# Nearest Neighbor Algorithm Overview

## **A.** Nearest Neighbor

## **B.** Algorithm Overview:

The nearest neighbor algorithm follows these steps:

1. Mark all vertices as unvisited
2. Select a starting vertex, mark it as visited
3. Find the shortest edge connecting the current vertex and an unvisited vertex
4. Set the nearest vertex as the current vertex and mark it as visited \*
5. Repeat steps 3 and 4 until all vertices are visited

In our case, the vertices are our packages, and visiting them means delivering them. In order to meet the requirements of the project, the algorithm requires some modifications to prioritize packages that have delivery deadlines or other special conditions. Specifically, it modifies step 4 such that if there is a package with priority which is one of the three nearest packages, it will select that package instead of simply the nearest one. Additionally, under certain conditions, a priority package must be selected regardless of distance to ensure all instructions and requirements are met.

**B.1.** Pseudocode

Here's the pseudocode for my implementation of the algorithm, with the modifications mentioned:

**For package in package list:**

**Add package to available list if available**

**Parse package info, make list of priority packages**

**While length of available packages > 0**

**Select start package from available list**

**Remove start package from available list**

**For package in available list:**

**Set shortest distance variable**

**Get distance between current location and package**

**If distance == 0:**

**Return package**

**If distance is less than shortest distance:**

**Shortest distance = distance**

**Update nearest three list**

**If package is in priority list:**

**Every third package, return from priority list**

**Check nearest three:**

**If any in nearest three list are in priority list return it**

**Return shortest distance**

## **B.2.** Programming environment:

IDE: PyCharm Community Edition 2021.3.3

Python version: 3.10

Operating System: Windows 10

Processor: Intel Core i7-3770 CPU @ 3.40GHz

RAM: 8 GB

System type: 64-bit operating system, x64-based processor

## **B.3.** Space-Time complexity in Big-O notation:

See the code for details: all major segments throughout the code have the Big-O value declared in the comments, including at the start of files (for space complexity) and for each function (for time-complexity). The following is a summary including the space-time complexity for each file, the hash map, and the Nearest Neighbor algorithm.

Overall space-complexity: O(n)

Overall time-complexity: O(n^2)

Main file space-complexity: O(n)

Main file time-complexity: O(n)

CSV reader file space-complexity: O(n)

CSV reader file time-complexity: O(n)

Schedule file space-complexity: O(n)

Schedule file time-complexity: O(n^2)

Hash file space-complexity: O(n)

Hash file time-complexity: O(n)

Hash map insertion, lookup, and deletion functions time-complexity: O(n)

Hash map space-complexity: O(n)

Nearest Neighbor algorithm time-complexity: O(n)

## **B.4.** Scalability:

Overall my solution is pretty scalable, since I attempted to use relative lengths and sizes when possible, and avoided hard-coding if I could. This means that if, for example, there are more than 40 packages, I won’t have to re-write the code, since I referenced the number of packages, not the number 40, in cases where that was necessary. A few adjustments would need to be made as the number of packages increases, but overall it would be relatively minor adjustments.

In terms of hardware scalability, as the number of packages increases, the space needed would increase proportionally (O(n)), while the processing speed for insertion and search functions should remain constant, since the buckets in the hash map should be evenly distributed, which should result in O(1) for time complexity. So overall, in terms of hardware, if we increased the number of packages, more storage space would be needed, but the computational power needed would remain basically the same.

## **B.5.** Maintainability and efficiency:

The code is efficient because I opted to use a relatively efficient algorithm (Nearest Neighbor), and once the main sorting algorithm is completed, very little else needs to happen. In terms of maintainability, I made sure to comment frequently, especially on loops and functions, so it should be easy to go back later and understand what’s going on.

