# A:

I chose the nearest neighbor algorithm for my program. The algorithm is self-adjusting by iterating through the list of unvisited vertices and selecting the closest one. Whichever vertex is closest to the current vertex is added to the sorted list and then removed from the list being sorted.

# B1: Nearest Neighbor Algorithm Pseudocode

while path to sort is not empty:

for all locations in the path to sort:

if the distance to travel to location is shorter than the previous distance to travel:

replace the previous location with the new shorter distance location

if the shortest location is not in the sorted path:

add the location to the sorted path

remove location from path to sort as that location/vertex has been visited

return sorted path

# B2: Development Environment

The programming language used was Python 3.8.3 and it was programmed in the VSCode IDE. The operating system used was Windows 10.

Hardware used:

CPU: i7-9700k

GPU: AMD 5700XT

RAM: 32GB

Memory: 1.5TB

# B3: Space-Time and Big-O

The space-time complexity/big-O of each major block of code and for each function has been added to the code comments.

# B4: Adaptability

For the current scale of the business, the program is adequate. The distances between package deliveries are not extreme and are contained to a very small area. Also, since only ~40 packages are expected to be delivered per day, the program should remain adequate for the near future.

The problem appears when scaling up. If, for instance, another city was added to the delivery possibilities the way data is handled in this program presents a very real challenge. The chaining hash table that the package data is stored in should be fine for scalability to this level. At worst, the hash table will perform at O(N). As such, purely increasing the number of packages without growing the number of possible locations is not much of a concern for this program and it would be well suited to that role.

The graph, however, is the real problem to scaling this program. As locations are added to the graph, the graph scales at O(N^2) due to it being a complete graph. Every point is connected to every other point by a unique and undirected edge. This is necessary for how the algorithm calculates an optimal path. We need to know the distance from any location to any other location. As such, the graph will suffer as the number of locations grows.

The algorithm is the other problem facing the expansion of this program. With the nearest neighbor algorithm, because it can’t look any farther ahead than the next step you can end up very far from where you need to be on a very inefficient path. This problem is lessened when all the locations are close together, but as more locations are added and the space between possible locations gets larger, the algorithm will be worse and worse. The algorithm also scales at O(N^2), so performance will suffer as more locations are added.

# B5: Software Efficiency and Maintainability

This program is efficient. It runs at a time complexity of O(N^2). For the scale of the use of this program, that should be sufficient.

The code should be very simple to maintain. The algorithm and data structures were chosen for their simplicity in implementation and maintenance. The code is relatively short and flows in a logical manner. The code itself is broken into 4 separate files, each with a distinct purpose as explained in the comments. The code is well commented, and all functions are explained, and the complexity of each function is noted in the comments.

The code should also be easy to modify and perform maintenance on. For all of the reasons noted above and because the underlying data structures can be kept the same, even if the data they hold is processed in a different manner.

# B6: Self-Adjusting Data Structures

The chaining hash table that the package data is stored in should be fine for scalability. At worst, the hash table will perform at O(N). As such, purely increasing the number of packages without growing the number of possible locations is not much of a concern for this program and it would be well suited to that role. Since the hash table is chaining, it will never be full and the upper limit for packages is infinite (in theory). The downside of this chaining hash table is that to find any element (besides the first one) you must iterate through the hash table until you find the element you want since keys are stored in a linked list. This contrasts with the much speedier O(log(n)) average search of a data structure like a binary search tree or even O(1) of a perfectly hashed hash tree.

# C: Original Code

All Code is original and runs without error or warnings. The program is interfaced with through the console after running the main.py file.

# C1: Identification Information

The author’s name and student ID are commented at the top of the main.py file.

# C2: Process and Flow Comments

The code is relatively short and flows in a logical manner. The code itself is broken into 4 separate files, each with a distinct purpose as explained in the comments. The code is well commented, and all functions are explained, and the complexity of each function is noted in the comments.

# D: Data Structure

A chaining hash table was created to store the package data for this program. The hash table is self-adjusting because it can take in different numbers of packages and still function the same. It adapts based on different input sizes.

# D1: Explanation of Data Structure

The hash table implemented is a chaining hash table that stores package data in 10 different “buckets” or lists. Each added element is added to a bucket based on the outcome of a mod(%) operation between the key and the length of the hash table. Once a bucket is selected, the item is appended to the associated bucket. During a search, the function first finds which bucket the value should be in based on the key. Then the list of keys is iterated through to see if the key exists in the list. If it is found, the associated value is returned. For this hash table, the package ID is the key.

# E: Hash Table

This hash table has an insertion function (add) that uses the package ID as the key and the rest of the package information as the value. All required components of the package are stored in the hash table.

# F: Look-Up Function

The hash table has a look-up function (get) that returns all the required package components based on a given package ID as the key.

# G: Interface

The program is interfaced with through the console. The user enters any valid time, and the current state of all packages is output through the console to the user. The status of all packages includes the location of the package and if it has been delivered, includes the time the package was delivered and by which truck. The miles traveled by all the trucks to the desired time are also output to the user, even if all packages are not yet delivered.

# G1-G3: Status Checks

The screenshots of the program output are included in the file submission. “0920status.png” satisfies G1, showing the state of all packages at 09:20 and the miles traveled by all trucks up to that point. “1020status.png” satisfies G2, showing the state of all packages at 10:20 and the miles traveled by all trucks up to that point. “allpackagesdelivered1300status.png” satisfies G3, showing the state of all packages at 13:00 and the miles traveled by all trucks up to that point. All packages are delivered at this point.

