Benjamin Gamman
Data Structures and Algorithms: “WGUPS” Program

A: Algorithm Identification

The core of my algorithm to route the packages is a greedy selection algorithm, though it is embedded in much longer and more complex algorithms for sorting and routing. Those sorting and routing algorithms employ greedy selections along with other constraints and checks to sort and deliver all packages for which information is provided.

B1: Logic Comments

Core Greedy Selection Algorithm (a method of the TruckList class in Truck.py):

The fundamental greedy selection algorithm that I used to choose most stops along each route functions as follows:

```
Set next stop to default value (100 miles away)
For each stop remaining in the load’s route:
	If closer to the preceding stop than the currently selection:
		Set this location as the next stop
For each stop remaining in the load’s route:
	If further from the previous stop by less than 0.5 miles
	and closer to the hub than current selection:
		Set this location as the next stop
Return next stop
```

This selects the nearest available location as the route’s next stop, then uses another loop to check for options that might be better to temper the “greediness” of the algorithm and take the big picture into account just a bit by prioritizing locations further from the hub. This helps to avoid leaving the last stop far away from the hub and adding a significant distance to the end of the route. The greedy selection algorithm on its own has worst case space complexity of $O(N^2)$ and time complexity of O(N) and for each time it is called: it utilizes the matrix of distances which is $O(N^2)$ in size, and each loop checks up to N points on the route giving O(2N) = O(N) time. This core algorithm is incorporated into much longer algorithms for sorting and routing packages. Those are outlined below, with in-depth explanations of their execution in the code’s comments.

Add Package to Load (“add” method of the Load class in Load.py):

This method is used to add a package, and recursively other packages that belong with it, to a load (called as Load.add(Package). It is called repeatedly in the following sorting algorithm.

```
If package not already sorted and load not full and valid truck requirement:
	Add package ID to load’s package list
	Update package with assigned load index
	Mark package as sorted
	If necessary update load’s truck requirement
	If package is bundled with other packages:
		For each bundled package:
			If bundled package not sorted and load not full:
				Add package to the same load (recursive call)
	If package’s destination known:
		Add destination to load’s stops
		For each package in that location’s package list:
			If package is not delayed and not sorted and load not full:
				Add package to the same load (recursive call)
```

The add method can call itself recursively, but also has safeguard checks built-in to only do so if the target package is not already sorted. This means that while a call to the function not resulting in recursion would have time complexity O(N) (checking each other package) and space complexity $O(N^2)$ (utilizing the $O(N^2)$ PackageTable), a call triggering recursion will still only have worst case time and space complexities of $O(N^2)$. This is because in a worst case in which a package is connected to every other package by either bundle or shared location, the function will end up calling itself for each other package, but not calling itself for the same packages repeatedly, for a total of O(N) checks made by each of O(N) packages for a maximum of $O(N^2)$ time complexity. Meanwhile, while still taking up the $O(N^2)$ space of the PackageTable, adding potentially O(N) Package objects each taking up O(N) space leads to space complexity of $O(2N^2) = O(N^2)$.

Sorting of Packages into Loads ("sort" method of the LoadList class in Load.py):

```
While less than three Loads in LoadList:
	Add new load to LoadList
```

The algorithm I designed requires three loads, so this generates additional loads if less were created initially. It would function with a different set or number of packages though, provided that the assumption that none have more than one "special note" holds true.

```
For each package:
	If package has a deadline before 10:30:
		Add package to first load (index 0 in LoadList)
```

Packages with deadlines before 10:30 are sorted into the first load, to ensure they make the early deadline on time. Dependencies (bundled packages or those going to the same location) are also added automatically by recursive calls in the add method.

```
For each package already in first load:
	For each location:
		If within 3 miles of package's destination and in ring 2 and no delay at location:
		(locations were assigned to rings based on distances from the hub during import)
			For each package at location:
				Add package to first load
```

Locations were assigned to "rings" based on their distances from the hub during the import process (see LocationTable.import_csv() in Location.py for more detailed process). Packages with destinations in ring 2 and within 3 miles of already-added stops (and with no delays at the location) are added to the first load. The first load will have to range far from the hub to deliver early deadlines and dependencies, so this takes care of stops that might be remote for other routes but will add less mileage here. Loops are also constructed in code so as to break if a load is full, for efficiency. (These "if full, break" lines appear repeatedly in multiple blocks of code for the algorithm; they are omitted here for brevity.)

```
Sum region values of destinations currently in first load
(regions, each roughly half of the map, were determined and assigned during data import)
Assign dominant region as average of first load's current stops' region values
For each package:
	If destination in dominant region and ring 2 and no delay at destination:
		Add package to first load
```

Locations were assigned to "regions," each roughly half of the map, based on their distances from two "pole" locations during the import process (see LocationTable.import_csv() in Location.py for more detailed process). After determining which region is "dominant" in the first load, packages at those locations with no delay are added. Adding these stops from ring 2 will make the shape of the route more of a circuit, while keeping them in its dominant region prevents it from traversing the entire perimeter of the map.

```
For each package:
	If delayed:
		Add to second load
```

Packages with a delay are added to the second load. This is essential because they cannot leave with the first load (deadlines before these packages are available), but leaving in the third load may not get them to their destinations in time.

```
Sum region values of destinations currently in second load
Assign dominant region as average of second load's current stops' region values
For each package already in second load:
	For each location:
		If within 2 miles of package's destination and in dominant region
		Or within 1 mile of package's destination and in non-dominant region:
			For each package at location:
				Add package to first load
```

Adds packages to the load from destinations that are either in its dominant region and within 2 miles of existing stops (being in the dominant region means they are likely to be near multiple stops, so a wider range of distances is okay), or in the non-dominant region within 1 mile of existing stops (some regional crossover is okay, and revisiting these areas later would likely be less efficient, but they need to be very close to existing stops to be worth it).

```
While any packages left unsorted:
	If last existing load is full:
		Add new Load to LoadList
	For each package:
		If not sorted:
			Add to last existing load
```

While there are any packages left unsorted, they are placed in the last load in the list. If that load is full, a new Load object is generated and added. Most of the packages remaining with known destinations should be grouped relatively well, or at least in clusters that were not reached by the geographical sorting done on the earlier loads. This section also covers packages with unknown destinations (applies to those with incorrect information provided and new address not yet available as well). These packages are thereby delivered toward the end of the day, when their destinations will hopefully be known/updated, and in a load that is not strictly bound geographically to avoid disturbing more carefully sorted routes.

Routing and delivery of each Load on Trucks ("deliver" method of the TruckList class in Truck.py):

The routing process iterates through each load in LoadList, determining their routes sequentially as later loads' delivery times may be affected by earlier loads' times.

```
For each Load:
	If the load has a truck requirement:
		Assign that truck to the load
	Else:
		Assign the truck with the earliest time available to the load
	Set the load's departure time to the assigned truck's availability time
	For each package in the load:
		If the package's delay time is later than the load's departure time:
			Set the departure time to the package's delay time
```

First, the load is assigned to a truck.(Only trucks with drivers are considered and used.) If the load has a nonzero truck requirement, it is assigned to that truck. Otherwise, it is assigned to whichever truck is available to depart from the hub next. The load's departure time is then set to the time that its assigned truck will be available to pick it up from the hub. If any packages in the load have a delay time (delayed arrival at the hub and thereby availability to load onto a truck) later than that time, the load's departure time is changed to the latest such delay time.

```
	Add the hub as the beginning of the load's route
	Create a set of remaining stops from all of the load's known stops
```

The route must start at the hub, so the initial stop (stop "0") added to the load's route data member is there (location 0). The stop's location key is added as a list because the route data will later become a matrix of information, so this will allow other "columns" to be appended later. A new set is created of remaining stops that can be modified, leaving the load's original set of stops intact.

```
	Set the first deadline as the earliest deadline of any package in the load
	and the first stop (tentatively) as the location closest to the hub with that deadline
```

The earliest deadline on the route and the location of the package with that deadline are determined. If packages are tied with the same earliest deadline, the location closest to the hub is chosen. If that deadline is within an hour of the load's departure time, the following steps are used to determine a route.

```
	If the first deadline is within 1 hour of the load's departure time:
		Add that location as the first stop (after the hub) in the route
		Remove that location from the set of remaining stops
		While other packages in the load have the same deadline:
			Add the nearest stop with that deadline to the route next
			(break ties by choosing greater average distance)
			Remove from remaining stops
```

The first stop added to the route is the first deadline location determined previously. While any remaining stops have that same deadline, the nearest of those locations from the preceding location is added to the route next. This is a greedy selection, but not using the method above because it must consider only locations with the same deadline, and this variation of the greedy selection is employed only here. It breaks ties based on average distances, choosing the location with greater average distance first, on the basis that it will be more advantageous to leave shorter distances available for later at no cost to distance here.

```
		While any (known) remaining stops:
```

```
                Choose next stop with greedy selection algorithm (call method above)
                Add location returned by greedy selection to route next
                Remove location from remaining stops
```

After those stops have been added to the route, the greedy selection method above selects next stops to add to the route until the remaining stops set is empty. This means that all known destinations have been visited, but does not necessarily complete the route; destination keys of packages with unknown/incorrect addresses were not added to the load's set of stops, so they may not have been visited yet. This will be accounted for later in the portion of the algorithm determining times.

```
Else:
        Determine a “pole” among the load’s stops with the greatest average distance
        Determine a second “pole” that is the furthest stop from the first
        Set the first stop as the stop with the greatest sum of distances to the two
        route poles, minus distance to the hub
        Add that location to the load’s route as the first stop (after the hub)
        Remove that location from remaining stops

        While any (known) remaining stops:
                Choose next stop with greedy selection algorithm (call method above)
                Add location returned by greedy selection to route next
                Remove location from remaining stops
```

If the load's first deadline is more than an hour after departure, these steps are taken instead. The route's first stop is determined based on two poles within the load's required stops are determined that are far from each other and on average from other locations. (This is similar to the process used to assign locations to regions during import but with some differences, most importantly that it is restricted to stops in the load being considered.) Pole 1 of the route is set as the stop with the highest average distance, and Pole 2 is set as the stop furthest from Pole 1.