## **B.6.** Self-adjusting data structures:

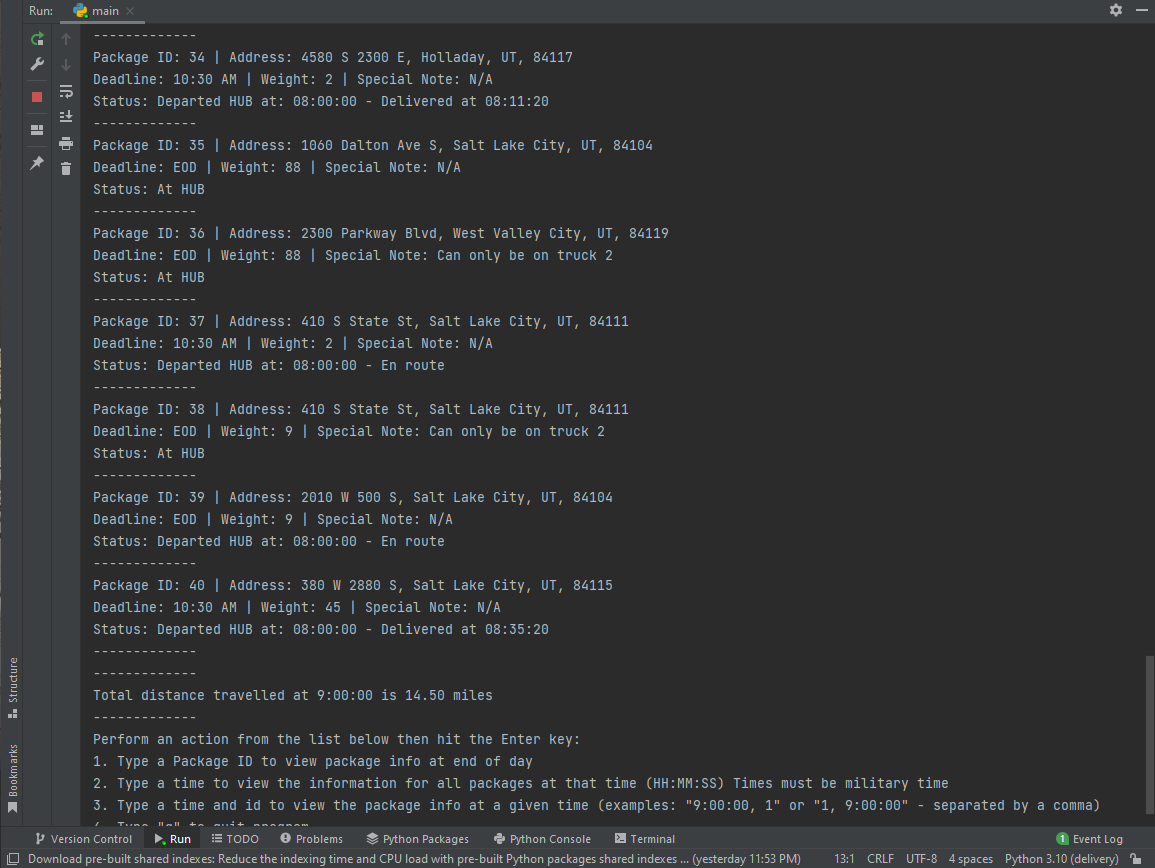
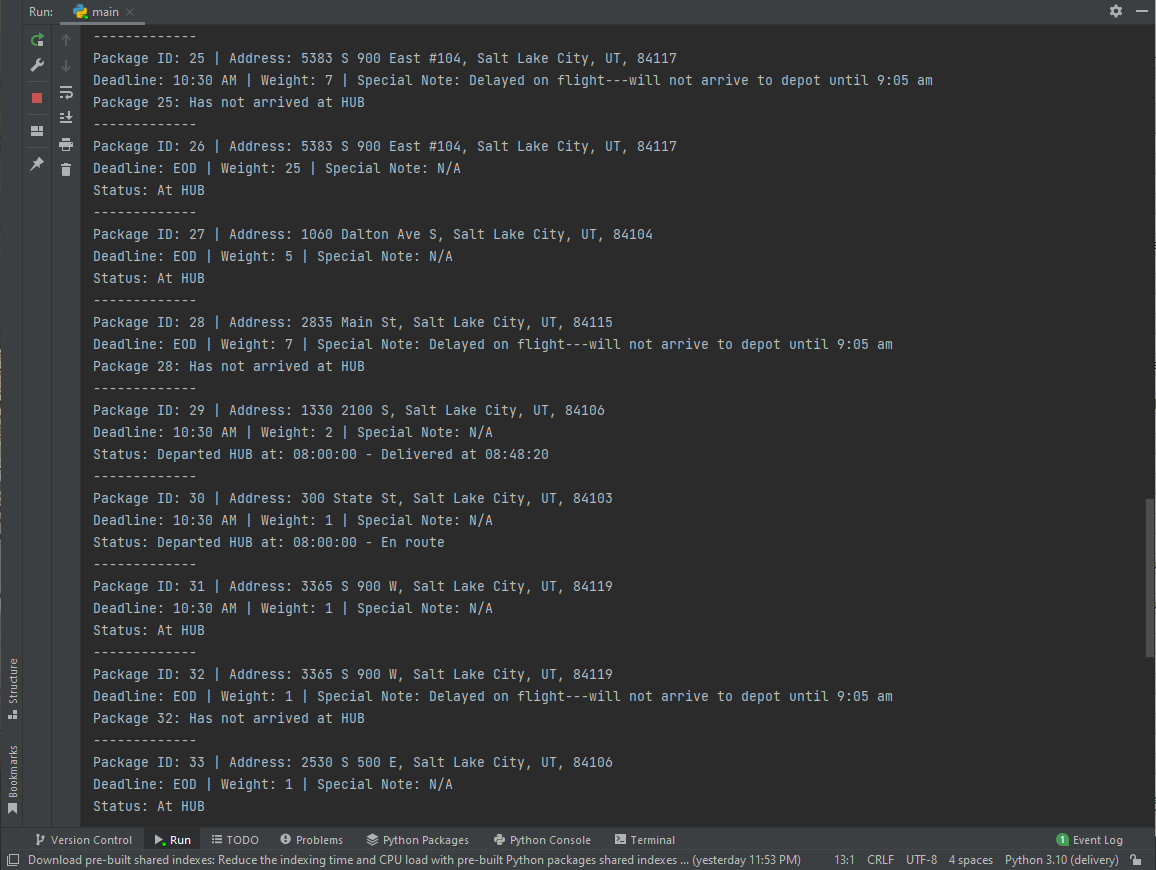
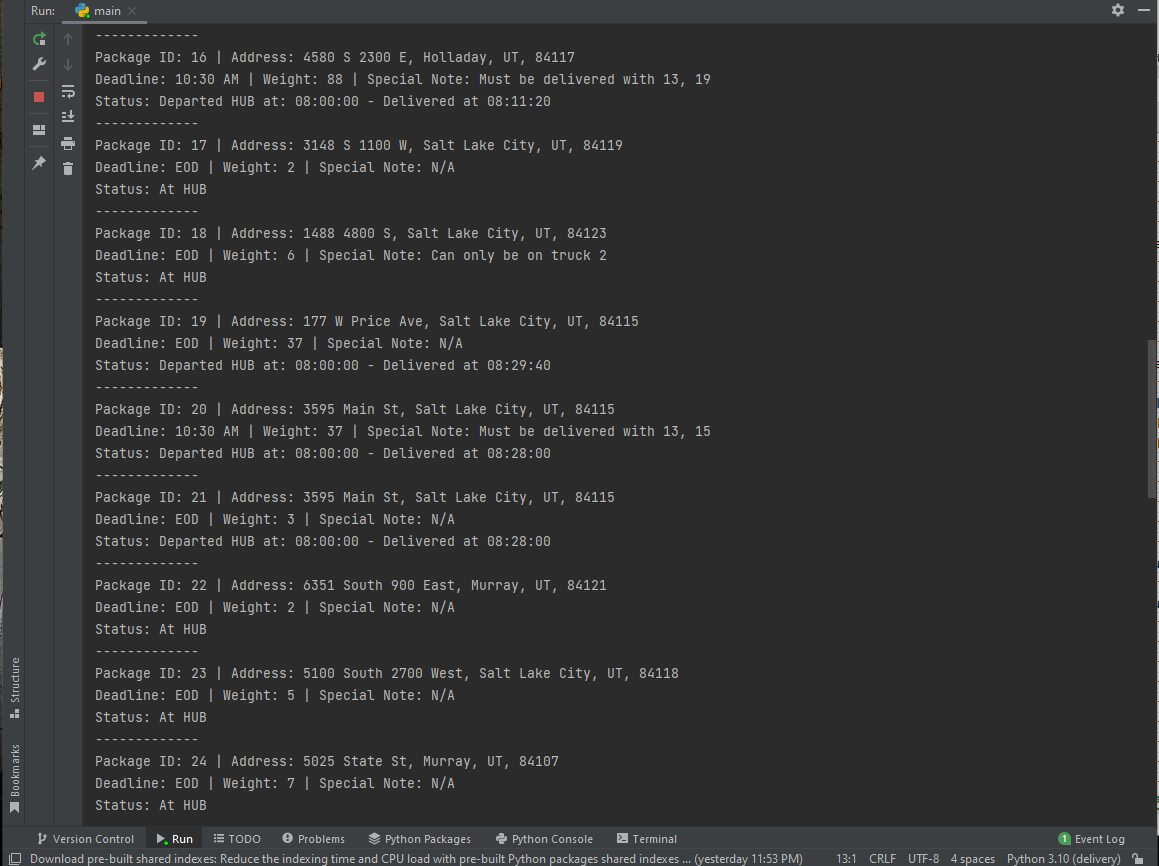
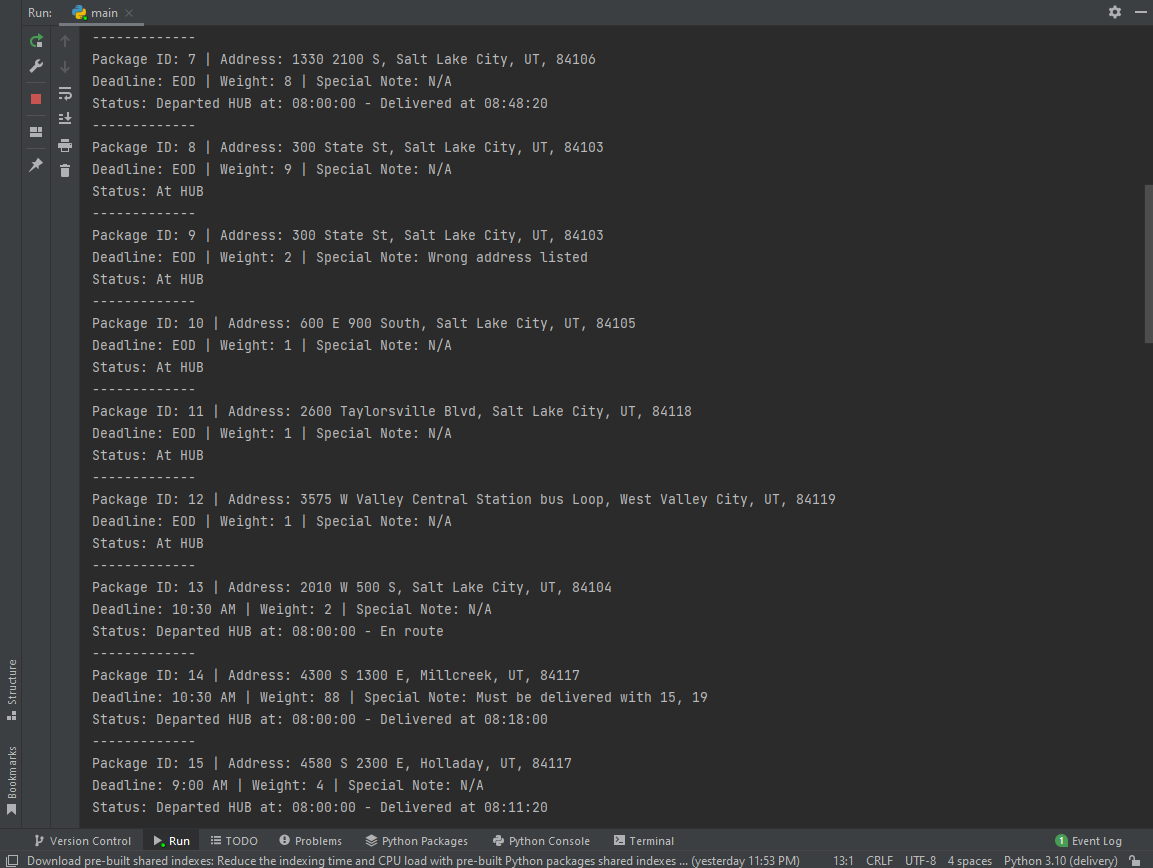
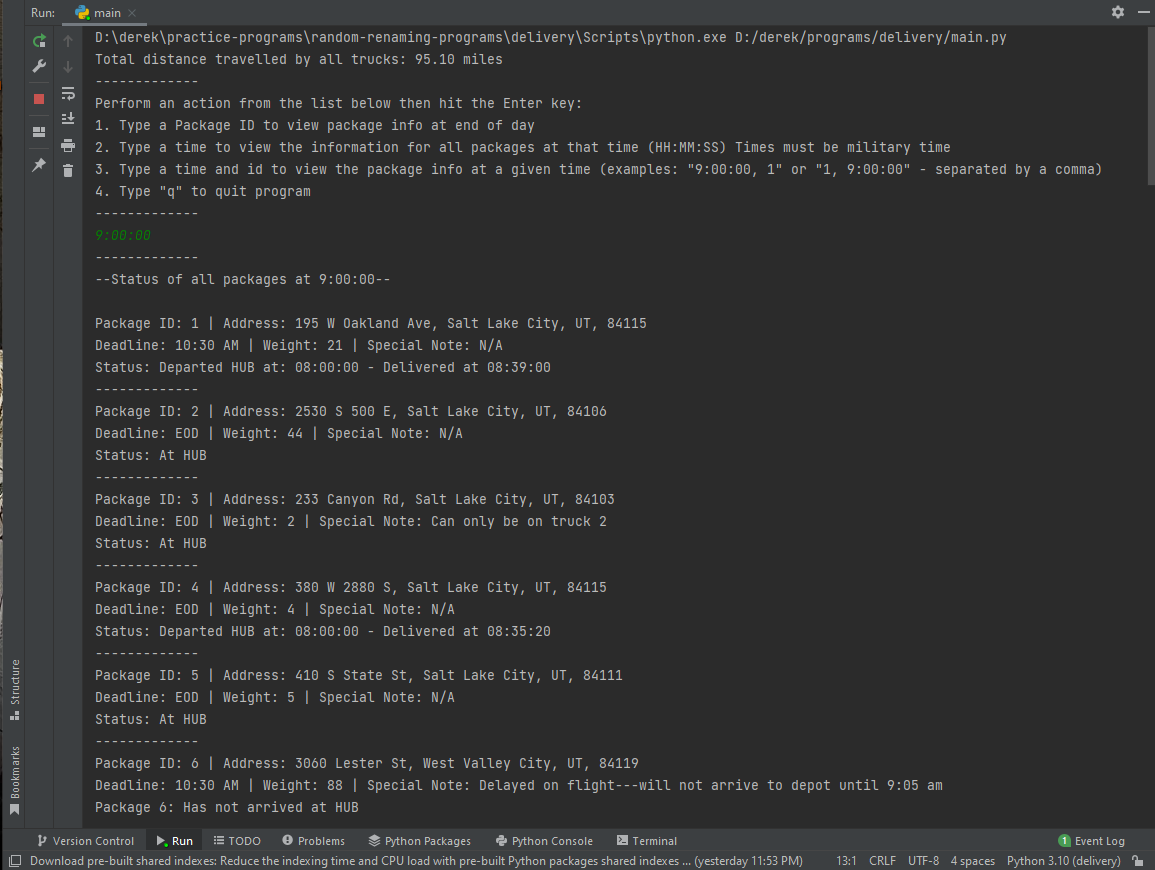
Generally, one of the main strengths to most self-adjusting data structures is the efficiency of performing operations even with a very large dataset. Typically these are more efficient than non-self-adjusting data structures. For example, insertion into an array vs insertion into a hash table would, on average result in the hash table being much more efficient, and this is true of most operations with a hash table compared to an array. In this comparison, the worst cases would be the same (O(n)), but on average, the hash table would be much more efficient at O(1) compared to the array which is still O(n).

