C950 WGUPS Algorithm Overview

Student Name

ID #010241173

WGU Email: aluc167@wgu.edu

27 June 2023

C950 Data Structures and Algorithms II

# Introduction

In this project, I implemented data structures, functions, classes, objects and algorithms in order to load and delivery 40 packages to various locations across the Salt Lake region. Through this project I learned about how to correctly implement efficient algorithms and create data structures to accurately store and retrieve data. Following the requirements for this program I successfully developed a program that takes packages, loads them onto trucks and delivers them in a timely manner by following the shortest distance according to the constraints in place. This project helped me to improve my logical thinking, development skills with the python language, and knowledge and implementation of data structures.

# A. Algorithm Identification

The algorithm used in this project resembles the nearest neighbor algorithm. It does so by selecting the nearest address from the current point and creating a path in that direction, eventually making a full loop, starting at the hub “4001 South 700 East (84107)” and returning to the hub.   
However, there are some differences, wherein my delivery function is working on delivering the packages in the most efficient way.

# B1. Logic Comments

IMPORT csv, datetime from system

IMPORT truck, csvLoader, hasher from respective files

READ distances csv file, addresses csv file, and package csv file and add them to separate lists

INITIATE a new hash table

Use loadPackageCSV, from the csvLoader.py file, with the package csv file and your new hash table as the parameters

DECLARE packageIds as an array and WRITE into it with numbers 1 through 40

DECLARE three arrays for each truck and WRITE into them ids from packageIds, assure that each array has no more than 16 items and follows special constraints

INITIATE the truck objects making three of them and ASSIGN each truck’s package array to the respective truck

findDistance(index1, index2):

DECLARE distance is equal to distanceData with both parameters and indices

IF distance is blank

Swap indices

ENDIF

RETURN distance as a float

returnAddress(address):  
 FOR each item in addressData enumerated

IF the first index of item is equal to address

RETURN index

ENDIF

ENDFOR

checkStatus(package, time)

IF package has left facility and package has been delivered

SET status of package to string “delivered”

ELSEIF package has left facility and has not been delivered

SET status of package to string “on the way”

ELSE   
 SET status of package to string “hub”

ENDIF

delivery(truckObject):  
 DECLARE undelivered packages array

FOR each id in the truck’s packages

DECLARE package as package data from hash table

Add each package to undelivered packages array

ENDFOR

Clear all packages from truck object

While length of undelivered packages array is greater than 0 or empty

ASSIGN next point as 10000

ASSIGN next package as None

FOR each item in undelivered packages array

ASSIGN package address as address index and zip index from item

CALL findDistance() using returnAddress and ASSIGN mileage the OUTPUT

IF mileage is less distance than closest point

THEN ASSIGN closest point as mileage and ASSIGN next package as item in undelivered packages array

END IF

ENDFOR

add delivered package to truck

remove delivered package from undelivered packages array

INCREMENT mileage total

ASSIGN next address using index from next package

SET truck’s current address to next address

SET truck’s traveling time to current time since truck left hub

SET departure time of next package as the same as truck’s departure time

SET next package’s delivery time to truck current travel time

ENDWHILE

ASSIGN returning distance equal to the distance between truck’s last delivery and hub

Add return distance to truck’s total mileage

SET truck’s current address to the hub

runProgram():  
 CALL delivery with first truck as parameter

CALL delivery with second truck as parameter

CALL delivery with third truck as parameter  
 PRINT welcome to my delivery system

PRINT total miles traveled

TRY

PROMPT for user to enter time in 24-hour format

Convert time to timeDelta

FOR id in packageIds array

DECLARE package as package data from hash table

CALL checkStatus using packages and inputted time as parameters

PRINT packages

ENDFOR

PRINT total mileage for truck 1

PRINT total mileage for truck 2

PRINT total mileage for truck 3

ENDTRY

EXCEPT

IF time was not entered correctly THEN close program

ENDEXCEPT

csvloader.py

loadPackageCSV(file, hashtable):

OPEN file

READ file info to variable

Skip first two header lines

FOR item in file info

ASSIGN package id

ASSIGN package address

ASSIGN package city

ASSIGN package delivery deadline

ASSIGN package zip code

ASSIGN package weight

ASSIGN package status

CREATE new package object

Insert package id as key, and package object as value into the hashtable

ENDFOR

hasher.py

DECLARE hash class

CREATE hash table

truck.py

DECLARE truck class

CREATE truck

package.py

DECLARE package class

CREATE package

# B2. Development Environment

The programming environment used is Pycharm, by IntelliJ. It is an IDE used to develop code in the Python programming language. It has a built-in debugger and terminal making it great for analyzing code and running it all within the application.