The first stop is selected as the location in the load's set of stops that has the highest sum of distances to the two route poles, minus its distance from the hub. (The route poles themselves are excluded from consideration as a start point.) Considering the sum of pole distances results in a start point that is more or less midway between the poles, and somewhat out from a direct line between them (based on the idea of the Pythagorean theorem, it will be further from the poles than a point along that direct line). This helps make the route more of a circuit in shape, while also factoring in the distance to the hub helps avoid a long initial distance to get to the start point.

After that first stop is selected and added first, the above greedy selection method is used to choose the order of the known remaining stops. As with the previous case, packages without known destinations are not considered for now.

```
Expand the first row of the route (hub at index 0) to include 0 (miles from last stop)
Expand the first row of the route to include 0 (miles on the route so far)
Expand the first row of the route to include 0:00 (time of the route so far)
For each stop on the route (row of the matrix):
        Calculate and store distance from previous stop
        Calculate and store distance of the route so far
        Calculate and store time of the route so far
```

Next, the route data member is expanded into a matrix including the location key, distance from previous stop, total distance of the route so far, and elapsed time from the start of the route for each stop. Actual delivery times are still excluded for now, as these pieces of information will be used to adjust the route's start time. The initial information for starting at the hub is added first (0 miles from itself, 0 miles travelled so far, and 0:00 elapsed on the route). Then a loop calculates those values based on each other sequentially combined with the previous stop's values, for each stop currently on the route.

```
Determine whether any packages in the load have updates expected
If an update is expected:
        Determine the last stop along the route with a deadline
        Calculate the latest time that the route can begin while still reaching that
        deadline with 1 minute to spare
        If that time is later than departure time:
                Set departure time to that time
```

This section of the algorithm begins to consider packages with unknown/incorrect addresses. If such a package is included in the load, a marker variable is set reflecting that an update is expected. In that case, the load's departure time is updated to be as late as possible while still meeting the last deadline along the route. This maximizes the chance that any package loaded onto a truck without a known destination at that time will be updated with the correct destination before reaching that stop, so that it can be delivered with any others that may be at the same location and avoid returning and making redundant stops later.

```
Create a set of undelivered packages from the load’s package list
Create an empty set of delivered packages
Expand the first row of the route to include departure time (current time along route)
For each stop (row) in the route:
	Calculate and store the current time at the stop
	For each package in the undelivered set:
```

This begins the handling of the actual "delivery” of packages to the stops on the load's route. All of the load's packages are initially added to the undelivered set, while the delivered set is initially empty. For each stop along the route, the time of the stop (departure time plus time on the route so far) is added to the next column of the route matrix. Then, each of the packages in the undelivered set are checked.

```
		If an update is expected
		and the package’s expected update time is before the current stop’s time:
			Update the package’s destination to the correct location key
			Reset the package’s update time to reflect none
			Add the package to the correct location’s package list
```

If an update is expected (meaning to a package's destination) and the update time is at or before the current stop's time, the package's destination data is changed to the location key matching the correct address. This reflects the system relaying updated information to the truck/driver as it comes in, so that even though a package might have been loaded without knowing its destination, and the route was determined without that knowledge but included the same destination for other packages, if the truck goes to that stop after the package's information is updated, it can be delivered along with the others. Its expected update time is then reset so that it isn't re-updated at each stop, and it is added to the correct location’s package list as well.

```
		If the package’s destination is the current stop:
			Mark the package delivered at the current time
			Add the package to the delivered set
	Remove packages in the delivered set from the undelivered set
```

Then, if a package's destination matches the current stop, it is delivered (added to the delivered set and marked with the stop's time). After delivering packages to a stop, the set of undelivered is updated to remove any that are now in the delivered set. This improves efficiency by reducing the number of packages that must be checked at each subsequent stop, and keeps track of any leftover packages after the route’s initially known locations have all been visited.

```
While packages remain in the undelivered set:
	Set next stop to default to hub
	Create known destinations set (package destinations in undelivered>0)
	For each package in undelivered:
		If destination is known:
			Add destination to known destinations
```

This section handles what happens if a package is left undelivered after the previously calculated route. (This would occur if an update was expected, and the correct stop either was not already on the route or was reached before the update was made.) First, a set of known destinations is made (undelivered packages' positive location keys). (There would be known destinations at this point if updates had already been made but were not for stops along the portion of the route following the update time.) As long as undelivered packages remain in the load, the truck will continue extending the route.

```
	If known destinations is not an empty set:
		Choose next stop from known destinations with greedy selection algorithm
		Add location returned by greedy selection to route next
		Calculate and store distance from previous stop
		Calculate and store distance of the route so far
		Calculate and store time of the route so far
		Calculate and store current time
		For each package in the undelivered set:
			If an update is expected
			and the package’s expected update time is before the current time:
				Update the package’s destination to the correct location key
				Reset the package’s update time to reflect none
				Add the package to the correct location’s package list
			If the package’s destination is the current stop:
				Mark the package delivered at the current time
				Add the package to the delivered set
		Remove packages in the delivered set from the undelivered set
```

If there are known destinations, the greedy selection method is used to choose the next stop from among them, which is then added to the route and the package is delivered as above.

```
Else:
        Add a stop at the hub to the route
        Calculate and store distance from previous stop
        Calculate and store distance of the route so far
        Calculate and store time of the route so far
        Calculate and store current time
        Determine the earliest update time of any package in undelivered
        If current time is before the earliest update time:
                Add another stop at the hub to the route
                Store 0 as distance from previous stop
                Store same distance as before as distance of the route so far
                Calculate the route time so far as the difference between the
                departure time and the earliest update time
                Store the earliest update time as current time
```

Otherwise, if there are no known destinations at this point, meaning that packages are still awaiting updated information, the truck returns to the hub to wait. The next expected update time is determined, and if it is later than the time the truck returns to the hub, the truck continues to wait until that time. This case adds a second consecutive "stop" at the hub to the route if needed, the first showing the truck's arrival time and the second showing its departure. After that, the loop will repeat, and at least that package's location will be added to known destinations, triggering the previous case.

```
Add a stop at the hub to end the route
Calculate and store the distance from the previous stop
Calculate and store the route's total distance
Calculate and store the total time of the route
Calculate and store the end time of the route
```

Lastly, a final stop is added to the route to mark the truck's return to the hub and display the route's final length and end time. The assigned truck's availability time is also updated to the route's end time, signifying that it will then be available to deliver another load as this method repeats for any following. When all loads have been routed and "delivered," the method ends.

B2: Development Environment

This program was developed using IDLE for Python 3.7.8 64-bit, on an HP laptop with an Intel Core i5 CPU @1.00GHz and 12 GB RAM running 64-bit Windows 10 Home.

B3: Space-Time and Big-O

The following assumptions are made in my analysis of the program's complexity:

- No extraneous locations (without packages to deliver) are included in the location data imported
- No extraneous stops (at which no package is delivered, other than the hub) will be made.
- Number of packages>=Total stops across all routes>=Number of locations>=Stops on any particular route
- Number of packages>=Highest number of packages in any one load>=Stops on any particular route
- Number of packages>=Number of packages delivered to any single location
- Number of loads and packages per load have an inverse relationship (as more packages are allowed per load, less loads will be required, and vice versa)

Because the total number of packages is the greatest meaningful variable, and other important values are dependent on it (significantly more packages would require more loads and would likely introduce more locations), complexity will be assessed in terms of this value (N=total number of packages) and will be substituted for other variables as appropriate. (For example, where L=number of locations, O(L) => O(N), O(L) + O(N) = O(2N) = O(N), etc.) Analysis of each major segment of the program follows here, with specific assessments of each individual class and method found in the code comments.

<u>Creation and population of the LocationTable object from import of a csv file</u>

Space complexity: $O(N^2)$

The class is composed of two data structures. The matrix of distances will be $O(N^2)$ as each location's row contains an entry for each location. The table holding the location objects themselves will be O(N). Although one individual location may also be O(N) in the case that its package list contains all packages, in that case all other location objects would have empty package lists and be O(1), so O(N)+N*O(1) gives O(N). Another way to arrive at this conclusion is to reason that altogether, the table will contain a set amount of data for each location, O(N), plus one entry for each package in their package lists collectively, O(N), giving O(N)+O(N)=O(N). Nevertheless, the distance matrix makes the LocationTable $O(N^2)$ overall.

Time complexity: $O(N^2)$

The most complex portions of the locations import process involve taking actions in nested loops through locations in the form:

```
For each location:
      For each location:
            Execute O(1) task
```

to populate and fill in the matrix of distances. The complexities of the loops (each O(N)) multiply to give $O(N^2)$.

Creation and population of the PackageTable object from import of a csv file

Space complexity: $O(N^2)$

It is possible for each Package object to have N items in its bundled packages set, in the case that it is bundled with every other package. These can (and often will) be repeated in the corresponding packages' sets, so the table's size will be N packages times O(N) for each package, giving $O(N^2)$.

Time complexity: $O(N^2)$

The most complex portions of the packages import process involve taking actions in nested loops for each line in the import file (each package) and locations in the locations table in the form:

```
For each line in csv file (corresponding to a Package object being created):
      For each location:
            Execute O(1) task
```

This process is used to match packages' addresses to the correct locations. The complexities of the loops (each O(N)) multiply to give $O(N^2)$. There is also a loop for each package through its bundle list, which could reach $O(N^2)$ in a worst case that all packages are bundled together.

Use of the LoadList object to sort packages into loads

Space complexity: $O(N^2)$

The sorting process accesses the PackageTable and LocationTable objects, which use $O(N^2)$ space.

Time complexity: $O(N^2)$

The most complex portions of the sorting process are of the form:

```
For each package in the load:
If load is full, break
      For each location:
            If load is full, break
            If near package's destination:
            For each package with that destination:
                  Add that package to the load using the Load.add(Package) method
                  If load is full, break
```

This is done to add packages to a load that are near stops already on its route. Although multiple loops are used, some of these simplify when combined. The "for each location… for each package with that destination" portion is easily reducible to "for each package" because each package has one and only one destination, so this will check each package exactly once. (It is actually somewhat more efficient split as it is written, because it allows skipping over packages at irrelevant locations.) This leaves:

```
For each package in the load:
If load is full, break
	For (up to) each package:
		Add that package to the load using the Load.add(Package) method
		If load is full, break
```

The number of times the add method will be called in total is limited by the "if load is full, break" lines, to be no more than the capacity of the load. Further, while that method's worst-case runtime is $O(N^2)$ as described above, after this its runtime would be reduced to O(1) for all future calls. This is because the case in which its runtime is $O(N^2)$ loads all packages, so all future calls fail its initial check whether package is loaded and exit. It is also possible that it could be called N times, each of which may result in O(N) operations (if all other packages are checked but not called recursively).