# H: Screenshots of Code Execution

“allpackagesdelivered1300status.png” satisfies H, showing the state of all packages at 13:00 and the miles traveled by all trucks up to that point. All packages are delivered at this point. The code is shown as completing without warnings or errors.

# I1: Strengths of the Chosen Algorithm

As this program has a relatively limited scope for the near and foreseeable future, I prioritized simplicity. The nearest neighbor algorithm isn’t the fastest or most optimal algorithm, but it is the easiest to implement and get the program into the company’s hands as fast as possible and start optimizing their routes.

Efficiency is another reason the nearest neighbor algorithm was chosen. Although it has a O(N^2) worst-case complexity, compared to other possible solutions to this problem it is relatively speedy. This is because this algorithm only provides an approximation of the optimal route. Calculating the actual optimal route would be much more expensive, complexity-wise. As such, this gives the algorithm some headroom for the expandability of the program. Especially if a new hub is used for each new city the company expands into.

# I2: Verification of Algorithm

The nearest neighbor algorithm satisfies all requirements.

1. The packages were delivered with a total driven distance, of all trucks, of 115.5 miles.
2. All packages were delivered before end of day and all packages were delivered by their specified required delivery time if one was supplied.
3. All packages were delivered in accordance with any special notes or requirements.
4. All the above information is accurate and output to the user about every package at any valid time the user inputs.

# I3: Other Possible Algorithms

One alternative algorithm would be Dijkstra’s shortest path algorithm. The algorithm is similar in that it is greedy. It will always make the locally optimum choice and not consider if a choice is optimal for the entire solution. What Dijkstra’s algorithm does is find the shortest path from the start to all other points on the graph. While the nearest neighbor no longer considers the starting point past the initial step, choices are only made between that location and other adjacent locations. Dijkstra’s algorithm has the advantage in that it will consider the node visited before the current location, the predecessor when making its choice of the next location. Dijkstra’s algorithm was considered, but nearest neighbor algorithm was chosen purely for simplicity of application, even if it has a greater potential to be suboptimal due to its slightly more limited scope when making choices.

The farthest insertion algorithm would also likely satisfy the requirements. The farthest insertion algorithm selects a starting point and connects to the location that is the farthest away from it. After that, it will constantly search for the location that is farthest from any location already selected. It will then place that location between the two selected locations that will make the overall path the shortest. The algorithm repeats until all locations are selected. This algorithm has the same efficiency as nearest neighbor, at O(N^2). Just like Dijkstra’s algorithm though, it was not selected due to the relative complexity of implementation. Simplicity was the most important factor considered in selecting an algorithm.

# J: Different approach

Since this is a simulation of time and not actual time passing, I struggled with how to handle time. Right now, the time is handled by using datetime objects with today’s date put into the year, month, and day values. This isn’t ideal as this program would not be able to be used in a real-world setting unless that part of the code is changed every day. I could have used datetime.time objects instead to track time, but then I could not use timedelta to simulate the passage of time. So, my solution, while it works, is not a very elegant one. I would try to find a more real-world solution to the simulated passage of time problem if I did the project over again.

Also, I think the farthest insertion algorithm would be interesting to implement as I think it could have some real boosts to the optimization of the path. It wouldn’t be any more efficient in relation to complexity, but it would likely cut down on miles. If I wasn’t so focused on simplicity, I would try to implement that algorithm.

# K1: Verification of Data Structure

The chaining hash table used satisfies all requirements.

1. The packages were delivered with a total driven distance, of all trucks, of 115.5 miles.
2. All packages were delivered before end of day and all packages were delivered by their specified required delivery time, if one was supplied.
3. All packages were delivered in accordance with any special notes or requirements.
4. All the above information is accurate and output to the user about every package at any valid time the user inputs.
5. An efficient (O(N)) hash table is used and there is a look-up function included in the hash table.

# K1A: Efficiency

Since the hash table used is a chaining hash table where all items are sorted into buckets based on their key, the effect of more packages being added is O(N) in terms of complexity. The correct bucket must be searched for the proper key, so each package added increases the time and storage needed in a linear manner.

# K1B: Overhead

Since the hash table used is a chaining hash table where all items are sorted into buckets based on their key, the effect of more packages being added is O(N) in terms of complexity. The correct bucket must be searched for the proper key, so each package added increases the time and storage needed in a linear manner.

# K1C:

Adding more trucks or more cities would not affect the chaining hash table used. The scope of the hash table in relation to more cities or trucks would be unchanged. That functionality is handled by the graph in the program. So, no matter what other cities or trucks are added, the hash table will still function the same at O(N).

# K2/K2A: Other Data Structures

A direct hashing table could have been used instead of the chaining hash table used. In a direct hashing table, each key is assigned to a unique bucket. This makes searching much faster as you don’t have to search inside the bucket for the correct key. This gives a search and insertion complexity of O(1). The downside of this type of hash table though is that it can become very large since each key has its own bucket. The reason this type of hash table was not chosen, even though it is adequate for the problem, was because of scalability. If the company eventually scaled to 40,000 packages instead of 40, a chaining hash table would keep the size of the hash table much smaller.

A linear probing hash table is another style of hash table that could have been used. If a collision occurs in a linear probing hash table, then the search will start at the bucket where the collision occurred and continue sequentially until an empty bucket is found. Like the direct hashing table, the search complexity and insertion complexity of the linear probing table is O(1). The downside is the same as well, as the number of packages grows, the linear probing hash table is also going to get bigger and bigger. As such, the chaining hash table continues to make more sense from a simple future-proofing perspective as it will be able to keep the size of the hash table relatively small if the company were to expand.

# L: Sources

No outside sources were used.

# M: Professional Communication

All submitted content is organized with attention to detail. Grammar was checked before submission.