Warren Harper - C950

Student ID: 001326657

**A: IDENTIFY THE ALGORITHM THAT WILL BE USED TO CREATE THE PROGRAM**

The two underlying algorithms are Dijkstra's algorithm and a nearest neighbor greedy algorithm. Dijkstra's is used to calculate the shortest route between two points. The greedy algorithm uses Dijkstra's in order to determine the closest neighbor and from that constructs a route for the truck.

**B1: COMMENT USING PSEUDOCODE TO SHOW THE LOGIC OF THE ALGORITHM APPLIED TO THIS SOFTWARE SOLUTION.**

The distance table is parsed and stored in a weighted graph data structure.

The package information is parsed and stored in a hashtable data structure.

Packages are selected by determining the nearest package (using Dijkstras) to the current location and adding it to a list which will make up the delivery route.

**Dijkstra’s Algorithm - Shortest path:**

input : start node, destination node, graph

Set start node to zero and all other nodes to arbitrarily high value

Visit each node from start node and set shortest distance from previous node according to graph weight.

Move through each node and update node if sum of weights is smaller than current weight.

Output : path route and sum of shortest path

**Greedy algorithm - route selection:**

Input : start, destination, graph

Iterate through package list:

Use dijkstras to select the nearest package by address

Add package to list

Update current location to address of current package

Output : list of packages in order to be delivered

**B2: DEVELOPMENT ENVIRONMENT**

Hardware: MacBook Pro 15 inch 2018

Operation system: macOS 11.3.1

IDE: PyCharm Professional 2021.1

Language: Python 3.8.2

**B3: SPACE-TIME AND BIG-O**

Entire program: O(n^2)

Hashtable: Add, set, retrieve, delete item O(1)

shortest\_route(dijkstras): O(n^2)

calculate \_route(greedy): O(n^2)

Sort\_packages, deliver\_packages, fill\_truck: O(n)

**B4: ADAPTABILITY**

The package data are parsed, and the constraints are met in an automated way so that if more packages were added to the CSV file, the program would automatically process them.

The weighted graph structure is constructed at run time from data parsed from the CSV file, therefore if more destinations were added to the CSV file or a completely different set of addresses used, the program would still find an optimal route.

The Hashtable data structure has a resizing function so it can handle an infinite (within reason and memory constraints) amount of packages.

Shortest paths and delivery routes are calculated at runtime using diijkstras and a greedy algorithm, therefore the program can adapt to different destinations or sets of addresses.

**B5: SOFTWARE EFFICIENCY AND MAINTAINABILITY**

The program runs in O(n^2) which would be considered efficient for the TSP problem.

The program is split up into multiple files with the main.py file being short and succinct, and therefore easy to understand. All files contain multiple comments in order to help someone looking at the code for the first time. The main hashtable data structure has its own file as well as the package object and algorithms. All files use easy to understand naming conventions.

**B6: SELF-ADJUSTING DATA STRUCTURES**

The hashtable is a self adjusting table structure. It handles collisions using the chaining method via lists. It also has a load factor variable which calculates how full the table is and resizes the table (increases the size of the array) if need be.

Strengths:

Data setting, retrieval, deletion is executed in constant time

Data structure can handle an infinite amount of objects (within reason)

Data structure handles collisions

Weaknesses:

Hashing function is only optimized for unique integers.

If hashing function is not optimal, the chaining method could lead to long lists within the array which would cause execution time to be linear instead of constant.

