C950 Performance Assessment

# A:

I have chosen a variant of the Nearest Neighbor algorithm, which is a type of greedy algorithm, as the basis for my system to deliver packages for WGUPS.

# B:

1. The basic form of my algorithm is as follows:
   1. Sort the packages by distance to the starting location
   2. Add the closest package to the path
   3. Sort the remaining packages by distance to the location of the most recently added location in the path
   4. Add the first package to the path
   5. Repeat steps c and d until there are no packages remaining
2. I used Sublime Text to edit all the python files used in my application and tested and run the application from the command line. At all times I have used the most recent version of python available.
3. The most significant block of code is the one generating a path for each truck according to the packages assigned to it which has a worst case time complexity of O(N2 \* logN). Since this is the slowest section of the code, this is also the time complexity for the application. The report generation code is approximately O(N) because it must iterate through each package, while the hash table structure is O(1) because of the efficient nature of hash tables.
4. Since the worst-case time complexity of the algorithm is O(N2 \* logN), the program will slow down significantly as the number of packages increases. However, on modern hardware it will still run quickly enough to be functional even with hundreds of packages instead of the 40 currently being delivered. Manually assigning packages to the trucks will get very difficult if the number of packages increases too much, so if this software needed to be scaled in the future an automated system for assigning packages will be necessary.
5. I have mainly followed object-oriented design principles and split the major classes into separate files, making the software easy to understand and maintain. In addition, the simple nature of the nearest neighbor makes it easy to understand as well. The pathfinding code is split into its own function, so if the future maintainer wished to use a different algorithm to meet changing business needs it would not be overly difficult.
6. The hash table I designed for this application should have a low number of hash collisions as the size can be adjusted for different numbers of packages, meaning that even with a large number of packages all operations should execute in nearly constant time. However, its addressing space will grow as quickly as the number of packages, so if memory is a significant constraint this structure has that disadvantage.

# C:

See comments in code

# D:

For this project I implemented a hash table, found in the hash\_table.py file. The hash table is customized for the parcel data structure used in the project and implements insert and lookup operations using objects of the parcel class. All information relevant to parcels is stored and retrieved.

D1: My hash table implementation uses the package ID in the hash function calculating which “bucket” to assign to the parcel object. It does this by taking the modulo of the package ID and the expected number of packages, which is given when the hash table is created. This aspect of a hash table makes it more efficient than a linear data structure because the location of the data is found with a single hashing operation instead of enumerating every object in a linear data structure until the correct one is found, making lookups O(1) instead of O(n).

# E:

The hash table is found in hash\_table.py and the insert function is named insert(self, package).

