A. Identify a named self-adjusting algorithm that you used to create your program to deliver the packages.

Nearest Neighbor Algorithm

B1. Explain the algorithm's logic using pseudocode.

```
AlgorithmToDeliverPackages(Truck, HashTable, UserTimeLimit)
    for each package id in Truck:
        change package status to "en route"
    while Truck has packages AND Truck time is less than UserTimeLimit:
        next_location = use AlgorithmToFindNextLocation to find next closest address to visit
        potential_mileage = the distance it would take to travel from Truck location to next_location
        potential_arrival_time = Truck time + hours it would take to travel potential_mileage
        if potential_arrival_time is greater than UserTimeLimit:
            return Truck
        else:
            change Truck location to next_location and add potential_mileage and potential_arrival_time
        for each package id in Truck:
            Package = package_id
            if Package address equals Truck location:
                remove Package from Truck
                change Package status to "delivered" + the time it was delivered
                update HashTable with Package
    potential_mileage = the distance between Truck location and the "hub"
    potential_arrival_time = Truck time + hours it would take to travel potential_mileage
    if potential_arrival_time is greater than UserTimeLimit:
        change Truck location to next_location and add potential_mileage and potential_arrival_time
    return Truck

AlgorithmToFindNextLocation(HashTable, CurrentLocation, AllPackageIDs
    result_address = address of first package in AllPackageIDs
    min_distance = the distance between CurrentLocation and result_address
    for each package id in AllPackageIDs:
        potential_address = address of package id
        distance = distance between CurrentLocation and potential_address
        if distance is less than min_distance:
            make min_distance equal to distance
            make result_address = potential_address
    return result_address
```

B2. Describe the programming environment you used to create the Python application.

IDE: PyCharm Community 2022.1
OS: "Debian GNU/Linux 11 (bullseye)" Linux Development Environment on "Chrome OS Version 100.0.4896.133"
Hardware: Chromebook 14a-na1043cl

B3. Evaluate the space-time complexity of each major segment of the program, and the entire program, using big-O notation.

Each segment time complexity is written above the function using a comment.
Entire program time complexity is written in main.py

To reiterate, my entire program is n^2.

My package delivering function runs for each trip a truck takes and is O(n) because for each truck stop (max 16), it searches the HashTable for the address field of each package. The HashTable is O(n) because it has chaining with linked lists.
My method to load packages onto trucks is manual but in the worst case, it would load 1 package on each truck, meaning the package delivering function would run n times.
This means O(n*n) would lead to O(n^2) for my entire program.

B4. Explain the capability of your solution to scale and adapt to a growing number of packages.

The solution can scale/adapt to an increase of packages. The data structures that handle the packages and distances can scale without change, but one might want to increase the size of the hash table to improve package lookup speeds. The functions that read the CSVs and populate the program with packages and distances would need no change. The truck and package objects would need no change but an alteration to the package print function would improve visibility of the console UI. The algorithm would need no change. The main method which handles the UI and package loading would have to be altered because the packages are manually loaded onto trucks and sent out in manual orders and times.

B5. Discuss why the software is efficient and easy to maintain.

The software is efficient.
One list stores address strings, and their index in this list is used to find the distance between two addresses using a second 2d list which stores floats. This replaces the need to store the distance between two addresses as many variables.
The packages are stored in a hash table which allows for fast lookup speed.
The algorithm is efficient in finding a solution because it uses the Nearest Neighbor Algorithm, which reliably finds a fast solution.

The software is easy to maintain because the functions and objects are stored in multiple modules instead of one big python file. The code is organized and structured, allowing for easy changes.

B6. Discuss the strengths and weaknesses of the self-adjusting data structures (e.g., the hash table).

The strength of the hash table is that it allows for constant lookup speed. Compared to iterating an entire list of packages, the hash table allows for faster time complexity.
The weakness of the hash table is that if the number of packages grows too large, there can be many collisions, slowing down the lookup speed.

C1. Create an identifying comment within the first line of a file named "main.py" that includes your first name, last name, and student ID.

DONE

C2. Include comments in your code to explain the process and the flow of the program.