# B3. Space-Time and Big-O

Thes space-time complexity of this program is mainly composed of O(n^2). This is due to the multiple loops required to read the data of each declared array. Each time a loop is written inside another loop this increases the Bit-O notation.   
The delivery function is the majority code in this function and it runs with a space-time complexity of O(n^2). This is because each item in the undelivered packages array has to be iterated over multiple times until there is nothing left in the array.

The run program function also has a lot of code; however, it does contain one for loop making it O(n) space-time complexity, the majority of the code in the function has the space-time complexity of O(1) as it is composed of mainly print statements.

The find distance function has a simple space-time complexity of O (1). It does not iterate through any data, but instead uses exact coordinates to return the correct distance.

The return address function uses a for loop to find the correct index of an address, because of this it becomes O(n) space-time complexity because it is a constant iteration until it has read all the data.

The part of the code that is reading csv files is a O(n) space-time complexity. This is due to the unknown size and data of the files. Making the csv reader method look over the data in every line and assign it accordingly.   
  
The rest of the code in the main.py file that is not inside a function has a space-time complexity of O(1). This is because all the code is doing is assigning data to a variable.

# B4. Scalability and Adaptability

This program has the ability to be self-adjusting due to the usage of hash tables and O(n) notation loops. If more packages are added to the program the hash table can easily adjust using chaining to never have any conflicts and still create the key value pair necessary to organize and use the data.

With the many for loops in this program, it has the ability to adjust to what ever size of array is being looped over. For example, if there are 100 packages, as long as they are manually loaded onto a truck, they will be iterated over in the undelivered packages array in the delivery function. With this adaptability the growth of this program is endless.

# B5. Software Efficiency and Maintainability

This software is easy to maintain and will remain efficient due to the usage of many functions. Once a package or a list of packages is added to a truck, you can call the delivery function and everything else will fall into place and the program will run successfully.   
The only thing the user has to do in order to maintain the code is to make sure the packages are added to the trucks and the data for the packages is correct. With simple tasks like this, the program becomes very efficient and anyone could maintain it.

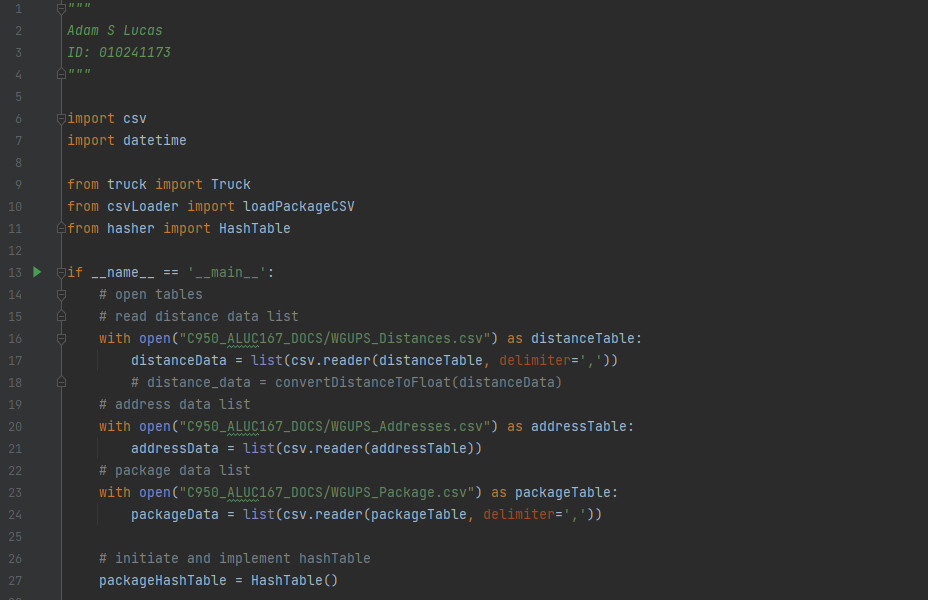
# B6. Self-Adjusting Data Structures

The self-adjusting data structure used in this code was a chaining hash table. The weaknesses of using a chaining hash table are collisions, storage. A chaining hash table is prone to collisions especially with large data, it does it’s best to distribute the data efficiently, however this is not always possible depending on the size, and the amount of data being entered into the hash table. While a hash table is being loaded with data.   
Some strengths that come with a chaining hash table are easy implementation, flexibility, and scalability. The easy implementation makes them good for simple programs such as this one and for beginner programmers as well. A chaining hash table is also flexible because of its ability to hold various data types. The scalability of a chaining hash table can easily be upgraded by having more buckets, increasing the capacity.

# C. Original Code

I have attached a zip file of my program named C950\_ALUC167zip.zip. This covers in C1 and C2 and I will provide a screenshot to correctly demonstrate usage of requirements.