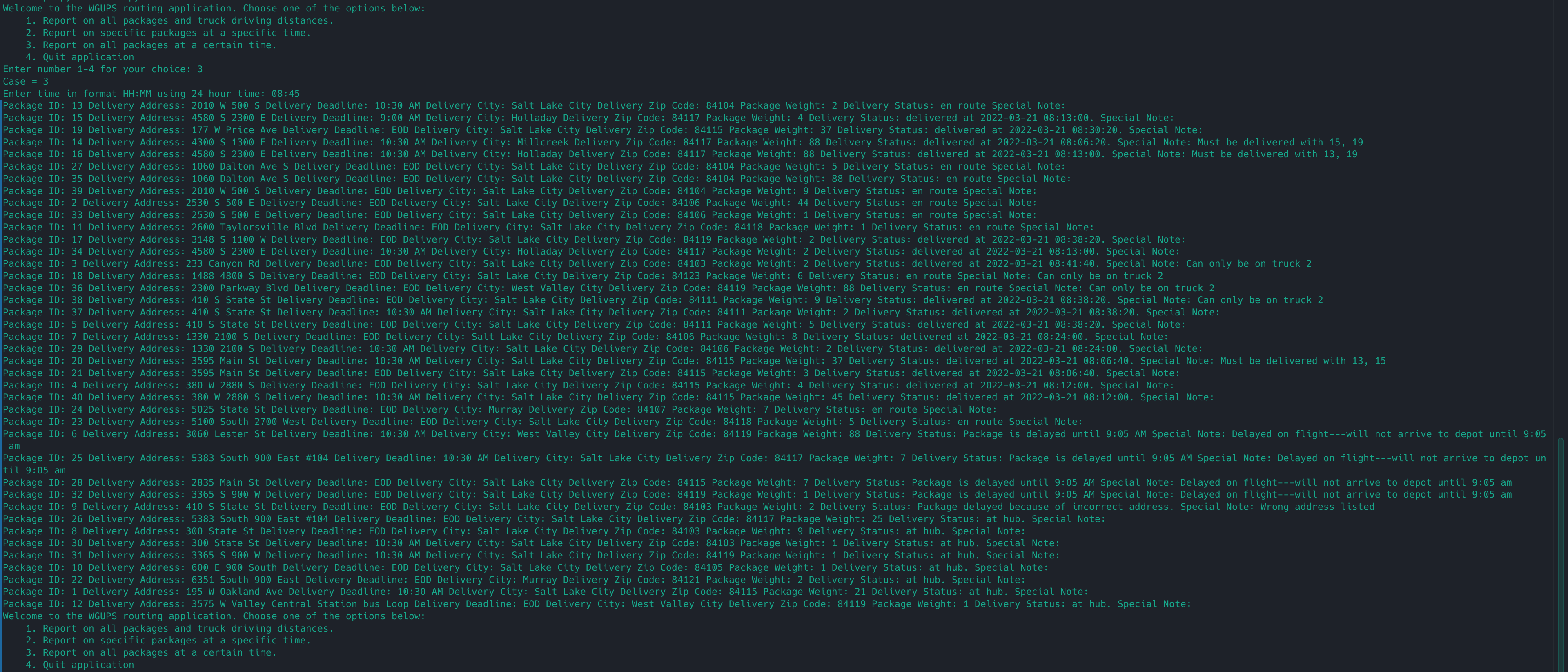
# F:

The hash table is found in hash\_table.py and the lookup function is named lookup(self, key).

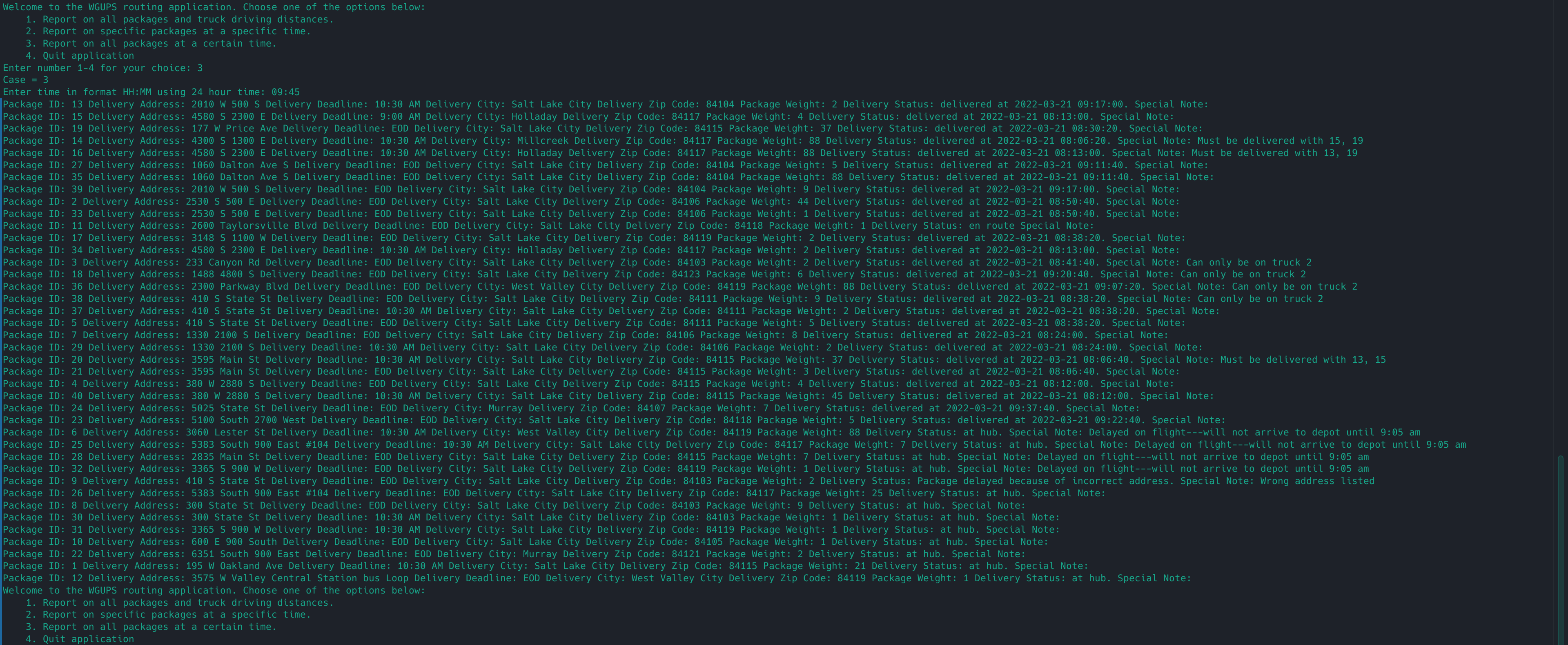
# G:

The program is run with a console interface. All that must be done is to run ‘main.py’.

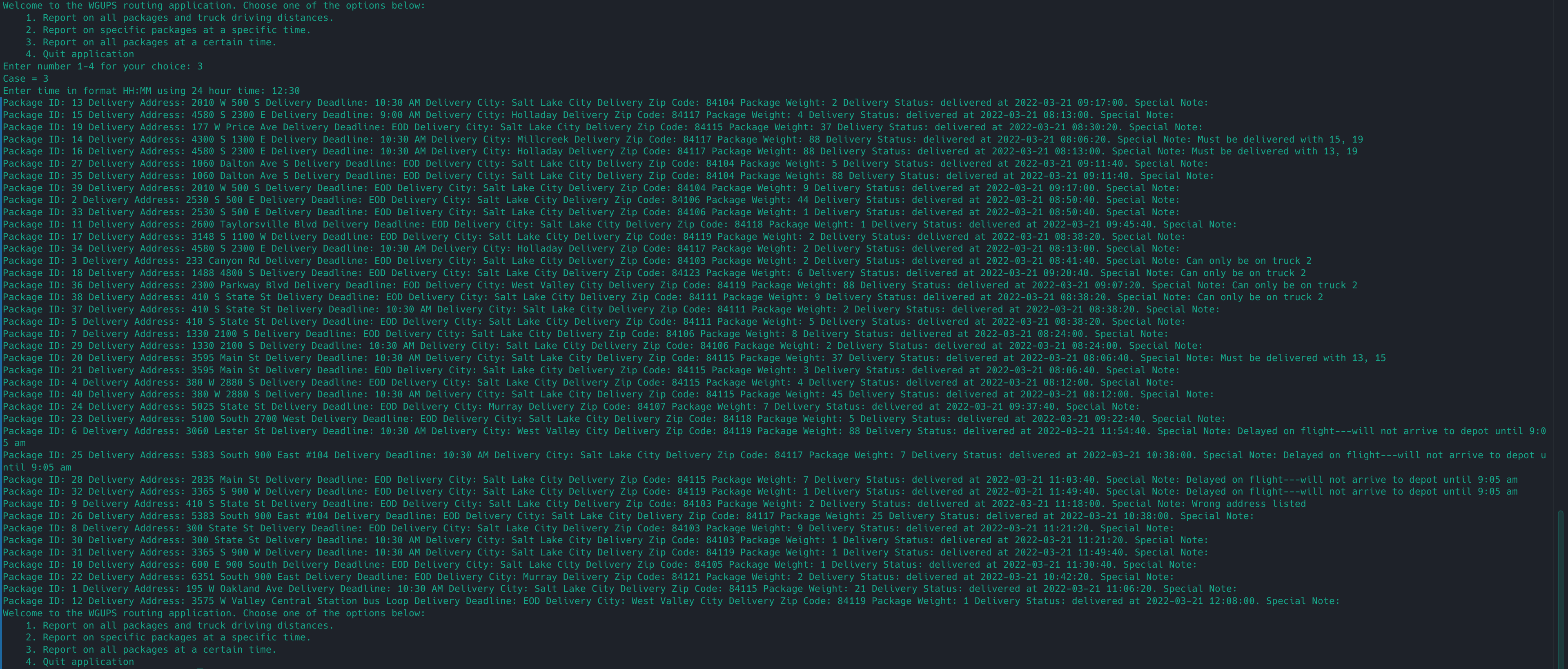
G1: This screenshot shows the information and status for all packages at 8:45 AM