With all of this, the code can be simplified to:

```
For no more than the remaining capacity of the load:
	Execute either one O(N²) and N*O(1) tasks or N*O(N) tasks
```

This results in the following potential worst cases for this block of the sorting algorithm:

- The load's capacity is equal to the total number of packages, only one package has been loaded, and it is bundled with all other packages: The add method executes once at $O(N^2)$, then the loop exits, making its runtime complexity $O(N^2)$.
- The load's capacity is equal to the total number of packages, only one package has been loaded, and no packages are bundled but all fit the criteria to be added: The add method is called O(N) times and executes at O(N) each time, resulting in $O(N^2)$ runtime overall.

In either case, this shows the block's time complexity to be $O(N^2)$. This is the most complex portion of the sorting algorithm, so its time complexity is $O(N^2)$ as well.

Use of the TruckList object to route and deliver each load of packages

Space complexity: $O(N^2)$

The routing and delivery process accesses the PackageTable and LocationTable objects, which use $O(N^2)$ space.

Time complexity: $O(N^2)$

Two sections of the routing and delivery algorithm stand out as having the most potential for runtime complexity. First, slight variants of this basic block are used in multiple places to generate routes:

```
For each load:
	.
	.
	.

	While stops remaining unvisited:
		Use greedy selection algorithm to choose next stop
```

While the number of loads and the stops remaining at any given time may not be known up front, this block can be simplified logically. The total number of stops visited, in all loads combined, cannot exceed the number of packages to be delivered. However the stops are distributed among loads, and no matter how many loads there are, this will therefore not cause the greedy selection algorithm to execute more than N times across all loops. So it becomes:

```
For (up to) each stop:
	Use greedy selection algorithm to choose next stop
```

The core greedy algorithm method itself has O(N) time complexity, as described above, looping through the set of stops provided (not exceeding N in length), so this makes this section's time complexity O(N) * O(N) = $O(N^2)$.

The next potentially complex block is the process to dynamically check for and apply updated package destination information while it has been loaded and is en route:

```
For each load:
        .
        .
        .
        For each stop on the route:
                For undelivered packages in the load:
                        If an update is expected and available:
                        For each location:
                                If matching the corrected address:
                                        Update package's destination
```

The outer two loops can be simplified as with the previous block discussed. Regardless of the distribution of stops among loads and number of loads, the rest of the loop will execute once for each stop, for no more stops than the total number of packages. Then, the inner loop is only executed when a new update is available, and it is primarily checking each location with just assignment operations when found. So this block can be simplified to:

```
        For each stop (total):
                For each undelivered package with a new update:
                        Check each location for match
```

In a worst-case scenario for this there would be as many stops as packages, all packages would be in need of updates, and each package would be going to a different location. Even then, because locations will only be checked for each package once, across all stops and loads (after updating their update time is reset so they will not be checked again), the innermost loop will only execute once for each package across the algorithm's entire run, simplifying finally to:

```
        For each package:
                Check each location for match
```

This gives a time complexity, ultimately, of O(N) packages * O(N) locations = $O(N^2)$.

Another way of reaching this conclusion is to abstract that the location check loop will execute only once for each package that needs to be updated, therefore, even though this task is nested within other loops handling other functions that will check $O(N^2)$ times, it will still only *add* (number of updates*number of locations) = O(N) * O(N) = $O(N^2)$ operations to the program, not multiply that number by the number of loops those operations are split between, and so the overall time complexity of the section it is contained within is $O(N^2)$ checks + $O(N^2)$ inner loop iterations = $O(N^2)$.

Use of the Schedule object to execute menu commands

Space complexity: $O(N^2)$

The Schedule object itself holds only three reference variables, making it O(1) space, but its methods use those references to access the PackageTable and LocationTable objects, which use $O(N^2)$ space, during this part of the program.

Time complexity: $O(N^2)$

At this point in the program, most of the more complex calculations have been completed and results stored. The most complex part of this section of the program is the method to print the entire schedule, which must print each stop along all routes, and for each of those check each package in the load, nested loops that produce $O(N^2)$ time complexity. Other methods called here to look up and display packages' information and status are O(N), looping through each package to perform checks and/or print operations. They execute one at a time, not looped with each other, so their overall complexity remains O(N).

Overall Program Complexity

Space complexity: $O(N^2)$

Throughout much of the program, the PackageTable and LocationTable objects are accessed and manipulated, each of which is a data structure with $O(N^2)$ space complexity. This is essentially unavoidable with the type of adjacency matrix used when each location in the graph is connected to every other location.

Time complexity: $O(N^2)$

Many operations throughout the program employ $O(N^2)$ time complexity, commonly checking a set of packages against a set of packages, a set of packages against a set of locations, a set of locations against a set of locations, etc. Although many instances of more deeply nested loops exist, their complexity is offset and reduced by the fact that although the size of any one of the loops may be O(N), the other related loops' sizes will have an inverse relationship. For instance, "for each load, for each stop along the route" can become at most at most N stops on 1 route, or 1 stop on at most N routes, giving O(N) rather than $O(N^2)$ time complexity in the end. This enabled relatively efficient calculation and checks of only O(N) objects to initiate operations with O(N) complexity (like applying greedy selection to determine a series of stops) while keeping complexity within $O(N^2)$ throughout the program.

B4: Scalability and Adaptability

The program is easily scalable to handle more packages in most ways. Computationally, it is of polynomial time and space complexity, specifically $O(N^2)$, so its usage of resources will increase by a factor on the order of x^2 for each increase by a factor of x in the number of packages it handles. This scalability is achieved by checks before executing the same loops or functions for the same packages over and over, to preserve a low polynomial complexity and relatively high efficiency.

The algorithms employed are set up so that they could continue sorting many more packages than the list provided, and continue taking into account the "special notes" required for each package (provided it remains true that each package has no more than one such restriction). If too many such requirements are included however, they may begin to overlap too much and some may end up failing to be met (for example, if numerous packages have a very early deadline, the express delivery route may not handle all of them in time).

This is in large part due to the sorting algorithm, which is designed to maximize efficiency for the provided set of packages on the provided map and specifies at least three loads to do so. With more packages, more loads would be created, but would not be as well-sorted into geographically. I experimented with several other sorting algorithms based on similar geometric and geographic principles (use of rings, regions, poles, etc.), some of which were more general and may perform better with larger sets of packages but were somewhat less efficient for the small set provided (mileage in the high seventies or low eighties, rather than 72.8). To significantly scale up the program, it would be better to update the sorting algorithm to such a similar but more generic method.

The routing and delivery algorithm is more readily applied to larger sets of packages without much loss of efficiency. Even if provided with loads that are not sorted as well, it will continue to prioritize the most imminent of deadlines and otherwise develop routes that are well-designed geometrically to maximize efficiency in mileage, though each may cover a larger area if the effectiveness of sorting drops.

Potentially the most time-complex portion of the program is the dynamic updating of package information as available en route. If large numbers of updates are expected, these could simply be delayed instead, reducing the efficiency of total mileage but increasing the program's runtime efficiency, if necessary to avoid bogging down when scaled up.

Overall, the capacity of the program to simply handle larger volumes of packages scales well, and its ability to do so most effectively would require only some tweaks to the sorting algorithm. Most of the important processes will continue to account for things like package dependencies, delays, deadlines, truck requirements, and updates no matter how many packages are handled, devising a near-optimal route for each load, and continue routing until all packages are delivered, all in polynomial time and using $O(N^2)$ space for the required tables.

B5: Software Efficiency and Maintainability

The program is fairly efficient, maintaining polynomial complexity in time and space as described above. This $O(N^2)$ complexity is achieved by using checks to prevent processes that are O(N) from executing more than O(N) times (for example, the greedy selection method being used at most once per stop).

The program is made easy to maintain in part by automating the application of each package's "special notes" requirements, rather than needing to input these manually for each package. (This requires only that the initial Excel spreadsheet/csv file from which packages are imported by formatted properly before import, with different types of requirements recorded in different columns.)

It also uses constants defined once in main.py to hold situational variables that remain constant throughout a given application of the process (for instance truck speed, truck capacity, start of day time, and number of trucks). This makes the program easy to maintain because instead of having these values hard-coded into each class and method where they are used, they can be changed once if needed and changes to the rest of the program's execution will take place automatically as a result.

The sorting and routing algorithms also make the program easy to maintain by being automatic, rather than requiring, for instance, the manual sorting of packages before routing can take place. Only one update of the sorting algorithm to make it more general rather than tailored to this situation, as described above, may be needed to ensure the process functions well for larger numbers of packages and using varied maps. (This would involve using the same principles to sort, but specifying a repeating pattern of loads rather than just three initially; it could easily be set up as repeating approximately the same pattern of steps for every three loads in a longer sequence). After that update, it would function in different situations with no need to make any modifications beyond changing the constants in main.py to fit if necessary.

(B6: Self-Adjusting Data Structures on next page following part D)

D: Data Structure

The PackageTable object functions as a direct access hash table to store each package's information in an entry based on its ID number.

D1: Explanation of Data Structure

In the PackageTable direct access hash table, each list item stores a reference to the Package object with the ID number corresponding to its index. Each Package object stores the data of an individual Package, most of it directly while the destination information (street address, city, state, and zip) is abstracted to a location key that corresponds to an entry in the LocationTable object's list, a direct access hash table containing references to Location objects with keys corresponding to their index in the list. This structure was logical to store the data provided because each package had a unique, sequential ID number starting with 1, and the locations are used to construct a matrix of distance information, so it is helpful to assign each a key corresponding to their placement in it.

Given the sequential package IDs starting at one, it is easy to establish a direct correspondence with indices and practical because every entry in the table is used (other than index 0), the length of the table does not exceed N+1, and it is easy to adjust the size of the table, simply increasing by 1 for each package added. This allows direct access to easily create a perfect hash of the packages with no collisions in insertion.