# C1. Identification Information

As stated in section C, here is an attached screenshot that provides evidence of my name and WGU number as the first lines of code:   


# C2. Process and Flow Comments

As stated in section c, here is an attached screenshot to demonstrate usage of comments to direct flow and explain the process:

# D. Data Structure

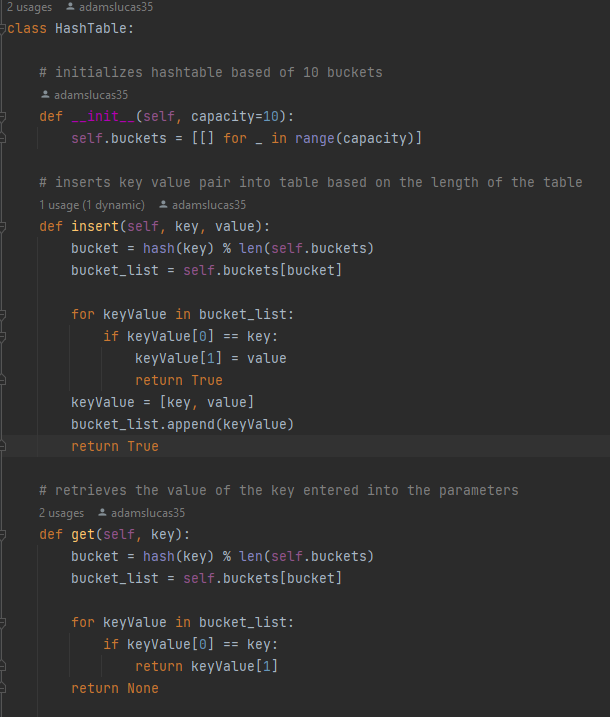
A hash table was used in this program to correctly implement the Nearest Neighbor Algorithm replica as stated in section A.

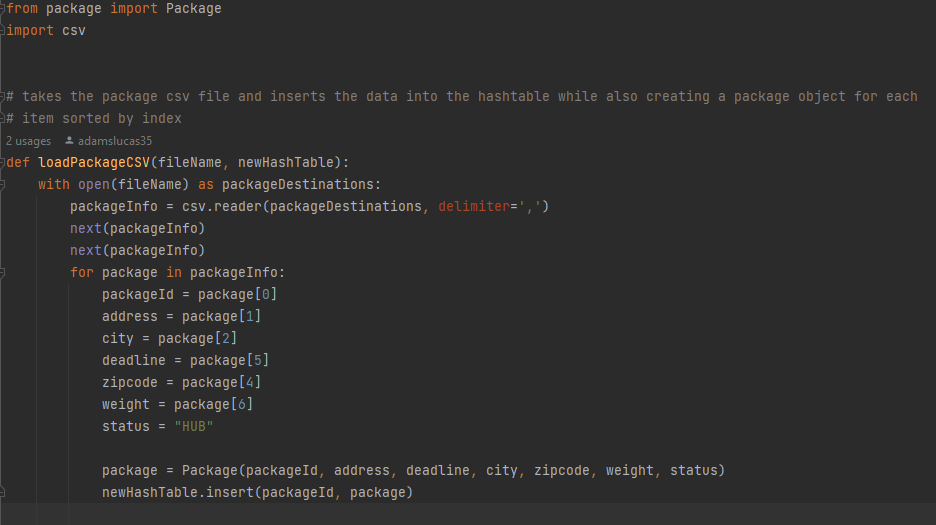
# D1. Explanation of Data Structure

I found a hash table to be most efficient in this program because, similar to a dictionary, it has key value pairs. This means that when I search for the key, I can return the value. Each package has a unique id, which I used as the key. When implemented inside the delivery function it simply had to search through the hash table by an incrementing index loop until all values had been evaluated and the distances had been checked.

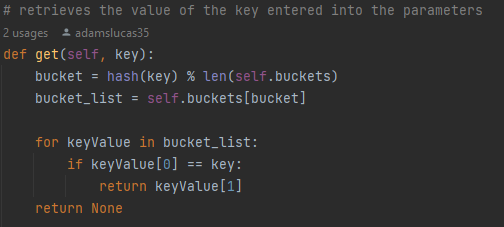
# E. Hash Table

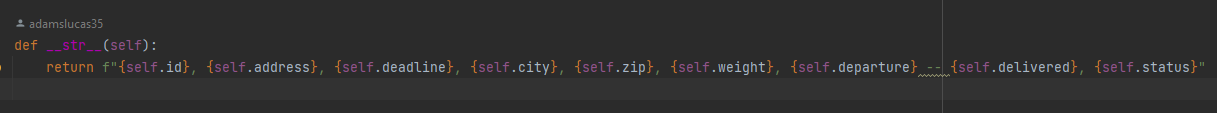
Here is a screenshot of the hasher.py file indicating the construction of the hash table:



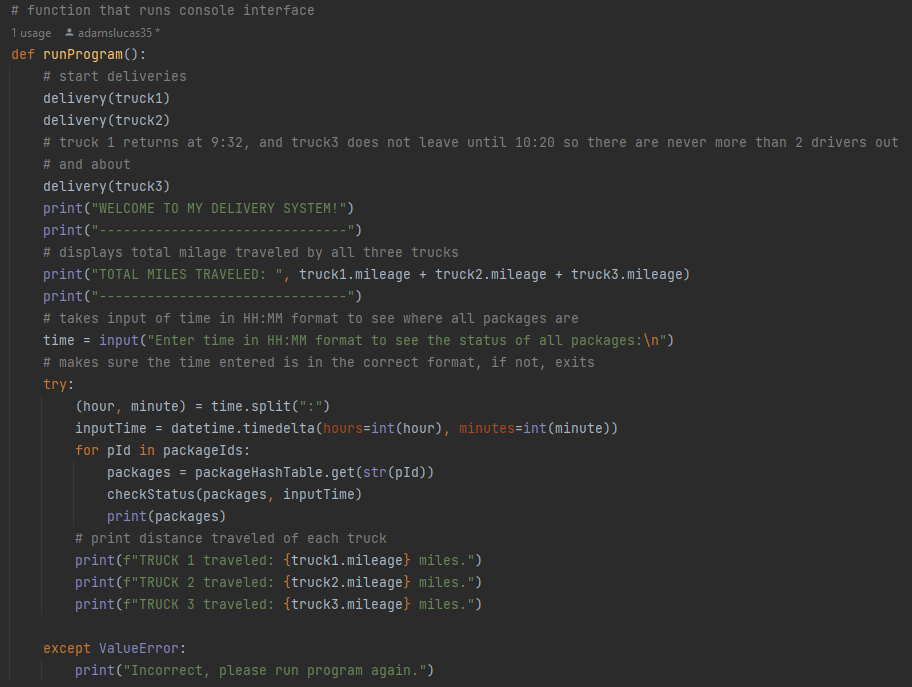
And below is the csvloader.py file, this takes in the hash table, and inserts that required data:   


# F. Look-Up Function

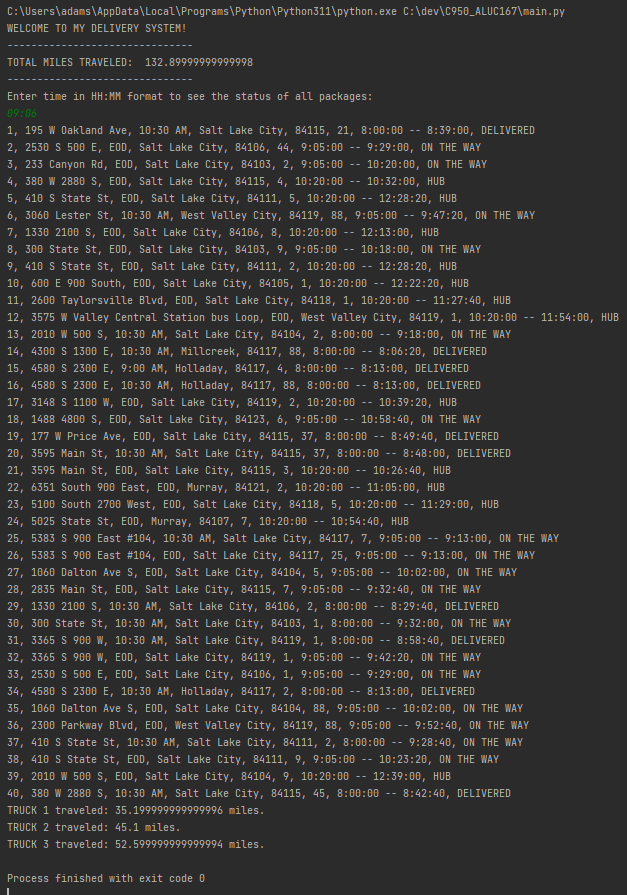
The look-up function was established in the creation of the hash table class, with the method named “get”, here is a screen shot of that code:   


The package class uses the \_\_str\_\_(self) method to output specific data when the class is called. Attached below is a screenshot of the method:   


