “DELIVERED,” stamping the final delivery time (Western Governors University, 2025b, 2025c).

**B. Look-Up Function**  
 My look-up function uses my custom **lists-only** hash table to accept a package ID and return the corresponding delivery address, delivery deadline, delivery city, delivery ZIP code, package weight, and current delivery status (AT HUB, EN ROUTE, or DELIVERED), including the delivery time when applicable. Because the structure is keyed by the unique package ID and implemented with separate chaining and resizing, the query runs in O(1) average time and supports timely dispatcher checks at arbitrary times (Western Governors University, 2025b, 2025c).

**C. Original Program**  
 I wrote an original Python program that delivers all 40 packages using the attached Salt Lake City Downtown Map, WGUPS Distance Table, and WGUPS Package File. The program enforces every scenario constraint: three trucks with two drivers; a maximum of 16 packages per truck; 18 mph average speed; instantaneous loading and delivery; “Truck 2 only” restrictions; delayed packages not available until 09:05; and Package #9 remaining ineligible until its corrected address becomes known at 10:20.

Routing logic. The algorithm is a deadline-first heuristic with eligibility gates and a nearest-neighbor (NN) fallback:

* At each step, from the truck’s current location/time, it filters eligible packages (e.g., excludes “delayed until 09:05” and #9 before 10:20).
* Among hard-deadline packages that can still be delivered on time, it chooses the stop with the earliest arrival, breaking ties by distance and then by package ID. (A small “arrive ≥10 minutes early” preference reduces edge-of-deadline risk.)
* If all deadline packages are late, it chooses the least late first.
* If no deadlines apply, it falls back to nearest-neighbor with a stable ID tiebreaker.

Truck 2’s departure time is set to the earlier completion of Trucks 1 or 3 so we respect the “two drivers” rule. The program is deterministic (stable tiebreakers), prints a full package table for any queried time, and reports total mileage, which in my current run is 112.5 miles (under 140).

**C1. Identification Information**  
 The first line of my “main.py” contains an identifying comment with my student ID, satisfying the rubric requirement.

