**A. Algorithm Identification**

I have chosen to use the nearest neighbor algorithm to choose the delivery path of each truckload.

**B.1 Explanation of Core Algorithm's Logic**

Stated Problem:

The purpose of this project is to create a Python program to simulate the delivery of forty packages using only two drivers and their respective trucks. This program should contain an algorithm that will allow all of the packages to be delivered in less than 140 miles. It also needs to adhere to the constraints of each package. I have chosen to implement the nearest neighbor algorithm in the program that I have created.

Algorithm Overview:

The nearest neighbor algorithm has been implemented in the following manner:

* 1. A local variable called current\_location is initialized with the location ‘Hub’.
  2. A list of up to 16 packages has been created to represent a truckload.
  3. The next package in the list (The first time doing this step the next package is the first package.) is selected as the sort\_partition. This represents the first unsorted item in the list at this point.
  4. This package is then compared against the other packages to determine the closest address to the current\_location.
  5. Once the next package is determined, it is swapped with the package at the sort\_partition. Current\_location is then updated to the address of this package.
  6. Steps 3 through 7 are repeated until the sort\_partition has reached the end of the list.

Operation time for the nearest neighbor algorithm is O(n^2), and it is not dependent on the list being sorted ahead of time. The structure of the program and operational time of the nearest neighbor algorithm will be demonstrated by an analysis of pseudo-code that represents the basis for this program.

**current\_location = ‘HUB’**

**packages[1...n], n = length(packages)**

**for each element in packages:**

**sort\_partition = element index**

**find\_min\_distance(element, current\_location)**

**for each element starting at sort\_partition + inner\_index:**

**temp\_distance = distance(current\_location, element.address)**

**if temp\_distance < distance:**

**distance = temp\_distance**

**swap (package [sort\_partition], package[j])**

**current\_location = closest\_address**

The nearest neighbor algorithm that I have implemented is a modified selective sort algorithm. For every element in the list, the inner loop will run and swap() will be called once. For every element, the inner loop distance will be called (n/2) times.

**Methods:**

swap(element1, element2) :: swaps element's positions in the list

distance(address1, address2) :: looks up distance from a dictionary

**Running Times:**

swap: O(1)

distance: O(1)

entire algorithm: O(n^2)

**B.2 Development Environment**

Operating System: Linux Mint

Python Interpreter: Python 3.6

IDE: PyCharm Community 2020.3.2

Hardware: Local Machine (Dell Latitude E5450)

**B.3 Space and Time Complexity**

In this document, I will list the space and time complexity for the major blocks of code. However, I have done a more granular analysis and documented it with comments in the code.

Chaining Hash Table Class:

* Time Complexity: O(n)
* Space complexity: O(n)

Package Class:

* Time Complexity: O(1)
* Space complexity: O(1)

Time Class:

* Time Complexity: O(1)
* Space complexity: O(1)

ImportData Class:

* Time Complexity: O(n^2)
* Space complexity: O(n^2)

Truck Class:

* Initialization of Truck Class
  + Time Complexity: O(1)
  + Space complexity: O(1)
* Once the truck is started (contains nearest neighbor algorithm)
  + Time Complexity: O(n^2)
  + Space complexity: O(n)

Main Class:

* truck\_factory
  + Time Complexity: O(n)
  + Space Complexity: O(n)
* **entire program**
  + Time Complexity: O(n^2)
  + Space Complexity: O(n^2)

**B.4 Scalability and Adaptability**

Nearest Neighbor Algorithm:  
  
The time complexity of the nearest neighbor algorithm is O(n^2) so as n increases the number of operations will increase n times. This means it can still perform correctly even if n increases significantly.

Hash Table:

* + Scalable - It can easily adjust to a growing number of packages. The chaining feature will support a large increase of packages by simply appending a package to the end of its respective bucket\_list.
  + Adaptable- If the program adds features that require another instance of the hash table besides the one used for the packages, then it can simply create another instance. This new instance can take any data type as a key since the hash() method in the insert, look-up, and remove functions. Also, the new instance can be told to contain any number of buckets.

