Core Algorithm Overview

**Stated Problem:**

This project will create an algorithm to optimally design a route for WGUPS deliveries in Python. The requirements include 40 packages with access to 3 drivers and 2 trucks. Each truck may only carry 16 packages at a time. Of those 40 packages, some may have constraints such as being required to be delivered with certain others or delivered after a certain time. The solution to this comes in a 2-step process. We will first sort the packages to be on specific trucks then each truck will use a greedy algorithm in which it will always pick the next shortest stop to determine the most efficient path of deliveries.

**Algorithm Overview:**

After the packages are sorted and sent to the individual trucks in an unordered manner each truck will execute its greedy algorithm. The truck will assume its starting location is the hub. The truck will then look at each package and measure the distance to that location. After measuring all possible stops the truck will accept the shortest of those as its next stop and then use that packages’ delivery destination as the next cycle's starting location. From that point, the algorithm is repeated until all packages have been ordered.

1. **Within the Truck object, the algorithm will operate on the unordered\_cargo list of that truck with a starting location at the hub.**
2. **All packages within the objects unordered\_cargo list will be checked to remove packages and add them to early\_delivery\_packages as needed. This set consists of packages that must be delivered early in the day such that they must be prioritized to accomplish this.**
3. **WHILE length(early\_delivery\_packages) greater than 0:**

**Set shortest\_distance to 100.0 (this is an arbitrarily long distance to check packages against)**

**Set chosen\_item to Null (this sets an arbitrary next stop location)**

**FOR package in early\_delivery\_packages: (This will cycle through each remaining unordered package to assure the true shortest distance is chosen)**

**If distance between current\_location and package\_location < shortest\_distance**

**Set shortest\_distance as this new value**

**Set chosen\_item as this package**

**APPEND chosen\_item to ordered\_early\_delivery\_packages**

**(This adds the next shortest stop to the ordered list)**

**REMOVE chosen\_item from early\_delivery\_packages**

**(Finally, we remove the chosen\_item from the unordered list to repeat the process)**

Complexity at this point is O(N^2) across some number of the original unordered\_cargo

1. **WHILE length(unordered\_cargo) greater than 0:**

**Set shortest\_distance to 100.0 (this is an arbitrarily long distance to check packages against)**

**Set chosen\_item to Null (this sets an arbitrary next stop location)**

**FOR package in unordered\_cargo: (This will cycle through each remaining unordered package to assure the true shortest distance is chosen)**

**If distance between current\_location and package\_location < shortest\_distance**

**Set shortest\_distance as this new value**

**Set chosen\_item as this package**

**APPEND chosen\_item to ordered\_cargo**

**(This adds the next shortest stop to the ordered list)**

**REMOVE chosen\_item from unorded\_cargo**

**(Finally, we remove the chosen\_item from the unordered list to repeat the process)**

Complexity at this point continues to be O(N^2) across the rest of the original unordered\_cargo

1. **JOIN ordered\_early\_delivery\_packages and ordered\_cargo to create full and final ordered\_cargo list with all members of ordered\_early\_delivery\_packages preceding ordered\_cargo.**

The worst-case and expected complexity of this algorithm is 0(N^2) since each package must be cycled through for each unordered delivery. Comments throughout the provided code show the complexity of individual methods and blocks. We can see that complexity is nearly linear through the entire runtime. This provides a very scalable algorithm. Bandwidth challenges are of no concern due to this program entirely operating within the local machine.

**Advantages of chosen Algorithm:**

The greedy algorithm is very fast and scalable. Any size input can be used with a greedy algorithm with minor modifications at worst making the maintenance of code very easily. The algorithm can satisfy the delivery requirements and maintaining a very low total distance needed for trucks to travel. Another advantage is that since the algorithm is static it allows very easy lookup functions to be created much like that which is currently used in the user interface which outputs all necessary information of all packages at any given time.

An alternate method that is available is called Dynamic Programming. This reduction to smaller problems would have the potential to find an even shorter total path by checking for shorter paths from other locations. However, this would have come at an extensive cost to the complexity of the program.

One final algorithm that could have been used would be of a more heuristic style. This would find the shortest paths from hub to available destination and add all packages going to that destination to the truck. This would have very similar complexity to the greedy algorithm which I have used but it would be less scalable and more difficult to maintain with regards to the addition of new delivery destinations.

**Programming Models:**

This program is written in Python 3.8 and executed in Pycharm. It is isolated exclusively to operating on the local machine so there are no communication protocols or network flow relevant to a project of this type. All input data is retrieved from a CSV in the project folder directly into python.

**Ability to Adapt:**

This program was designed with adaptability in mind. Any changes to the number of trucks, drivers, packages, or destinations could be accounted for quickly with very minor changes. New CSV files could be provided in almost all situations. The sorting algorithm already uses regular expression searches to find packages with special instructions so new packages with special instructions would only need to be verified that they were contained in one of the existing classes. The challenge in the future is in the case of packages with changing addresses like the provided package #9. I manually catch that package and adjust its destination in the code. If I were improving this software, I would add a method to dynamically catch “wrong address” packages and fix them while the sorting method is taking place.

**Efficiency and Maintainability:**

The overall efficiency of this program is very good, with only 1 instance of a greater than O(N) complexity at O(N^2) on line 83 in trucks.py. Maintaining this software is very easy because data is always neatly contained inside of a method allowing for reuse and modification of those functions and methods. This structure also aids in simplifying debugging.

**Data Structures:**

The data structure I used to meet the requirements was a list of lists operated as a Hash Table in which the package number is used as a key and the value of the hash is a package object containing all other data. This allowed me to store each package at an index that was predictable based on that package's number. This was an extremely fast approach given that all methods of my hash table are O(1) complexity except for those methods intending to have multiple return values in which the complexity is still only O(N). This speed is easily the biggest benefit of this approach allowing immense flexibility in being able to retrieve packages by key, adding virtually no overhead when accessed. The challenge of a list of lists is the difficulty in handling potential collisions.

Other Data structures could have also filled this role such as the Binary Search Tree or a Graph. The BST would have forced me to presort all packages by some criteria. This would have allowed quick access to an individual node but would have forced a complex sorting situation an extra time in the program. The Graph would have allowed clustering of data nodes by maybe zip code in which we could have loaded the trucks by traversing the graph with a walk length of 16. This would have allowed very good scaling of additional packages at the cost of fine control over optimal distance.