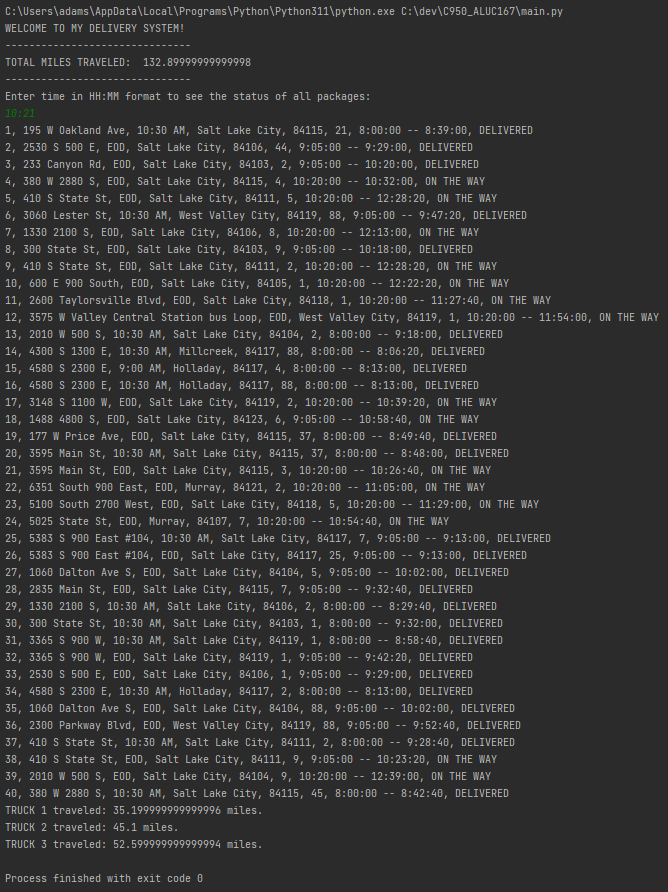
# G. Interface

I implemented a function called runProgram, this function controls the console interface and receives input from the user. Below is a screenshot of the function:   


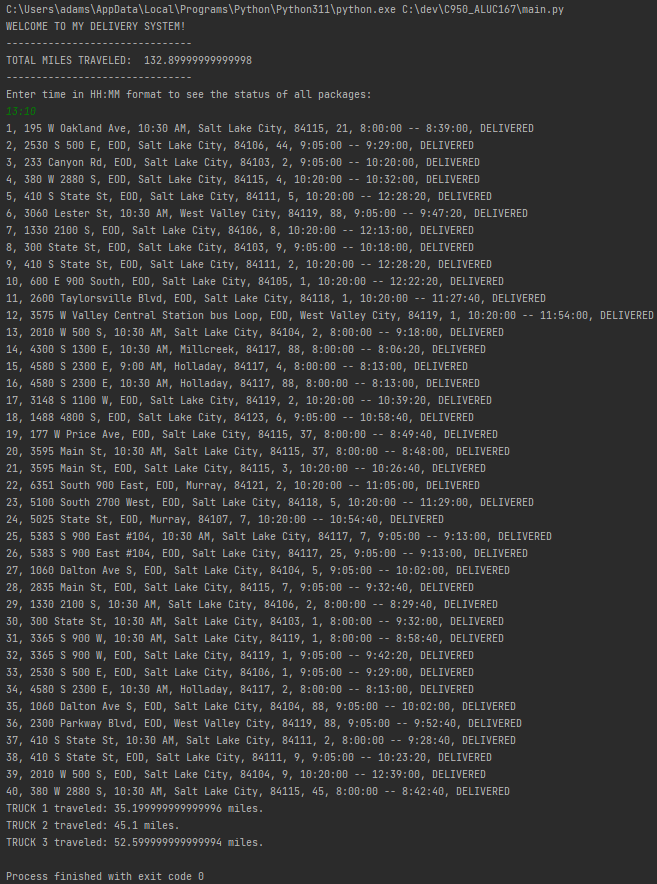
# G1. First Status Check

Below is a screenshot of the first status check at the time of 09:06:

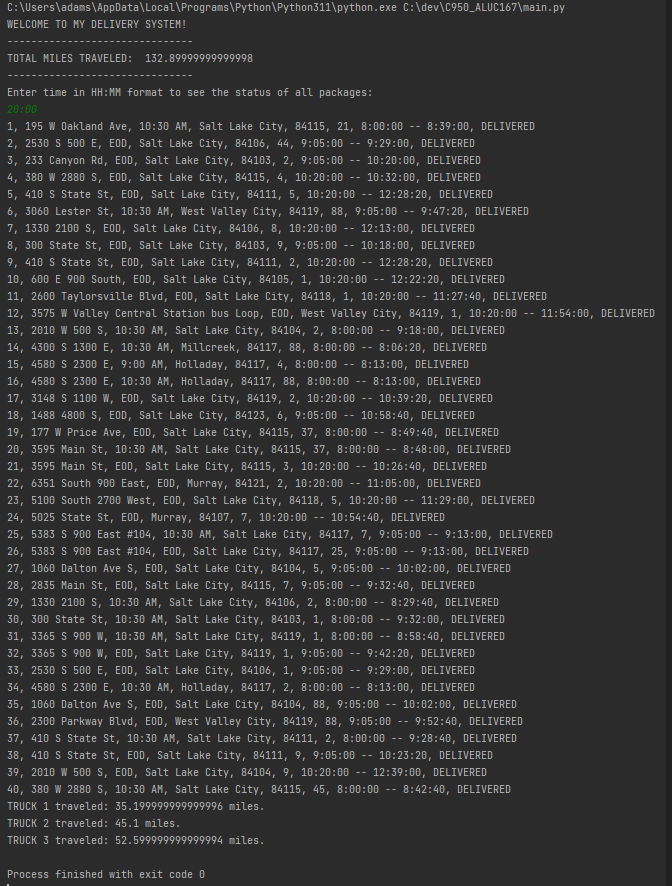
# G2. Second Status Check

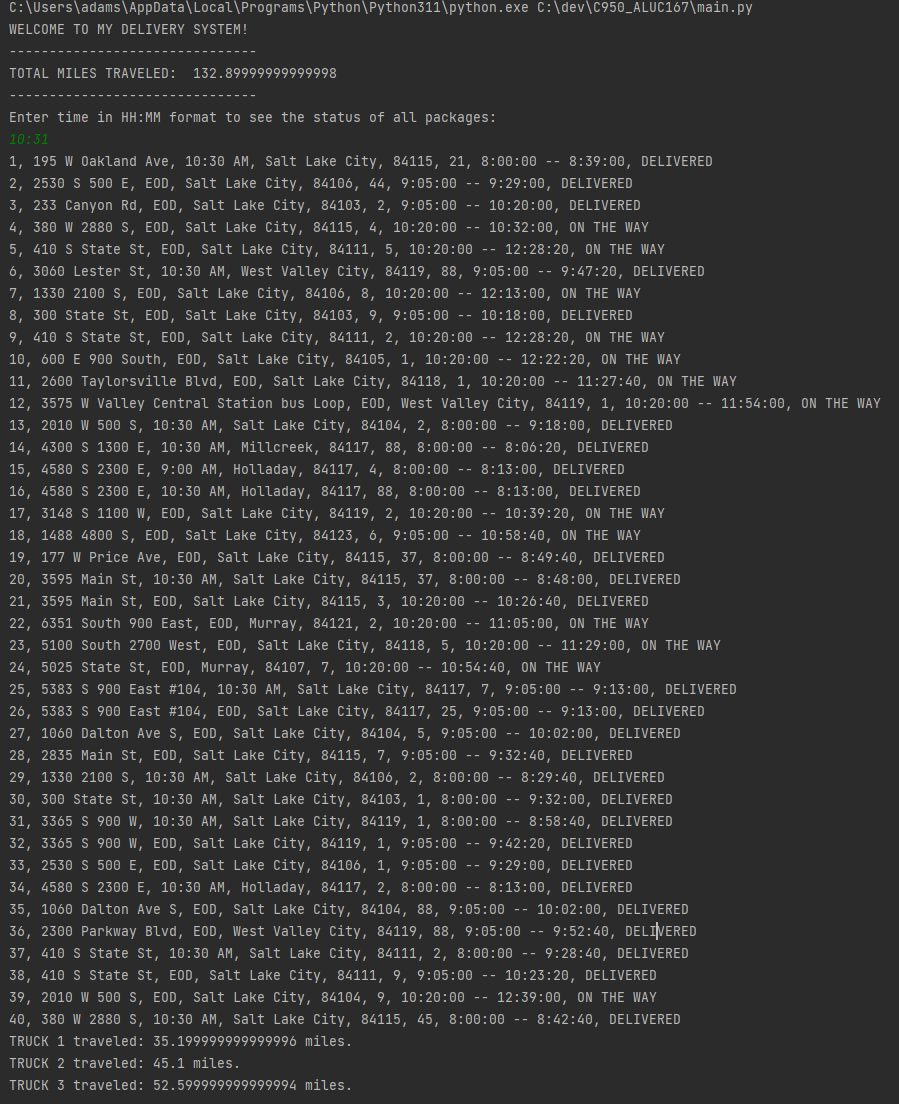
Below is a screenshot of the second status check at the time of 10:21: 