**D1: EXPLANATION OF DATA STRUCTURE**

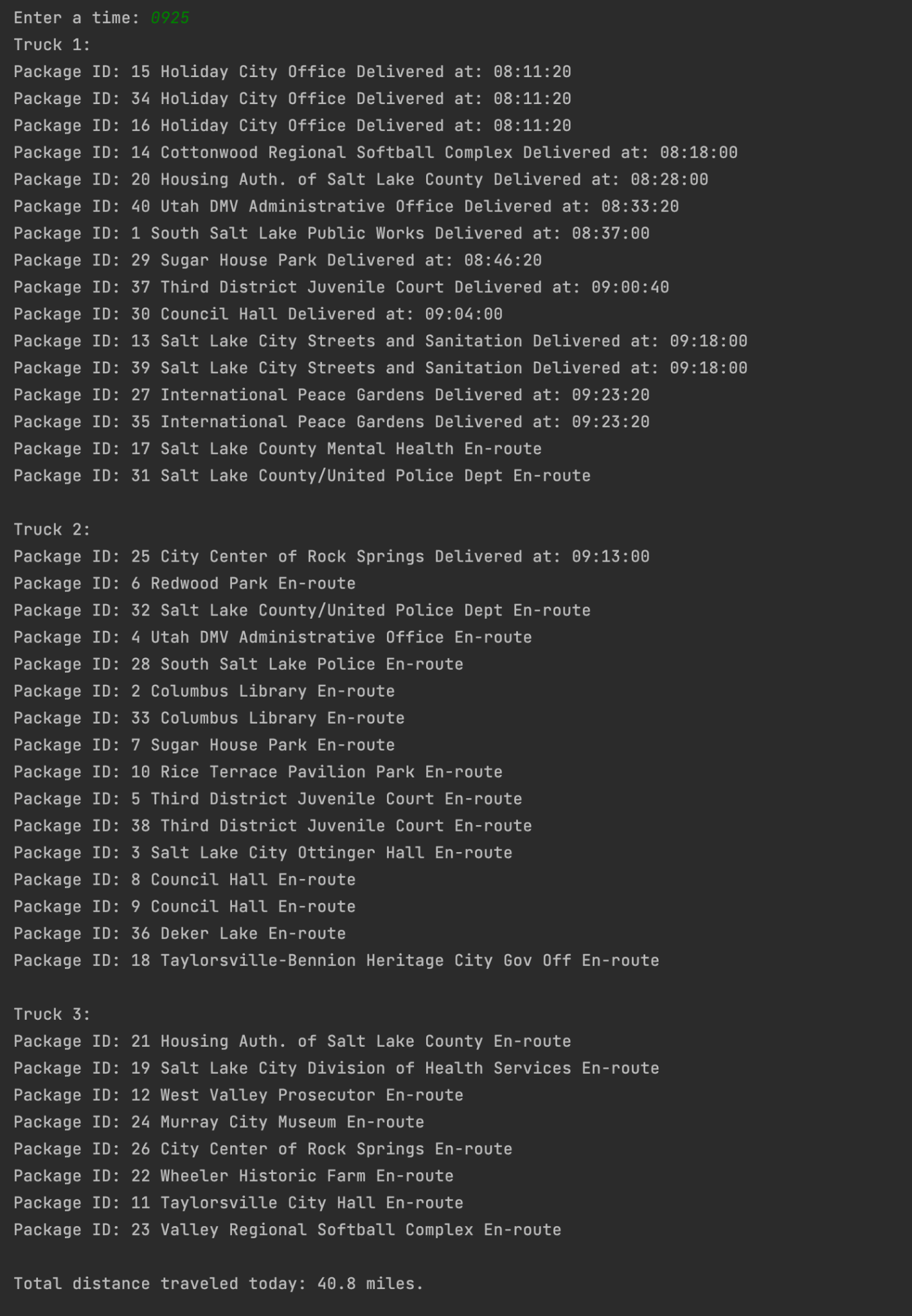
The hash table creates an array of lists which hold objects. A simple hashing function takes the key, in this case,a unique integer ID, and converts it into an index in order to access the list. The list is able to store multiple objects (packages) in case there are any duplicate indexes produced by the hashing function. This is a form of collision control known as chaining.

The hash table also has a resizing function based on load factor. Load factor is calculated by dividing the number of elements by the size of the current table. Once the load factor exceeds a load of one, the table is resized.

