**Program Overview**

**Overview:**

The program is written using Python 3.7. The flow of the program is written to simulate an entire work day of deliveries. I originally thought writing the program to run in real time would be too cumbersome so chose this method. All calculations are done prior to loading the user interface. The flow is as follows:

1. Import package, address, and routes from CSV files into memory data structures.
2. Create a graph using each delivery address as a vertex, and distances as edge weights.
3. Create 3 truck objects to carry the packages and track delivery times.
4. Packages are loaded into the trucks using the most restrictive rules first:
   1. All packages that are Truck 2 only.
   2. Any that are delayed or wrong address are set for truck 3 which won’t leave until one of the other two return.
   3. All early deliveries go to the first empty truck.
   4. Once the special rules are all loaded, any remaining packages with the same address as a package on truck 1 are loaded into it, then set aside Truck 1 for only early deliveries.
   5. For the two remaining trucks, match packages to the truck by zip codes of already loaded packages.
   6. Load all remaining packages into Truck 3, allowing either 1 or 2 to get back in time to get Truck 3 early deliveries out.
   7. If any packages remain, load into Truck 2.
5. Run the delivery algorithm on Trucks 1 and 2, both with start times of 8 AM.
6. Run the delivery algorithm on Truck 3 with the start time of Truck 1 return time to simulate that driver taking out Truck 3, prioritizing early delivery packages.
7. Run user interface to interact with the generated program data.

The program runs locally, so no sever or communication protocols are utilized.

**Screenshots:**

Most of the program consists of ‘under the hood’ calculations, making screenshots difficult. The program opens with a listing of the final package delivery times, miles driven by the trucks, and finally lands on a options menu. The screenshots show the program flow in this order:

* PackageDeliveryTimes.png : Lists all packages ID, deadline, and final delivery time.
* TruckDistances.png : Lists each total truck distance as well as total distance for all trucks.
* SinglePackageStatusExec.png: Shows execution of menu option 1, getting the status of a single package based on ID and time provided.
* The three package status screenshots show menu option 2 used at 3 separate times.
* Menu option 3 is shown in the truck distances screenshot.
* RoutesFromHub.png shows execution of menu option 4. You can put in any known address and it will output the most efficient route to all other known addresses.
* QuitProgram.png : Shows program completion and exit.

**Algorithm:**

I chose the **Dijkstra’s Shortest Path** as the core delivery algorithm. I chose this algorithm for two reasons:

1. Because the algorithm can accept and process a list of any size, it allows for scalability. This will be advantageous if the company using this software continues to grow and finds they need to process more packages.
2. This algorithm offers better efficiency than a standard greedy algorithm by backtracking distances between the start and all vertices to ensure that each path is the absolute shortest distance from the start vertex rather than just the most expedient path from vertex to vertex.

Another algorithm that could have been used for a simpler implementation would be a standard **greedy algorithm**. This algorithm would pick whichever vertex was the next closest, regardless of overall efficiency. For example:

Vertex A -> Vertex B = 2

Vertex A -> Vertex C = 1

Vertex B -> Vertex D = 1

Vertex C -> Vertex D = 5

Using these edge weights, if we want to travel from Vertex A to Vertex D by picking the shortest route at each vertex, we would go:

Vertex A -> Vertex C -> Vertex D = 6

Picking Vertex C makes the most sense from A because 1 < 2 making A -> C the shorter path than A -> B. However, A -> B -> D = 3, making it the shorter path overall. Dijkstra’s Algorithm takes this into account, making it the more efficient algorithm.

Another option would be to try every possible permutation of routes to find the shortest path for delivery of each package. This is an example of a **brute force** approach. The efficiency of this approach would be a factorial of V, O(V!). This approach would be wholly impractical with only ten to 20 vertices.

**Routes Data Structure**

For the routes table, I first created a **graph object**, and added each address in as a **vertex object**. The vertices are stored in a dictionary attribute of the Graph class using the address string as the key, and the vertex object as the value. This allows each vertex to be referenced by the address string.

The distances between addresses are then stored as a two-dimensional adjacency matrix using the edges attribute of the Graph class. Each index in the list is another list with all distances from one location to the next. Each nested list represents one row of the graph. This allows us to get the distance between two vertices by referencing two indexes as graph coordinates.

**Package Hash Table**

In order to easily organize package data, **package objects** are created for each package with attributes to hold each data point. These objects were stored in a simple **hash map** constructed as a list of tuples with index 0 as key and index 1 as value:

[(key, value), (key, value)]

The **hash function** is a simple (key % length of table) function. If the key is an integer data type, it is used directly in the function. For any other data type, each character is converted into it’s ascii value and added together before being entered into the function. This method allows the table to be collision free for the given project, while still allowing for use in storing other types of data. The **hash table** self-adjusts by evaluating the number of items to be inserted initially into the table and setting its length to that value.

The **hash table** uses a **linear probing** algorithm to handle any possible collisions. If it tries to insert an item into a bucket containing another item, it will shift to the next bucket up and continue until an empty bucket is found.

Retrieving data works in a similar fashion. Each bucket in the table is marked either EMPTY\_SINCE\_START or EMPTY\_AFTER\_REMOVAL, or it contains a data item. When searching for an item, the key is hashed, and the index is checked for the data. If the data exists, index 1 of the tuple contained int hat bucket is returned. If not, the search continues to the next bucket until the data is found, or an EMPTY\_AFTER\_REMOVAL is found in which case the data is not in the table. On average this hash table design should perform with a constant runtime O(1), and worst case every insertion causes collision, O(n).

In addition, the space complexity of this simple hash table is directly relative to the number of buckets, making it O(n).

No bandwidth is used in the manipulation of data as all computation takes place locally.

A good alternative to this data structure would be a **Binary Search Tree.** This tree could be implemented using the same (key, value) tuple as the hash table. This data structure would yield similar efficiency with a worst case of O(H) comparisons, where H is the height of the tree. Searches can potentially be made in O(log N), a bit slower than the hash table but still pretty good. The biggest drawback would be complexity of implementation.

Using the built in Python dictionary would also be a good alternative. This is essentially a built-in hash table and would be essentially interchangeable with the hash table used.

**Scalability and Final Observations**

If I were to do this project over again, I would write the program to run in real time rather than simulating time. The simulation created several complications that could have been avoided by just changing the clock to get delivery statuses.

There are also a few efficiency and scalability improvements that can be made. Using a heap rather than a list for the core algorithm would increase runtime performance for larger package sets, and the entire function of creating and loading the trucks could be re-worked for better scalability.

The special package instructions were the biggest hurdle in sorting and loading the trucks. Without those it would be easy enough to group packages based on address and zip codes. I think a better implementation would be to have a system of codes for special instructions. That way code can be written to handle them specifically.

There are a number of places where I hard coded variables, such as the hub address. For scalability, a much better way to do this would be with a config file that could be edited to change things like the hub address, number of trucks, number of drivers, route and package data file locations, etc.

In addition, a config file would allow the use of this program for different cities. The local hub data, number of trucks, and number of drivers could be added into the config file with each city running it’s own local version of the program. That data could then all be transmitted to a central server location in order to consolidate company wide data.

In order to accommodate these changes, I used classes and functions wherever possible in the program run-time. This will allow whoever maintains the program to change single classes or methods in order to effect the entire program, rather than change multiple places.