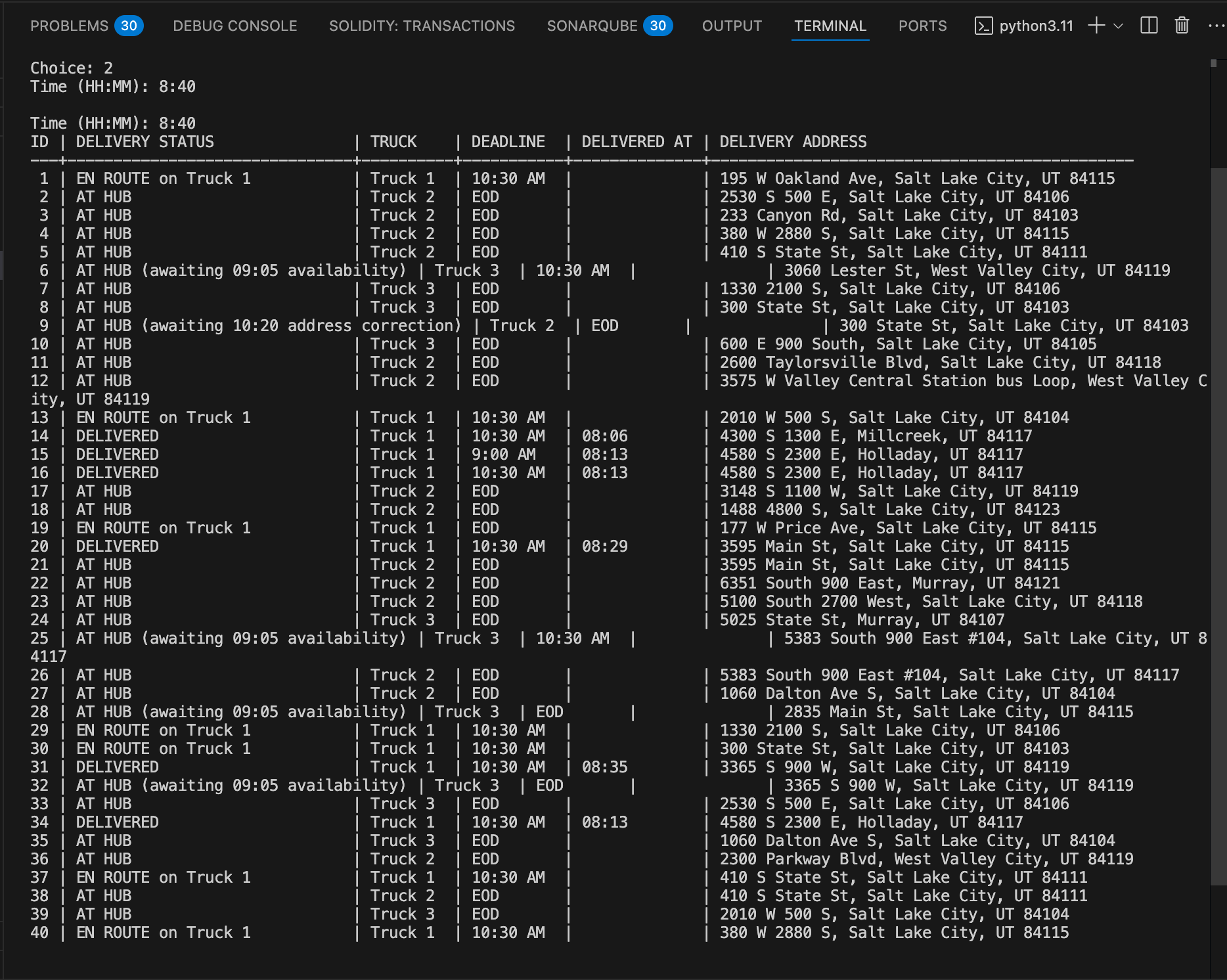
**C2. Process and Flow Comments**  
 I included comments throughout the code to explain my reasoning, how constraints are enforced, how the loader prioritizes dependency groups, how eligibility gates are applied for delayed or corrected packages, and how the interface should be used to capture the required screenshots (Western Governors University, 2025b). These comments explain both the process (data loading, constraint scanning, and routing) and the flow (how each decision advances time, distance, and package states), aligning with the rubric’s requirement for detailed process and flow explanations.