# G3. Third Status Check

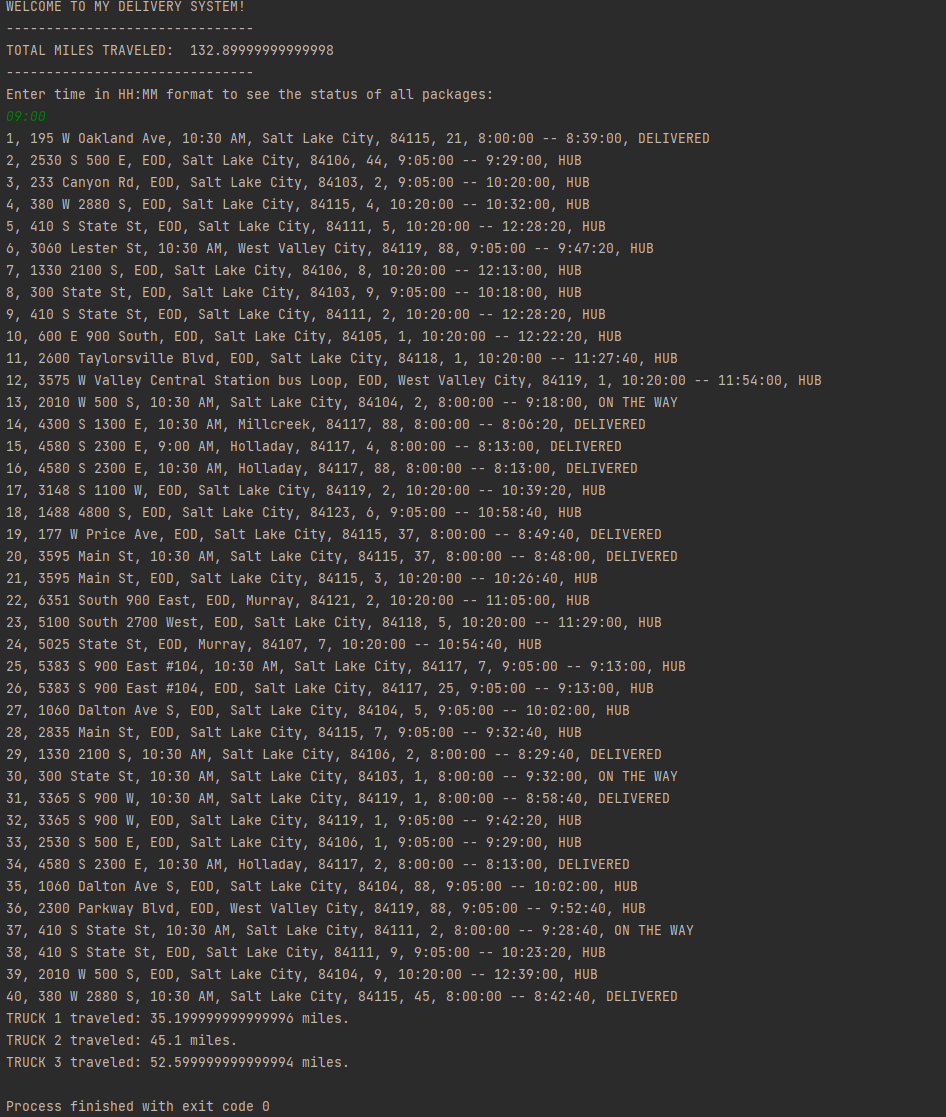
Below is a screenshot of the third status check at the time of 13:10:  


# H. Screenshots of Code Execution

Similar to the screenshots in found in section G, below is a screenshot a successful execution of the code, indicating in the top header that the total milage traveled was 132.9 miles. Each truck traveled, 35.2, 45.1, and 52.6 miles respectively:   


Below is screenshot of correct execution showing all packages with 10:30 deadline have been delivered on time:   


Below is another screenshot showing the package that needed to be delivered before 9:00 had been delivered on time:



# I1. Strengths of Chosen Algorithm

The algorithm used in this project has a few strengths that made an efficient use case. First, the constant looping assured that every item in the list was easily read over and analyzed for specific data such as the address. Using the address, it was then able to iterate through the loop again and check the distances making sure everything was checked before going on to the next step of moving the truck. Another strength in the algorithm was the flexibility and scalability, other than a real-life time constraint the list of packages to be delivered could essentially have been thousands and thousands of packages, due to the structure of the algorithm, it simply looked at the data (current address) and either assigned it as the next closest point or skipped over it. By doing this we know that any size works in the algorithm as it would just skip over anything that is bigger than the smallest point.

# I2. Verification of Algorithm

This algorithm uses the provided distance table as part of the findDistance function to correctly verify the distances between the addresses assigned to each package in the package file. Using these distances, we can see the truck traveled a total of 132.9 miles and stayed under the maximum distance allowed of 140 miles.   
Also, according to the package file and scenario, various packages had deadlines of 10:30am, one package had a deadline of 09:00am, and the remaining packages had a deadline of the end of the day. Each package was delivered before the deadline by using this algorithm.

# I3. Other possible Algorithms

There are many other algorithms that could have worked for this project; the Dijkstra’s Algorithm is a common algorithm that could have been implemented with its usage being common in creating a loop of nodes. The greedy algorithm, which was shown in the course material, is another that could have been used to deliver packages and help sort them before loading. The Bellman-Ford is a less known algorithm; however, it could have been implemented as well especially for points that are far away from the mean of the data.