Program:  
  
One drawback to the program is that the times for the trucks to leave the hub are hard-coded. This does hinder the adaptability and scalability of the program since a knowledge of the packages and their constraints is needed to manually determine the optimal time for each truck to leave the hub.

**B.5 Software Efficiency and Maintainability**

Efficiency:

The element of the program with the largest time complexity is the parsing of the distance table. It is O(n^2) with the worst-case scenario being all packages have a different address. At this point, that scenario is not occurring, but it could occur if this was a real-life scenario. The nearest neighbor algorithm also has a time complexity of O(n^2). However, because each truckload is limited to 16 packages and the algorithm isn't used until the packages have been divided into truckloads this algorithm has operates in constant time. All of my time complexity and space complexity calculations were based on 'n' being the number of all packages delivered that day. The program's time complexity is O(n^2) due to the parsing of the distance table.

Maintainability:  
  
I kept three elements in mind while writing the program. These elements are abstraction, encapsulation, and readability. I accomplished abstraction and encapsulation by creating classes that each had their own responsibilities and data members. I built these classes in such a way that internal changes to the classes would not require changes to be made outside of the class. I also tried to enforce private methods and variables. However, in Python, I have learned there is no way to strictly enforce the privacy of these items. Breaking the program into classes and keeping methods inside of those classes relatively small also helps with readability. I also added comments to ensure the flow of the program and function of each part could be understood by someone seeing the code for the first time. Adhering to these three elements of good coding practice will make it easier for future developers to make changes and additions to the program.

**B.6 Self-Adjusting Data Structure**

I chose to use a chaining hash table for the self-adjusting data structure requirement. I have set the default initial capacity to 80 to decrease the chance of a collision. I chose chaining because it can gracefully handle a collision by simply appending the new item to the bucket\_list. I also used the hash() method as part of the hashing algorithm in case the package id’s needed to contain characters as well as or instead of numbers.

Chaining Hash Table Strengths:

* It allows the hash table to self-adjust as the number of packages increases.
* It can be used for numeric or non-numeric keys
* It has faster lookup, insert, and removal times compared to non-hash table data structures. The most likely time complexity for these operations is constant time.

Chaining Hash Table Weakness:  
  
The bucket\_list will be more likely to grow as the number of packages increases. Each collision will make a bucket\_list longer, which will increase the amount of time needed for each insert, look-up, and remove request. To combat this the hash table’s initial capacity can be expanded to contain more buckets, which will reduce the chance of collisions. However, there is still a chance the time complexity of the previously mentioned requests could be linear.

**C. Original Code, Identifying Information, Process Control Comments**

See attached program for the fulfillment of these requirements

**D. Identify Data Structure**

As mentioned in section B.6 and B.4 I have chosen to use a chaining hash table for the self-adjusting data structure in my project

**D.1 Explanation of Data Structure**

The chaining hash table is built by creating a 3d array. The innermost array is a tuple containing the key-value pair. The middle array is the bucket\_list, which hopefully only contains one key-value tuple. However, because this hash table uses chaining to handle collisions the middle array could contain a list of key-value tuples. The outer array contains a list of bucket\_lists. The hash table has methods to insert, remove, or look-up key-value pairs. The insert function can also be used to change a value associated with an existing key. At this point, the program does not use the insert or remove method. However, these methods may become useful as features are added to the program.

Relationship Between Data Points:

The nearest neighbor algorithm frequently uses the chaining hash table to find a package by providing the hash table's look-up function with the package id as the key. The algorithm then uses the package's address to determine the distance between the truck's current location and the package's address.