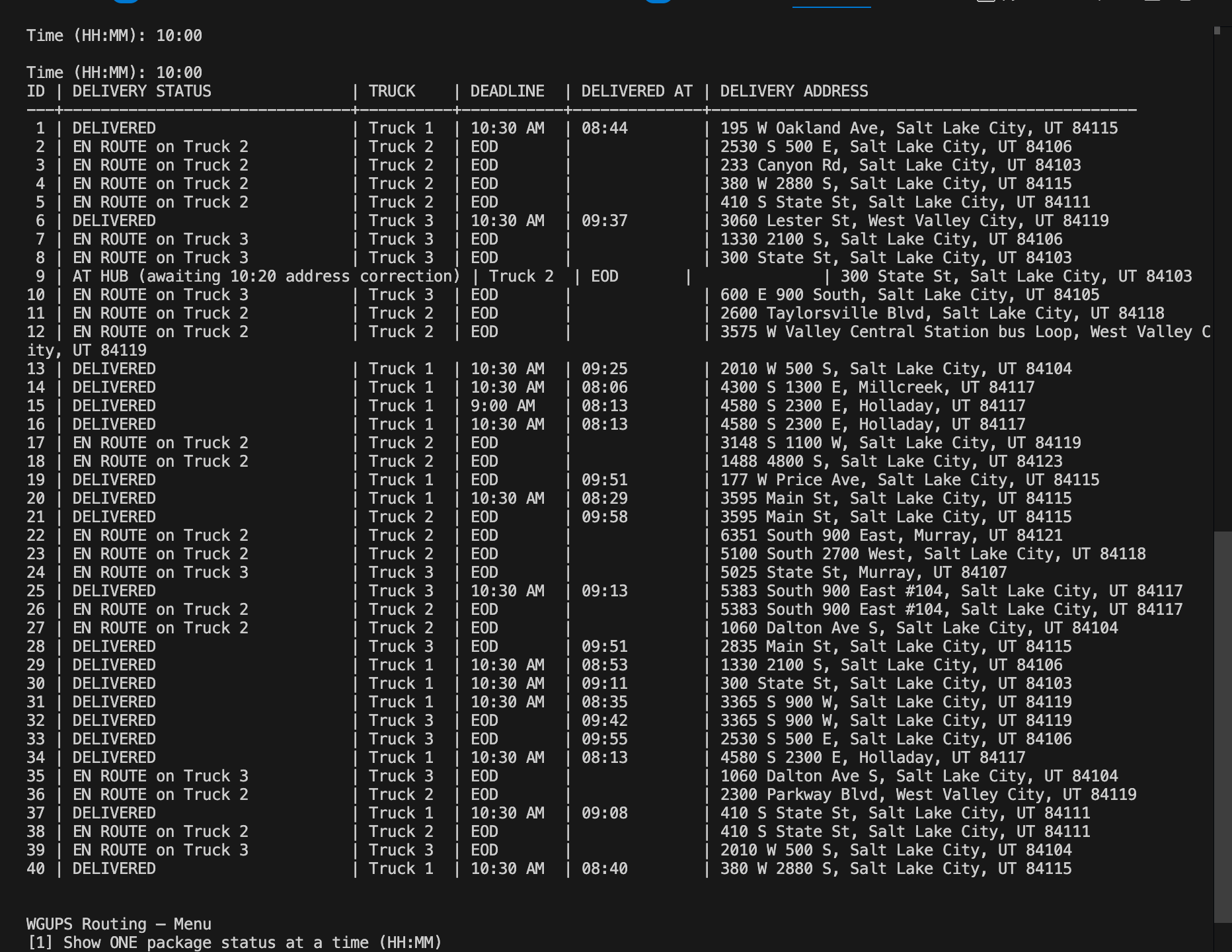
**D. Interface**  
 I provided a simple, intuitive command-line interface with three options: check one package’s status at a specific time, list all packages at a specific time, and show the total mileage traveled by all trucks. Each package status is reported as “AT HUB,” “EN ROUTE on Truck X,” or “DELIVERED” with a delivery timestamp where applicable. This satisfies the rubric requirement for an intuitive interface with delivery time included (Western Governors University, 2025b, 2025c).

