**C950 Project Document**

**Section 1: Programming/Coding**

**A: IDENTIFY THE ALGORITHM**

To create this program a modified nearest neighbor (greedy) algorithm was used (see class method named “send\_out\_truck” in TruckManager.py). This algorithm, unlike a regular greedy algorithm, does not pre sort packages. The program will iterate through the list each pass to stay dynamic since it is also checking for approaching deadlines to appropriately reprioritize as needed.

**B1: COMMENTS ACCURATELY EXPLAIN LOGIC**

Comments have been written for every function and for some of the core algorithm functions there will be additional comments describing major logic blocks as well.

A high-level explanation of the application:  
1. Data is parsed from .csv files and stored into hashmaps

2. Trucks are loaded with packages  
3. The truck fleet is sent out to deliver packages

1. Each truck is assigned it’s departure time and sent out accordingly (accounting for package arrival at warehouse and driver availability)
2. Using a modified greedy algorithm the trucks find the closest available delivery location or if needed a delivery with an approaching deadline will be expedited.
3. Once the truck has zero packages left undelivered it will drive back to the hub

4. The report is compiled and printed to screen with results for each truck.

3.b. Is where the core algorithm is used in this program. This function is named “send\_out\_truck()” and resides in TruckManager.py. The pseudo-code for it is below:

**Def** send\_out\_truck(truck):  
 ***For each:*** package in a given truck

***If:*** any packages need to be delivered to the “current” location  
 ***Then*** go ahead and drop them off.  
 ***Else If:*** any delivery deadlines are approaching.  
 ***Then*** complete the delivery found with an approaching deadline

***Else:***

***Then*** Delivery the package that is closest to the current location

The truck returns to the hub

**B2: APPLICATION ARCHITECTURE**

Considering that all the needed information was read from .csv files located within the project directory there were no communication protocols or defined interaction semantics used in this program. The program was written in python using the pyCharm IDE. All data was stored in memory using hash tables and queried from there as needed. Every dependency for this application can be housed on a local machine and run independently from a host server.

**B3: TIME COMPLEXITY**

All methods are annotated with their respective Big-O notation within the source code. The upper bound for the entire application is O(N^2). This is because the greedy algorithm that is the core algorithm runs with an upper bound of O(N^2) or quadratic time. No other function in this project runs with a space-time complexity greater than quadratic time.

**B4: APPLICATION SCALABILITY**

The core algorithm used in this application is scalable considering that its time complexity is quadratic time or O(N^2). The core algorithm uses heuristics to provide timely delivery with optimal drive distance. This algorithm could easily handle many more deliveries given that there were more trucks to meet all the deadlines.

Although the core algorithm can scale easily the logic for loading the trucks and managing truck departures based on driver availability will not scale considering that the application is tightly bound to the constraints and assumptions given for this project. In other words, this program only handles 3 trucks, 2 drivers and can only load 40 packages in its current state.

**B5: EFFICIENCY & MAINTAINABILITY**

Using heuristics, the difficulty of this problem was reduced to being solvable in quadratic time. The slowest function in this program runs with an upper bound of O(N^2) or quadratic time.

The logic is separated into class files to help simplify the application using object-oriented design. Every function is commented with its intended purpose and core functions have additional comments that annotate some of the more complex logic within them. This project should be considerably readable and easy to maintain or add onto.

**B6: SELF-ADJUSTING DATA STRUCTURES**

Within this application a custom Hashmap data structure was used to store package and distance data. The package data was read into a one-dimensional hashmap for constant-time read and write operations on package data while the distance data was stored into a two-dimensional hashmap or matrix for constant time-distance calculations. Both of these hashmaps create a hash for any key passed in through a read or write operation and uses that key hash to access a specific index in constant time.   
  
The only two downsides to this data structure were that it could be difficult to manage collisions and read data while avoiding array indexes with no stored values. Although these complexities could be hidden behind an interface for this class, writing new methods for this data structure often required additional logic to manage the complexities of unused hash indexes and hash key collisions that were stored in the same index as sub-elements in an array.

**C: APPLICATION MEETS ACCEPTANCE CRITERIA**

The application delivers all packages on time and according to the constraints defined with a total mileage of 133.3 and with the final truck returning back to the hub at 2:03 PM.

(Seen on the report from the main menu: option 1 “Complete entire day of deliveries (See report)”)

**C1: INITIAL COMMENT w/IDENTIFICATION INFORMATION**

Name and student id are located at the top of the file in main.py.

**C2: COMMENTS EXPLAINING PROCESS AND FLOW**  
Comments have been added to each function in the project with additional comments for some of the core methods that may be slightly more complicated.

**D: DATA STRUCTURE IDENTIFICATION**

A hashmap was created to be used alongside the greedy algorithm. This allows for constant-time read and write operations. This hash map data structure was implemented using only primitives in the python language (excluding the dictionary type). (See HashMap.py for implementation)

**D1: DATA STRUCTURE EXPLANATION**

To store the package data into the hash map data structure I create the PackageEntity interface to convert the package data from the .csv file into in-memory objects. These objects are stored into the hash map using their package id value as the key. (See PackageEntity.py for implementation of the interface)

The hash map data structure was also used to create a matrix for the destination data. This was done by creating a two-dimensional hashmap and using the to and from destinations as an x and y coordinate pair for constant time distance loop-ups. (See distance\_table\_to\_hash\_map() function in CsvHelper.py for creation of matrix using the hashmap).

