* Nearest Neighbor algorithm logic in pseudocode

**Input**: a start\_location as a pkg object, a list of package objects

**Output**: a shortest path

|  |
| --- |
| Function find\_fast\_route(start\_location, packages):  Initialize an empty\_route list  Initialize an empty nearest\_address  If start\_location object == Hub location object:  Set travel distance attribute inside the object to 0  Set current location to a start location  While package list is not empty  initialize a location\_holder variable to None  initialize and set a nearest\_distance variable to an infinity  for each package in the package list  get the specific address from package object  find the distance between the specific address and the current location  if the distance is not empty or None  cast the *distance* as a float  if the distance < the nearest\_distance variable  set location\_holder to hold this package object  set the nearest\_distance to the distance  repeat until all package looped through  if the location\_holder != None  if the current\_location = the nearest\_location  update travel\_distance = 0  else, update the travel\_distance to the current package that has the nearest distance  append the nearest location package to the route list  remove this package from the packages list  set current\_location to the nearest location  repeat until the packages become empty and get out of the while loop  return a route |

* + Time complexity is O(N^2)

A nested loop, the for loop, loops through the entire list again to find the nearest package while the outer loop, the while loop, loops through n times, so it is n\*n = n^2

* + Space complexity is O(N)

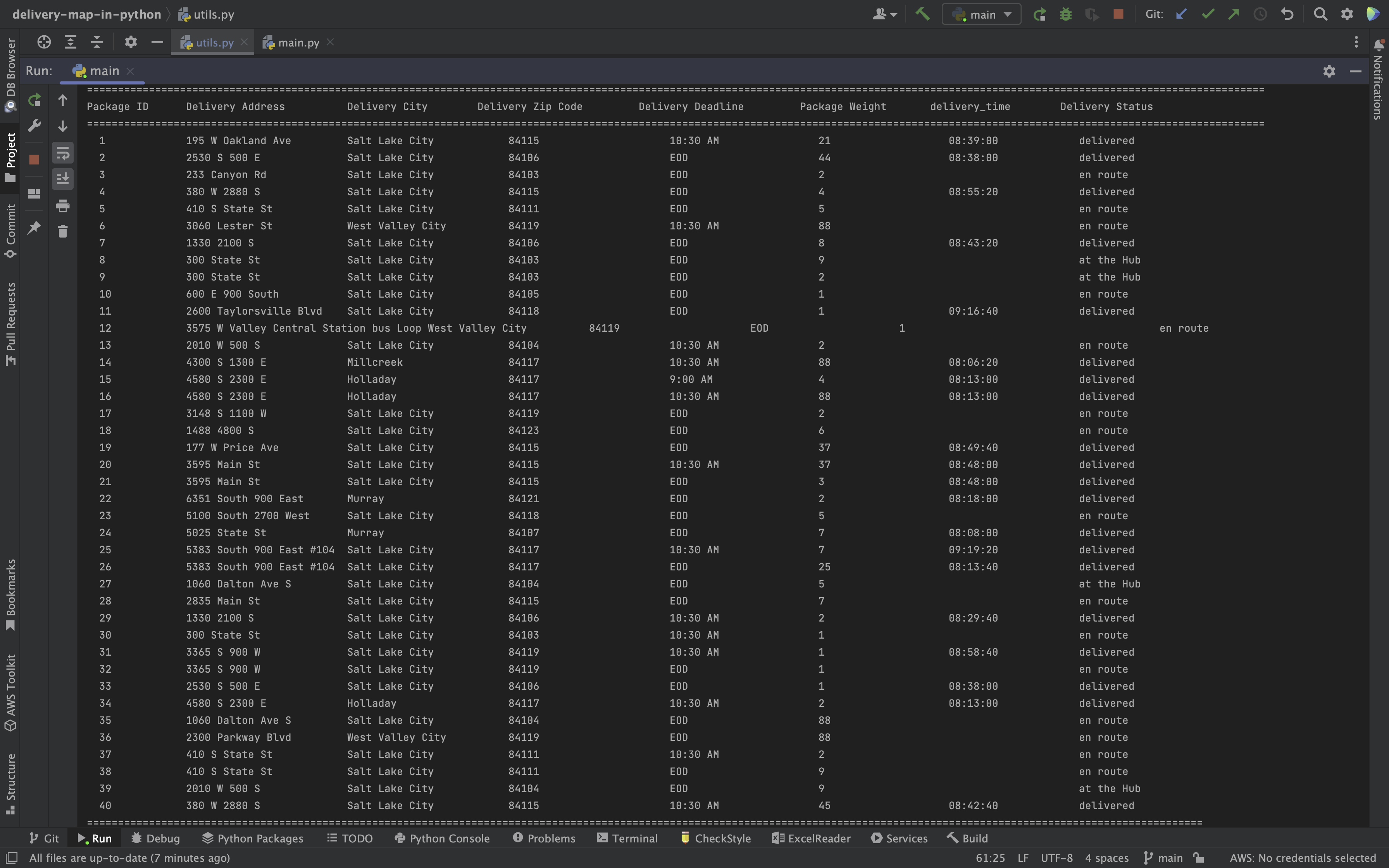
The route variable is a new list that created within this function. The list stores each package that is visited. The size of the ‘route’ list grows linearly with the number of packages in the input list, which means that the space used by the function is directly proportional to the number of packages.

