A. For this program, I used a Nearest Neighbor algorithm to deliver the packages. This algorithm delivers all the packages.

B. I used a closest neighbor algorithm in my program to deliver the packages.

1. Pseudo code for the closest neighbor algorithm:

Function Closest neighbor:

Pass in: Truck, address, hash of packages

J = 0

Total\_distance = 0

While the length of packages in truck is greater than 0:

For address\_index and name in the list of distances:

If the address in the list matches the address

J equals the address index

End For

Distance = 100

Next\_package = 0

For package\_index and package in the trucks current packages:

For index and item in the list of distances:

If the package address matches the address in the list of distance:

Try to find the distance using J and the current index:

If the distance of item J on the current Index is less than Distance:

Distance equals item J on the current index

Next\_package equals the current package

End Try

Except find the distance swapping J and the current index:

If the distance of item current index on index J is less than Distance:

Distance equals item current index on index J

Next\_package equals the current packages

End Except

End if

End For

total\_Distance equals the total\_distance + the current distance

Calculate time using the distance traveled

Using the hashmap, set the current packages delivered time to the current time p plus the calculated time

If the packages delivered time modulo 100 is greater than 60:

Add 100 to the time and subtract 60

End if

Set the truck time to the trucks time plus the calculated time

If the trucks time modulo 100 is greater than 60:

Add 100 to the time and subtract 60

The address equals the current packages address

Delete the package from the truck

End For

End While

After the final loop, calculate the distance back to the hub

Add this to the trucks time and distance traveled

End Function

2. I created my Python application using PyCharm 2022.3.2 (Community Edition). I backed up my program using GitHub. I alternated between programming on my personal laptop and PC. I used GitHub to pull my project which allowed me to program on both the laptop and PC.

3. My program has a time complexity of O(n^3) space complexity of O(n^3)

The major functions are:

* Main.main.get\_option() = time complexity O(1) space complexity O(1)
* CSVRead.hubDistance.createDistList() = time complexity O(n^2) space complexity O(n^2)
* CSVRead.packages.setPackageList() = time complexity O(n^2) space complexity O(n^2)
* hashMap.packageHash.packageHash() = time complexity O(1) space complexity O(1)
* hashMap.packageHash.add() = time complexity O(n) space complexity O(n)
* loadTrucks.trucks.load\_truck() = time complexity O(n) space complexity O(n)
* deliveringAlgorithm.deliveries.closestNeighbor() = time complexity O(n^3) space complexity O(n^3)
* Interface.interface.display() = time complexity O(1) space complexity O(1)
* Interface.interface.report() = time complexity O(1) space complexity O(1)

4. My program currently has capabilities to deliver as many packages and perform as many runs as necessary. Additionally, each truck and package can be created in a class and easily be put into the nearest neighbor function allowing capability for more packages, runs, and trucks.

5. My software is efficient and easy to maintain because of the use of algorithms and separate classes. This allows the main.py to be easier to read. The files are separated into packages making everything easy to find. Additionally, more packages and trucks can be easily created or removed using functions.

6. The self adjusting data structure (hash map) allows for easier storage and retrieval of values. You can also store a large amount of data and still access it very quickly. A weakness is the additional time needed to create the hash map. Additionally, the hash map needs a hashing function that adequately stores the values throughout the hash map. The hashing function must be catered for the type of data being stored.

C. My program delivers all packages and meets all requirements. I used the Distance Table and Package File to help deliver the packages.

1. I created an identifying comment on the first line of my main.py that includes my first name, last name, and student ID.

2. My code includes comments for every function and most lines to explain the process and flow of the program. The comments also include time complexity in Big O notation.

D. I used a Hash Map to store data for each package. I created buckets to store the data. I created a package class that stored all package data which made it easier to store each package.

1. Each package was stored using the Package ID as the key. This ID was modulo 10 to create the hash. The formula to create the hash is extremely basic. However, the package ID is already unique and there are only 40 packages. A higher level hash algorithm is not necessary.

I used the following for help in making my hash map:

Joe James. (2016, January 22). *Python: Creating a HASHMAP using Lists.*[Video]. YouTube.