**E: HASH TABLE INSERTION METHOD**

The hash table has an “add” method that performs the insertion operation in constant time. This method will receive any object, but for the package hash table, the PackageEntity object is used to organize the package information and then store the entire object into the hash table with the package id being used as a key. (See package\_table\_to\_hash\_map() method in CsvHelper.py for usage)

The PackageEntity interface organizes/stores the following values:

1. package\_id
2. delivery\_address
3. delivery\_city
4. delivery\_state
5. delivery\_zip
6. delivery\_deadline
7. package\_weight
8. special\_instruction

**F: HASH TABLE LOOK-UP METHOD**

The hash table uses a “get” method for look-up operations. This method uses the key and returns the value in the hash table. For the package table, this would be the package id. When the get method is called using the package id the entire PackageEntity is returned from the hash table  
(See the “load\_truck()” method in TruckManager.py for an example of the look-up method being used.)

**G: USER INTERFACE**

A user interface has been created that has 3 available options.   
 Options:

1. Complete entire day of deliveries

2. See status of deliveries at a specified time

3. Quit program

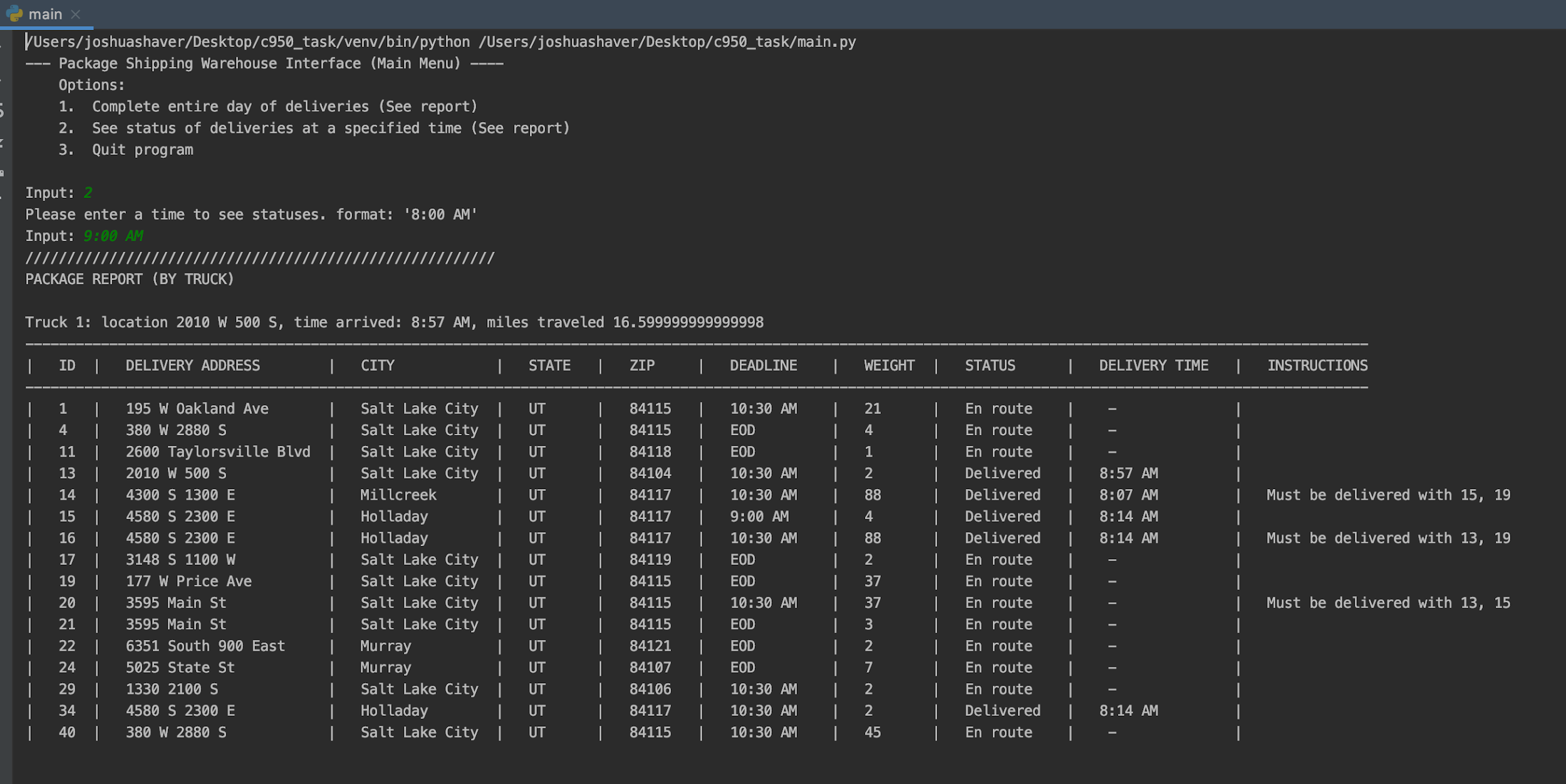
Option 1 will just run the program for the whole day and print the outcome of the program.