One of the weaknesses of self-adjusting data structures is that they are not as useful for small datasets when the difference between O(n) and O(1) would essentially be negligible. In that case a simpler data structure would likely work just as well and may be easier to implement and use.

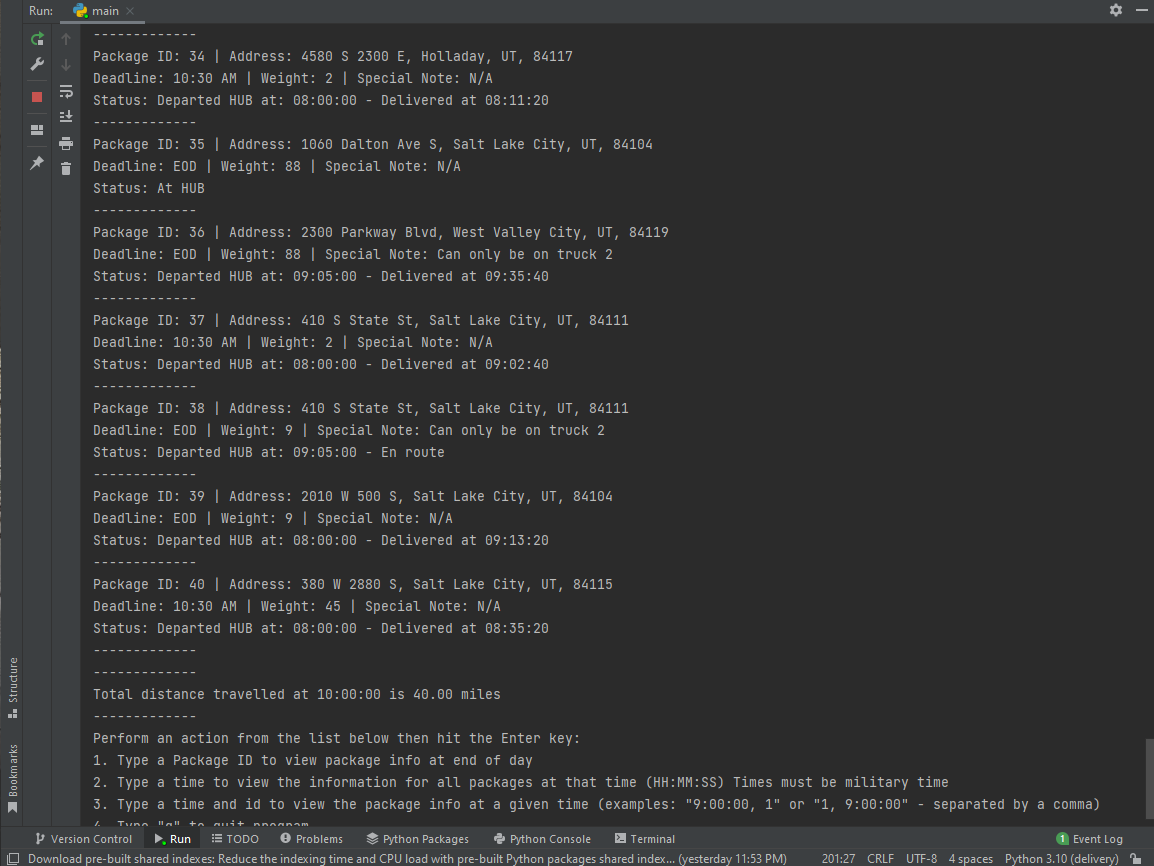
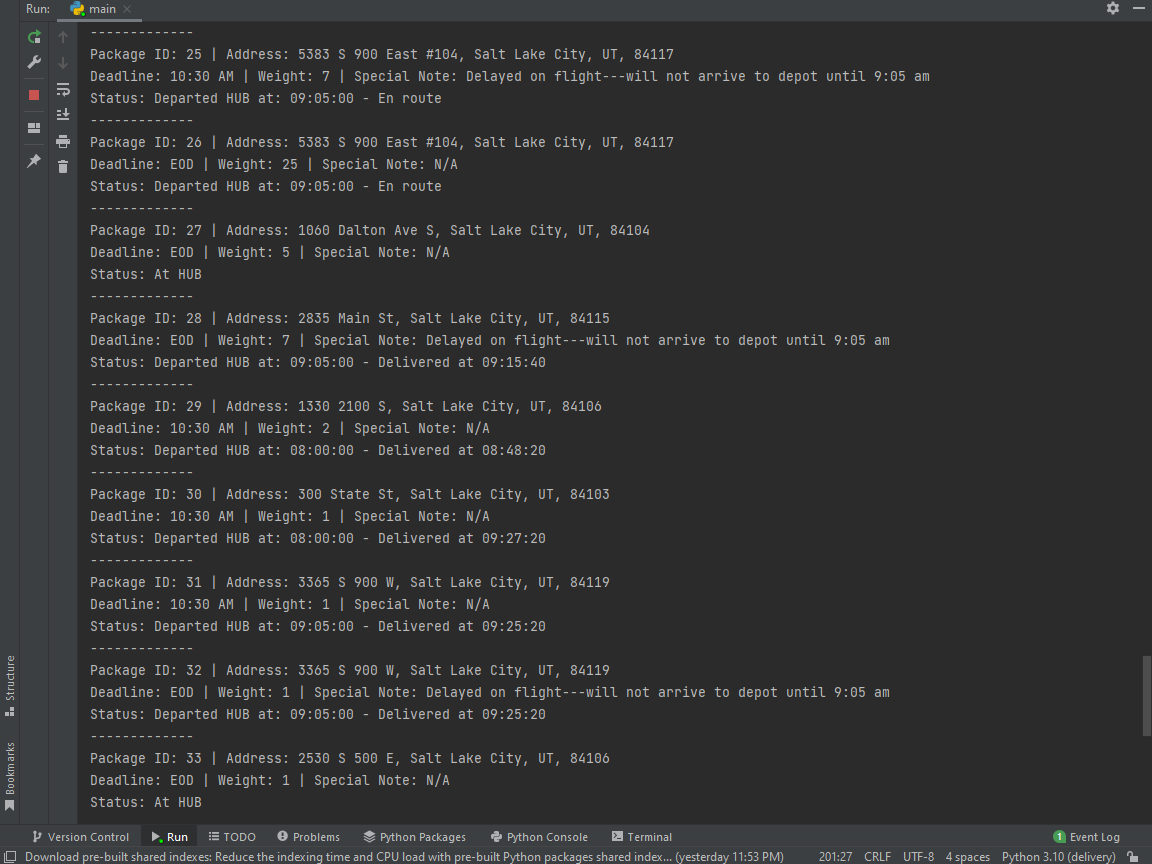
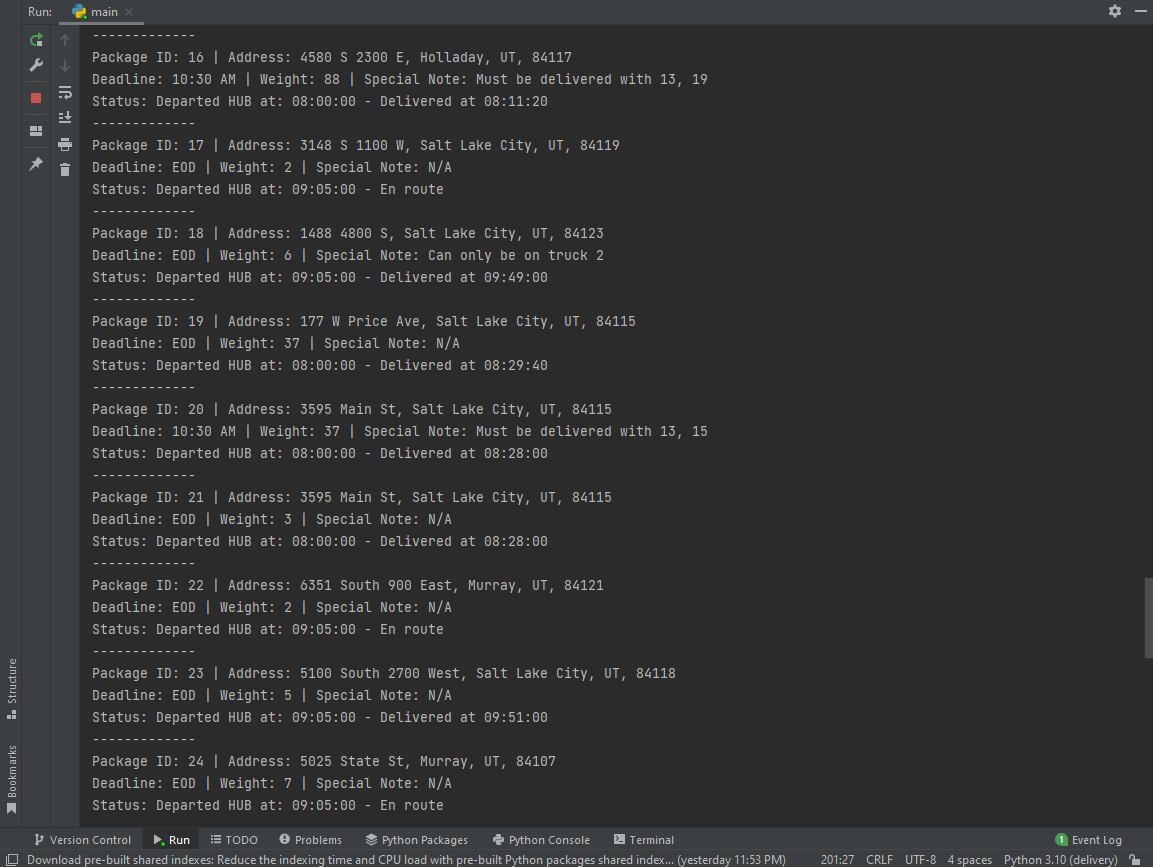
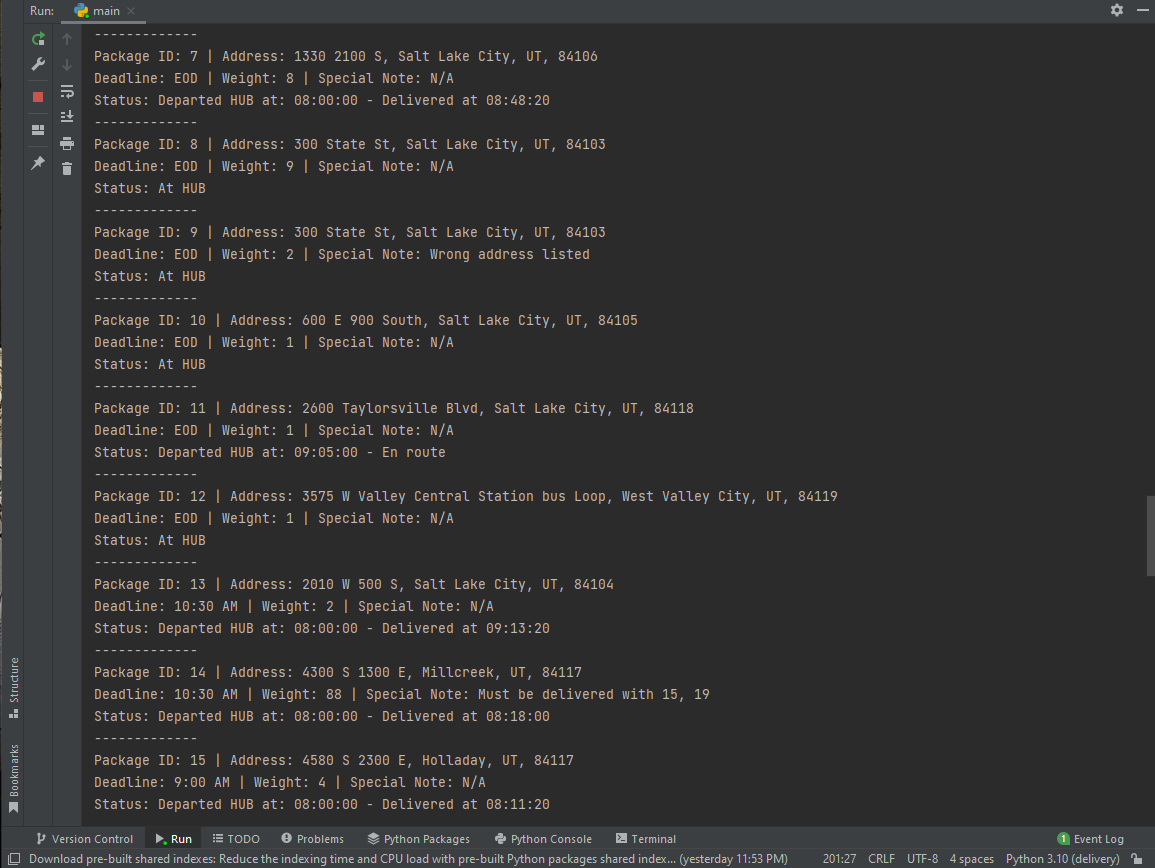
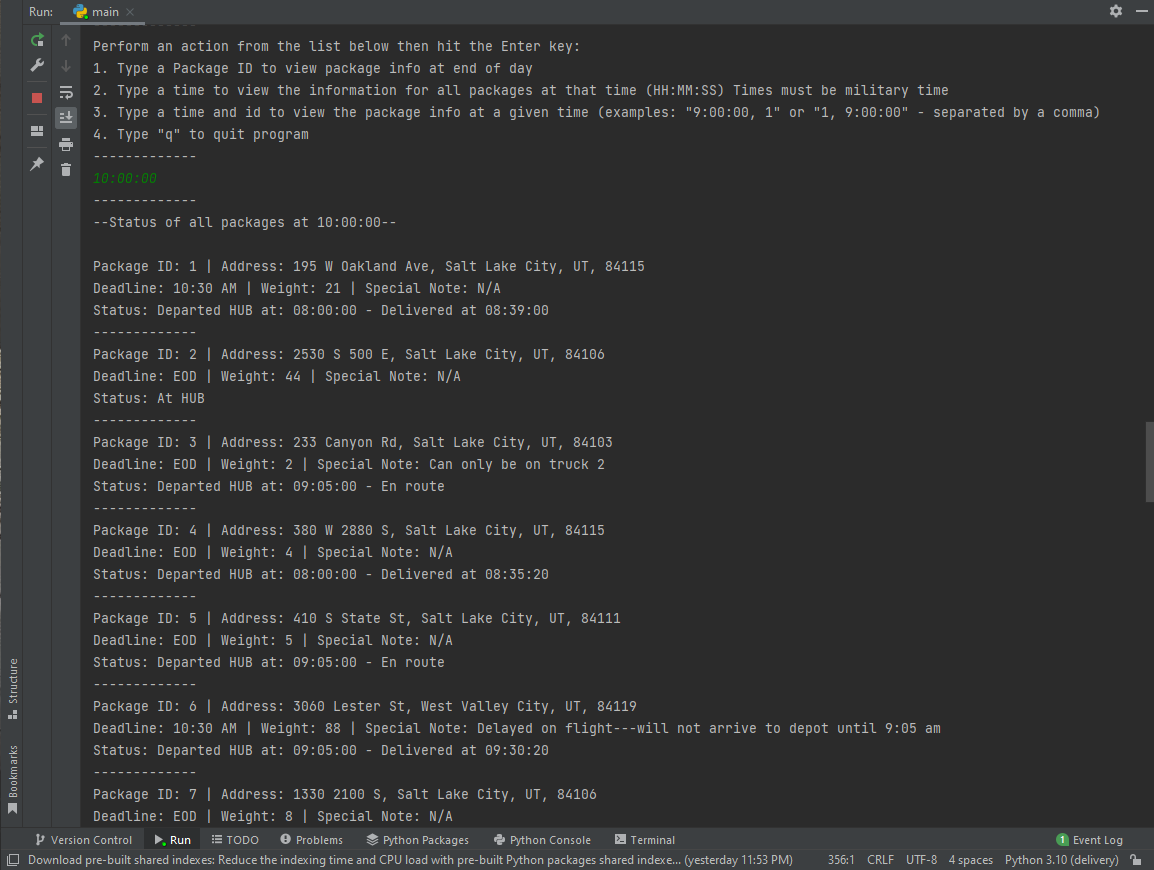
## **D.** Hash Map