<https://www.youtube.com/watch?v=9HFbhPscPU0>

E. The following is the code for my hash file:

| Class packageHash: |  |
| --- | --- |
|  | def \_\_init\_\_(self): |
|  | self.size = 64 |
|  | self.map = [None] \* self.size |
|  |  |
|  | def \_get\_hash(self, key): |
|  | hash = 0 |
|  | hash = int(key) |
|  | hash = hash % 10 |
|  | return hash |
|  |  |
|  | def add(self, key, value): |
|  | key\_hash = self.\_get\_hash(key) |
|  | key\_value = [key,value] |
|  |  |
|  | if(self.map[key\_hash] is None): |
|  | self.map[key\_hash] = list([key\_value]) |
|  | return True |
|  | else: |
|  | for pair in self.map[key\_hash]: |
|  | if pair[0] == key: |
|  | pair[1] = value |
|  | return True |
|  | self.map[key\_hash].append(key\_value) |
|  | return True |
|  |  |
|  | def get(self, key): |
|  | key\_hash = self.\_get\_hash(int(key)) |
|  | if self.map[key\_hash] is not None: |
|  | for pair in self.map[key\_hash]: |
|  | if pair[0] == key: |
|  | return pair[1] |
|  | return None |
|  |  |
|  | def delete(self, key): |
|  | key\_hash = self.\_get\_hash(key) |
|  |  |
|  | if self.map[key\_hash] is None: |
|  | return False |
|  | for i in range (0, len(self.map[key\_hash])): |
|  | if self.map[key\_hash][i][0] == key: |
|  | self.map[key\_hash].pop(i) |
|  | return True |
|  |  |
|  | def print(self): |
|  | print('---Packages----') |
|  | for item in self.map: |
|  | if item is not None: |
|  | temp = item |
|  | print(temp.pack) |

Because each item stored in the hash map is already a package, the following is stored within the hash map:

• package ID number

• delivery address

• delivery deadline

• delivery city

• delivery zip code

• package weight

• delivery status (e.g., delivered, en route)

To access each specific item, the package is retrieved from the hash map followed by the specific part of the package needed. I.e. h.get(‘1’).zip would return package 1’s zip code.