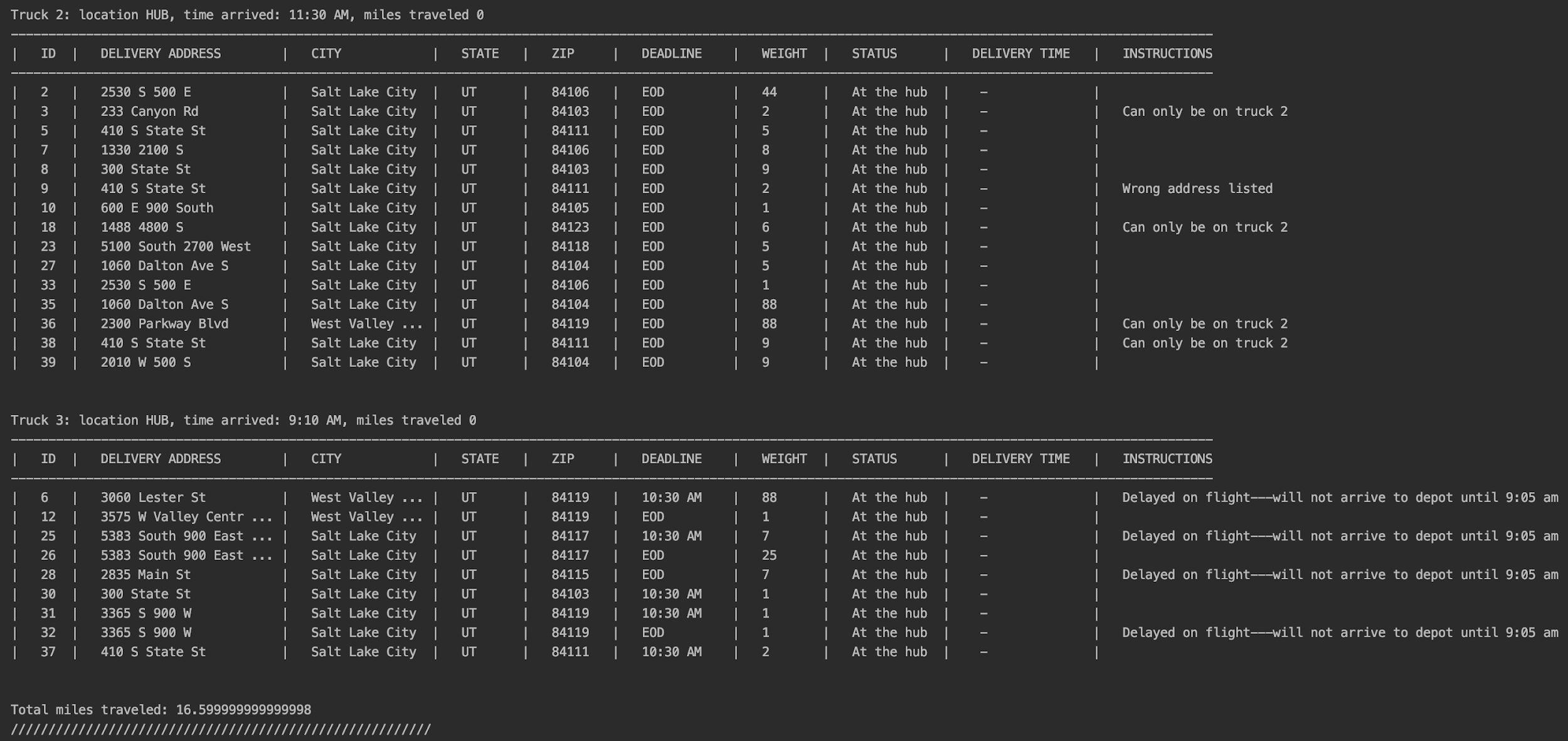
Option 2 can be used to see the program’s progress at any point in time. After selecting option 2 the program will prompt for a time and requires a format like “9:00 PM” (HH:MM AM/PM) to be entered.

Option 3 simply exits the program.