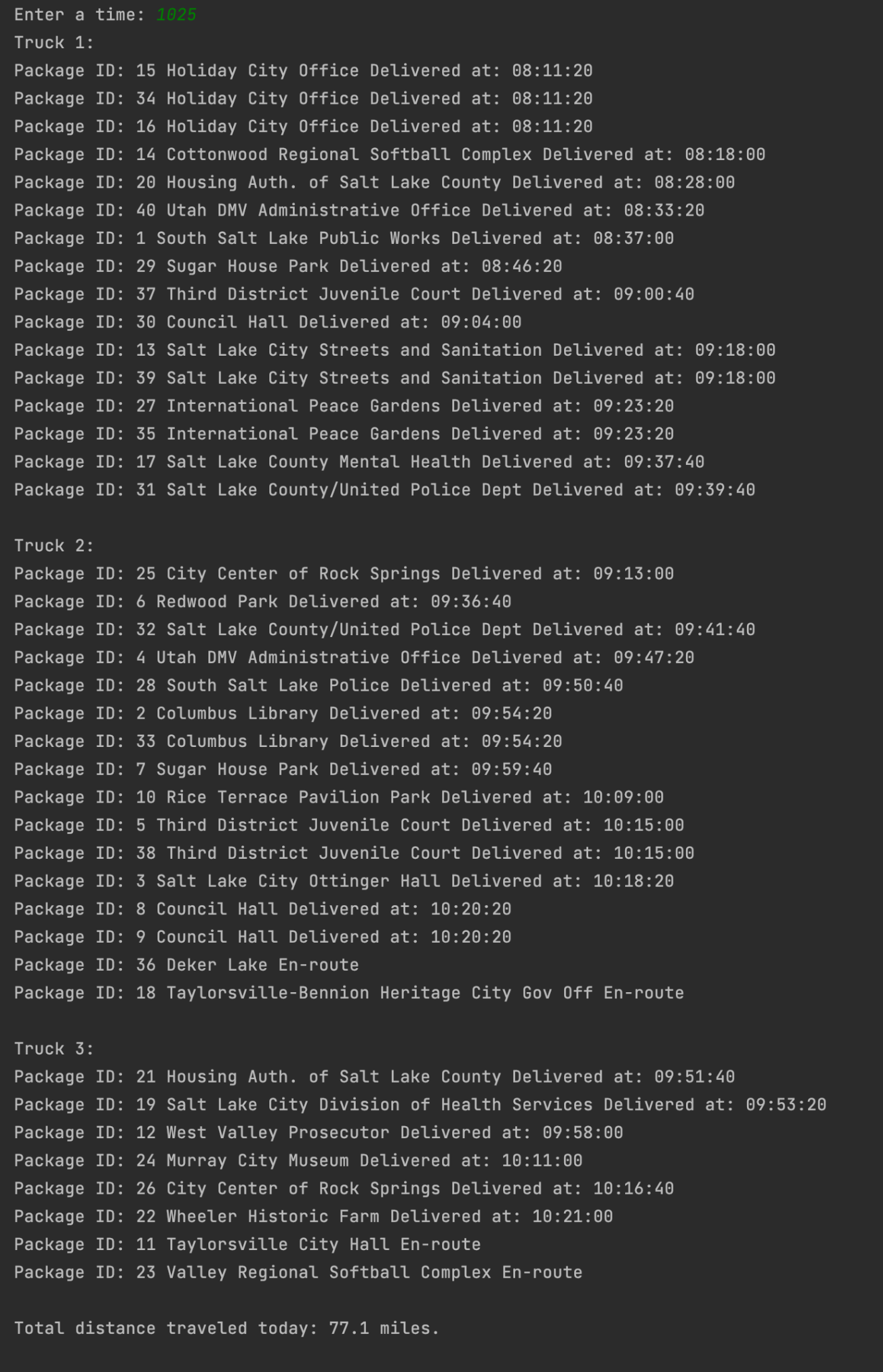
The hash table uses the key value to access the corresponding object in constant time versus a list which will need to be iterated through and has a linear access time o(n).