Storing location-specific information in Location objects rather than directly in Package objects makes sense because multiple packages share destinations, so this information would otherwise be repeated in packages with the same destination, storing unnecessarily redundant data. Use of Location objects, with simple integer location keys stored in each Package object, thereby reduces the overall usage of space by the program and also protects the data integrity of locations (avoiding slight variations of addresses or typos in different Package objects' fields). These location keys will correspond to the location's row and column indices in the matrix of distances, making it easy to access distance information based on a package's location key as well.

B6: Self-Adjusting Data Structures

The self-adjusting data structures of the PackageTable and LocationTable objects each utilize a direct access hash table as described above. A major strength of the direct access table structure, combined with the links between them (via each Package object's destination location key, and each Location object's package list), allows for fast and easy access to many pieces of information from either a given package or a given location, without having to run through any loops to find the matching item from the other set. These pieces of information include data about a given package's destination (access package directly via ID, then destination directly via key), the distance between one package's destination and another (access package via ID, then use each location key as row and column indices in the distance matrix), all packages at the same destination (access one package via ID, then via location key get the list of packages at that location, without needing to access and check any other packages), etc.

Another strength of these direct access table structures is that each will use space efficiently for the provided scenario, matching the number of entries in the PackageTable list to the number of packages (+1 for index 0) and the number of entries in the LocationTable list to the number of locations, and utilizing every entry in the distance matrix to hold a required piece of distance data.

The direct access structure does have weaknesses that would be problematic if the package IDs were assigned differently:

- If the packages' IDs were not sequential, provided out of order, the method of adjusting size by appending a new entry for each insertion would be impractical (packages with high ID numbers may be inserted before the correct "bucket" is appended), but a table of the required length could be generated based on the length of the input up-front instead as long as the IDs still composed a continuous range of integer values.
- If the IDs were sequential and continuous, but did not start at 1 (for instance continuing where the previous day's numbering left off), a direct access hash table would not be practical (all entries between index 0 and the first package ID would be extraneous), a simple mapping function, for instance by subtracting the first ID from each and assigning to the resulting entry in the table, could still easily create a perfect hash function into a table of length N without making access operations significantly more complex either.
- If the IDs did not compose a continuous range, a direct access table would be impractical and the structure could not be easily adapted; an alternative structure for the hash table would have to be used.

The PackageTable object has a potential weakness that can give it $O(N^2)$ space usage in a worst-case scenario. If each package is bundled with every other package, each package's bundle list would use O(N) space, which for O(N) packages results in $O(N^2)$ space overall. This is necessary to ensure that bundled packages are sorted together though, and this worst-case complexity will almost never be the case as it would be impractical in real situations for all packages to be bundled together. The bundle lists will generally tend to be short in general producing typical runtime closer to O(N).

The LocationTable, though, will always be $O(N^2)$ given the structure used, storing distances as a matrix with dimensions equal to the number of locations, which is O(N), squared. Storage of this $O(N^2)$ amount of distance data is necessary as long as each location is connected to every other location, and the matrix is an efficient way to represent this situation. However, if this was not the case, and locations connected to only a limited number of others, this structure may become a weakness, taking up potentially large amounts of space unnecessarily. Instead, it might become more efficient to store distances from a given location to each connected point as a dictionary in the Location object.

E: Hash Table

The insertion method for the PackageTable object takes a Package object, containing the required package data items, as input. (Each Package object is created during the import process from the csv file and stores the data of a given package either directly or, in the case of destination information (street address, city, state, and zip) indirectly by abstracting it as a location key used to access a Location object storing that information directly.)

The insertion method increases the hash table's size by 1 to accommodate the new entry, then stores the Package object's reference. The table is initialized with None at index 0, so package #1 is placed at index 1, and because ID numbers are sequential and continuous in the imported data, each subsequent package that is added is placed at the correct index that is generated in the same call to insertion.

F: Look-Up Function

The look-up function takes as input a user-entered search term, and a user-entered time at which to check packages' statuses. It returns a list of Package object references using one of three cases to search:

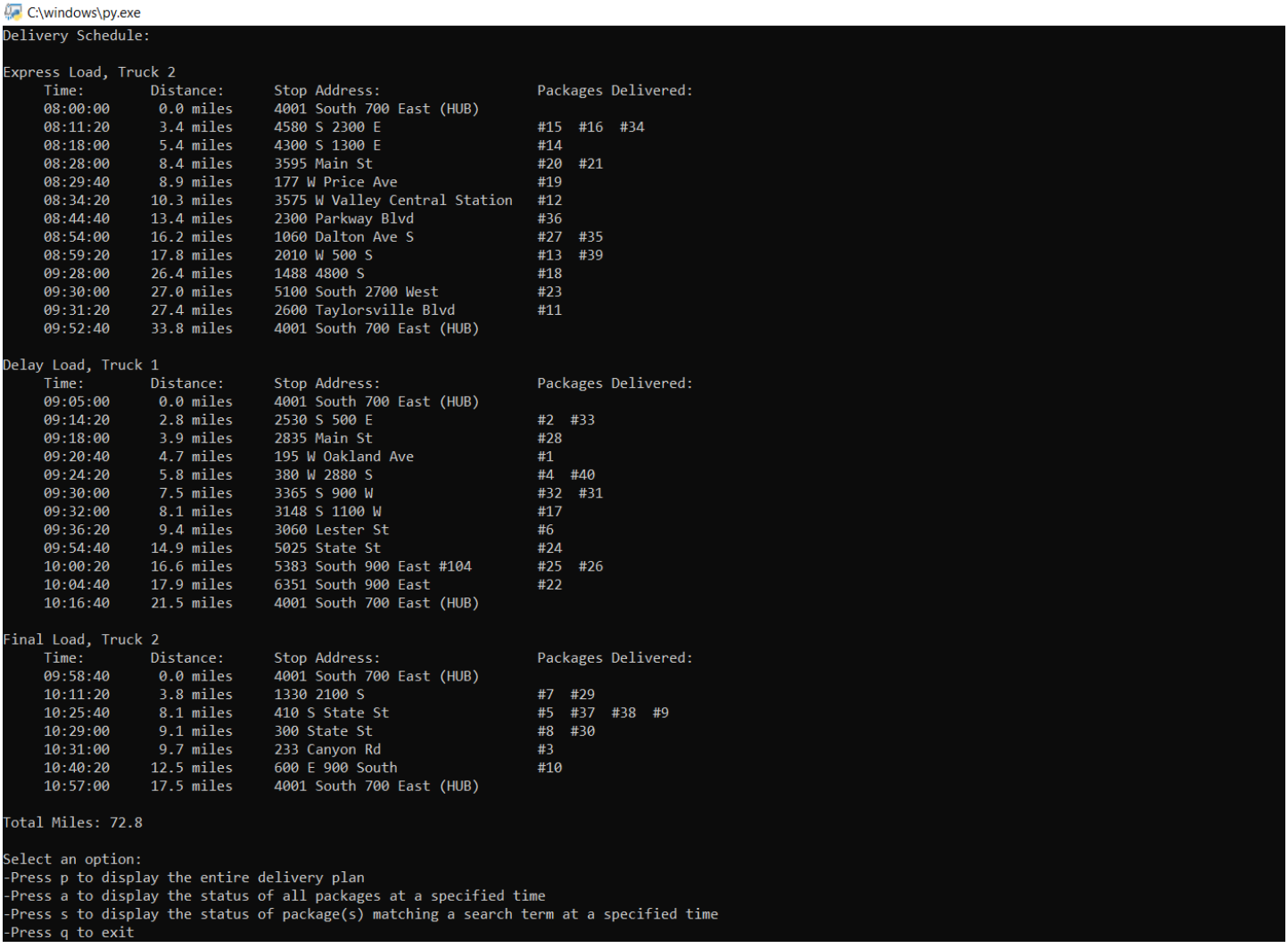
- If the search term is recognized as a valid package ID number, it adds only that package to the list of matches which is then returned.
- If the search term is recognized as one of the designated statuses, the method loops through each package, comparing its delivery time and the departure time of its load to the chosen time, and adds packages to the list of matches that have the correct status at that time.
- Otherwise, it loops through each package, comparing the search term to other data and adding any package for which the search term is a partial match of street address or city, or an exact match of state, zip code, or weight, or deadline.

The list of all matches produced by the appropriate checks is returned by the lookup function, which will then be used by the calling function to display the information and status at the provided time for each package in the list.

G: Interface

The program includes a console menu interface to execute three tasks: display the entire schedule (each package delivered at each stop along each route travelled by all trucks, concluding with the schedule's total mileage), display information and status for all packages at a specified time, and display information and status for packages matching a search term at a specified time.

This screenshot shows the print schedule function, including total mileage travelled:



G1: First Status Check (9:00 am)