DONE

D1. 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. Explain how your data structure accounts for the relationship between the data points you are storing.

My solution uses a hash table as the self-adjusting data structure to store packages for access in constant lookup time. The key is the package id and the value is a Package object which contains the package id, address, deadline, city, zip code, weight, and delivery status. My Truck object stores the packages using their package IDs, which allows for fast lookup of a truck's package's address, status, etc. The hash table is used to search for nearest addresses, change delivery status, etc. My algorithm only reads the hash table but for correcting package 9's address, the hash table is modified once.

E. Develop a hash table, without using any additional libraries or classes, that has an insertion function that takes the following components as input and inserts the components into the hash table

DONE

F. Develop a look-up function that takes the following components as input and returns the corresponding data elements

DONE

G. Provide an interface for the user to view the status and info (as listed in part F) of any package at any time, and the total mileage traveled by all trucks. Provide 3 screenshots.

DONE (screenshots included)

H. Provide a screenshot or screenshots showing successful completion of the code, free from runtime errors or warnings, that includes the total mileage traveled by all trucks.

DONE (screenshots included)

I1. Describe at least two strengths of the algorithm used in the solution.

One strength of the nearest neighbor algorithm is that it is simple to implement and understand. This algorithm is specifically simple because essentially, all it does is find the lowest number in a list of numbers, similar to a min() function. This simplicity applies to this scenario because even without an algorithm, a truck driver would be able to find his next package destination by simply looking for the next closest drop-off location. This would allow one to approximate the next location without having to run countless simulations.

Another strength of the nearest neighbor algorithm is that it delivers packages in "clusters" since delivery locations closest to each other get completed. This means that when a delivery driver finishes his route, he will have completed an area and reduce the chance that the same "area" will have to be visited by the next truck driver. Instead of zig-zagging across the city/town, the driver stays in a central area so that even if the route is suddenly canceled, the next driver knows most packages in the area have been dropped off already.

I2. Verify that the algorithm used in the solution meets all requirements in the scenario.

As seen in screenshot "verification_proof.png", all packages have been delivered and the total mileage of all trucks combined is only 124 miles by the end of the day (5pm). The algorithm meets the requirement of being self-adjusting because the route determined for the truckload is dynamically chosen based on what packages the algorithm is given.

I3. Identify two other named algorithms, different from the algorithm implemented in the solution, that would meet the requirements in the scenario. Describe how each algorithm identified in part I3 is different from the algorithm used in the solution.

Two other named algorithms are the Cheapest Insertion and the Nearest Insertion algorithm. These two would meet the requirements of the solution because they are self-adjusting, find a “optimal” shortest path, don’t have collisions,

Both algorithms do not have collisions, since nodes are inserted into the path, not visited again. The self-adjusting part comes from the algorithms deciding dynamically which node to go to next.

The Cheapest Insertion algorithm starts off with 2 points. The algorithm constantly inserts the cheapest point between two existing points already in the route. The cheapest point is the point that increases the total “path” of the route the least.

The Nearest Insertion algorithm also starts off with 2 points. The algorithm constantly inserts the nearest point between two existing points already in the route. The difference is that the nearest point is the point that is the least amount of distance to a point already in the route.

But both algorithms are different from my Nearest Neighbor algorithm because both algorithms must decide the path before the truck starts delivering. With the Nearest Neighbor algorithm, the truck can go along with the algorithm as it decides the next path. However, the Cheapest Insertion and Nearest Insertion algorithms insert new locations into the route and so the truck must wait for the route to finish before traveling. When running countless delivery simulations, this slight delay might add up in the grand scheme of things.

Another difference is that the Nearest Neighbor algorithm does not insert a node into the path, it only appends it to the end of the path.

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

I would make the loading of packages into the trucks self-adjusting as well. Currently, I have hand-selected which packages go on which trucks and exactly what time they leave. Coding this new self-adjusting package loading would be extremely difficult. The details of modifying the project to be able to load projects automatically would include the need for more algorithms.