**G1-G3: 1ST, 2ND, AND 3RD STATUS CHECKS**

* *8:35 a.m. and 9:25 a.m*

**

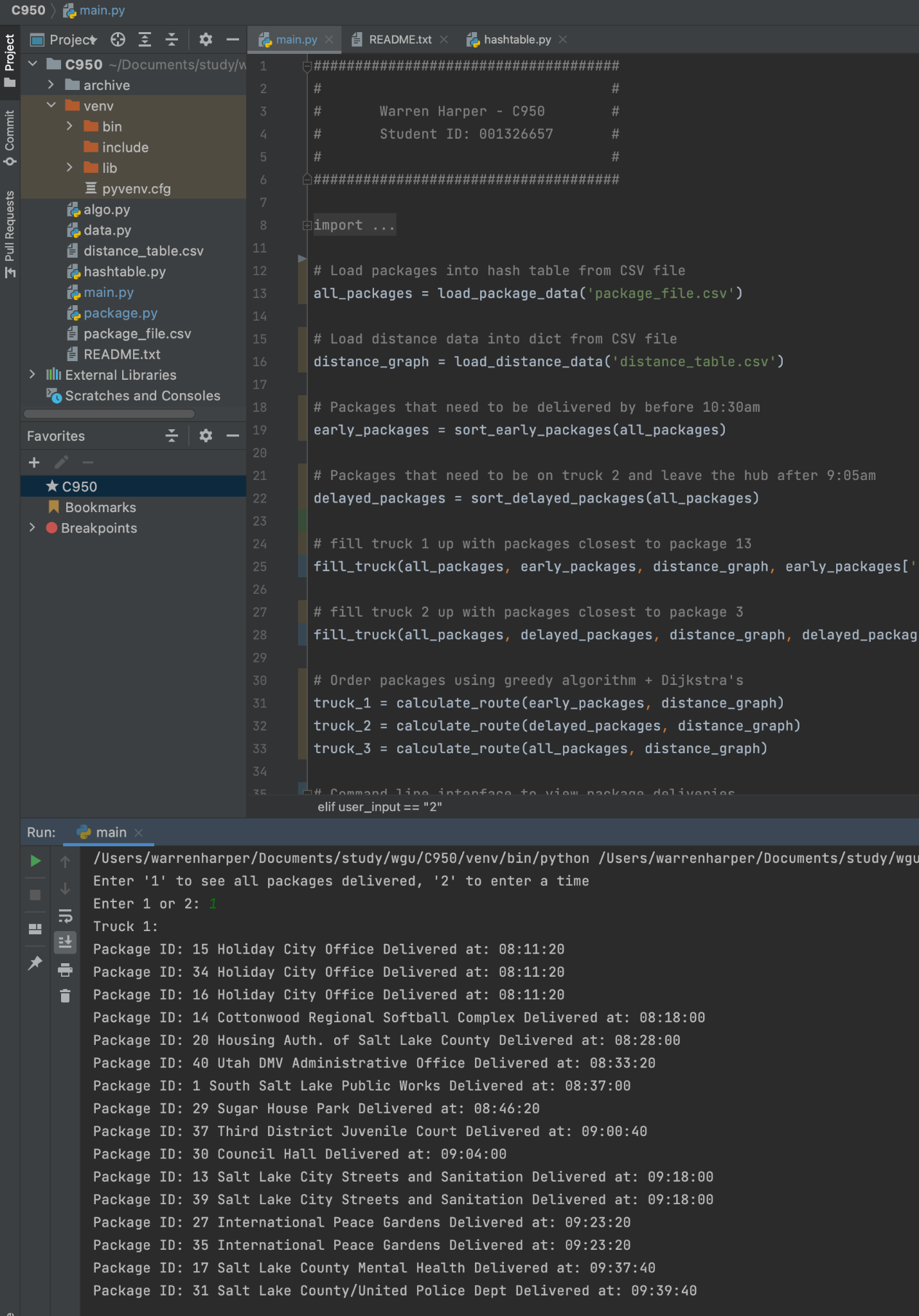
* *9:35 a.m. and 10:25 a.m.*

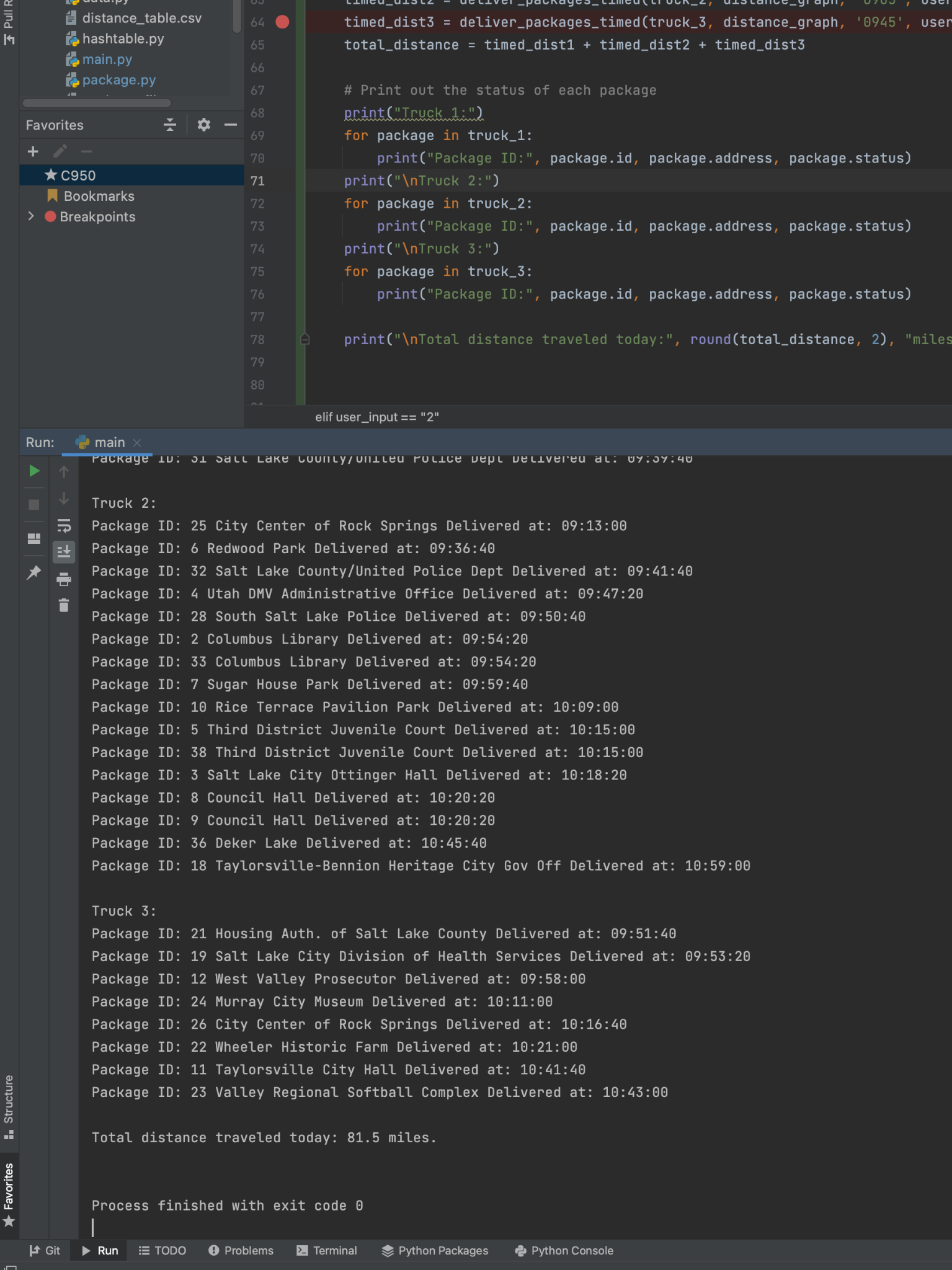
**

* *12:03 p.m. and 1:12 p.m.*

**

**H: SCREENSHOTS OF CODE EXECUTION**

****

****

**I1: STRENGTHS OF THE CHOSEN ALGORITHM**

***shortest\_route(dijkstras):***

Strengths:

Well know algorithm

Easy to understand

Used in multiple parts of the program, not just final route

Finds optimal routes in distance table where direct route is not necessarily the shortest

Weaknesses:

Adds complexity to the program by adding in an extra step instead of just using distance values

Time complexity is O(n^2), large data sets could make the program exponentially slower

***calculate\_route(greedy):***

Strengths:

Easy to understand and maintain

Small amount of code

Weaknesses:

May not produce the optimal route

Relies on another algorithm(Dijkstras) to calculate nearest neighbor

**I2: VERIFICATION OF ALGORITHM**

Total miles driven: 81.5 miles

All packages were delivered according to their constraints and specifications by 10:59AM

Run the program and type ‘1’ in the console to see a print out of all the packages, the truck they were on and the time they were delivered.