C:\windows\py.exe

```
Enter command:a

Enter a time for which to view status (enter as "X:XX am" or "X:XX pm"):9:00 am

Status of all packages at 09:00 AM:
Package ID:   Weight:     Destination:                                                Deadline:     Status:
         1     21 kg                195 W Oakland Ave    Salt Lake City, UT 84115     10:30 AM     At the Hub
         2     44 kg                  2530 S 500 E       Salt Lake City, UT 84106          EOD     At the Hub
         3      2 kg                233 Canyon Rd        Salt Lake City, UT 84103          EOD     At the Hub
         4      4 kg                 380 W 2880 S       Salt Lake City, UT 84115          EOD     At the Hub
         5      5 kg               410 S State St       Salt Lake City, UT 84111          EOD     At the Hub
         6     88 kg               3060 Lester St  West Valley City, UT 84119     10:30 AM     Delayed, arriving at the Hub at 09:05 AM
         7      8 kg                  1330 2100 S       Salt Lake City, UT 84106          EOD     At the Hub
         8      9 kg                 300 State St       Salt Lake City, UT 84103          EOD     At the Hub
         9      2 kg               410 S State St       Salt Lake City, UT 84111          EOD     At the Hub
        10      1 kg              600 E 900 South       Salt Lake City, UT 84105          EOD     At the Hub
        11      1 kg          2600 Taylorsville Blvd    Salt Lake City, UT 84118          EOD     En route, loaded onto Truck 2 at 08:00 AM
        12      1 kg  3575 W Valley Central Station  West Valley City, UT 84119          EOD     Delivered by Truck 2 at 08:34 AM
        13      2 kg                  2010 W 500 S       Salt Lake City, UT 84104     10:30 AM     Delivered by Truck 2 at 08:59 AM
        14     88 kg                4300 S 1300 E          Millcreek, UT 84117     10:30 AM     Delivered by Truck 2 at 08:18 AM
        15      4 kg                4580 S 2300 E          Holladay, UT 84117     09:00 AM     Delivered by Truck 2 at 08:11 AM
        16     88 kg                4580 S 2300 E          Holladay, UT 84117     10:30 AM     Delivered by Truck 2 at 08:11 AM
        17      2 kg                3148 S 1100 W       Salt Lake City, UT 84119          EOD     At the Hub
        18      6 kg                  1488 4800 S       Salt Lake City, UT 84123          EOD     En route, loaded onto Truck 2 at 08:00 AM
        19     37 kg              177 W Price Ave       Salt Lake City, UT 84115          EOD     Delivered by Truck 2 at 08:29 AM
        20     37 kg                3595 Main St       Salt Lake City, UT 84115     10:30 AM     Delivered by Truck 2 at 08:28 AM
        21      3 kg                3595 Main St       Salt Lake City, UT 84115          EOD     Delivered by Truck 2 at 08:28 AM
        22      2 kg            6351 South 900 East           Murray, UT 84121          EOD     At the Hub
        23      5 kg           5100 South 2700 West       Salt Lake City, UT 84118          EOD     En route, loaded onto Truck 2 at 08:00 AM
        24      7 kg                5025 State St           Murray, UT 84107          EOD     At the Hub
        25      7 kg        5383 South 900 East #104       Salt Lake City, UT 84117     10:30 AM     Delayed, arriving at the Hub at 09:05 AM
        26     25 kg        5383 South 900 East #104       Salt Lake City, UT 84117          EOD     At the Hub
        27      5 kg           1060 Dalton Ave S       Salt Lake City, UT 84104          EOD     Delivered by Truck 2 at 08:54 AM
        28      7 kg                2835 Main St       Salt Lake City, UT 84115          EOD     Delayed, arriving at the Hub at 09:05 AM
        29      2 kg                  1330 2100 S       Salt Lake City, UT 84106     10:30 AM     At the Hub
        30      1 kg                 300 State St       Salt Lake City, UT 84103     10:30 AM     At the Hub
        31      1 kg                3365 S 900 W       Salt Lake City, UT 84119     10:30 AM     At the Hub
        32      1 kg                3365 S 900 W       Salt Lake City, UT 84119          EOD     Delayed, arriving at the Hub at 09:05 AM
        33      1 kg                2530 S 500 E       Salt Lake City, UT 84106          EOD     At the Hub
        34      2 kg                4580 S 2300 E          Holladay, UT 84117     10:30 AM     Delivered by Truck 2 at 08:11 AM
        35     88 kg           1060 Dalton Ave S       Salt Lake City, UT 84104          EOD     Delivered by Truck 2 at 08:54 AM
        36     88 kg           2300 Parkway Blvd  West Valley City, UT 84119          EOD     Delivered by Truck 2 at 08:44 AM
        37      2 kg              410 S State St       Salt Lake City, UT 84111     10:30 AM     At the Hub
        38      9 kg              410 S State St       Salt Lake City, UT 84111          EOD     At the Hub
        39      9 kg                2010 W 500 S       Salt Lake City, UT 84104          EOD     Delivered by Truck 2 at 08:59 AM
        40     45 kg                 380 W 2880 S       Salt Lake City, UT 84115     10:30 AM     At the Hub

Select an option:
-Press p to display the entire delivery plan
-Press a to display the status of all packages at a specified time
```

G2: Second Status Check (10:00 am)

C:\windows\py.exe

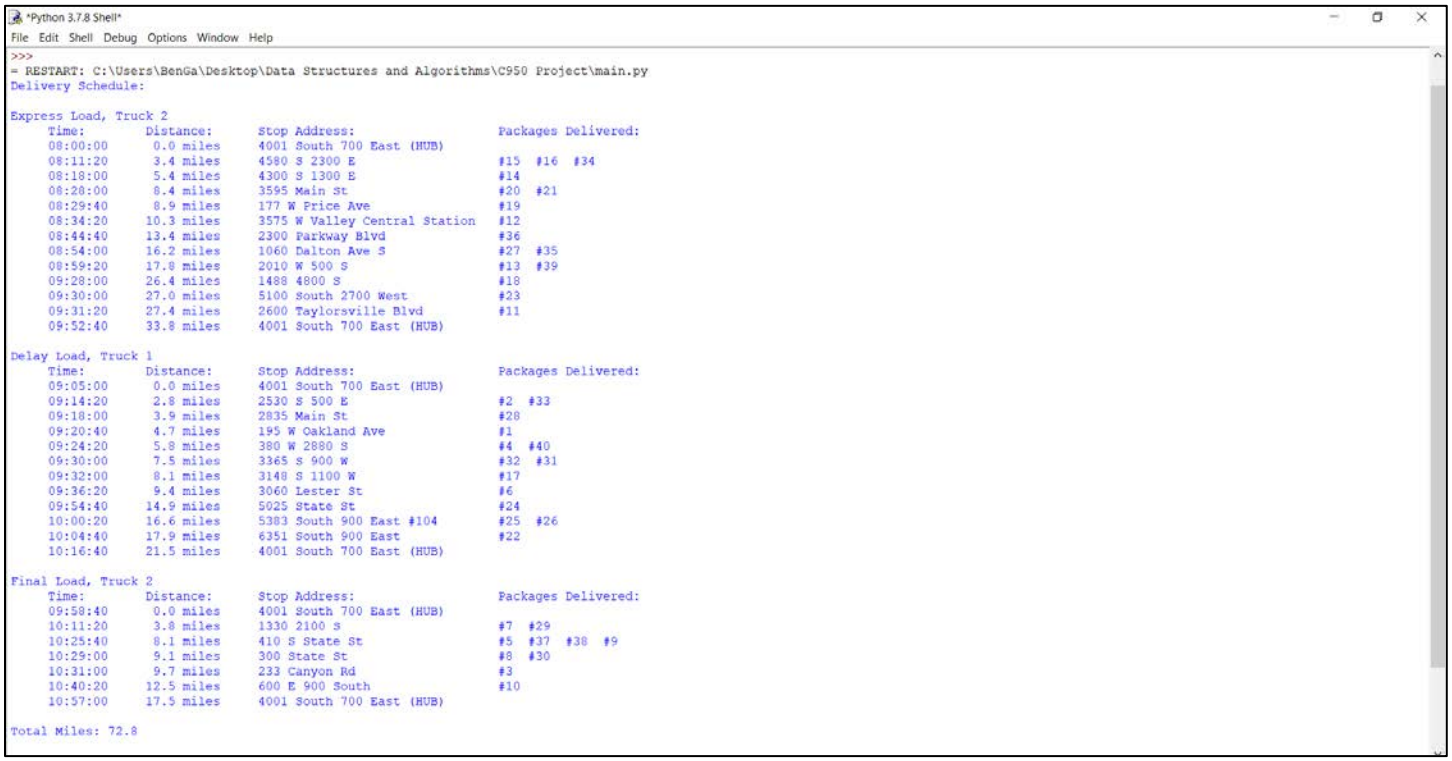
```
Enter command:a

Enter a time for which to view status (enter as "X:XX am" or "X:XX pm"):10:00 am

Status of all packages at 10:00 AM:
Package ID:   Weight:     Destination:                                                Deadline:     Status:
         1     21 kg                195 W Oakland Ave    Salt Lake City, UT 84115     10:30 AM     Delivered by Truck 1 at 09:20 AM
         2     44 kg                  2530 S 500 E       Salt Lake City, UT 84106          EOD     Delivered by Truck 1 at 09:14 AM
         3      2 kg                233 Canyon Rd        Salt Lake City, UT 84103          EOD     En route, loaded onto Truck 2 at 09:58 AM
         4      4 kg                 380 W 2880 S       Salt Lake City, UT 84115          EOD     Delivered by Truck 1 at 09:24 AM
         5      5 kg               410 S State St       Salt Lake City, UT 84111          EOD     En route, loaded onto Truck 2 at 09:58 AM
         6     88 kg               3060 Lester St  West Valley City, UT 84119     10:30 AM     Delivered by Truck 1 at 09:36 AM
         7      8 kg                  1330 2100 S       Salt Lake City, UT 84106          EOD     En route, loaded onto Truck 2 at 09:58 AM
         8      9 kg                 300 State St       Salt Lake City, UT 84103          EOD     En route, loaded onto Truck 2 at 09:58 AM
         9      2 kg               410 S State St       Salt Lake City, UT 84111          EOD     En route, loaded onto Truck 2 at 09:58 AM
        10      1 kg              600 E 900 South       Salt Lake City, UT 84105          EOD     En route, loaded onto Truck 2 at 09:58 AM
        11      1 kg          2600 Taylorsville Blvd    Salt Lake City, UT 84118          EOD     Delivered by Truck 2 at 09:31 AM
        12      1 kg  3575 W Valley Central Station  West Valley City, UT 84119          EOD     Delivered by Truck 2 at 08:34 AM
        13      2 kg                  2010 W 500 S       Salt Lake City, UT 84104     10:30 AM     Delivered by Truck 2 at 08:59 AM
        14     88 kg                4300 S 1300 E          Millcreek, UT 84117     10:30 AM     Delivered by Truck 2 at 08:18 AM
        15      4 kg                4580 S 2300 E          Holladay, UT 84117     09:00 AM     Delivered by Truck 2 at 08:11 AM
        16     88 kg                4580 S 2300 E          Holladay, UT 84117     10:30 AM     Delivered by Truck 2 at 08:11 AM
        17      2 kg                3148 S 1100 W       Salt Lake City, UT 84119          EOD     Delivered by Truck 1 at 09:32 AM
        18      6 kg                  1488 4800 S       Salt Lake City, UT 84123          EOD     Delivered by Truck 2 at 09:28 AM
        19     37 kg              177 W Price Ave       Salt Lake City, UT 84115          EOD     Delivered by Truck 2 at 08:29 AM
        20     37 kg                3595 Main St       Salt Lake City, UT 84115     10:30 AM     Delivered by Truck 2 at 08:28 AM
        21      3 kg                3595 Main St       Salt Lake City, UT 84115          EOD     Delivered by Truck 2 at 08:28 AM
        22      2 kg            6351 South 900 East           Murray, UT 84121          EOD     En route, loaded onto Truck 1 at 09:05 AM
        23      5 kg           5100 South 2700 West       Salt Lake City, UT 84118          EOD     Delivered by Truck 2 at 09:30 AM
        24      7 kg                5025 State St           Murray, UT 84107          EOD     Delivered by Truck 1 at 09:54 AM
        25      7 kg        5383 South 900 East #104       Salt Lake City, UT 84117     10:30 AM     En route, loaded onto Truck 1 at 09:05 AM
        26     25 kg        5383 South 900 East #104       Salt Lake City, UT 84117          EOD     En route, loaded onto Truck 1 at 09:05 AM
        27      5 kg           1060 Dalton Ave S       Salt Lake City, UT 84104          EOD     Delivered by Truck 2 at 08:54 AM
        28      7 kg                2835 Main St       Salt Lake City, UT 84115          EOD     Delivered by Truck 1 at 09:18 AM
        29      2 kg                  1330 2100 S       Salt Lake City, UT 84106     10:30 AM     En route, loaded onto Truck 2 at 09:58 AM
        30      1 kg                 300 State St       Salt Lake City, UT 84103     10:30 AM     En route, loaded onto Truck 2 at 09:58 AM
        31      1 kg                3365 S 900 W       Salt Lake City, UT 84119     10:30 AM     Delivered by Truck 1 at 09:30 AM
        32      1 kg                3365 S 900 W       Salt Lake City, UT 84119          EOD     Delivered by Truck 1 at 09:30 AM
        33      1 kg                2530 S 500 E       Salt Lake City, UT 84106          EOD     Delivered by Truck 1 at 09:14 AM
        34      2 kg                4580 S 2300 E          Holladay, UT 84117     10:30 AM     Delivered by Truck 2 at 08:11 AM
        35     88 kg           1060 Dalton Ave S       Salt Lake City, UT 84104          EOD     Delivered by Truck 2 at 08:54 AM
        36     88 kg           2300 Parkway Blvd  West Valley City, UT 84119          EOD     Delivered by Truck 2 at 08:44 AM
        37      2 kg              410 S State St       Salt Lake City, UT 84111     10:30 AM     En route, loaded onto Truck 2 at 09:58 AM
        38      9 kg              410 S State St       Salt Lake City, UT 84111          EOD     En route, loaded onto Truck 2 at 09:58 AM
        39      9 kg                2010 W 500 S       Salt Lake City, UT 84104          EOD     Delivered by Truck 2 at 08:59 AM
        40     45 kg                 380 W 2880 S       Salt Lake City, UT 84115     10:30 AM     Delivered by Truck 1 at 09:24 AM

Select an option:
-Press p to display the entire delivery plan
-Press a to display the status of all packages at a specified time
```