**G1: SCREENSHOT (*8:35 a.m. and 9:25 a.m.*)**

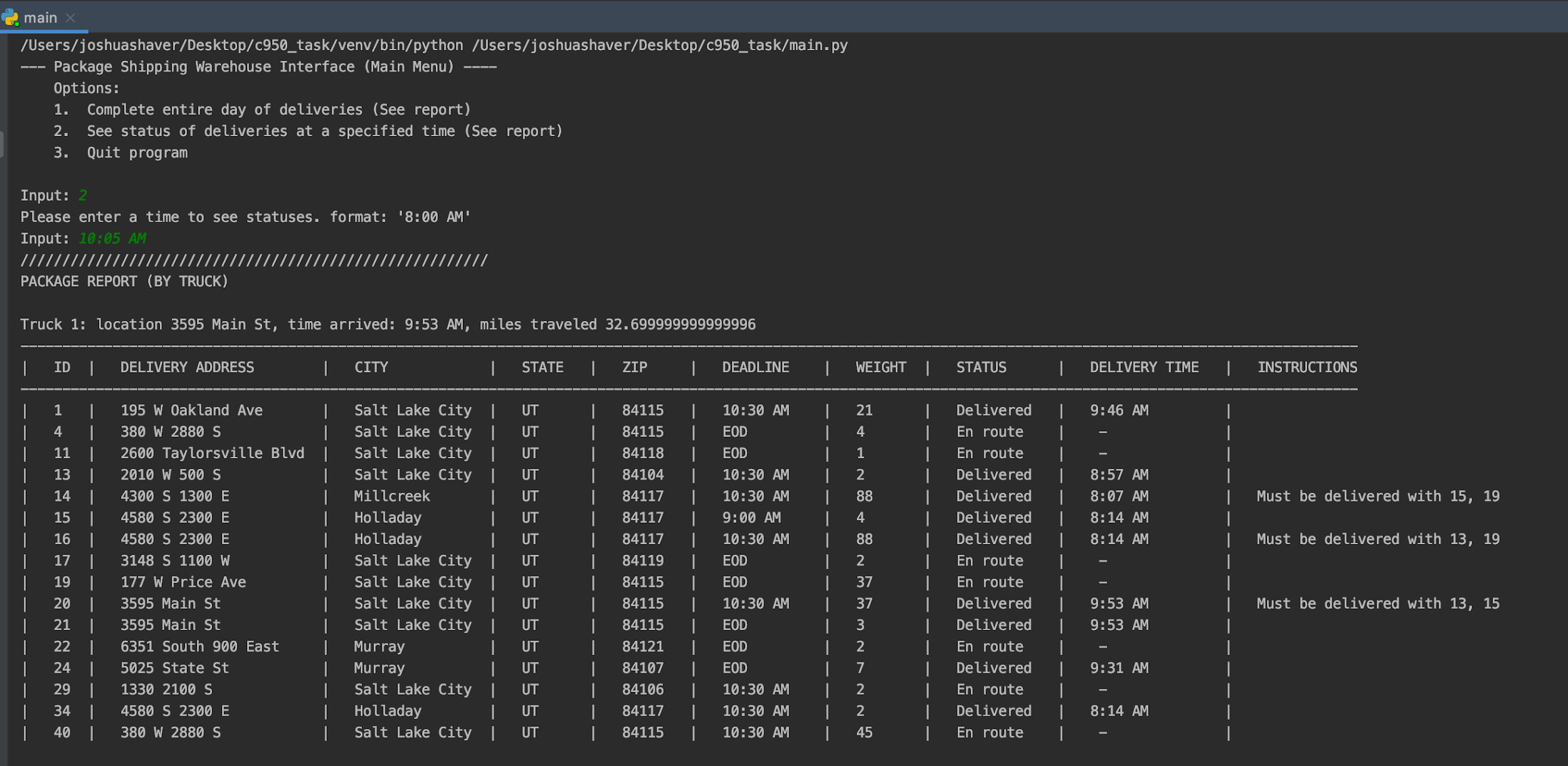
Ran for 9:00 AM



****

**G2: SCREENSHOT (*9:35 a.m. and 10:25 a.m.