**Deadline-first Algorithm Overview**

**Requirements:**

**A1.** My algorithm which I will refer to as my Deadline-First algorithm, Is a greedy algorithm, split into 2 functions, next\_delivery(), and route\_delivery()

**B1.** next\_delivery is the core part of the algorithm which decides what package to deliver next, it works as follows:

**next\_deliver(truck, current\_location):**

**for package on truck**

**if wrong address  
 continue loop**

**first package = best package**

**continue loop**

**if distance to address < 1 mile**

**return package & distance**

**if package.deadline < best\_package.deadline**

**best\_package = package**

**continue loop**

**elif package.deadline == best\_package deadline**

**OR (package.deadline & best\_package deadline == None)**

**if package.distance < best\_package.distance**

**best\_package = package**

**continue loop**

**return best\_package, distance**

This pseudo-code is highly simplified, but covers the most important checks. We prioritize packages with the earliest deadlines first, unless a package is within 1 mile of current location. If no packages have a deadline, or 2 deadlines match, we use whichever is the closest to our current location. This function is nested inside the route\_delivery() function which re-runs the function and updates the current location on each iteration, until every package loaded is delivered.

**route\_delivery(truck, departure)**

**to\_be\_delivered = truck.packages**

**current\_time = departure**

**current\_location = “HUB”**

**while to\_be\_delivered # iterates until list is empty**

**returned = next\_delivery(truck, current\_location)**

**# returned is a list to store a package and a float (distance)**

**current\_time =+ time(distance/truck\_speed)**

**Truck distance += distance**

**to\_be\_delivered.remove(package)**

**# distance & package both from returned**

**package.delivered = current\_time**

**current\_location = package.address**

**distance = address\_distance(“HUB”, package.address)**

**# address\_distance returns the distance between two addresses on**

**# the distance map**

**current\_time += time(distance/truck\_speed)**

**truck\_distance += distance**

The core function of the algorithm is nested inside route\_delivery, which re-runs it (next\_delivery) until it’s finished routing every package that was on the truck. The list to\_be\_delivered is made to mirror truck.packages (altho here we use reference assignment, in the program we iterate through truck.packages and append each package to create and independent list) and we use that list as a boolean to loop until every package is delivered, and then we run some more code to route the truck back to the hub to finish it’s day.

**B2.** I created this program in PyCharm community edition 2022.3 using Python 3.9.5 on my home desktop. My desktop is running Windows 10 Pro 64-bit, on an asus rog strix z370-e motherboard, an i7 8700k cpu, with 32gb ram.

**B3.** The average complexity of next\_delivery is O(n), while the worst case complexity would be O(n^2), altho worst case isn’t technically possible as there is only one group of packages. Route\_delivery()’s complexity is O(n^2), the while loop in main used for user input is O(n\*(m+k) worst case, but is highly dependent on user input. The complexity of every other function in my program is notated in the comments of the program. The overall complexity of the program is heavily dependent on the input loop as it is the highest complexity function, so the program would also be O(n\*(m+k)).

**B4.** While the program as is cannot be scaled much larger, it would need minimal adjustments to be able to be scalable. For example in a similar way packages are loaded from a CSV file, trucks could be as well. With some additional truck attributes, such as a priority attribute for trucks that leave first thing at start of day to deliver packages with an early deadline, or to account for trucks with a faster average speed or capacity. Some other manual package assignments to Trucks 1 and 2 would have to be adjusted to check for trucks with specified attributes, such as a specific departure time. It would also be conducive to increase the number of buckets within the hashmap, as currently it only has 20, and 2 packages get assigned to each bucket. With many more packages, it would reduce computation time to increase the number of buckets to the largest reasonable size. With these changes the program should have no issues scaling to larger sizes.

**B5.** The software is segmented into various python files. This makes for easy locating of code that is responsible for various attributes of the program, and makes it easier to maintain than if the program was all within main.py. From an end-user perspective, the prompts make clear what inputs do what, and allow the user to easily select a numerical option to perform the operations they wish, to retrieve any information they may need.

The program tracks key moments in each packages time-line, so that when data is requested such as where the package is/was at a given time, only the time in which its status was changed needs to be checked. For example a package that is en route at 9:00 and delivered at 9:30 would be on the truck at any time between those two times, or at the hub anytime before 9:00. The program Also maintains efficacy by enforcing deadlines first, except in exceptional cases where the distance to a package would be miniscule, ensuring packages w/o a deadline but that have the same address as a package with one are delivered at the same time. In cases where all deadlined packages are delivered, or there are none, distance is prioritized, to minimize mileage, and the loading algorithm also loads packages by deadline and distance in an effort to ensure that all packages with a deadline are on the first truck leaving in the morning, and that all other packages are sorted onto trucks based on relative distance to one another, further minimizing truck mileage.

**B6.** Self adjusting data structures, such as the hash-table implemented in this project are very useful for cutting down on time complexity when a specific object needs to be retrieved and operated on, since we don’t have to iterate through a full length list of all the objects until we arrive at the one we want, and can simply use a key to locate the sublist it is contained within. However we are still using lists and with a smaller number of buckets, the sublists of each bucket get larger as the total number of objects increases. While we can alleviate some of this by adding more buckets, there is an upper limit on the number of buckets we can have. Additionally we are forced to use a nested for loop to iterate over all the objects within the data structure, which technically does not increase the final operational complexity, as we are still iterating through the same number of objects, it does affect the big-o notation and makes the function seem more complex. Having to iterate through nested lists in such a manner does also mean we must create additional reference lists if we use to sort these objects, and easily manipulate the objects within. Lastly if we want to manipulate specific objects within the hash-table, we must write our own functions to select the object we want and perform the desired operations, instead of using the stock append, remove or setattr functions.