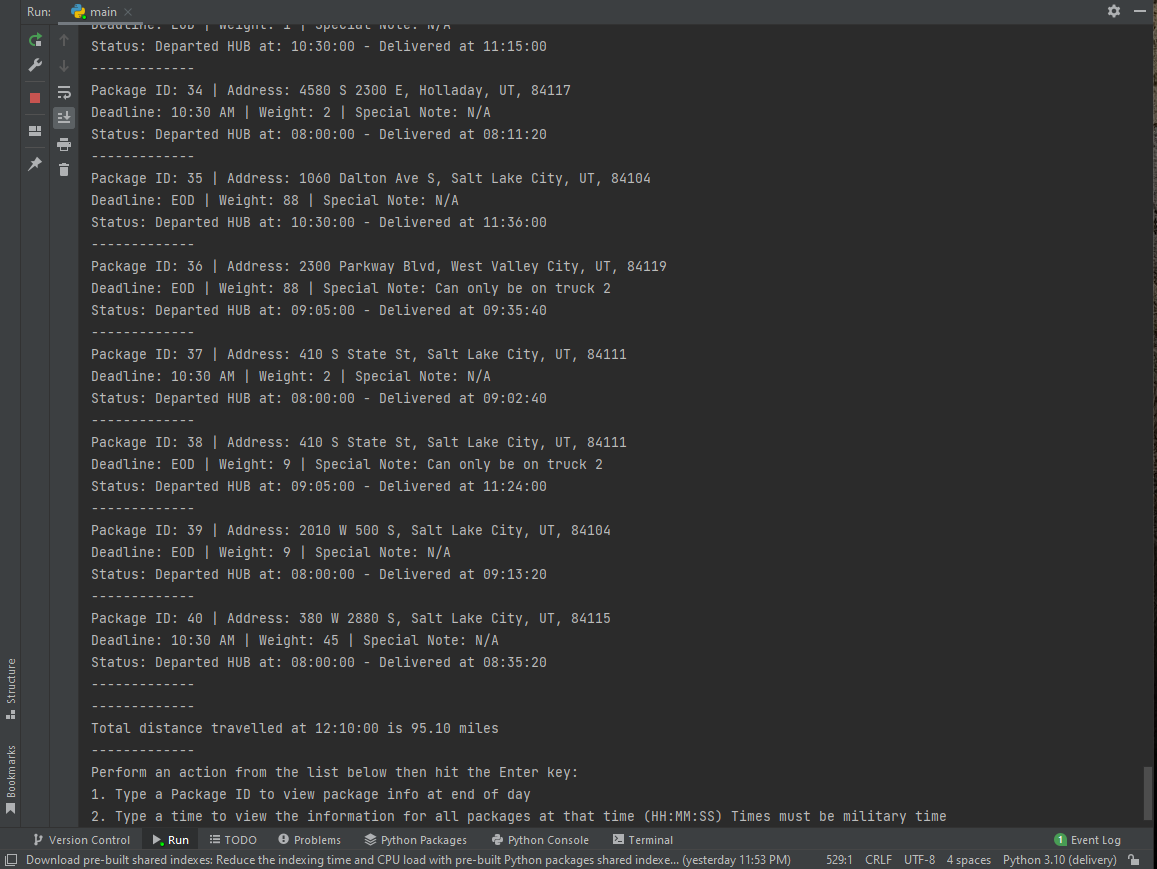
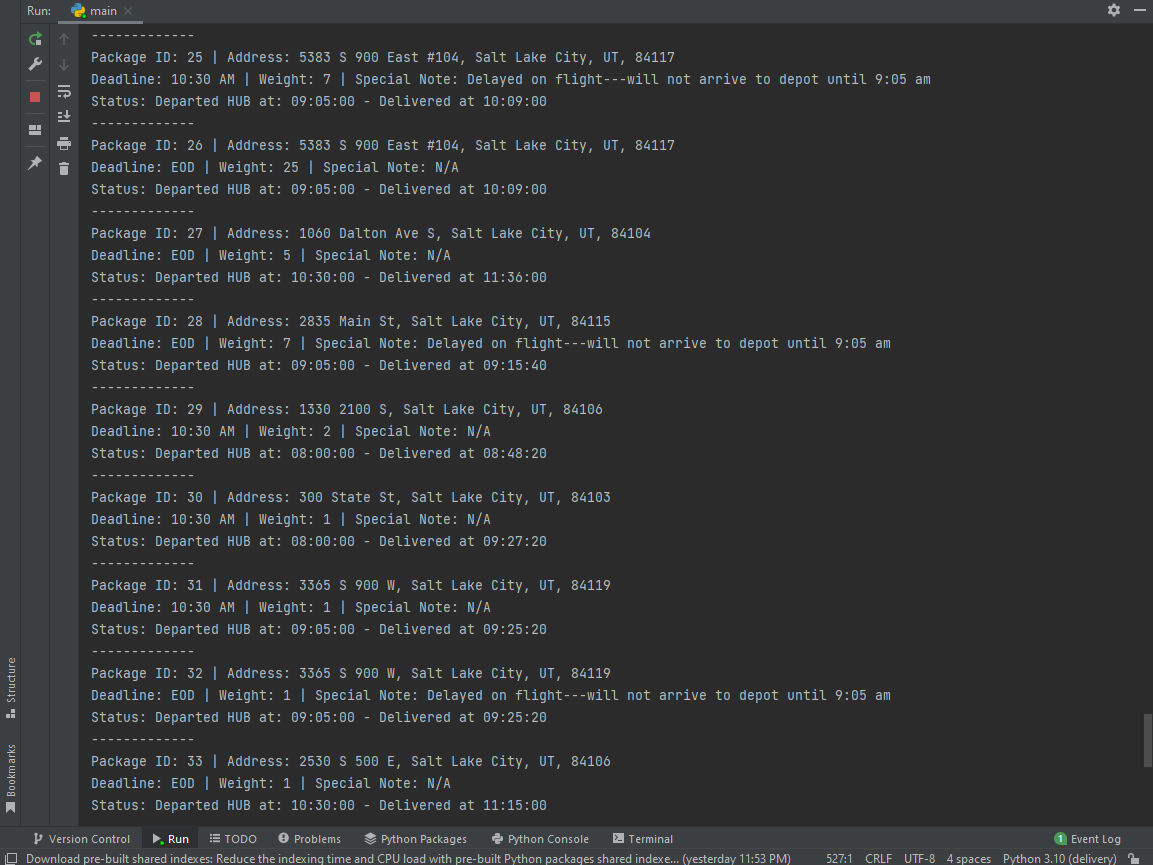
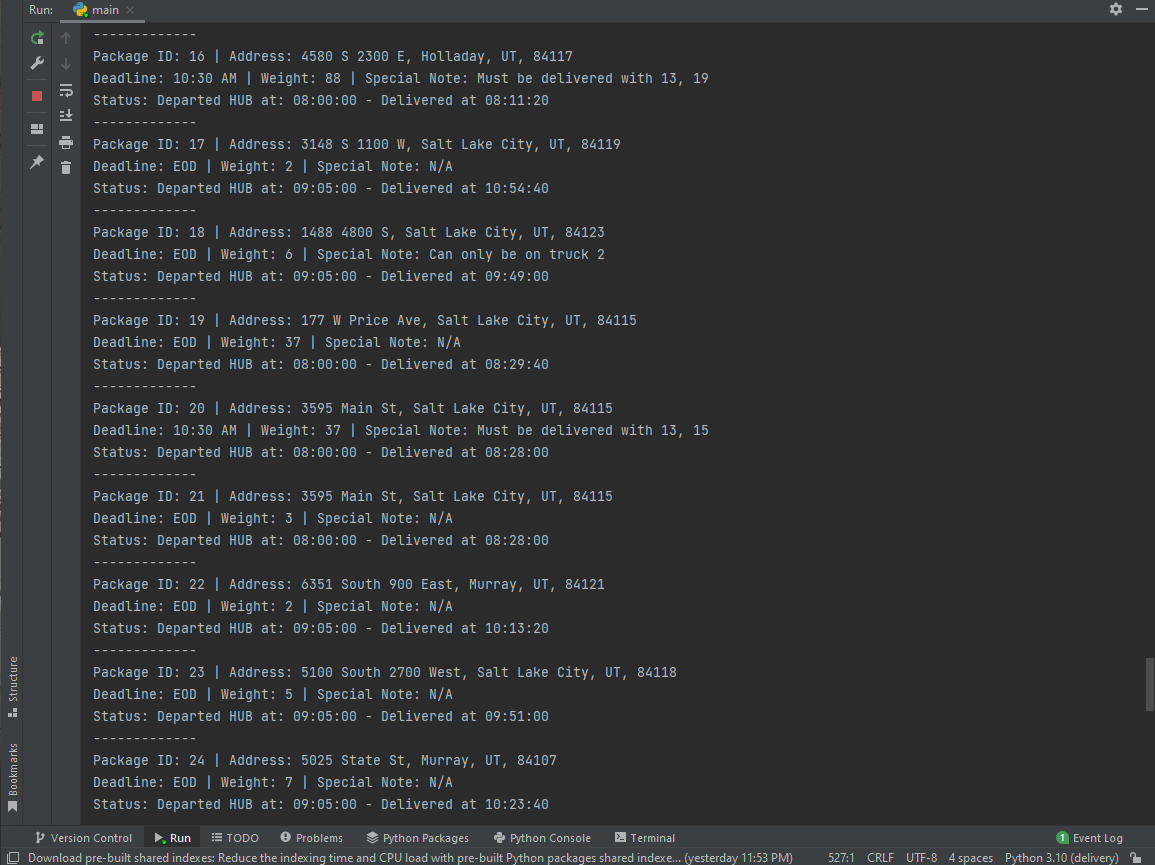
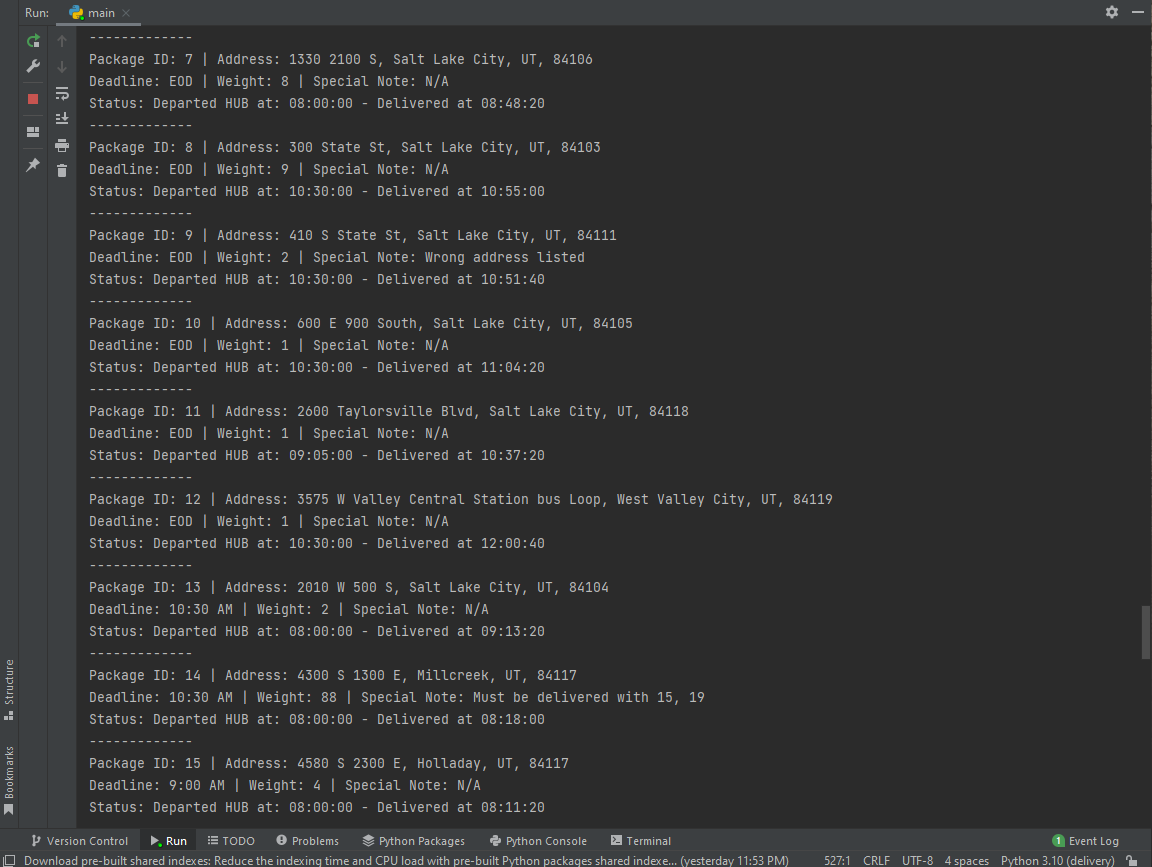
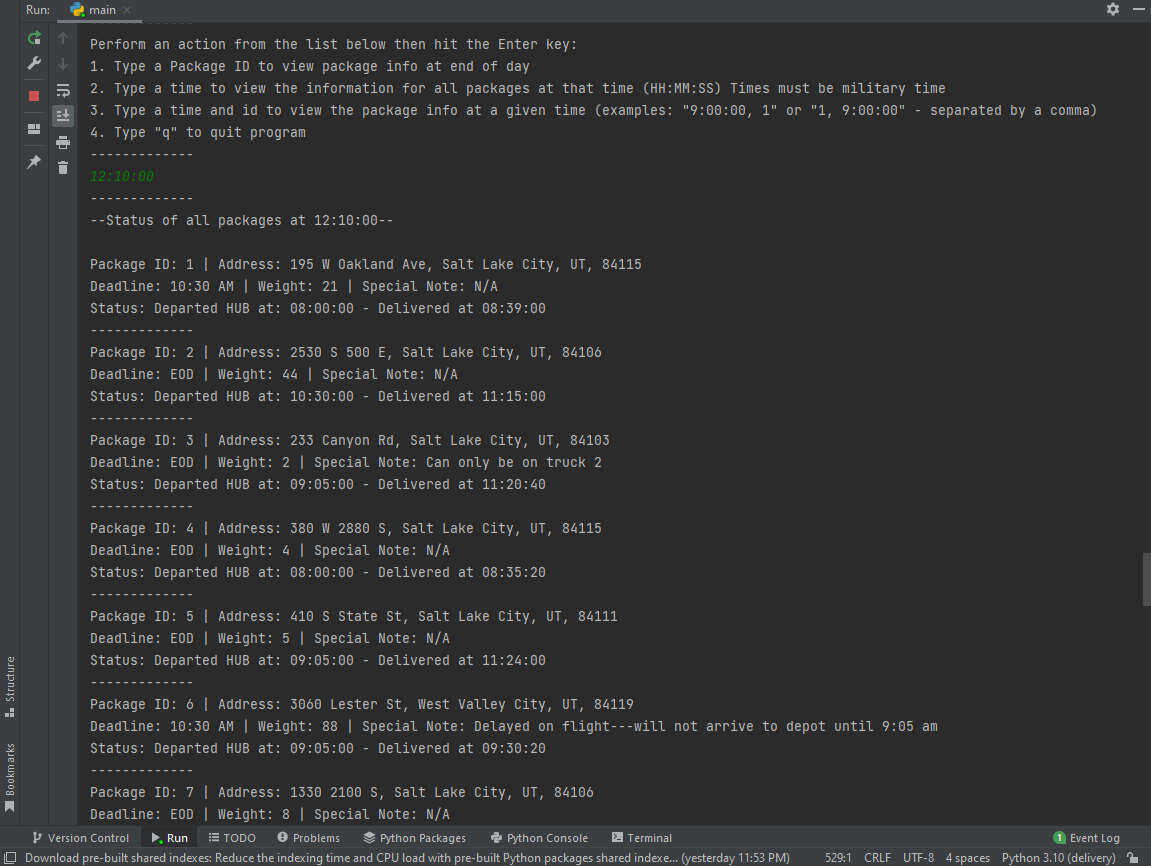
I used a hash map to store the package information. This is accomplished using buckets. In this case, there are 10 buckets. Each package and all its information is stored in a bucket in an array format. The bucket is decided by using a key and a hash function to determine which bucket a package should go into. The key for each bucket is its package ID, and the hash is 10 (the number of buckets). Since the package IDs are sequential, the buckets should each contain 4 packages. In this way, all the information for a specific package is stored with the ID in its respective bucket, so each data point can be accessed easily by plugging the package ID into the lookup function.