G2: This screenshot shows the information and status for all packages at 9:45 AM

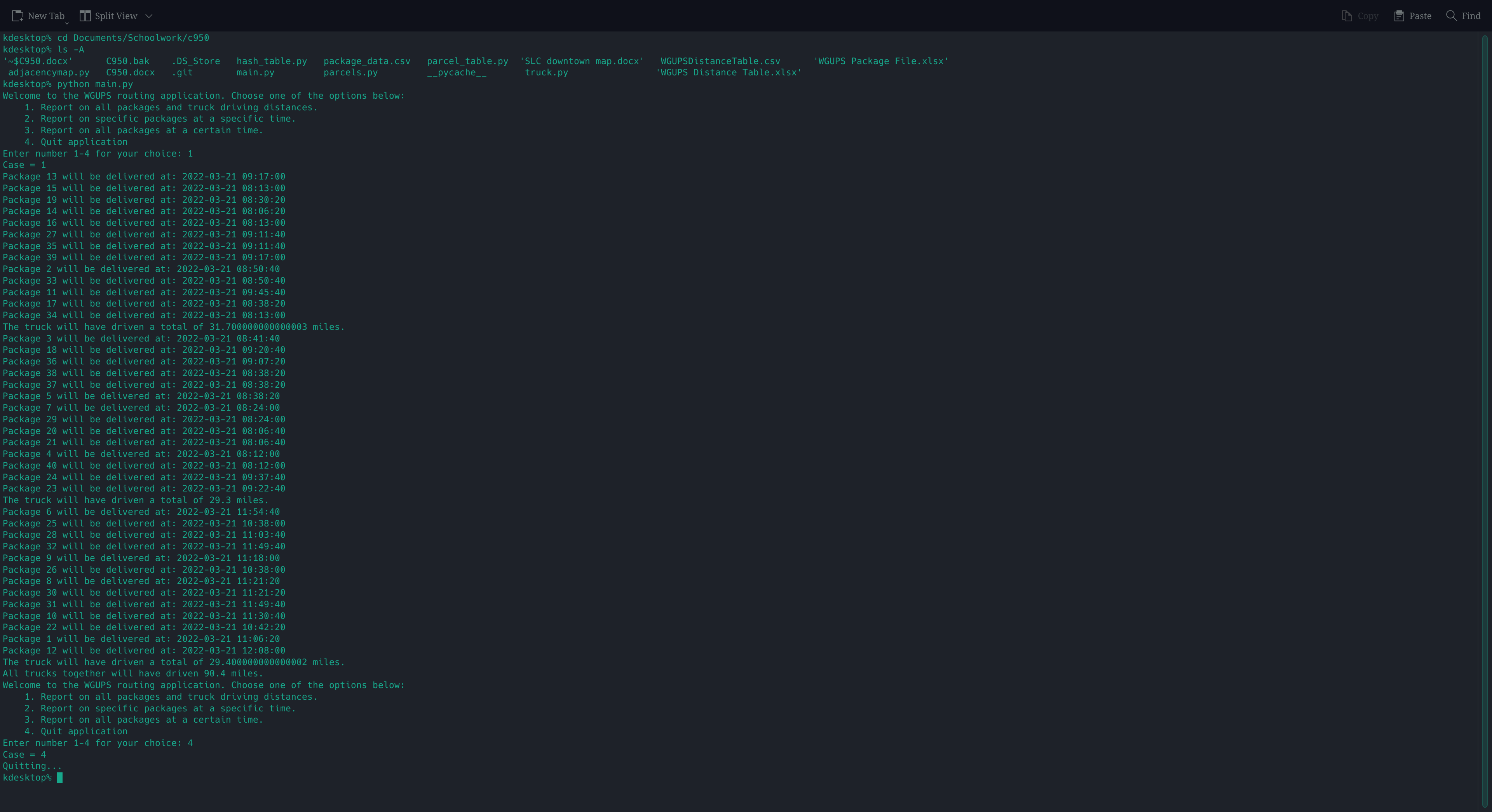


G3: This screenshot shows the information and status of all packages at 12:45 PM



# H:

This screenshot shows the code running to completion, the delivery times for all packages, and the total mileage driven by each truck as well as the miles driven by all trucks together.