**E. Hash Table**

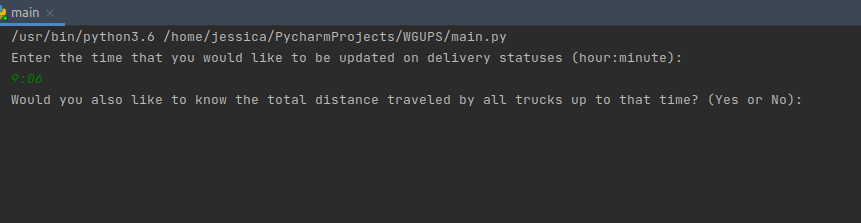
See attached program for the fulfillment of this requirement

**F. Look-up Function**

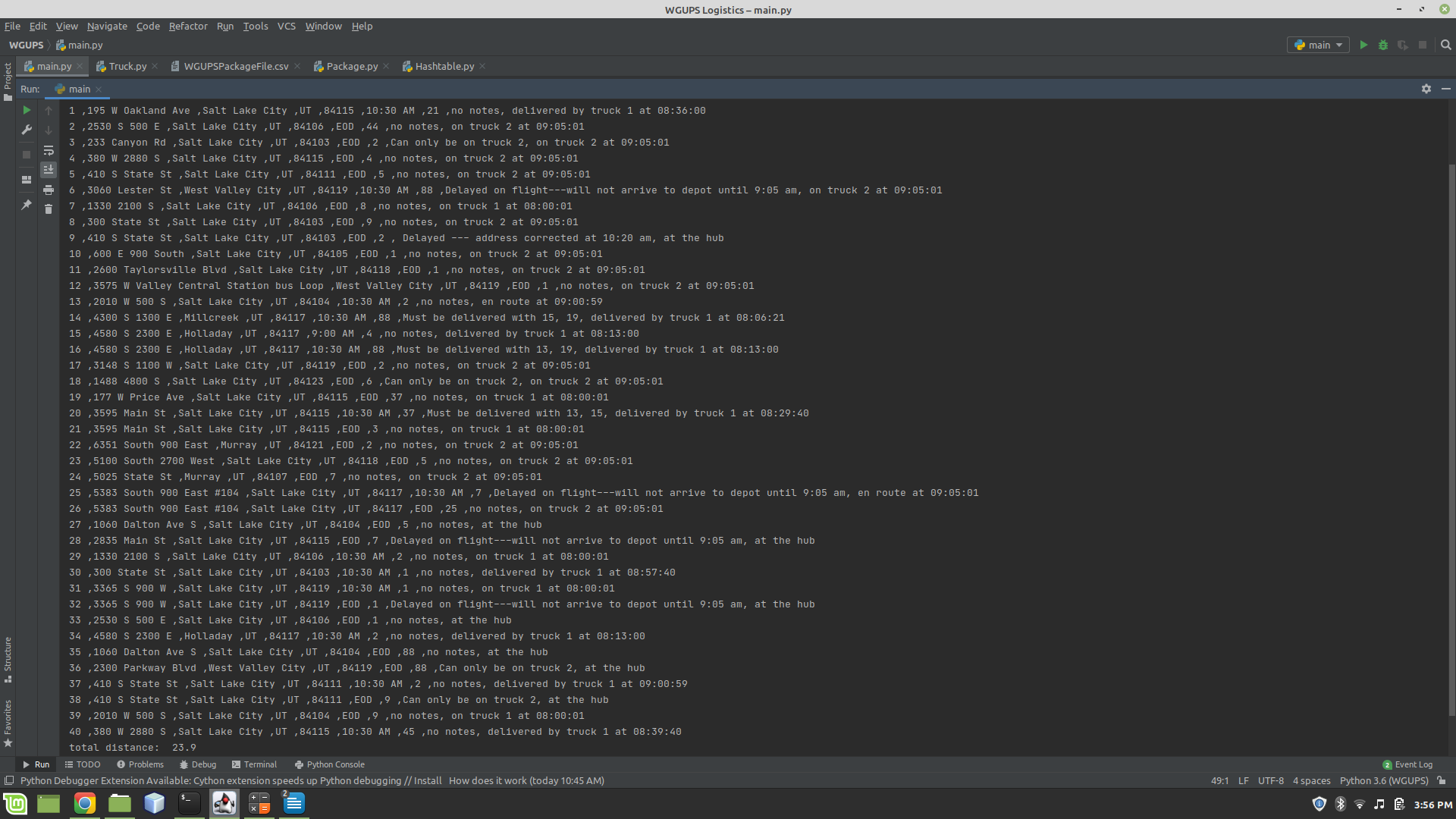
See attached program for the fulfillment of these requirements