## **G. Screenshots of the status of all packages at given times:**

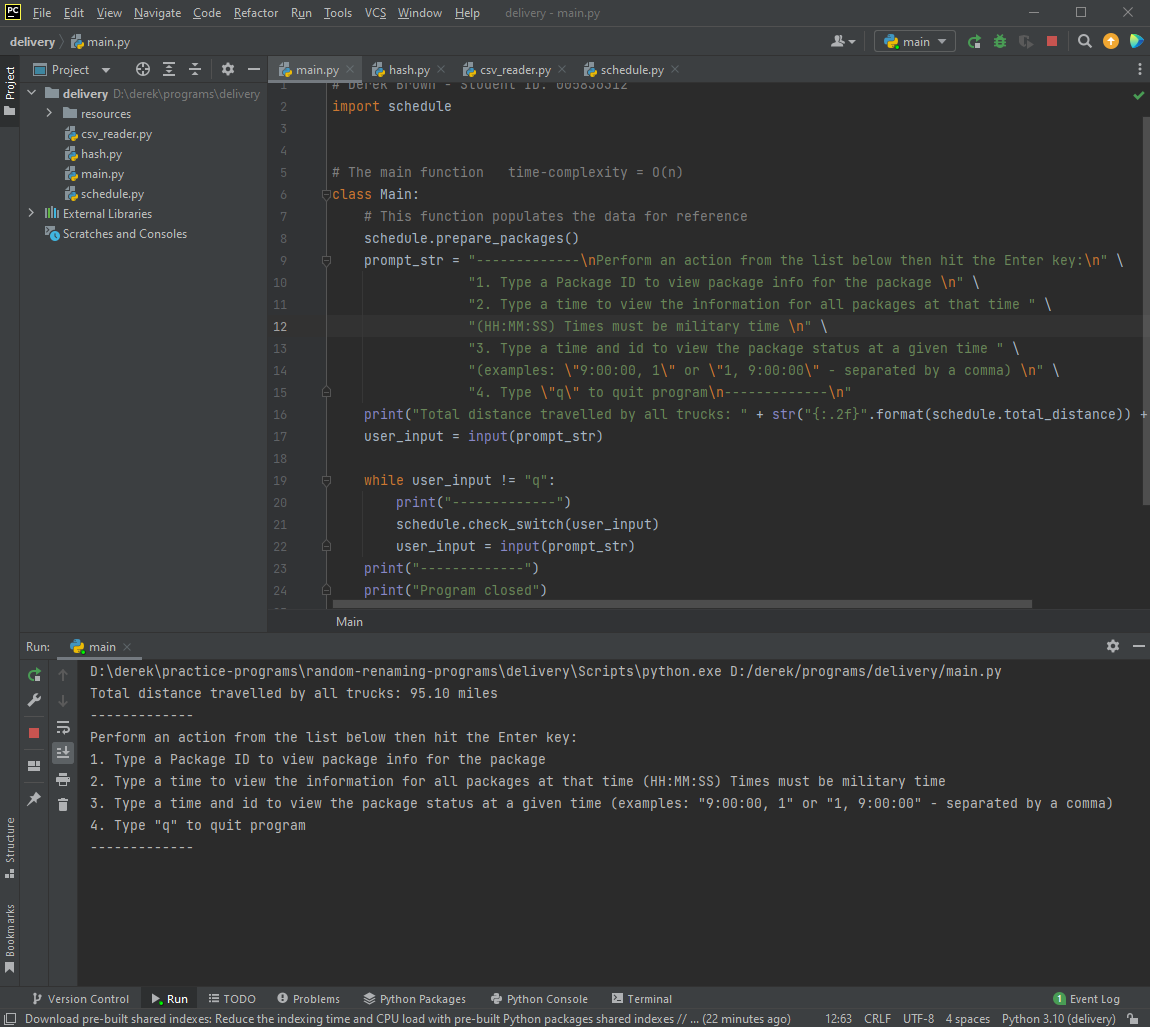
**9:00:00 screenshots:**

**10:00:00 screenshots:**



**12:10:00 screenshots**

## **H. Screenshot showing successful completion of code, free from runtime errors or warnings, including the total mileage traveled by all trucks:**



## **I.** Justification for my choice of algorithm – Nearest Neighbor

**I.1.** This algorithm’s big strengths are that it is efficient, and it usually gets a pretty good path. In this scenario, both of these apply. In addition, it is relatively easy to implement. However, it does not ensure that the path chosen is the shortest.

**I.2.** In my case, to meet the requirements in this scenario, I needed to add some modifications when selecting the next package, so it’s not always the nearest package which is selected. That said, with some modifications I was able to meet all the requirements. The total distance needed was 95.10 miles and all packages were delivered on time.

**I.3.** Other algorithms that could have been used include the Dijkstra’s shortest path, and nearest insertion. The shortest path algorithm is a very good algorithm for finding the best path, but it comes at the cost of efficiency. It checks every path possible and selects the shortest one, whereas the nearest neighbor algorithm just checks the next package and returns the shortest. This means it has to loop through the list multiple times and has a time complexity of O(n^2) which becomes exponentially slower with larger datasets. In this case, I opted for the nearest neighbor since it is more efficient and gets a pretty good route. The nearest insertion could be a viable alternative, and works similarly to the nearest neighbor algorithm, but instead of adding the next nearest package at the end of the list, you would insert it in between two packages already on the list. This approach would be a bit more difficult to meet all the requirements, however, and would require a different approach overall. It would require a lot more attention to ensure that special notes are taken into account, but it could work. However, the nearest neighbor algorithm is much simpler to implement and would do about as well, which makes it the better option.

## **J.** Things I would do differently if I did this again

One thing I would be sure to do is organize my data structures a bit better. In some cases, my code is a bit messy because I used a few different lists with corresponding elements. I would make sure to organize it a bit better to ensure better maintainability and ease of use. I may also split up the files differently. In my implementation, one file is about 350 lines of code, the rest are about 50 lines, which means I could probably do a better job organizing my code by splitting up the large file, which would also make it easier to maintain.

## **K.** Justification for my choice of data structure – Hash map

**K.1** The data structure used meets all requirements in the scenario.

**K.1.a.** The time to complete the look-up function is really not affected by changes in the number of packages to be delivered. It remains at an average of O(1) regardless of the number of packages to be delivered, with the worst case being O(n).

**K.1.b.** Space usage is only slightly affected by the number of packages to be delivered, since in my implementation, I do not delete the packages from the hash map. Instead, I append the times of departure and delivery to the bucket containing the corresponding package, which allows for lookup later.

**K.1.c.** Increasing the number of trucks or cities would not impact the lookup times for the hash map, since that information is not stored in the hash map in my solution. It would have an effect on disk space usage, which would increase as the number of trucks or cities increases. Assuming each city has the same number of trucks and packages, the space complexity would essentially be O(n), meaning that as we increase the number of cities and trucks, the amount of disk space needed would increase linearly.

**K.2** Two alternative data structures that could have been utilized are a BST tree and an AVL tree. In a BST tree, each node has up to two child nodes. You essentially just add one node at a time. Each time a node is added, you compare it to the root node, if it is smaller, it goes to the left, if it’s larger, it goes to the right. If there’s a node there, compare it and do the same process again until the new node finds an empty spot. Once it does, that’s its new home. Essentially, what this creates is a data structure which is more efficient when performing operations compared to a simpler data structure. In the worst case scenario, the BST tree would have the same time complexity as a typical array – O(n). On average, however, it would be closer to O(logn) which is pretty good. A BST tree could be used to store packages with their package information and would be able to meet all the requirements of the scenario.

An AVL tree is another option that would work. It functions like a BST tree, but it has a few additional steps. In an AVL tree, when a node is added or removed, it completely rebalances the tree if the difference between subtrees is greater than 1, in which case a rotation is made and each node moves in order to rebalance the tree. The big advantage to this type of tree is that it is very efficient in any operations, with the max number of comparisons being equal to the height of the tree. This makes it much more efficient than a lot of other data structures, but takes a few more steps to implement than a BST tree. It would also be a good alternative for the scenario.

In this case, the Hash Map was chosen because it is easier to implement than the AVL tree, since rotations are not necessary, but it performs better in most cases than a BST. In the worst case, it would perform the same as the worst case BST, however, since we know the package IDs are sequential, we know that the Hash Map should be evenly distributed, and hence it will be a better choice than the BST, which is less likely to be unless we made some specific modifications. An AVL tree could perform about as well as the Hash Map, but it takes a bit more to implement and would perform a bit less efficiently due to having to rotate and rebalance.

## L. Sources

No sources were cited in this overview.