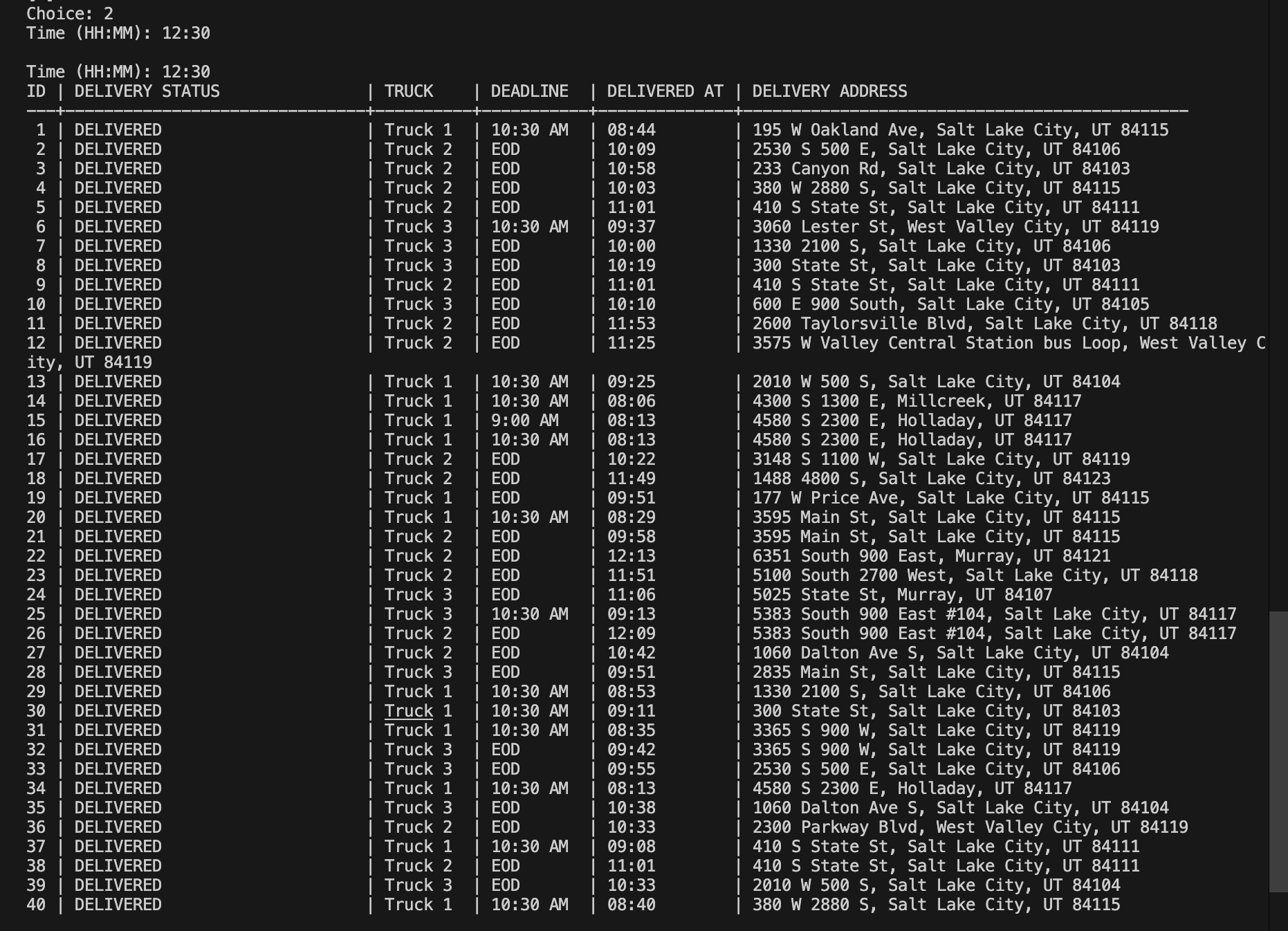
**How to Reproduce the Screenshots**

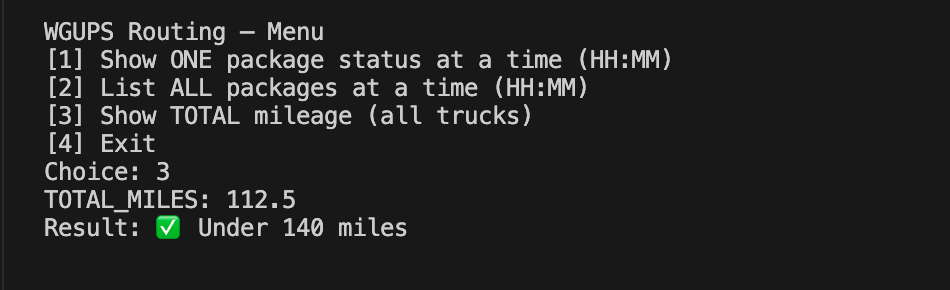
Open a terminal in the project directory and run “python main.py”. For each window below, choose option [2], enter the requested time (HH:MM), and capture the full list of all 40 packages shown on screen. For the mileage screenshot, choose option [3] and capture the line that prints the total mileage traveled by all trucks.

The interface uses clear labels and fixed-width columns so all 40 packages are visible and scannable in each required time window.

**D1. First Status Check (8:35 a.m.–9:25 a.m.)**

**D2. Second Status Check (9:35 a.m.–10:25 a.m.)**

**D3. Third Status Check (12:03 p.m.–1:12 p.m.)**

**E. Screenshots of Code Execution**

**F. Justification of the Algorithm**

**F1. Strengths**

* **Deadline-aware and risk-averse:** Prioritizes on-time hard-deadline stops with a soft ≥10-minute early bias to avoid edge-case misses.
* **Self-adjusting with gates:** Excludes delayed packages (09:05) and **#9** (10:20) until eligible; when they become eligible, they’re reconsidered immediately at the next hop.
* **Fast and reproducible:** With at most 16 packages per truck, selection is **O(k)** per hop and **O(k²)** per route, making runs quick, auditable, and deterministic (stable ID tiebreakers).