# I3A. Algorithm Differences

The algorithm that I created is not guaranteed to find the shortest path, but it will find the closest point to the current node. Both Dijkstra’s and Bellman-Ford algorithm would have found the shortest path and uses the distance from the source to the current node, this makes the implementation very similar but will be exact in finding the shortest distance.   
The greedy algorithm has more similarities to my algorithm than the other two, however the difference is the guarantee of the shortest point. The greedy algorithm will always return the closest point even if it is not the shortest path, my algorithm may do the same, but does not always.

# J. Different Approach

If I were to redo this project, I think I would like to implemented a machine learning algorithm to adjust the packages by shortest distance before loading the trucks. This would have made the total distance traveled significantly shorter and would have made the trucks deliver in the same areas without having to cross town.   
I also would have created more separate files and classes, I found that my organization of having the majority of my code in the main.py file caused me to often get lost in which function I was working with, the creation of more files would have made it much easier to keep track of what each function did.

# K1. Verification of Data Structure

The data structure I used meets all the requirement by being manually created and non-usage of foreign libraries and is self-adjusting. As seen in the attached screenshot in section H, in the header the total mileage traveled by all trucks was under the threshold of 140 miles.   
Packages with id’s: 1, 6, 13, 14, 16, 20, 25, 29, 30, 31, 34, 37, 40 had to be delivered before 10:30am, as seen in the second screenshot in section H, all of the indicated packages had been delivered by that time. Also, package number 15 has a deadline of 9:00am, the third screenshot of section H indicates that it had also been delivered on time.   
In hasher.py, or as seen in the screenshot in section F, the look up function (named “get”) is present and has 2 declared usages in the code.   
Included in section F is another screen shot indicating the required data needed to report the status and information of the package and display it in the console terminal.

# K1A. Efficiency

Any change in the number of packages added to this program could significantly impact the time-efficiency of this application. As stated priorly, the majority of the code is writing in O(n^2) notation. This implies that with the 40 packages each item has to be iterated over even if the first package address is the closest point. If 80 packages were to be delivered this would instantly double the time needed to iterate over ever item. If 120 it would triple the time, 160, 200. Etc... the more packages needing to be delivered the more time needed to iterate over every package, multiple times throughout the code.

# K1B. Overhead

Each package takes up space in the hash table, with a capacity of 40 buckets, ideally each package would fit in 1 bucket, however the more packages added the more chaining that needs to be implemented in order to prevent collisions, with a small number such as 40 collisions are very unlikely, nevertheless, if more packages are needing to be delivered this could cause significant issues when storing the data in the hash table as collisions become more likely, and the time needed to locate the item could become quite slow.

# K1C. Implications

Changing the number of trucks or the number of cities could affect the look-up time dramatically, as it now adds another loop to the look-up function. Each truck has packages on it, having another truck would distribute the packages accordingly, however it now causes all functions to be run at least one more time in order to deliver all packages. One more loop can be O(n) notation causing even more time needed to process the data.   
Adding more cities to the program could change the look-up and space of the data structure as well by increasing the number of items in the hash table. Each package has a city assigned, there are many repeating cities, however having more cities would cause more data to be used in the end making the look up have O(n) more iterations to sort through before returning with the data need.

# K2. Other Data Structures

Another data structure that could have been used, if my data was being updated frequently, would be a linked list, this would allow me to insert data and remove data from the middle of the list without having to shift other items around before execution.   
I could have also used a tree, because of the distances, a hierarchy tree would have been to implement as distances could have stored in the tree using the distance from the next point as the key. This would have allowed a faster lookup time for the point closest to the truck’s current position.

# K2a. Data Structure Differences

The difference between my data structure and those mentioned in section K2 are few, the main difference is where the data is stored, with the linked list, I can store the data anywhere in the list with quick and easy insertion and deletion. In the tree the information would be stored in a hierarchical structure making sorting and searching nearly automatic as everything would be in order. The time complexity of the prementioned data structures and my own vary as well, a linked list consistently has O(n) time complexity, a tree uses O(log n) for all actions, and my hash table uses O(1) for retrieval, and O(n) for everything else. For space complexity, all three data structures are the same with O(n)

# M. Professional Communication

This document was written in formal English, and was proofread twice in order to assure accuracy of data and grammar.

# L. Sources - Works Cited

Below are sources of where I found help for my algorithms and code:

Lysecky, R., & Vahid, F. (2018, June). C950: Data Structures and Algorithms II. Retrieved June 28, 2023, from <https://learn.zybooks.com/zybook/WGUC950AY20182019/>

baeldung. (2023, March 11). Binary Trees vs. Linked Lists vs. Hash Tables. Retrieved June 28, 2023, from <https://www.baeldung.com/cs/binary-trees-vs-linked-lists-vs-hash-tables>