*)**

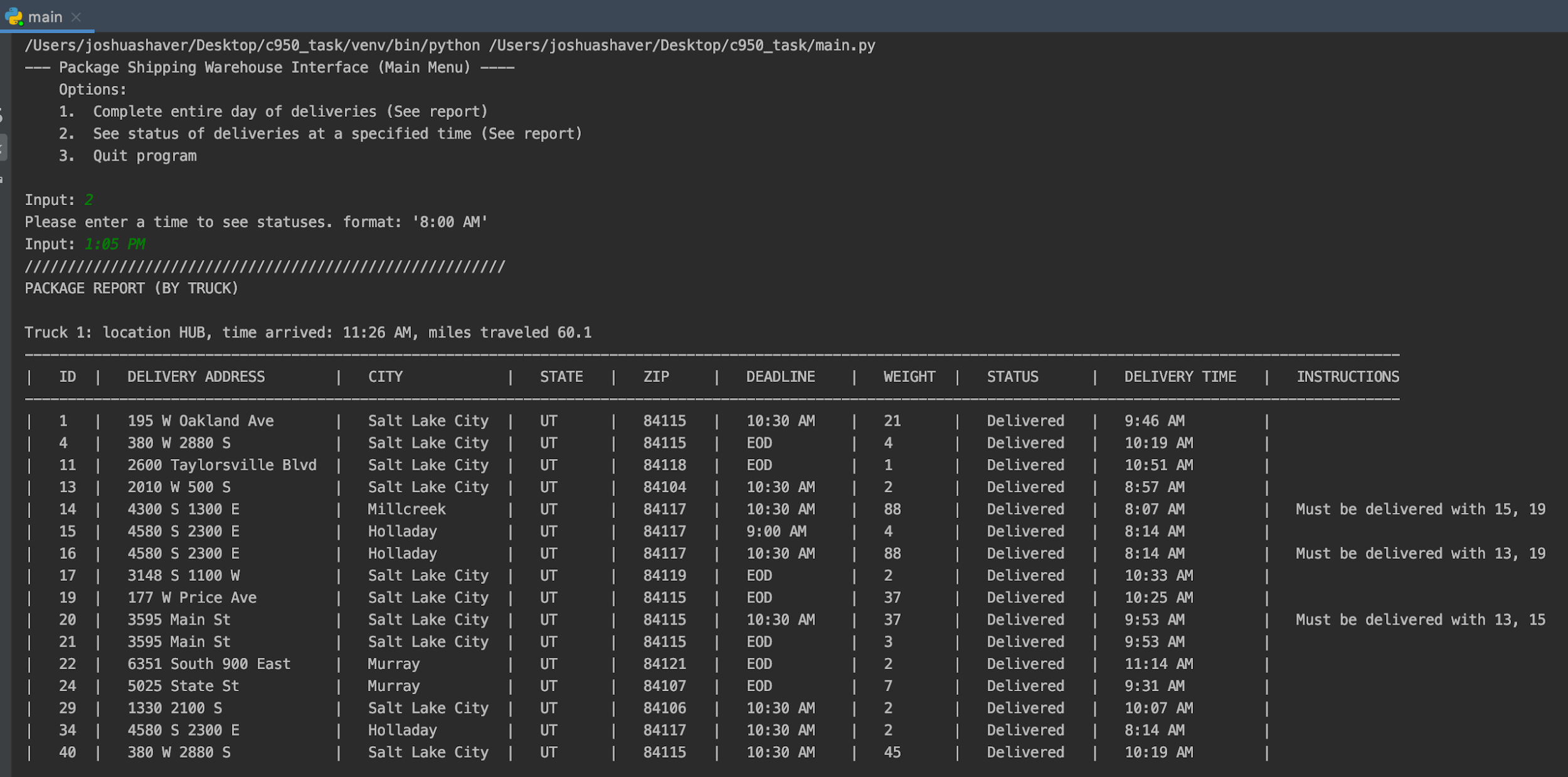
Ran for 10:05 AM

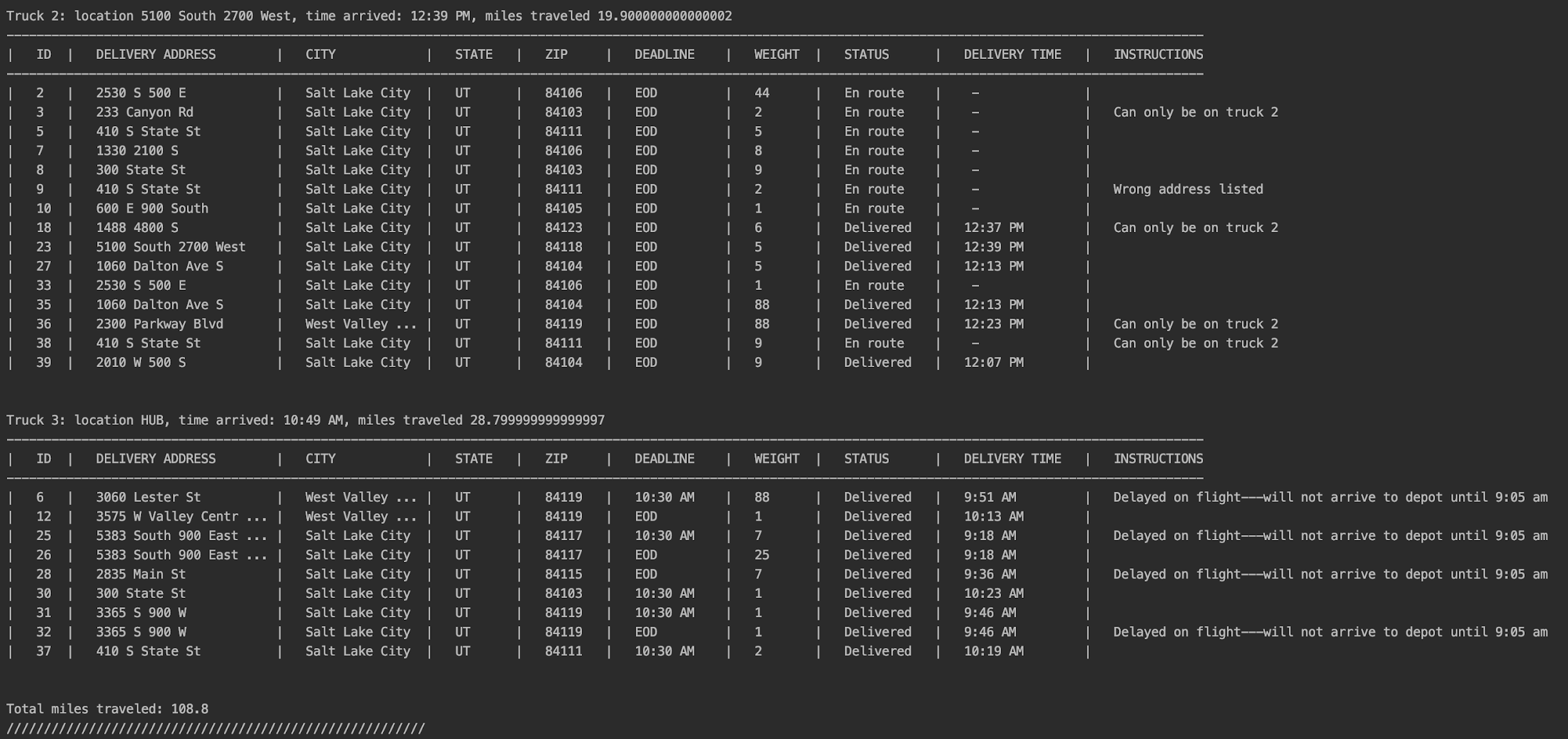