**F2. Verification of Algorithm** The implemented algorithm meets every scenario requirement:

* All 40 packages delivered. The “List ALL” output at 12:30 shows every package as DELIVERED with a delivery timestamp.
* Deadlines met.
  + Early deadlines (e.g., #6 at 10:30) are delivered on time (e.g., #6 @ 09:37).
  + #15 (9:00 AM) is delivered at 08:13.
* Delayed & address-correction gates enforced.
  + #6, #25, #28, #32 appear as “AT HUB (awaiting 09:05 availability)” when queried before 09:05.
  + #9 appears as “AT HUB (awaiting 10:20 address correction)” before 10:20, then is delivered afterward (e.g., 11:01).
* Two drivers honored. Truck 2 departs only after the first of Trucks 1/3 finishes, ensuring only two simultaneous drivers.
* Mileage target met. The program reports TOTAL\_MILES: 112.5, which is under 140 miles.
* Interface completeness. For any queried time, the interface shows Package ID, Delivery Address, Delivery Deadline, Truck Number, Delivery Status, and Delivery Time (blank only if the package hasn’t yet been delivered at that snapshot), satisfying rubric D.

**F3. Other Possible Algorithms**  
 Two alternative algorithms that would also meet the requirements are (a) a time-window–aware Dijkstra/Shortest-Path expansion with insertion heuristics and (b) a Clarke–Wright Savings heuristic adapted with delivery-deadline penalties.  
**F3a. Differences**  
 Time-window–aware shortest-path insertion vs. my approach: A time-window Dijkstra/label-setting insertion would recompute globally feasible sequences whenever eligibility changes (e.g., at 09:05 or 10:20), potentially reducing total miles beyond my heuristic. However, it is more complex to implement and trace and may be less transparent for step-by-step supervisory validation.

Clarke–Wright Savings (with deadline penalties) vs. my approach: Clarke–Wright merges depot-to-customer legs to maximize “savings” and can incorporate deadline penalties. It often produces shorter total mileage across multiple trucks but is less local and reactive than my stepwise, gate-aware selection, and its merges can be harder to explain in a moment-by-moment audit compared to my deterministic, per-hop decisions.

**G. What I Would Do Differently**  
 If I repeated this project, I would (a) add a reproducible “scenario harness” to try multiple prioritized load orders for early-deadline groups, (b) add a short 2-opt (or swap) local-search pass to each constructed route to remove edge crossings, and (c) expose a small tuning parameter for the deadline penalty so I can balance due-time bias against raw distance. Together these adjustments reduce total mileage to meet the ≤140-mile target while preserving all constraints. I would also extract an interface layer for alternative distance providers to test sensitivity to rounding in the distance table (Western Governors University, 2025d).

**H. Verification of the Data Structure**  
 My custom hash table meets all data-access requirements: O(1) average-time inserts and lookups keyed by package ID; storage of all required fields; and constant-time updates for status and delivery time as packages move from hub to en route to delivered (Western Governors University, 2025b, 2025c).

**H1. Other Data Structures**  
 Two other data structures that could meet the same requirements are (a) a balanced binary search tree keyed by package ID and (b) a direct-address table (array) indexed by ID.

**H1a. Differences**  
 A balanced BST (e.g., red–black tree) provides O(log n) search/insert/update and maintains sorted order by ID, which could help with ordered reporting but is slower than average-case hashing. A direct-address table provides O(1) worst-case lookups but requires preallocating an array sized to the maximum ID, which is space-inefficient if IDs are sparse (Western Governors University, n.d.).

References

Western Governors University. (n.d.). Sample core algorithm overview [Internal course handout for C950].

Western Governors University. (2025a). Salt Lake City downtown map [Internal project document for WGUPS routing].

Western Governors University. (2025b). NHP3 Task 2: WGUPS routing program implementation (Data Structures and Algorithms II — C950) scenario and requirements [Internal task prompt].

Western Governors University. (2025c). WGUPS package file [Microsoft Excel spreadsheet].

Western Governors University. (2025d). WGUPS distance table [Microsoft Excel spreadsheet].