Instead of calculating a single route once, like my current Nearest Neighbor, I would have to repeatedly run multiple routes for every single combination of packages. The larger my input grows (number of packages), the more exponentially large the possible routes become because adding a single package does not linearly add a specific number of routes.

I would try to reach the maximum of 16 packages in one truck and also use “priority rankings” for the packages in the truckload so that packages with deadlines can go alongside other packages without deadlines. Currently, if you combine deadline packages with non-deadline packages, then there is no system to deliver the deadline packages first. So I would have to create multiple lists inside the already existing list of packages inside the truck.

I would have to create a loop to try every possible combination of package inside the truck AND also run the routing algorithm to see if that combination is the shortest.

K1. Verify that the data structure used in the solution meets all requirements in the scenario.

As seen in the “verification_proof.png” screenshot, the total miles added to all trucks is shown and less than 140, all packages are delivered on time, and the hash_table lookup function works because the console ui uses the lookup function for to print each package.

K1A. Explain how the time needed to complete the look-up function is affected by changes in the number of packages to be delivered.

As the number of packages increases, the lookup function is affected negatively and slows down. The time needed to complete the function increases because filling the hash table above capacity causes collisions to occur, which force the function to iterate through each bucket/linkedlist in each index of the hash table. So more packages causes more collisions, causing more iterations needed to reach packages stored at the end of the linkedlist.

K1B. Explain how the data structure space usage is affected by changes in the number of packages to be delivered.

The more packages added to the hash table, increases the size of the linked lists/buckets at each index. The hash table has a specified capacity and available indexes. But once all indexes fill up, then there are linked lists in each index that also fill up, meaning that each linked list get larger as more packages are added.

K1C. 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.

Adding more trucks to the solution would allow for more packages to be delivered quicker while removing trucks would make packages be delivered later. Regardless of how fast packages get delivered, the number of trucks would not affect the hash table containing the package data because the solution requires the user to be able to see ANY package status. This means regardless of the package being at the hub, enroute, or delivered, the hash table MUST still store the package data. Thus, the lookup time and space usage of the hash table would remain the same.

Adding cities would not affect package data because packages can be delivered to the same address and some addresses might not receive any packages at all. More cities does not directly correlate with the number of packages UNLESS the cities are being added as a direct result of new packages being added to the hash table that contain those new cities. Thus, since the hash table does not store city data, it would not affect the lookup time nor space usage. HOWEVER, if the cities were being added BECAUSE there were more packages being added with new addresses/cities, then lookup time and space usage would both increase because there is more data to sort through and store.

K2. Identify two other data structures that could meet the same requirements in the scenario.

One other data structure is a singe list that stores package data at each index and uses the UNIQUE package ID as way to index the package data into the list. It would be similar to the hash table except lookup time would be longer. The list would automatically fill with empty indexes to meet the demands of irregular package IDs
This data structure would be self-adjusting because the list would dynamically increase to meet the needs of the number of packages. The packages would also automatically assign themselves to an index instead of manually having to insert the package into a list.

Another data structure would be the use a priority queue that holds package objects. Each package would be inserted into the priority queue and given a higher priority the closer/earlier their deadline is. This would allow the trucks to quickly grab the most urgent packages to deliver. The priority queue would meet the requirements of the scenario because it is able to store package data and iterate by pop/pushing through package objects if a search is needed. The priority queue would be self-adjusting because it would dynamically change the priority of incoming packages based on deadlines. There would be no collisions as each package is its own object. It would allow for optimal/short paths based on the premise of prioritizing urgent deadline packages first.

K2A. Describe how *each* data structure identified in part K2 is different from the data structure used in the solution.

The single list (described in K2) is different from the hash table because the hash table uses a hashed key to index the package instead of the list's index using package id. Additionally, the hash table allows for linked lists at each index. The hash table allows for faster lookup than the single list since it has to iterate over the entire list and possible empty indexes.

The priority queue (described in K2) is different from the hash table because the queue does not use an indexing system. The priority queue allows grabbing of packages not by package ID but by deadline. One could iterate through the priority queue to find a specific package ID but it would again take longer to iterate through possibly the whole queue instead of the hash table.