A.  Identify a named self-adjusting algorithm (e.g., “Nearest Neighbor algorithm,” “Greedy algorithm”) that you used to create your program to deliver the packages.

To determine what route would be taken by a truck, the Nearest Neighbor algorithm was used as a self-adjusting algorithm. Nearest neighbor is a greedy algorithm to determine a circuit path where all given vertices are visited. This implementation of the algorithm has space complexity of O(n) and time complexity of O(n^2).

B.  Write an overview of your program, in which you do the following:

1.  Explain the algorithm’s logic using pseudocode.

CREATE location list, current location pointer, smallest location pointer, smallest distance pointer, and an unvisited queue list.

FOR package in package list:

CHECK to see if location in list already, and if not, add it to unvisited queue.

WHILE unvisited queue list is greater than 0:

FOR locations in unvisited queue:

CHECK for smallest distance and location from current location.

UPDATE location list, current location.

REMOVE current location from unvisited queue list.

IF unvisited queue list is empty:

ADD hub at end of list to complete circuit.

RESET smallest distance pointer for next iteration of WHILE loop.

2.  Describe the programming environment you used to create the Python application.

       PyCharm 2023.1.4 (Community Edition) was used to create the python files.

3.  Evaluate the space-time complexity of each major segment of the program, and the entire program, using big-O notation.

      Overall program space complexity was O(n) and time complexity was O(n^2).

4.  Explain the capability of your solution to scale and adapt to a growing number of packages.

A direct access table was used to hold package objects. The main advantage to this is no collisions, but at scale the data structure would have a large space requirement (one key per package). Larger package data sets would require hashing for (unique) keys, and handling collisions with either chaining or open addressing when the hash values are the same.  A greedy algorithm implementing a priority heap data structure could be used to sort which packages to load first as more packages are handled each day. The graph class implements the distances (edges) between all delivery locations (vertices), forming a complete undirected graph through an adjacency matrix. It also stores the delivery locations and their respective indexes for reference in the adjacency matrix, allowing for quick searching of individual distances. Data cleaning and formalized formats for data sets would be needed to ensure correct parsing of data into the program.

5.  Discuss why the software is efficient and easy to maintain.

The code is separated into various modules to make each functionality of the program distinct. Comments are used to succinctly describe each method, function, and module. All variables are also succinctly named to make clear the data and data type it's supposed to hold. The program uses classes under an object-oriented programming model, so future changes (such as adding a new algorithm) can be easily integrated as a new class.

6.  Discuss the strengths and weaknesses of the self-adjusting data structures (e.g., the hash table).

A direct access table was used to hold package objects. The main advantage to this is no collisions, but at scale the data structure would have a large space requirement (one key per package). Larger package data sets would require hashing for (unique) keys, and handling collisions with either chaining or open addressing when the hash values are the same.

D.  Identify a self-adjusting data structure, such as a hash table, that can be used with the algorithm identified in part A to store the package data.

1.  Explain how your data structure accounts for the relationship between the data points you are storing.

A hash table is an implementation of a dictionary that holds key value pairs (hash, data) for quick insert, removal, and search of specific data. The nearest neighbor algorithm uses the quick searching of package objects in the hash table to pull needed package attributes in calculating nearest distance between each delivery location using the graph attributes.

I.  Justify the core algorithm you identified in part A and used in the solution by doing the following:

1.  Describe *at least***two** strengths of the algorithm used in the solution.

The nearest neighbor algorithm is easy to implement into a program and intuitively understand. It also handles self-adjusting graph data structures well since it only checks for the nearest vertex at each iteration of the algorithm.

2.  Verify that the algorithm used in the solution meets *all* requirements in the scenario.

With various heuristic changes to the algorithm, the program solution meets the deadline and distance requirements of the project.

3.  Identify **two** other named algorithms, different from the algorithm implemented in the solution, that would meet the requirements in the scenario.

a.  Describe how *each* algorithm identified in part I3 is different from the algorithm used in the solution.

The Multi Fragment algorithm is a variation of the nearest neighbor algorithm that considers "clusters" (fragments) of vertices. MF takes advantage of the proximity within clusters to further optimize the distance traveled. A heuristic approach could be to do a quick search of undelivered package locations in the hash table until one with a distance under the next nearest distance is found (or other bounds), swap the packages (if the old/new package aren't restricted by certain conditions) and use the new package location in the algorithm. Depth First Search (DFS or brute force) exhaustively searches for the optimal path but takes considerably more time than nearest neighbor. DFS checks all combination of edges and vertex paths, while keeping track of the most optimal path until all combinations have been explored. A heuristic approach could be to use a less time complex algorithm (such as nearest neighbor) and then run DFS until a more optimal solution is found set by a bound. This could be a 5% decrease in path distance, or less than 46 miles (assuming 3 truck trips are made).

J.  Describe what you would do differently, other than the two algorithms identified in I3, if you did this project again.

 If the business had the resources available, I would implement machine learning on historical data, to find patterns in local delivery routes. Common delivery sub routes can be optimized and integrated into the NN algorithm, saving time in finding an acceptable route. I would also add thorough exception handling throughout the code, to reduce occurrences of bugs and aid in maintaining the program.

K.  Justify the data structure you identified in part D by doing the following:

1.  Verify that the data structure used in the solution meets *all* requirements in the scenario.

a.  Explain how the time needed to complete the look-up function is affected by changes in the number of packages to be delivered.

The direct access table would retain a search time of O(1) regardless of table size. The graph has a search time of O(n^2) so there would be polynomial increase with each new location and respective distances added to the graph.

b.  Explain how the data structure space usage is affected by changes in the number of packages to be delivered.

A direct access table key and value pair map 1 to 1, so the space usage would grow linearly as package numbers increase. The graph data structure would have its vertex dictionary increase linearly for packages and distances (edge weights).

c.  Describe how changes to the number of trucks or the number of cities would affect the look-up time and the space usage of the data structure.

The direct access table is static and would be unaffected by the number of trucks or cities as each respective module performs search functions independently and separately. The graph structure would not be affected by number of trucks or cities for the same reasons, with only space increasing linearly with more city key/value pairs added. The graph structure search time could increase with polynomial growth though if the number of packages loaded (indirectly increasing number of locations searched) increased.

2.  Identify **two** other data structures that could meet the same requirements in the scenario.

a.  Describe how *each* data structure identified in part K2 is different from the data structure used in the solution.

For the package hash table, a table implementing chaining or open addressing would scale better for larger data sets and would need to handle collision. Chaining stores keys with the same hash values in "buckets", so each bucket would be searched to find a specific key within an array of same hash values. Open addressing would search for the next available bucket if there's a collision through various implementations, such as linear or quadratic probing. The graph structure could have implemented an adjacency list instead of a matrix, with each vertex holding a sub array of adjacent vertices, with each vertex representing an edge. (distance). The advantage to an adjacency list is the space complexity of O(V+E) as each vertex is only represented once and each edge twice. The disadvantage is with sparse graphs, where searches could end up being closer to worst time complexity due to needing to search for adjacent vertices more exhaustively.