****



**G3: SCREENSHOT (*12:03 a.m. and 1:12 p.m.*)**

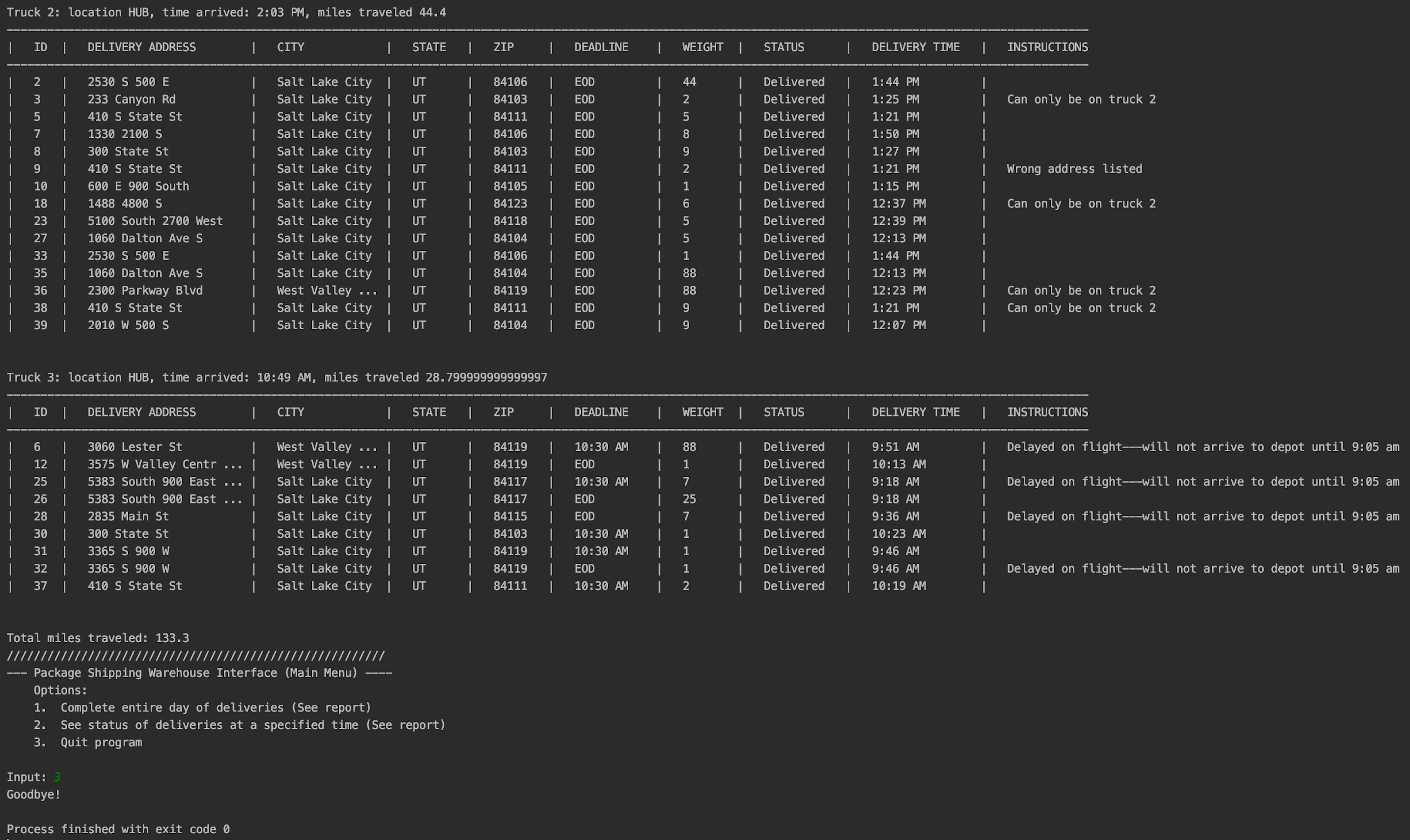
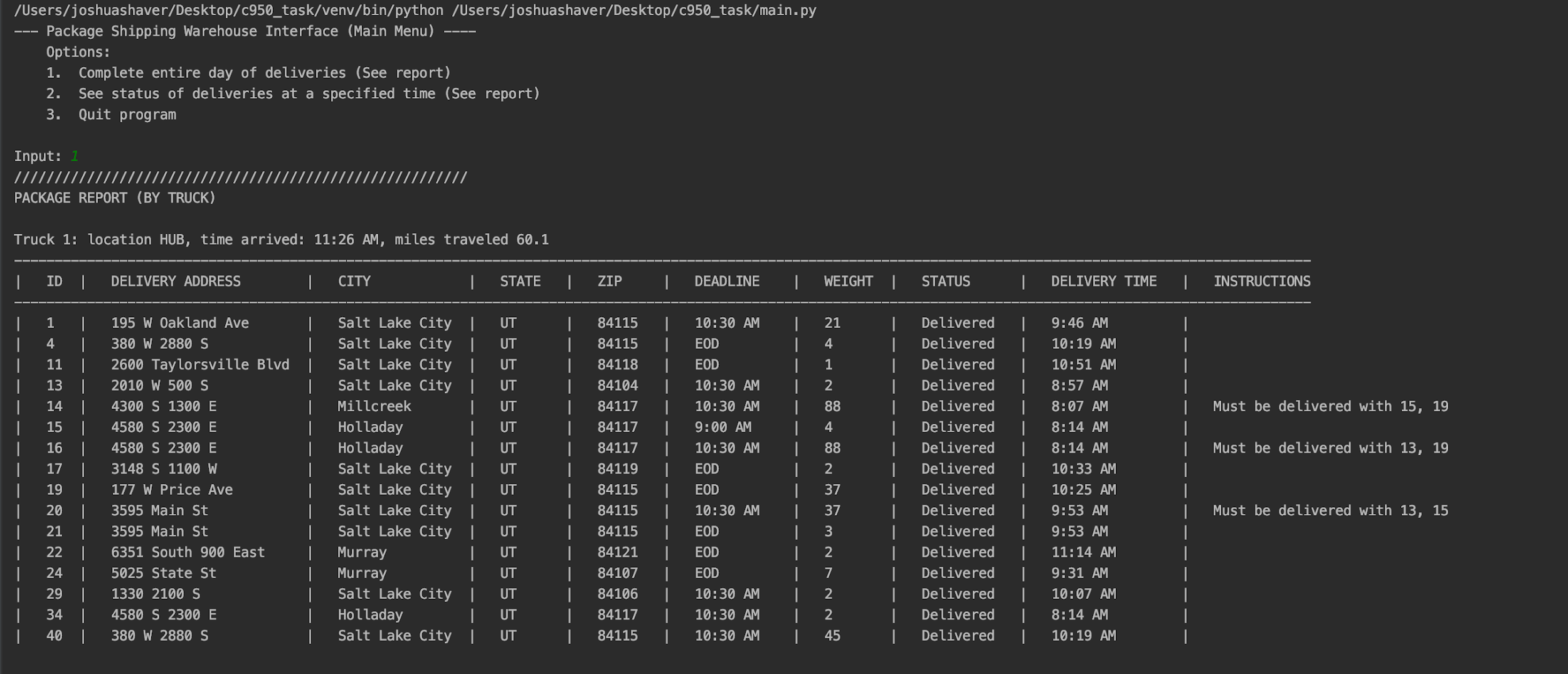
Ran for 1:05 PM