**D1.** In this Project I am using a hash-table as my self-adjusting data structure, and using it to store all my package objects, so that when needed I can retrieve a specific object without iterating through a long list to find the data I need. My hash-table itself does not automatically account for relationships to other objects in the program, but such functions do exist in other parts of my program to make comparisons between a package address and addresses in the address list to identify distances between 2 packages from the distance table. The Package object itself accounts for relationships to other objects as well, and includes none type initialized attributes that are later used to store status updates such as departure from the hub, what truck the package was loaded onto, and other necessary information.

**I1.** The algorithm I used in my solution accounts for many different variables to assure that all constraints and limitations are met. My algorithm checks for delivery deadlines first and foremost unless a package is found that is in the immediate vicinity (less than 1 mile away including packages addressed to current location). This ensures that packages are delivered on time, but also that extremely efficient options such as delivering multiple packages addressed to the same location at the same time, are not ignored to prevent the same truck visiting an address multiple times. It also accounts for packages that have the same, or no deadline, by comparing distance to destination when no packages with deadlines are left to deliver on the truck.

**I2.** My algorithm accounts for all requirements listed. It checks for deadlines first and ensures packages are delivered on time, and in the off chance that they are not, prints out a message to console. It also accounts for distance between packages to minimize the mileage of deliveries, ensuring that when all packages are delivered, the sum of all trucks mileage is well under 140 total miles. While not represented in the pseudo code, the algorithm also checks for packages that need an updated address, such as package #9. next\_delivery will check package notes, and if it finds ‘wrong address’ will skip over that package, while route\_delivery, will check for truck 3 where package 9 is loaded, and perform a time check to update package 9 to the correct address, and update it’s notes to reflect that the address has been corrected allowing next\_delivery to select it.

All other constraints are met and verified outside of my core algorithm, such as package groups, my separate sorting algorithm ensures that packages are loaded onto the correct trucks, and that package groups are respected.

**I3.** I also could have used a Nearest-Neighbor algorithm, that prioritizes the smallest distance between packages instead of the deadline, with some modifications to ensure that packages with an upcoming deadline are selected next regardless of distance or similar modifications to ensure delivery deadlines are met. This would’ve required my algorithm to pass more values in return statements and arguments between functions, or for the entire algorithm to exist within one function.

Another more complex algorithm I could’ve used is a derivative of Dijkstra's shortest path algorithm. Instead of simply going straight to the next package address, I could’ve gone through and compared multiple distances between packages to find the shortest distance to the package, while also delivering many packages along the route. Combining this with my deadline first algorithm could result in a much more efficient algorithm, that while prioritizing deadlines, could also deliver packages that are on the way to the package, reducing miles traveled, and finishing all loaded deliveries much quicker, although this would be much more complex to implement, and would greatly increase the space-time complexity of the algorithm, due to all the distance comparisons that would need to be made and stored.

**J.** If I were to do this project again, for starters I would manually load packages onto trucks instead of writing an algorithm to sort them onto trucks for me, as this doubled the time to write the program, as it was at least as complex as the routing algorithm if not moreso, and package groups complicate it and require less than efficient solutions, to account for. I might also try to work in some concepts from Dijkstra's, while my algorithm would still prioritize deadlines, I could add a function that would find packages that could be delivered between the two points that wouldn’t increase the total distance by more than a certain percentage, ensuring no long detours are taken, and ensuring that packages are still delivered by their deadlines. I would also certainly jump into the actual coding of the project much sooner, as it was much easier and faster to learn Python by actually writing the code and googling or referencing the class material when I encountered quirks of the language I didn’t understand.

**K1.** The hash-table I used meets all the requirements set forth by this assessment, it stores data objects based on a primary key, in this case package id. It can perform look-ups based on a key given to find the specific package matching an id given through a key-lookup function that identifies which bucket the object is stored in. Several supporting functions are written to support this data structure allowing look-ups, data updates, specific attribute retrieval, among others.

The Time needed for lookups is affected minimally by increases in the number of packages, as you would need to double the number of packages to increase the average look-up time by more than 50%, since each object is stored in a sublist, and currently there are 20 sublists each containing 2 packages (altho the number of sublists could be easily updated to much higher numbers to account for more packages and maintain low lookup times). Adding a single package would only increase lookup times for the specific bucket it gets assigned too, and to affect all buckets you’d need to at least add 20 packages (or more if we added more buckets)

Changes to the space requirements of the data structure would not meaningfully increase with new packages added when compared to a traditional list, although there is extra overhead space requirements for defining the structure of the list and it’s sublists, adding more buckets (or sub-lists) would likely be a more substantial increase in space requirements when combined with adding more packages, however this would help keep the time-complexity of the lookup low.

Changes to the number of cities or trucks, would not affect the lookup time of the data structure as these objects are not stored within the hash-table, the number of packages needed to be delivered within the extra cities or loaded onto the extra trucks, however, would.

**K2.** One data structure I could have used Besides the hash-table and class objects and 2D lists/vertexes, would be something such as a binary tree to store distance values between locations. Although this would have introduced some complications compared to the 2D list I used for storing distances and would not have worked smoothly with other elements of my Project.

Another data structure I could have used might be Nested Dictionaries instead of Class Objects. Ultimately this would have introduced various head-ache inducing complications, while class objects can perform a similar function under abstraction which I did not even end up needing.

**L2.** No sources were used except for purely educational purposes, and to my knowledge nothing from my project is quoted paraphrased or summarized from any other source.