F. The following takes an ID as input and returns the corresponding package from the hashmap along with the following:

• package ID number

• delivery address

• delivery deadline

• delivery city

• delivery zip code

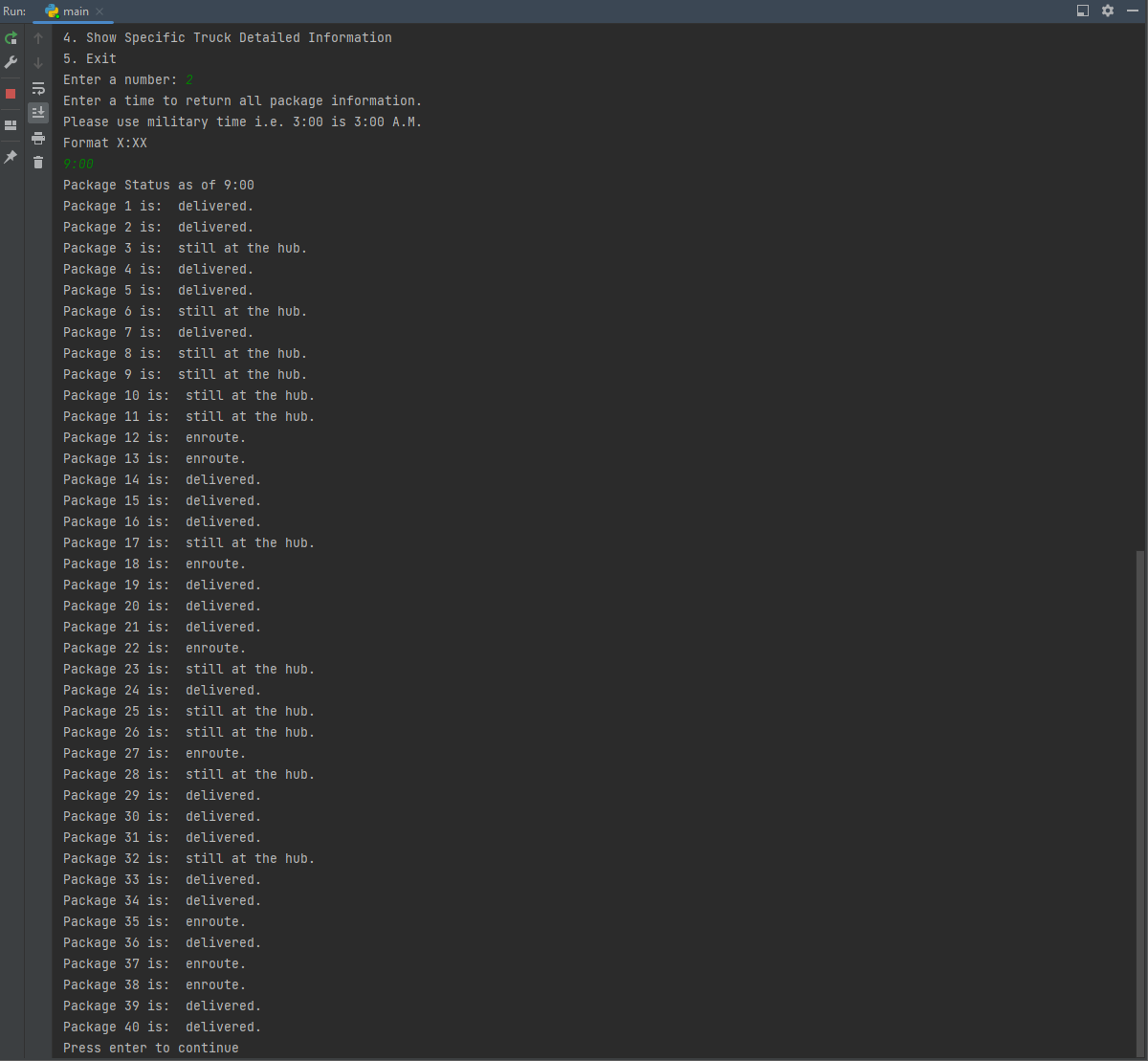
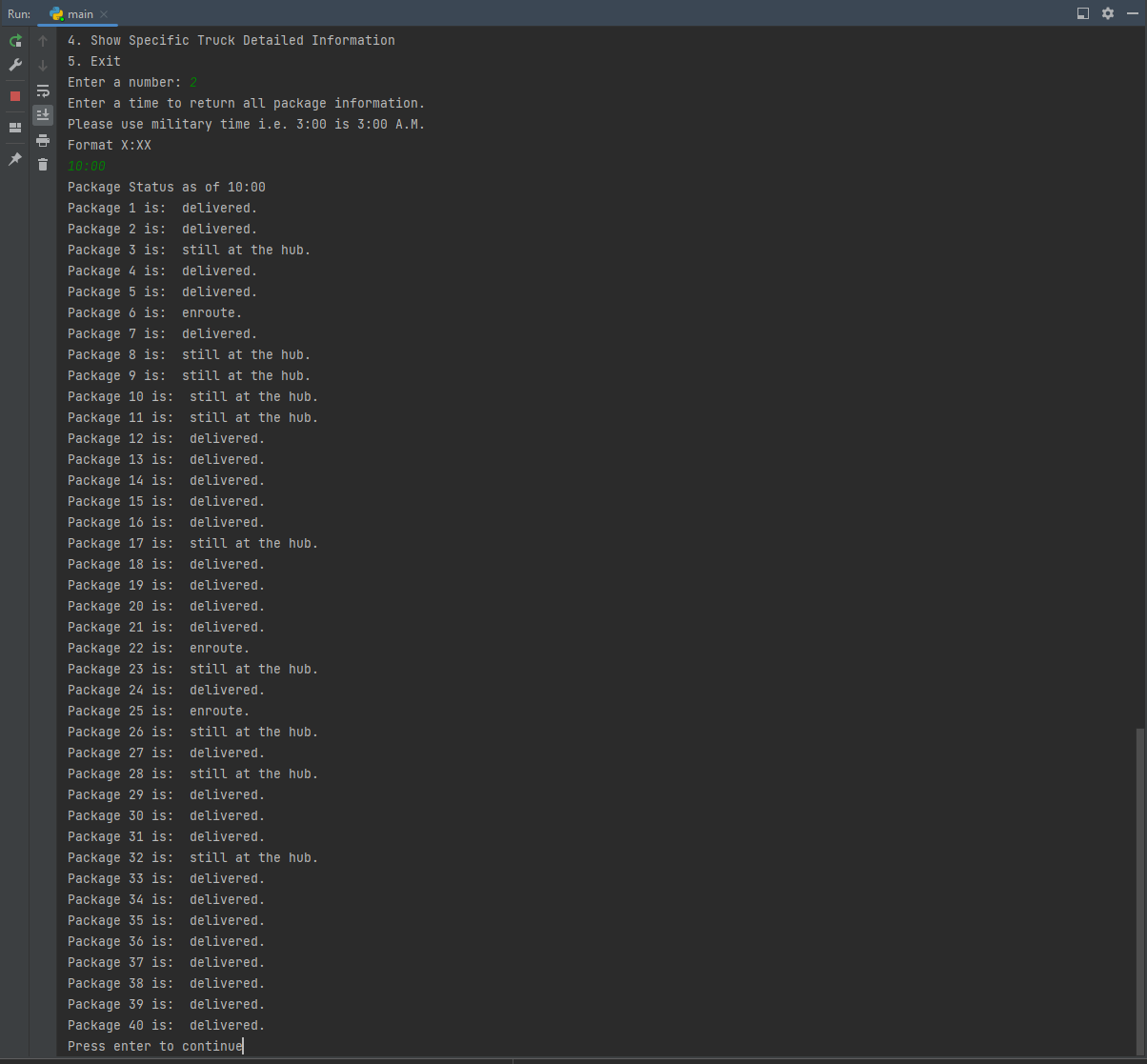
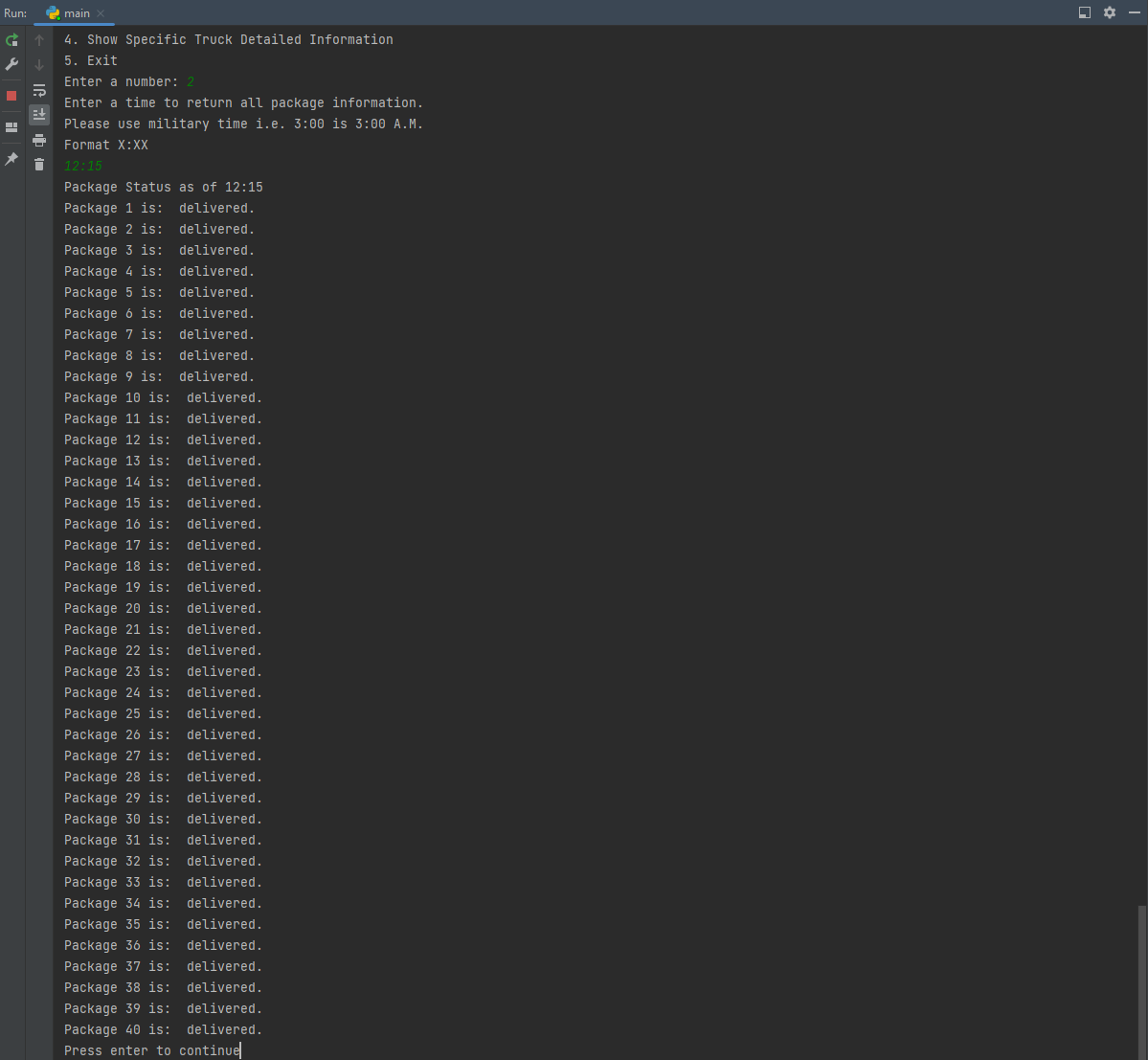
• package weight