# I:

1. The Nearest Neighbor Algorithm has two primary advantages: it is simple to understand and implement, and its execution time is relatively fast: O(n2). The algorithm’s code is short, and the simplicity allowed me to implement it quickly, and equally as quickly test the code and get it working. The time it takes to run is also an advantage, as it is much faster than algorithms that more exhaustively search for an optimal path.

2. The algorithm I used satisfies all requirements for the project.

* All packages were delivered by driving X miles, which is less than the required 140.
* All packages were delivered before the End of Day and before any individual deadlines
* All packages were delivered according to any specific instructions for the package
* The user can verify that all packages were delivered in less than 140 miles by running the “Overall Report” function in the application’s menu. To verify that all packages were delivered before their specific deadlines, the user can check on any individual package by using the “Package Report” function in the application’s menu.

3. There are other algorithms that could be used to route the packages according to the requirements as well. If the set of addresses grows larger and the most optimal possible route is required, a Branch and Cut algorithm could be used. If an approximate solution is acceptable, then the Multi-Fragment Algorithm could be used.

A: The Branch and Cut algorithm is known to find exact solutions to the Traveling Salesman problem, and on modern computing equipment it can be used on small to moderately sized numbers of addressed and packages in a practical amount of time. However, it is much slower than approximate solution finding algorithms like the Nearest Neighbor, and so is probably not the most practical algorithm to use for these requirements even though it would technically work. The Multi-Fragment Algorithm on the other hand finds an approximate solution like the Nearest Neighbor, but it typically finds a solution only 2-3% worse than the optimal solution. It is much slower than the Nearest Neighbor Algorithm with a worst case runtime of O(N2logN), but that is still in polynomial time and thus suitable for this application. It is also more complicated to implement which is why I chose to use Nearest Neighbor instead.

# J:

If I were to start this project over again, I would attempt to implement an algorithm for assigning the packages to different trucks. The manual approach I have used so far is suitable for the number of packages currently being delivered, but would quickly grow cumbersome if WGUPS started delivering many more. It would also be an interesting challenge to tackle.

# K:

1: The hash table data structure I implemented in part D allows my application to meet all requirements for the application. The packages were delivered with the trucks driving X miles, less than 140. All packages were delivered on time, and all packages were delivered according to any instructions given with the package. The hash table is efficient, and contains a lookup function, as well as an insertion function. All required information can be seen using the application’s user interface, and that information is accurate.

1A: Since the number of buckets in the hash table is automatically set to the number of packages given as an input to the program, adding more packages should not affect the lookup time making it O(1). If more packages are added without adjusting the size of the hash table, more collisions will be likely which could slowly increase the lookup time as the table must iterate through all entries in the bucket calculated by the hash function.

1B: Adding more packages linearly increases the memory required to store the hash table. The backing data structure is a list of lists, and in Python lists have dynamic size. Each time a new package is added to the hash table, the amount of memory taken up is increased by the same amount.

1C: Adding additional cities or trucks will not effect the amount of memory storage used by my hash table in any way; this is because only the package data is stored in the hash table while the trucks and cities are stored separately.

2: There are other data structures that could be used to meet the requirements of this application. One such data structure is a balanced tree, using the package IDs to sort and balance the tree. Another is a hash table using purely direct mapping.

2A: A balanced tree has several useful characteristics. No matter the size, a balanced tree has a lookup time complexity of O(logN), which contrasts with a hash table where hash collisions can cause the time complexity to increase if the load factor is too high. Since a balanced tree also maintains a sorted order of elements, it makes it easy to traverse the tree in either ascending or descending order, which could be useful in this application when displaying the information of all packages at once. A hash table with direct mapping on the other hand always has O(1) lookup times because there is no possibility of hash collisions. This does however require that all keys be non-negative integers, and it can cause extremely large table sizes because there must be as many buckets as there are possible keys. For this application I found these drawbacks to not be worth it, and so implemented my hash table with linear chaining.

# L:

No outside sources used.