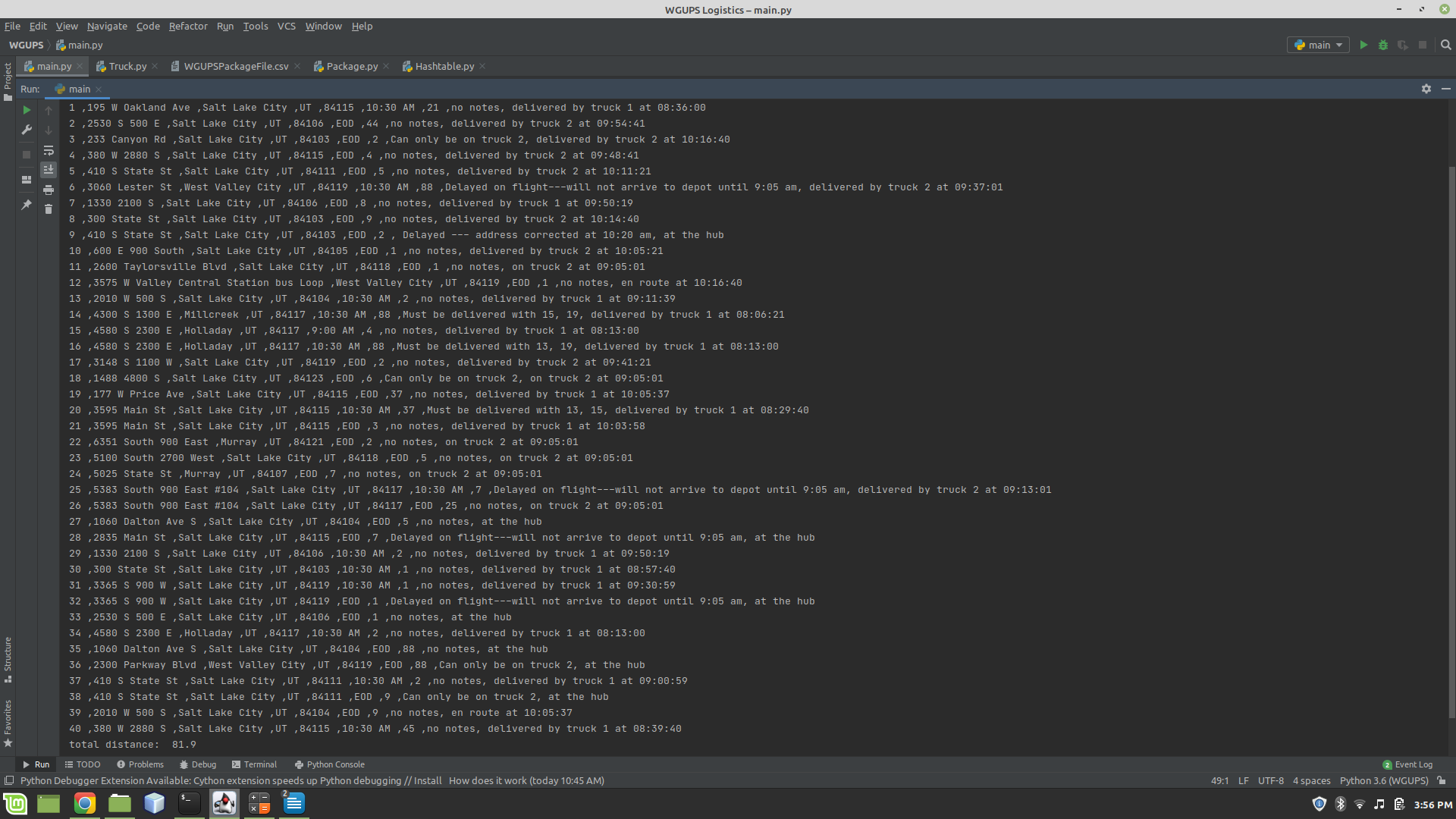
**G. User Interface, Three Screenshots for Status Check**