• delivery status (i.e., “at the hub,” “en route,” or “delivered”), including the delivery time

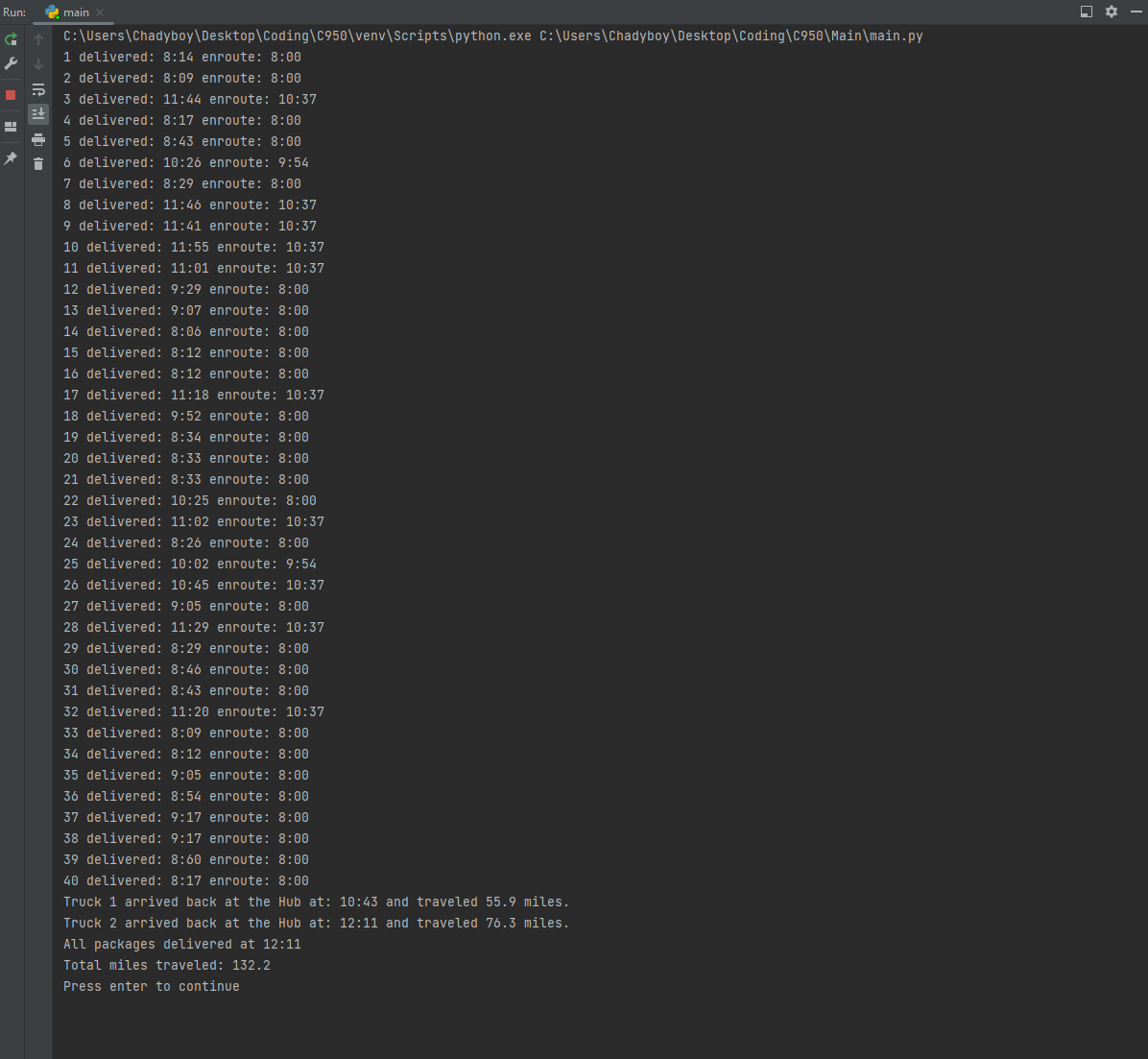
Code:

| def get\_package(id, h): |  |
| --- | --- |
|  | id = str(id) |
|  | time = h.get(id).atHub |
|  | hour = int(time / 100) |
|  | minute = int(time % 100) |
|  | print("Package " + id + " at hub: " + str(hour) + ":" + str(minute).zfill(2)) |
|  | time = h.get(id).enroute |
|  | hour = int(time / 100) |
|  | minute = int(time % 100) |
|  | print("Package " + id + " enroute: " + str(hour) + ":" + str(minute).zfill(2)) |
|  | time = h.get(id).delivered |
|  | hour = int(time / 100) |
|  | minute = int(time % 100) |
|  | print("Package " + id + " delivered: " + str(hour) + ":" + str(minute).zfill(2)) |
|  | print("Delivery deadline: " + h.get(id).delivery) |
|  | print("Delivery address: " + h.get(id).address + ", " + h.get(id).city + ", UT " + str(h.get(id).zip)) |
|  | print("Package weight: " + str(h.get(id).mass) + " Kilos") |
|  | print("Special notes: " + h.get(id).specialNotes) |

G. See Interface.interface.py. While running the code, you have the following options:

1. Display all packages
2. Show package status at a specific time
3. Show specific package detailed information
4. Show specific truck detailed information
5. Exit
6. 9:00 Package status
7. 10:00 Package status
8. 12:15 Package status

H. Screenshot upon completion of initial run.



I. For this program, I used a Nearest Neighbor algorithm to deliver the packages. This algorithm delivers all the packages.

1. The nearest neighbor algorithm has the following strengths:

* The algorithm does not take as long as other algorithms due to being more basic.
* The algorithm is easier to use for a broad range of data.

2. The algorithm delivers all packages. All packages are delivered on time. All packages that require leaving the warehouse at a later time go enroute on time.

3. Brute force search and 2-opt are algorithms that can be used to deliver all the packages as well.

1. Brute force would calculate every possible solution to deliver each package and use the most optimal one. Because of this, this algorithm would take the longest to complete.

2-opt begins to make a route and if the route crosses over itself, it swaps points. This could also be used to deliver the packages.

J. If I were to do this project again, I would spend more time testing and finding more optimal algorithms to deliver the packages. I would also create an algorithm for loading the trucks so the entire process is automated. Finally, I would add a GUI to make the program easier to use and more user friendly.

K. I used a hash map to store the package data.

1. The hash map I used meets all requirements by storing, adding, deleting, and retrieving all package information.

1. When multiple packages are stored in the same bucket, it increases the time needed to look up the package. If their were more packages, the buckets may have more items in them and increase the time needed to look up a specific package.
2. Depending on how many packages are added, their would need to be more buckets. Additionally, a better hashing algorithm would be needed to better spread the packages across the buckets.
3. More trucks and cities would not affect the look-up time and the space usage of the hash map. Packages are stored in the hash map, trucks are manually created using a truck class. Cities are stored as part of package information, but would not change the look-up time of the hash map.

2. A dictionary and array could be used to store package information.

1. A dictionary is a python class that stores a value using a key. Because each package already has a unique ID, packages could be stored using the ID as a key and the package (created using a custom package class) would be stored as a value.

An array could be used to store all of the packages as well. However, this would not be optimal and would increase the time taken to retrieve a value. The array would also have to be ordered to make retrieving packages easier.

L. The only source I used to assist with my project is the following I used for my hash map:

Joe James. (2016, January 22). *Python: Creating a HASHMAP using Lists.*[Video]. YouTube.

<https://www.youtube.com/watch?v=9HFbhPscPU0>

M. Thank You, Chad Draper