G3: Third Status Check (1:00 pm)

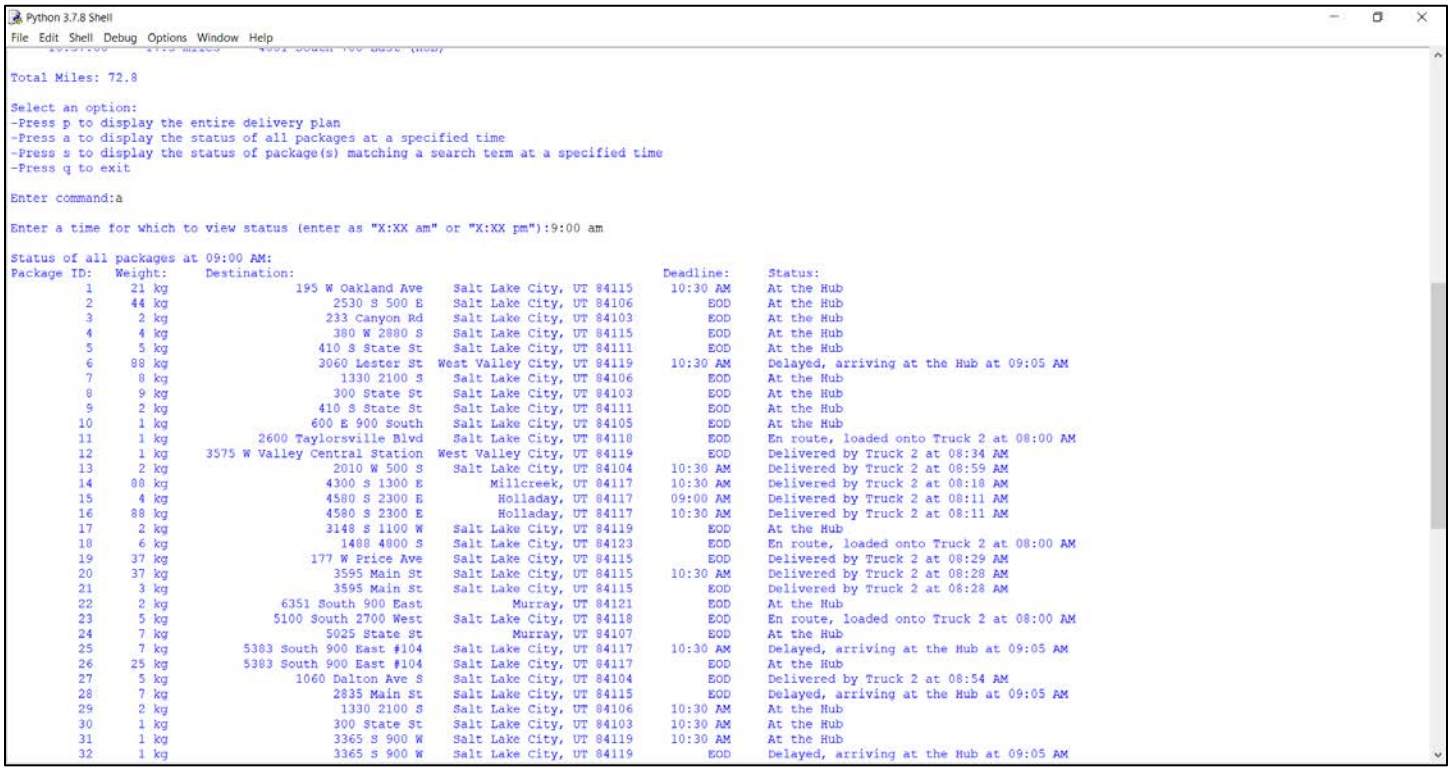
```
C:\windows\py.exe
Enter command:a

Enter a time for which to view status (enter as "X:XX am" or "X:XX pm"):1:00 pm

Status of all packages at 01:00 PM:
Package ID:   Weight:     Destination:                                                Deadline:     Status:
         1     21 kg                195 W Oakland Ave    Salt Lake City, UT 84115     10:30 AM     Delivered by Truck 1 at 09:20 AM
         2     44 kg                  2530 S 500 E       Salt Lake City, UT 84106          EOD     Delivered by Truck 1 at 09:14 AM
         3      2 kg                 233 Canyon Rd       Salt Lake City, UT 84103          EOD     Delivered by Truck 2 at 10:31 AM
         4      4 kg                  380 W 2880 S       Salt Lake City, UT 84115          EOD     Delivered by Truck 1 at 09:24 AM
         5      5 kg                410 S State St       Salt Lake City, UT 84111          EOD     Delivered by Truck 2 at 10:25 AM
         6     88 kg                3060 Lester St  West Valley City, UT 84119     10:30 AM     Delivered by Truck 1 at 09:36 AM
         7      8 kg                  1330 2100 S       Salt Lake City, UT 84106          EOD     Delivered by Truck 2 at 10:11 AM
         8      9 kg                 300 State St       Salt Lake City, UT 84103          EOD     Delivered by Truck 2 at 10:29 AM
         9      2 kg                410 S State St       Salt Lake City, UT 84111          EOD     Delivered by Truck 2 at 10:25 AM
        10      1 kg                600 E 900 South      Salt Lake City, UT 84105          EOD     Delivered by Truck 2 at 10:40 AM
        11      1 kg          2600 Taylorsville Blvd     Salt Lake City, UT 84118          EOD     Delivered by Truck 2 at 09:31 AM
        12      1 kg     3575 W Valley Central Station  West Valley City, UT 84119          EOD     Delivered by Truck 2 at 08:34 AM
        13      2 kg                  2010 W 500 S       Salt Lake City, UT 84104     10:30 AM     Delivered by Truck 2 at 08:59 AM
        14     88 kg                4300 S 1300 E         Millcreek, UT 84117     10:30 AM     Delivered by Truck 2 at 08:18 AM
        15      4 kg                4580 S 2300 E          Holladay, UT 84117     09:00 AM     Delivered by Truck 2 at 08:11 AM
        16     88 kg                4580 S 2300 E          Holladay, UT 84117     10:30 AM     Delivered by Truck 2 at 08:11 AM
        17      2 kg                3148 S 1100 W       Salt Lake City, UT 84119          EOD     Delivered by Truck 1 at 09:32 AM
        18      6 kg                  1488 4800 S       Salt Lake City, UT 84123          EOD     Delivered by Truck 2 at 09:28 AM
        19     37 kg               177 W Price Ave      Salt Lake City, UT 84115          EOD     Delivered by Truck 2 at 08:29 AM
        20     37 kg                 3595 Main St       Salt Lake City, UT 84115     10:30 AM     Delivered by Truck 2 at 08:28 AM
        21      3 kg                 3595 Main St       Salt Lake City, UT 84115          EOD     Delivered by Truck 2 at 08:28 AM
        22      2 kg            6351 South 900 East           Murray, UT 84121          EOD     Delivered by Truck 1 at 10:04 AM
        23      5 kg            5100 South 2700 West      Salt Lake City, UT 84118          EOD     Delivered by Truck 2 at 09:30 AM
        24      7 kg                5025 State St           Murray, UT 84107          EOD     Delivered by Truck 1 at 09:54 AM
        25      7 kg         5383 South 900 East #104     Salt Lake City, UT 84117     10:30 AM     Delivered by Truck 1 at 10:00 AM
        26     25 kg         5383 South 900 East #104     Salt Lake City, UT 84117          EOD     Delivered by Truck 1 at 10:00 AM
        27      5 kg               1060 Dalton Ave S      Salt Lake City, UT 84104          EOD     Delivered by Truck 2 at 08:54 AM
        28      7 kg                  2835 Main St       Salt Lake City, UT 84115          EOD     Delivered by Truck 1 at 09:18 AM
        29      2 kg                  1330 2100 S        Salt Lake City, UT 84106     10:30 AM     Delivered by Truck 2 at 10:11 AM
        30      1 kg                 300 State St        Salt Lake City, UT 84103     10:30 AM     Delivered by Truck 2 at 10:29 AM
        31      1 kg                 3365 S 900 W        Salt Lake City, UT 84119     10:30 AM     Delivered by Truck 1 at 09:30 AM
        32      1 kg                 3365 S 900 W        Salt Lake City, UT 84119          EOD     Delivered by Truck 1 at 09:30 AM
        33      1 kg                 2530 S 500 E        Salt Lake City, UT 84106          EOD     Delivered by Truck 1 at 09:14 AM
        34      2 kg                 4580 S 2300 E          Holladay, UT 84117     10:30 AM     Delivered by Truck 2 at 08:11 AM
        35     88 kg               1060 Dalton Ave S      Salt Lake City, UT 84104          EOD     Delivered by Truck 2 at 08:54 AM
        36     88 kg               2300 Parkway Blvd  West Valley City, UT 84119          EOD     Delivered by Truck 2 at 08:44 AM
        37      2 kg                410 S State St       Salt Lake City, UT 84111     10:30 AM     Delivered by Truck 2 at 10:25 AM
        38      9 kg                410 S State St       Salt Lake City, UT 84111          EOD     Delivered by Truck 2 at 10:25 AM
        39      9 kg                  2010 W 500 S       Salt Lake City, UT 84104          EOD     Delivered by Truck 2 at 08:59 AM
        40     45 kg                  380 W 2880 S       Salt Lake City, UT 84115     10:30 AM     Delivered by Truck 1 at 09:24 AM

Select an option:
-Press p to display the entire delivery plan
-Press a to display the status of all packages at a specified time
```