User Interface:

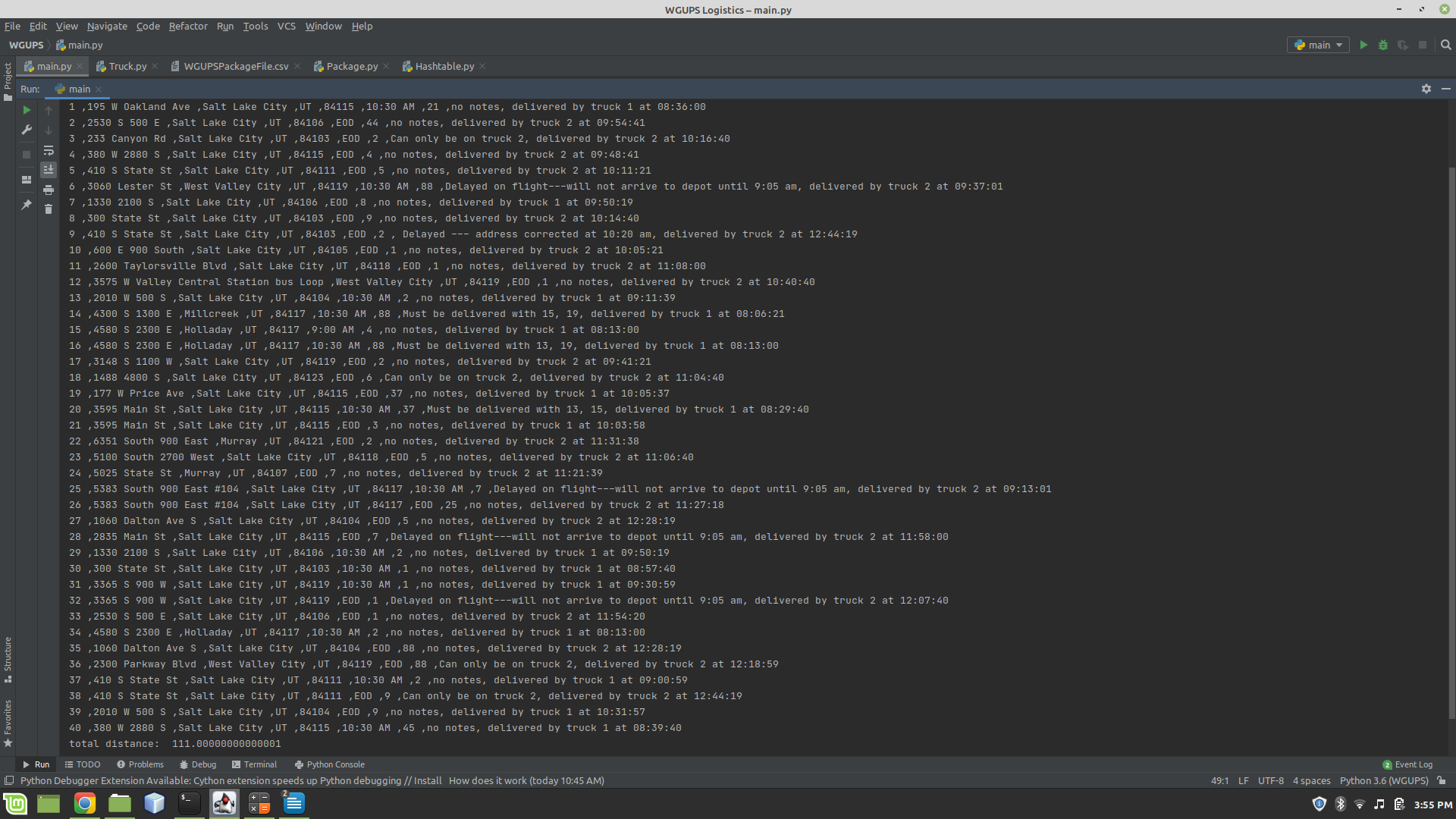


Update at 9:06 am:

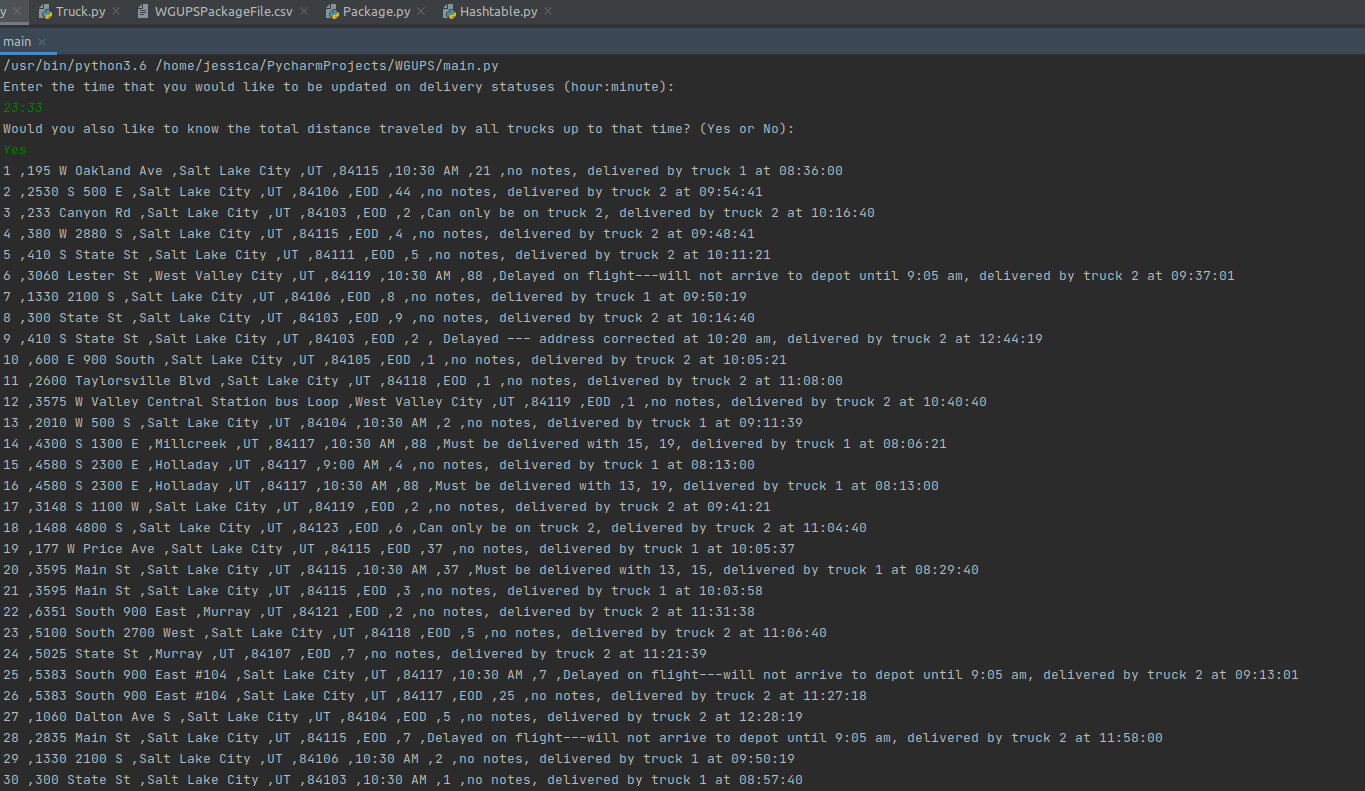
****

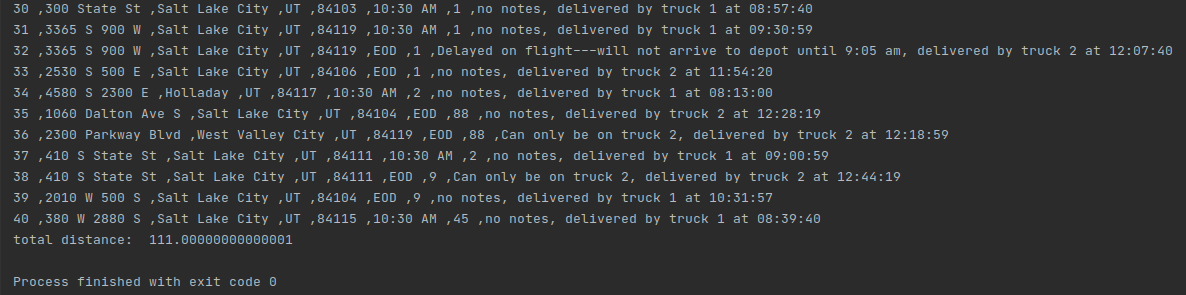
Update at 10:20 am:

Update at 1:10 pm:



H. **Screenshots of Code Execution**





**I.1 Strengths of Nearest Neighbor**

* It finds a solution in polynomial time, which is much more reasonable than the factorial time complexity needed to find the shortest route.
* It finds a reasonably short route.
* It is easy to implement and understand.

**I.2 Verification of Algorithm**

Please refer to the screenshots provided in sections and G and H as well as running the program itself for verification that the requirements of the program have been met.

**I.3 Two Alternative Algorithms Not Used**

Sub-tour Reversal Explained:[[1]](#footnote-2)

Phase 1:

Sub-tour Reversal if used in this project would start with a list of packages that represents a truckload. Then, it would swap a package's position with one of its neighbors. For example, the package in index 1 would swap places with the package in index 2. Next, it will calculate the total miles for that truckload based on the recent swap. The list then reverts to its original order, and another pair of neighboring packages are swapped. Then, the new mileage is calculated. This process will continue until each pair of neighbors has been swapped. The lowest mileage route will be kept and used in phase 2.

Phase 2:

In phase two, instead of switching neighboring indexes, a package will be swapped with another package that is two indices away. For example, the package in index 1 will be swapped with the package in index 3. The mileage is calculated for each route configuration and the lowest mileage route is kept like it was in phase 1.