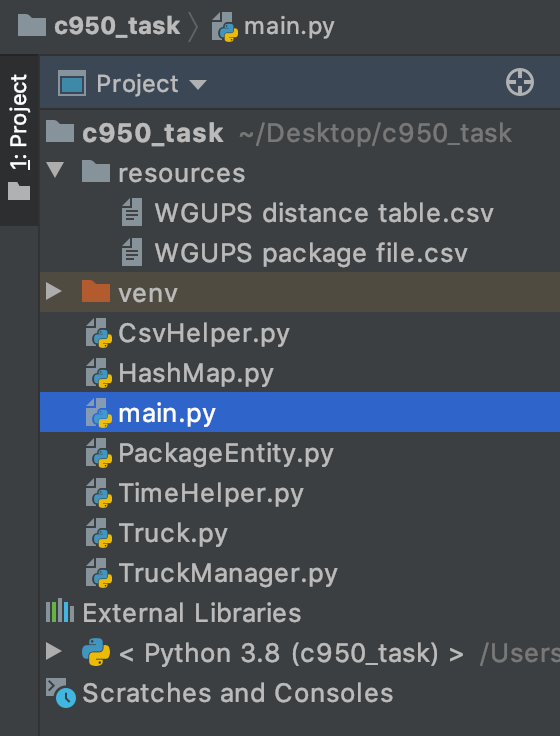
****

****

**H: SCREENSHOT (*successful code execution*)**

Completed in 133.3 miles

****

Screenshot of project file structure: ****

**Section 2: Annotations**

**I1: TWO STRENGTHS OF THE CHOSEN ALGORITHM**

The chosen algorithm was a modified greedy algorithm. The modification primarily lies in the fact that it will sometimes make a less “optimal” decision to get a different package delivered by a specified deadline. The benefits of this algorithm are its heuristic approach and its logical simplicity. This algorithm is able to reduce the complexity of this problem to O(N^2) by using heuristics and the simplicity of the algorithm makes it very easy to maintain and build upon or modify.

Two strengths:  
- heuristic approach (reduces problem complexity from O(2^N) to O(N^2)

- The simplicity of the logic makes it easy to modify and build on top of.