* Programming environment
* Python 3.10.9
* IntelliJ IDE 2022.2 (Community Edition)
* Only run on a local machine
* The capability of the solution to scale and adapt is logically and accurately explained, including the algorithm’s ability to scale to the number of packages.The discussion accurately addresses why the software is efficient and easy to maintain
  + The algorithm I am using is optimized for finding the most efficient route for drivers to deliver packages. It is a common approach for solving problems related to finding the shortest route that visits a given set of locations and returns to the starting point, making it suitable for the purpose of the program.
  + The program is easy to use as it can read packages and graph/distance data from CSV files without requiring changes to the code.
  + The code is well-documented, making it easy to follow and understand when reading it.
  + The program has a clean and clear user interface for interacting with users.
  + The current program is sufficient for handling deliveries with a deadline of 10:30 AM, as long as the urgent deadline package count does not exceed 16. However, for larger numbers of urgent deadline packages, additional resources such as more drivers and trucks may be required. The maximum capacity for the program is calculated at 48 packages, using 3 trucks each capable of carrying 16 packages. Note: the 16 package count is based on the latest delivery start time of 8:00 AM and the 10:30 AM as latest deadline. A recursive algorithm is implemented in the code to determine the number of packages that can be loaded on a truck for deliveries with a 10:30 AM deadline. (See deliverable\_package\_count() function in utils.py)
* The discussion accurately addresses both the strengths and weaknesses of the self-adjusting data structures and the hash table. I1: At least 2 strengths specific to the algorithm identified in part A are accurately described, and both strengths apply to the scenario.
* The algorithm I used to calculate a fast route can be applied to a wide range of input data and is also a very common solution for the purpose of this project. It has the following strengths:
  + flexibility and reusability given a start location and a list of all locations, it will find a sufficient route and return to the caller regardless of where it is been called through the program
  + Travel distance calculation: the function updates each package with the travel distance
  + Same address handling: if the next iteration has the same address, update the distance with a 0 as those packages can be delivered at one time without additional driving
  + A deep copy method usage: the function used a copy.deepcopy() method to make a package list to ensure the original data is not modified.

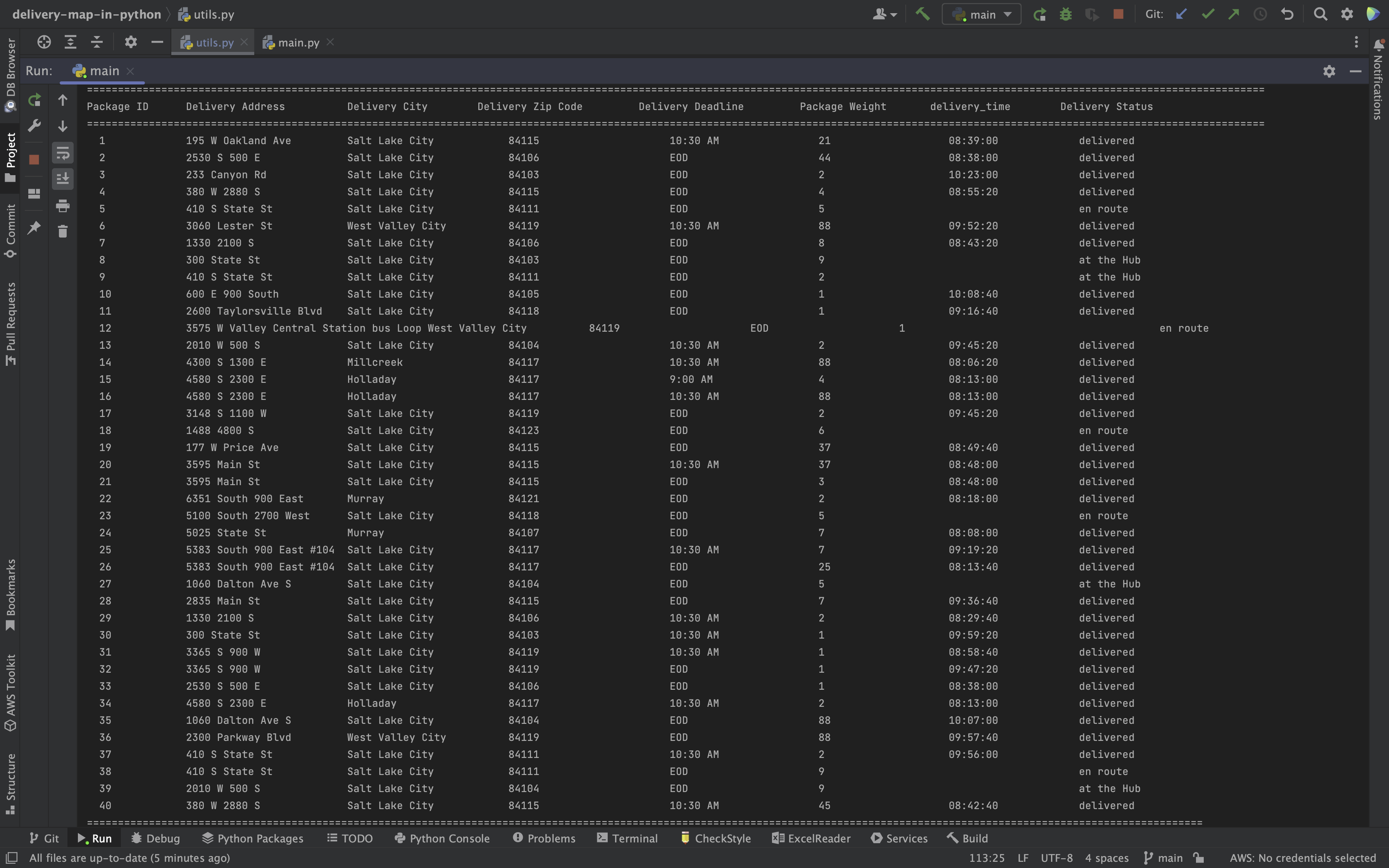
However, it also has the following weaknesses:

* + as data input gets larger, the runtime increases exponentially with the number of locations. For large number of locations, it may not be possible to find an optimal solution in a reasonable amount of time.
  + The nearest neighbor approach may not always find the optimal route, particularly if the locations are not distributed uniformly.
  + The function only consider distance between locations and not the time. If take the time as part of optimization, this may not result the best measure of the fastest route.
* Hash table is very efficient for lookups, insertions, and deleting. In this program, the key is the package ID, the value is the package object which contains all the package information. Thus, it is easy to find packages based on package ID. However, hash tables are sensitive to has collisions, which occur when two or more keys are mapped to the same index in the table. This can lead to poor performance if the number of the number of collisions is high. In my case, it would be the package ID increases. Also, the keys are not in order. Thus, I must use sort by package ID to get list the package information from 1 to 40.
* Screenshots shows a list of packages that are loaded on each truck and the status of each package at 3 different time

(the status of all packages at a time between 8:35 a.m. and 9:25 a.m.)



(the status of all packages at a time between 9:35 a.m. and 10:25 a.m.)



(the status of all packages at a time between 12:03 p.m. and 1:12 p.m.)

Meanwhile, the following screenshot also indicates all packages are delivered on time by comparing package delivery deadline and the package delivery time.

Calendar

Description automatically generated with medium confidence

* The screenshots capture a complete execution of the code and include the total delivery mileage.

Graphical user interface, text

Description automatically generated

* Identifies at least 1 aspect that would be done differently if the project were attempted again, and it includes details of the modifications that would be made.

If I were to start over, I would store the same references to the original packages list (maybe have a copy list as a backup) in the trucks instead of a copying of the packages. This way, when a package is updated during delivery, it would also update in the original list of packages, which is not only a better object-oriented programming but also can reduce an large amount of redundant code. Additionally, I would begin by creating a delivery high level flow diagram before writing any code. This would have saved me time, as I had to revise my program multiple times due to unclear high-level flow.

* Addresses how changes in the number of packages directly affect the time needed to complete the look-up function. K1B: The explanation accurately addresses how changes in the number of packages directly affect the data structure space usage.
  + The lookup/search function for look up packages has a time complexity of 0(1) on an average but can be as high as O(n) as the worst case scenario. It results 0(1) when it uses hash function to find the bucket in which is a key-value pair. Usually, the number of items in the bucket is small, so it is fast to search. However, if all items are stored in the same bucket, then the time complexity would be the max size of items which is the worst case scenario O(N).
  + The space complexity is 0(1) as it just require a constant amount of memory regardless of the number of key-value pair in the bucket. The variable, bucket, in the search function (see Hashtable.py, line# 33) stores reference to the bucket list and a single variable kv int the for loop stores the current key-value pair.
* The submission identifies 2 data structures other than the one used in part D that could meet the requirements in the scenario. K2A: The submission accurately describes attributes of each data structure identified in part K2, and it accurately compares these attributes to the attributes of the data structure used in the solution.
  + List data structure is used throughout the program. Such as packages in *truck1, truck2* and *truck2* are defined as list; they store a list of package objects. (See truck.py, line#2, 3, 4)
  + Dictionary data structure is used to store addresses and its distance (attribute: *address\_with\_distance*). The keys are each location, and the value for each key is a list of locations connected to that specific location. (See graph.py).
* The submission includes in-text citations for sources that are properly quoted, paraphrased, or summarized and a reference list that accurately identifies the author, date, title, and source location as available.

The reference for the HashTable.py is from ‘Hash Table class using Chaining’ in C950 ZyBook Chapter 9. The ZyBook is provided by WGU.