Phase 1 and 2 are repeated until the packages that are being swapped are n-2 indices away from each other. For example, the last instance would be index 1 swapping with index 15 if there are 16 packages in the list.

Then, the lowest mileage route out of all the operations from above is used to do this all over again. This continues to happen until the mileage doesn’t improve.

Differences between Sub-tour and Nearest Neighbor:

The nearest neighbor attempts to reduce each leg of the overall trip as an attempt to reduce the overall mileage. However, the sub-tour reversal creates a system to incrementally improve the route by trial and error.

Nearest Insertion of Arbitrary City:[[2]](#footnote-3)

Step 1: Choose a starting location. The starting location for this project is the hub, which I will refer to here as v.

Step 2: Choose the next closest location. Let’s call this location j.

Step 3: Arbitrarily choose another location that has not been chosen yet. Let’s call this location k.

Step 4: Among the chosen cities find an edge that minimizes the C(v) + C(k) + C(j) where C() is “cost of”. Insert k between v and j.

Step 5: Repeat steps 2 through 4 until all cities have been inserted.

Differences between Nearest Insert of Arbitrary City and Nearest Neighbor:

The first two steps of the nearest insert of arbitrary city algorithm (NIAC) and the nearest neighbor algorithm are the same. However, steps 3 and 4 are where the differences lie. In NIAC an arbitrary city is selected and then calculations are made to determine the best place in the list of already inserted cities that the new city should be placed. This is much different than the nearest neighbor algorithm, which never arbitrarily selects a city. Instead, it determines the next closest city to the most recently selected city and appends it to the route’s list.