**I3: OTHER POSSIBLE ALGORITHMS**

Brute-force / naive solution: This algorithm is arguably the most simple algorithm and it guarantees you the most optimum solution. The downside is that the time complexity is O(n!), meaning that the time to find the solution becomes impractical with more than a handful of destinations.

Minimum spanning tree + Depth first search: This algorithm is relatively simple to understand and implement provided you are somewhat comfortable with tree structures. It is known as a 2-approximation algorithm as it will produce a solution that is 2 \* optimal in the worst case.

**J: DIFFERENT APPROACH**

The current selection method for packages is naive in that it groups packages that need to be on the same truck with packages that are leaving the depot later in the day. An appropropriate algorithm could determine whether or not these grouped packages might be better off on

another truck, the one leaving earlier for instance based on location

**K1: VERIFICATION OF DATA STRUCTURE**

* Total miles driven: 81.5 miles
* All packages were delivered according to their constraints and specifications by 10:59AM
* A hashtable developed by myself was used to store package objects. The lookup function operates in constant time
* A report can be viewed via the CLI in order to see what time the packages were delivered. The user can also enter a time to see the status of all packages up to that point.

**K1A: EFFICIENCY**

Adding packages to the hashtable, in theory, does not affect the lookup time which is constant time O(1). However there is one caveat. The hashtable uses a chaining method to handle collisions. It does this by way of a list data structure. If the hashing function is really inefficient, there could be (n) collisions which would result in a linear lookup time O(n) because all data would be stored in a single list within the hashtable. The probability of this occurring is extremely low given that the hashing function has been optimized for package IDs (unique integer).

**K1B: OVERHEAD**

The space complexity of the hashtable is O(n). Adding packages causes the space usage to increase in a linear fashion.

**K1C: IMPLICATIONS**

Adding trucks or cities would not affect the lookup time of the hashtable. The lookup time complexity would remain constant O(1).

**K2: OTHER DATA STRUCTURES**

1. Binary search tree
2. List/Array

**K2A: DATA STRUCTURES DIFFERENCES**

1. Binary search tree: insertion and lookup in most cases would be O(h), where h is the height of the tree. Lookup is faster than an array but still slower than a hashtable. No hash function needed.
2. List/Array: Simple to implement and insertion would be constant time, however, look up times would be O(n), much slower than a hashtable. No hash function needed.