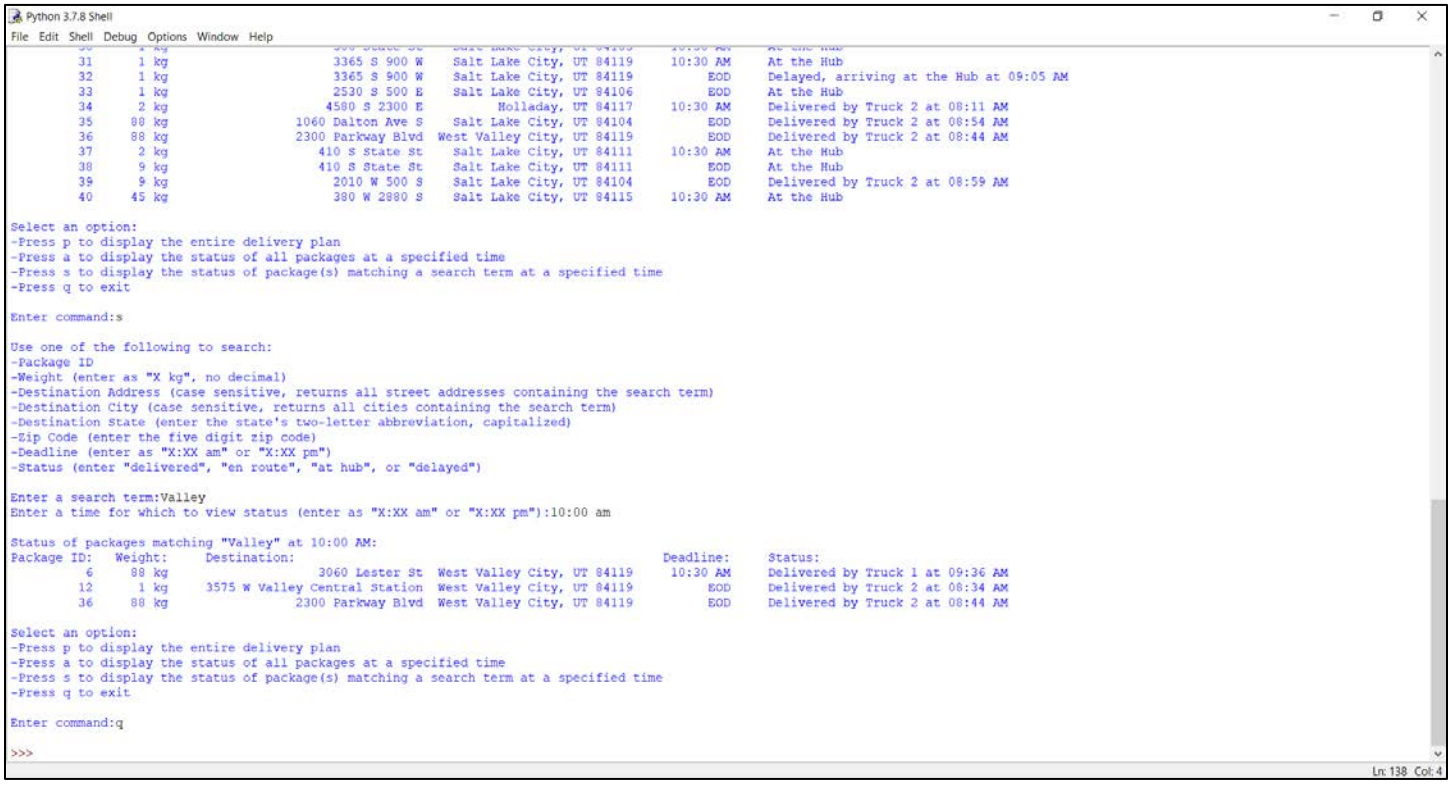
H: Screenshots of Code Execution

The preceding screenshots show the code running without errors in the command line interface, which closes upon choosing the menu option to exit. The following screenshots show an entire run of the program in the IDLE Python interpreter shell, using each of the menu options including exit at least once, without errors.



```
*Python 3.7.8 Shell*
File  Edit  Shell  Debug  Options  Window  Help
>>>
= RESTART: C:\Users\BenGa\Desktop\Data Structures and Algorithms\C950 Project\main.py
Delivery Schedule:

Express Load, Truck 2
     Time:        Distance:     Stop Address:                    Packages Delivered:
     08:00:00      0.0 miles    4001 South 700 East (HUB)
     08:11:20      3.4 miles    4580 S 2300 E                    #15  #16  #34
     08:18:00      5.4 miles    4300 S 1300 E                    #14
     08:28:00      8.4 miles    3595 Main St                     #20  #21
     08:29:40      8.9 miles    177 W Price Ave                  #19
     08:34:20     10.3 miles    3575 W Valley Central Station    #12
     08:44:40     13.4 miles    2300 Parkway Blvd                #36
     08:54:00     16.2 miles    1060 Dalton Ave S                #27  #35
     08:59:20     17.8 miles    2010 W 500 S                     #13  #39
     09:28:00     26.4 miles    1488 4800 S                      #18
     09:30:00     27.0 miles    5100 South 2700 West             #23
     09:31:20     27.4 miles    2600 Taylorsville Blvd           #11
     09:52:40     33.8 miles    4001 South 700 East (HUB)

Delay Load, Truck 1
     Time:        Distance:     Stop Address:                    Packages Delivered:
     09:05:00      0.0 miles    4001 South 700 East (HUB)
     09:14:20      2.8 miles    2530 S 500 E                     #2  #33
     09:18:00      3.9 miles    2835 Main St                     #28
     09:20:40      4.7 miles    195 W Oakland Ave                #1
     09:24:20      5.8 miles    380 W 2880 S                     #4  #40
     09:30:00      7.5 miles    3365 S 900 W                     #32  #31
     09:32:00      8.1 miles    3148 S 1100 W                    #17
     09:36:20      9.4 miles    3060 Lester St                   #6
     09:54:40     14.9 miles    5025 State St                    #24
     10:00:20     16.6 miles    5383 South 900 East #104         #25  #26
     10:04:40     17.9 miles    6351 South 900 East              #22
     10:16:40     21.5 miles    4001 South 700 East (HUB)

Final Load, Truck 2
     Time:        Distance:     Stop Address:                    Packages Delivered:
     09:58:40      0.0 miles    4001 South 700 East (HUB)
     10:11:20      3.8 miles    1330 2100 S                      #7  #29
     10:25:40      8.1 miles    410 S State St                   #5  #37  #38  #9
     10:29:00      9.1 miles    300 State St                     #8  #30
     10:31:00      9.7 miles    233 Canyon Rd                    #3
     10:40:20     12.5 miles    600 E 900 South                  #10
     10:57:00     17.5 miles    4001 South 700 East (HUB)

Total Miles: 72.8
```





Checks of input are also made to prevent errors from occurring if a user enters invalid commands, search terms, or status times.

I1: Strengths of the Chosen Algorithm

- One strength of the greedy selection algorithm is its simplicity, with just O(N) runtime complexity. This allows it to be embedded in a loop to choose each stop along the route without exceeding $O(N^2)$ time complexity overall.

- A greedy selection algorithm chooses the nearest available point for the next stop, the strength of which lies in keeping individual distances between stops low, in general. Its potential weakness is that inherent shortsightedness, which in this case could lead to zigzagging along the route if stops are arranged so that it moves from one side of the map in one direction and then must double back for leftovers, or leaving an unnecessarily long distance back to the hub at the end of the route. To compensate, in my algorithm packages are first sorted into routes meant to encourage the greedy algorithm to follow a more or less circuitous path around the route so that crossing back over is unnecessary, instead returning to the points not chosen from the start point, in the opposite direction of the route's initial path, more naturally at the end of the circuit. This maximizes the greedy algorithm's strength in choosing the next closest stop, and keeps it from becoming a liability.