**J. Different Approach**

These are the changes that I would make to the program if I did this project again:

1. time class- I would use the built-in Python time class instead of creating my own. The thought behind building my own time class was that I needed to keep track of pseudo time since this is a simulation. However, after finishing the program I realized the built-in Python time class is capable of doing everything I needed it to do. There were a few times in my code in which I had to use the built-in time class to help with formatting and parsing time. I should have used it exclusively
2. all\_keys() in the hash table class- Toward the end of creating this program I realized it would have been easier if I had created a method called all\_packages() instead of all\_keys() in the hash table class. I found myself converting keys to packages over and over again, which made the code harder to read and understand.
3. initial capacity of the hash table- The code would be more robust if I had included a way to read the length of the package table in the ImportData class. This would allow the program to know how many packages the CSV file contains and use that to determine the size of the hash table's initial capacity.
4. distance table- I filled in the missing fields of the distance table. If I were to do this over again, then I would have left those fields blank. I should have written code to determine whether or not the field is blank. If it is blank, then the program would look up the distance of the two cities in reverse order. This is a fairly small change in the code, and it would reduce the space needed for the distance table by half. It also would have saved me from copying and transposing the completed columns by hand.

**K. Verify Data Structure**

Please refer to the screenshots provided in sections and G and H as well as running the program itself for verification that the requirements of the program have been met.

**K1a. Time Efficiency of Look Up Function in Data Structure**

At this time there is only one key-value pair in each bucket\_list of the hash table, which means the lookup function works in constant time. However, as the number of packages grows the more likely the length of each bucket\_list will grow. It should however still be better than linear time especially if the package id's continue to be numbered sequentially. If all package I.D.s were a multiple of the tables outer array length, then all packages would be on the same bucket list. At that point, the time complexity would be O(n). This is the worst-case scenario, but it's also very unlikely to occur. The table can be adjusted to have a higher initial capacity, which may be necessary as the number of packages increases.

**K1b. Space Efficiency of Hash Table**

The hash table consists of a 3d array. The innermost array is a tuple containing key-value pairs. As the number of packages grows key-value tuples will be added to the hash table. The worst-case scenario growth is O(n). Since only a key-value pair will need to be added for every package that is added.

**K1c. Implications of Increasing Number of Trucks or Cities**

Increase the number of trucks nor the number of cities would increase the look-up time or space of the hash table since neither trucks nor cities are stored in the hash table.

**K.2 Identify Alternative Data Structures**

* Linear Probing Hash Table
* Quadratic Probing Hash Table

**K.2a Differences between Alternative and Chosen Data Structures**

One of the major differences between both probing techniques and chaining is that the probing techniques do not dynamically make the capacity of the hash table larger, but chaining does. If there came a need to insert key-value pairs to the hash table, but all of the tables' fields are occupied, then the probing hash table would not be able to handle the request.

Another difference is how they handle collisions. Chaining simply adds a new key-value pair to the bucket\_list. Both probing techniques will put the new key-value pair in an empty bucket if the target bucket is already occupied. This adds some complexity to the code to find the key-value pair in the future.

Also probing has a mechanism to determine if a key-value pair doesn't exist before looking at every n-items if certain conditions hold. Chaining does not have a similar feature.

**L. Acknowledge Sources**

BrainKart.com. "Heuristic Algorithms: nearest neighbor and sub-tour reversal algorithms - Traveling Salesperson Problem." Accessed February 2, 2021. https://www.brainkart.com/article/Heuristic-Algorithms--nearest- neighbor-and- subtour- reversal-algorithms---Traveling-Salesperson-Problem-(TSP)\_11247/.

Carravilla, Maria Ant´onia and Jos´e Fernando Oliveira. “Heuristics and Local Search.” Accessed February 2, 2021. https://paginas.fe.up.pt/~mac/ensino/docs/OR/HowToSolveIt/ConstructiveHeuristicsForTheTSP.pdf

1. BrainKart.com, “Heuristic Algorithms: nearest neighbor and sub-tour reversal algorithms - Traveling Salesperson Problem,” under “Subtour Reversal Heuristic.” [↑](#footnote-ref-2)
2. Carravilla and Oliveira, “Heuristics and Local Search,” under “TSP – Nearest insertion of arbitrary city.” [↑](#footnote-ref-3)