**I2: VERIFICATION OF THE CHOSEN ALGORITHM**

Using the User interface take option one to run the program to completion. The end result is 133.3 miles. Printed at the bottom of the report is the following string after taking option one on the main menu.

“Total miles traveled: 133.3”

This report will also show that all deadlines were met and special instructions/constraints were followed.

**I3: IDENTIFY TWO OTHER POSSIBLE ALGORITHMS**

Two other algorithms that could possibly solve this problem would be a branch and bound (BB) algorithm which is an exact algorithm, and a minimum spanning tree (MST) algorithm which is another heuristic algorithm

**I3A: ALGORITHM DIFFERENCES**

The chosen algorithm, nearest neighbor (NN) aka the greedy algorithm, is a heuristic algorithm.

Although the minimum spanning tree (MST) algorithm is also a heuristic algorithm it uses a weighted graph and finds a subtree to find an optimal solution. The MST algorithm would probably work just as well in this scenario as the greedy algorithm in terms of scalability.

The branch and bound (BB) algorithms are quite different in approach from NN and MST since it is not a heuristic algorithm but an exact algorithm that attempts to find a classical solution. In a worst-case scenario the BB algorithm is really just an exhaustive search. This algorithm does not scale as well as the heuristic ones, but it could solve a problem with 40 packages more precisely than the other two algorithms NN and MST.

**J: WHAT COULD BE DONE DIFFERENTLY**

Currently, the program has a TruckManager class that acts as a singleton object, and it acts as an orchestrator for the delivery trucks. If this were to be re-done it would probably be wiser to move the time calculation logic into this singleton class instead of in the individual truck objects. Having the time calculations within the truck objects made it difficult to orchestrate actions that were time-constrained between different truck entities i.e. sending the third truck out when a driver returned to the hub from another truck’s route.

**K1: VERIFICATION OF THE DATA STRUCTURE**

Using the User interface take option one to run the program to completion. The end result is 133.3 miles. Printed at the bottom of the report is the following string after taking option one on the main menu.

“Total miles traveled: 133.3”

This report also lists all the package object values that were stored in the hash table data structure.

**K1A: DATA STRUCTURE EFFICIENCY**

This application utilizes the hash table data structure for two data sets. The first dataset is the package data. This hash table provides O(1) -- constant-time operations for reading and writing to it. Without the hashing mechanism, every read and write operation would be O(N) -- linear time.

The second data set that this hash table is used for is a two-dimensional hash table for storing the location distance information. It does take O(N^2) operations to initially create this hash table from the provided distance data, but it can be utilized later for O(1) -- constant time distance loop ups from location to location. This lookup is occurring O(N^2) times for every package in this program. So the initial overhead pays off when the number of reads on this dataset is accounted for.

**K1B: DATA STRUCTURE EXPLANATION**

The hash table data structure performs read and write operations in constant time O(1). This does incur a very slight overhead during the creation of the hash table itself considering that the memory has to be allocated at that time. In this program, the memory requirements for the hash tables are nominal considering that the hash table does not need to exceed forty indexes to store the entirety of the dataset. That being said the overhead might be something to consider if the application were to need to scale immensely. No bandwidth will be needed for the data structure considering that everything required for runtime is available on the local machine.

**K1C: DATA STRUCTURE SCALABILITY LIMITATIONS**

The hash table data structure would require a different hash function that could create a unique hash more reliably if the number of packages grew by any order of magnitude.

Considering that a two-dimensional version of this hash table is used for distance information if the number of delivery addresses grew substantially the creation of that data structure would cost O(N^2) operations where N is the number of possible delivery locations. That being said if this delivery location list increased substantially, then it would probably be better to store this two-dimensional distance map in some sort of persistent memory like a database where it does not need to be created at runtime.

If more trucks were added to this operation the TruckManager singleton could move from a hardcoded 3 truck to a list of trucks. An additional heuristic algorithm for loading the packages would be beneficial for that as well.

**K2: IDENTIFY TWO OTHER POSSIBLE DATA STRUCTURES**

Two other data structures that could be used would be a binary search tree (BST) for the package data or possibly a weighted graph for the distance data.

A binary search tree could certainly be an option for storing the package data. Where a standard list would require linear time O(N) to find a specific element, BST would only require O(logN) time.  
  
For the weighted graph, each location could be adjacent, a fully connected graph and the edges would have values according to the distance to each adjacent node. From this list of values, one could determine the nearest neighbor and perform a similar greedy algorithm pretty easily.

**K2A: DATA STRUCTURE DIFFERENCES**

Currently, a one-dimensional hash table is used for the package data and a two-dimensional hash table is used for the distances between every location.   
  
When comparing a BST to the hash table for package data storage, the hash table would give constant-time read and write access while the BST would require O(logN). This means that the hash table is more suited for the package data in this scenario.

Now comparing the weighted graph to the two-dimensional hash table used for the distance data. The graph would operate in much the same way as the hash table, but without further optimization, it would require linear time O(N) to perform a read where the hash table can perform a read operation at constant O(1) time.

**L: SOURCES**

No sources cited.