I2: Verification of Algorithm

As seen in the first screenshot above, all of the situational requirements as follows are met by the program:

- All 40 packages are delivered in one of the three loads.
- All packages with deadlines are delivered on time, including package 15 by 9:00 am.
- Packages 3, 18, 36, and 38 are all delivered by Truck 2.
- Packages 6, 25, 28, and 32 do not depart the hub until at least 9:05 am.
- Packages 13, 14, 15, 16, 19, and 20 are all delivered in the same load.
- Package 9's correct location is not used to determine its placement in a load, or to calculate the initial route of the truck it is on, and it is not delivered until after the correct address is known at 10:20 am.
- No more than 16 packages are in any one load.
- No truck leaves the hub before 8:00 am.
- No more than two trucks (for the two drivers) are ever operating at the same time.
- Trucks always travel at 18 mph.
- Trucks return to the hub after completing delivery of each load.
- Total mileage is 72.8, well under 140.

I3: Other Possible Algorithms and Differences (I3A)

I could have used a variation of Dijkstra's shortest path algorithm to determine a near-optimal route along a given route's stops. This may have found slight improvements to some travel distances (perhaps where the distance from point a→b is greater than distance a→c→b, as my algorithm only considers direct distances). However, this would have a significant effect on the program's time complexity. Dijkstra's algorithm has complexity of $O(V^2)$ implemented with a list and O(E+logV) implemented with a fast heap, where V=number of vertices and E=number of edges (Lysecky & Vahid, 6.11). In this scenario (using N=number of packages), V is O(N) and E is O(N/2) = O(N), so the complexity of applying Dijkstra's algorithm would be either $O(N^2)$ using a list or O(N+NlogN) = O(NlogN) using a fast heap. Either case applied in a loop through each load (O(N)) would result in overall complexity greater than $O(N^2)$ as produced by the greedy selection algorithm, increasing the program's overall time complexity. This decreased efficiency may save a small amount of distance, but most likely very little for the provided set of data given the geographic sorting that occurs before routing and the correctional measures applied to the greedy selections. Although the improvements in mileage with a larger data set might be larger, increased time complexity would also have a greater impact there.

I also could have used a nearest neighbor algorithm to group stops into loads, instead of using the geographic "rings" and regions that I defined based on distances from fixed poles for each point. Applying nearest neighbor would have sorted locations into regions based on their previously assigned neighbors, rather than proximity to the same fixed points for each location (Adamczyk). This would have resulted in a different distribution of locations into regions, perhaps with regions whose points remained closer together. However, that may or may not have been significantly beneficial in the end given that in my sorting, sometimes the intent was to find nearby points but sometimes it was more to define geometric shapes within a route, which would not be accounted for directly by nearest neighbor. So, although nearest neighbor would have accounted for geographical separation of locations n a way that my core greedy selection algorithm does not, this was compensated for instead with my own geographic sorting of locations before the greedy algorithm is applied.

J: Different Approach

- One thing I would do differently were I to do this project over again would be to include in each data structure a reference to the data structure(s) created in the preceding section(s) of the program. (For example, the PackageTable class would have a self.LocationTable data member, LoadList class would have self.PackageList and self.LocationTable, etc.) This would remove the necessity of passing multiple references to these objects in calls to each other's methods, which would improve efficiency a bit and readability substantially, while taking up only a few more reference variables in space. I did employ this approach for the Schedule class and found it to be an improvement, but did not go back and change all the other classes.

- My program automates the application of "special notes" requirements to packages during import, but it does require that the Excel/csv file first be formatted to give each type of requirement its own field (or fields, in the case of an incorrect address to be updated). This may be a bit cumbersome, and if doing this over I could extend the program's ease of use by further automating the process to pick out different types of requirements from the same, single "notes" field. This could be done by checking for key words or phrases in a given note, then parsing the relevant data out from it. For example, during import it might check whether the note includes "with" and if so, parse out any numbers separated by commas within the field into the packages bundle list. Or it could check for the word "truck" then parse out an integer to assign as the package's truck requirement, or if "delay" is present not schedule that package to depart until a time parsed out by finding a substring in the form int:int, and so on.

K: Verification of Data Structure

The PackageTable, LocationTable, and LoadList data structures store all of the data needed to ensure that the requirements of the situation (as outlined in Part I2) are all met, and then after routes are calculated, store data about each route travelled allowing calculation of the total miles to be calculated by a loop through each stop along each load (as printed in the first screenshot in Part G). This verifies the total miles travelled by all trucks along all routes as 72.8, well below 140, as shown in the printout along with the completed delivery of all 40 packages in accordance with their requirements. The PackageTable object contains the required hash table, storing package information by direct access hashing of package IDs to Package object references, and has a look-up method as required to report package information and status (as seen in screenshots in Parts G1-G3).

K1A: Efficiency

The look-up function may perform a single loop through each Package object in two of three possible cases when it is called, making it O(N). This means that as the number of packages (N) increases, its time required to execute will increase linearly with it. This is reasonable, as if X times more packages must each have the same number of checks performed (none of which themselves require loops), the total time will also increase by a factor of X.

K1B: Overhead

The PackageTable object's data structure is of space complexity $O(N^2)$, so increasing the number of packages by a factor of X could increase its size by up to X^2 in a worst case. However, this is only the case if nearly all packages are bundled together, making the bundle list of each package objects approach N in length, which for N packages results in $O(N^2)$ space. Assuming small bundle lists of 0 to a few packages at most, each Package object will typically be closer to O(1) and the PackageTable's space complexity closer to O(N). So, while it is possible that an increase in packages by a factor of X will result in a factor of X^2 growth for the PackageTable data structure, it is more likely to be closer to a linear increase by approximately the same factor of X.

K1C: Implications

An increase in the number of trucks would have only a small impact on space usage as each Truck object has only three simple data members, and would have no effect on the time required for the look-up function, as it loops through each package, not each truck, and each Package object already contains a data member that will already have recorded the package's assigned truck, so this will not need to be checked against the list of trucks during look-up.

Increasing the number of loads taken/routes traveled would have a more significant effect on space usage than the number of trucks specifically, as each Load object contains a route data member that is a list object of variable length, with entries for each stop taken along the load's route. If the number of packages is not increased along with the number of loads though, this increase will still be mostly mitigated by the corresponding reduction in length of other loads routes, as all routes will combined will still be delivering the same number of packages, and thus making about the same number of stops, and so utilizing roughly the same number of list entries (amount of data) to store information about those routes altogether, perhaps with some extra to reflect "stops" at the hub at the beginnings and ends of the new routes at the hub. As to the time required for the look-up function, again, each package's assigned load has already been stored as an integer key in its Package object, along with its delivery time, by the time the look-up method can be called. So information about different loads is directly accessed rather than checked in loops during look-up, meaning that increasing the number of loads also does not impact the runtime for the look-up method.

Increasing the number of cities specifically, if not changing the overall number of destination locations, does not require any increase in space or runtime for look-up; the LocationTable object's structure would remain the same size, and regardless of what a package's city is, it is checked against the search term in approximately the same time (with negligible difference based on variance in the length of city name strings).

However, increasing the number of destination locations overall, whether in the same city or new cities, would have a significant impact on the program. This would not affect the size of the PackageTable, assuming the number of packages total remains constant, as each Package object would still only have one associated destination. The LocationTable's space usage would be affected though, increasing by a factor of X^2 for an increase by a factor of X in the total number of locations. This is because the matrix of distances that it contains uses $O(N^2)$ space, being of a size (number of locations)^2. An increase in the number of locations would still not impact the runtime of the lookup function though. Each Package object stores a location key allowing direct access to its destination Location object without checking/looping through any others, so no matter how many locations are added, once at the point of calling the look-up method, accessing each package's location data will take the same amount of time. The only variable that will cause an increase in the look-up method's runtime, because it only loops through Package objects, is the number of packages.

K2: Other Data Structures and Differences (K2A)

- One other structure of hash table that could have been utilized would be to develop a hash function for locations, and store each Package object's reference in the "bucket" mapped to its destination location, with chaining to add packages with the same destination to the same bucket. This would utilize less buckets than the direct hashing by package ID, but each would use more space, as they would be formatted as lists and each may contain more than one Package object reference. One advantage of this structure is that it allows easy access to all packages with the same destination, useful in portions that require, for instance, adding all packages with the same destination to a single load, or delivering all packages with the same destination at the same time. I considered using a structure like this for that reason, as it would allow me to avoid some loops through each package, but decided against it because of its significant downsides, requiring small loops to find a single package within a bucket and potentially large loops (through each list within each bucket) to find a package by ID number, a major drawback to efficiency in several parts of the program compared to direct access via that ID number. It would be more complicated to display the list of packages in order by ID number this way too. I also realized it would be redundant given that each Location object has a list of packages, so the rest of the packages as the same location of one given package can already be found using LocationTable.table[PackageTable.table[ID].destination].package_list to get the list of packages at that location.

- A non-direct hash function using package IDs mapped using a modulo function to a set number of buckets a bit larger than the number of packages, and handling collisions using linear or quadratic probing could also be used to insert packages into a hash table that would also be fairly efficient in space usage and easy to access packages from via ID number/in order. It would be more versatile as well, functioning even if packages' information is not provided in order by ID number, or if ID numbers were non-sequential, non-continuous, did not start at 1, etc. However, given the particular set of ID numbers in the provided scenario, this was not necessary and it was more efficient to use direct access to achieve a perfect hash inserting the packages sequentially starting with package 1.

- Distance information could also have been stored as adjacency lists instead of a matrix, with each Location having a dictionary of location: distance pairs. This could be more efficient if the map was represented by a sparse graph, where many locations do not connect to all other locations (Lysecky & Vahid, 6.3). This is because in such a case, a matrix of distances would contain many blank values where connections do not exist, wasting space. However, in a situation such as this where each point is connected to each other point, this would use more space than a matrix (the same number of distance values would have to be stored, but with the addition information, like keys, associated with dictionaries as well). It would also make access to distance values more complex operations, searching through the first location's dictionary of adjacencies for the correct key of the second location, rather than directly accessing a distance value as LocationTable.distances[first location key][second location key].

L: Sources

Adamczyk, Jakub. (2020, August 6). "k nearest neighbors computational complexity." *Towards Data Science.* https://towardsdatascience.com/k-nearest-neighbors-computational-complexity-502d2c440d5

Lysecky, Roman & Vahid, Frank. (2018). *C950: Data Structures and Algorithms II*. Zyante Inc. zyBooks.com