DSA2 PA

Nearest Nieghbor

George Kent Allen

Contents

A

The self adjusting algorithm used is Nearest Neighbor.

B

B.1

This implementation of the Nearest Neighbor Algorithm has the following steps.

```
1. Consider all vertecies 'unvisited'.

2. Pick a starting vertex and consider it 'visted'. In this code, it is WGU's location.

3. Compare all edges which lead to the remianing 'unvisited' neighboring verticies.

4. Traverse the edge with the least distance. Mark the destination vertex 'visited'.

5. Repeat Steps 3 and 4 untill all vertecies are visited.
```

After all vertices are visited, the path one took, or rather the order vertices are marked ‘visited‘, is the desired path. It may not be the shortest path possible. This can be represented in pseudo code but some data structures need to be explained. In this code, ‘Vertices‘ is either a List or, set of all vertices. An adjacency matrix is a 2 dimensional array where a cell at any given index pair contains the distance of an edge between vertices.

For Illustration →

M	0	1	2
0	0→0	0→ 1	0→2
1	1→0	1→ 1	1→2
2	2→0	2→ 1	2→2

If all cells are occupied in the matrix, the number of edges is $V \cdot V$ or V^2. E. A matrix has ‘Row‘ By ‘Column‘ cells. In an undirected graph a→b is the same as b→ a, halving the maximum number of edges, and consequently cells, that are needed to store a matrix. This reduction does not change spacial complexity because in Big O the following two expressions are equivalent as shown below.

$$O(\frac{V^2}{2}) \equiv O(V^2)$$

```
function nearest_neighbor(verticies, starting_vertex,  matrix)

    for vertex in verticies
        vertex belongs to unvisited
    path = [] # An empty List

    current_vertex = starting_vertex
    unvisited.remove(starting_vertex)

    while univsited != emptyset:
        minimum = infinity
        minimum_index = null
        for index, edge_value in matrix[current_vertex]
            if index not in unvisited or index == current_vertex
                continue

            if min > edge_value
                current_minimim_index = index
                min = edge_value

        unvisited.remove(minimum_index)
        path.append(minimim_index)

    return path
```

B.2

The environment used was PyCharm Community edition with a virtual environment running python 3.10. Python 3.11 was also testing on Ubuntu.

Here is the directory of the source code.

```
├── location.py
├── main.py
├── map_reader.py
├── package_hasher.py
├── package.py
├── package_reader.py
├── simulation.py
├── planner.py
└── resources
    ├── rawdistance.csv
    └── rawpackage.csv
```

B.3 - Space and Time Complexity

location.py

Let's start in the location.py. Location.py contains a Class that represents the graph of all locations and their edges. It also contains many high level functions and types to conveniently store and access vertices and their edges. Let's start from line 1.

```
1 from decimal import *
2
```

location.py only needs one type form the standard library. Decimal is used to store the value of each edge. Floating point is not used because when converting from cvs to a numeric value you would loose information.

Line 3 to 109 define the class LocationGraph. It is declared with its construction like so

```
class LocationGraph:
    def __init__(self):
        self.next_address_index = 0
        self.locations = dict()
        self.travel_times = list(list()) #0,0 exists
```

This constructor takes in no input and only initializes empty variables. It's space and time complexity is therefore $O(1)$.

Now we take a look at the next function in the Class.

```
9      def add_location(self, address):
10         self.locations[address] = self.next_address_index #  Constant O(1)
11         self.next_address_index = self.next_address_index + 1  # Constant O(1)
12         next_row = list() # Constant O(1)
13         for cell in range(0,len(self.travel_times)): # At most O(V)
14             next_row.append(None) #  Constant O(1)
15         self.travel_times.append(next_row) #  Constant(1)
```

The loop on line 13, can only run for the length of travel-times. Travel times is the outer list in our matrix. The only object that grows with our input is next-row which has a length equivalent to the length of our loop $O(V)$. Its length is always the carnality of vertices in our graph. Adding the time complexity of each line gives.

$$O(V) + O(1) + O(1) + ...$$

So the space and time complexity is.

$$\equiv O(V)$$

For future explanations, lines that are not given explicit complexity are implied to be constant. All complexity's of a lower power will be omitted from the final complexity of a function.

This function sets an edge between two vertices.

```
1     def set_travel_time(self, address_one, address_two, travel_time):
2
3         #  You set an edge to a vertecies that don't exists
4         if address_two not in self.locations:
5             raise Exception("Address One of value", address_one, " is not in locations")
6         if address_one not in self.locations:
7             raise Exception("Address Two of value", address_two, " is not in locations")
8
9         #  'Flip' matrix horizontally to fight redundancy
10        index_one = min(self.locations[address_two], self.locations[address_one])
11        index_two = max(self.locations[address_two], self.locations[address_one])
12
13        #   Implicit distance between a vertex and itself is 0.0
14        if index_one == index_two:
15            self.travel_times[index_two][index_one] = Decimal("0.0")
16
17        self.travel_times[index_two][index_one] = Decimal(travel_time) #Write to matrix
```

For lines 3 to 7, Checking the existence of a value in a dictionary is $O(1)$. Lines 10 to 11 both access a dictionary, $O(1)$. After that is done, the values are put in a both a min and max function. Taking the Minimum and Maximum of unsorted values is $O(N)$, where N is the number of values to be considered. But we know in all cases of this function, $N = 2$, so lines 11-10 are constant. Lines 14,15 are constant due to previously explained principles. The last line is just a write to a matrix which takes constant time.

All variables initialized in this function, index-one and index-two, do not scale with inputs given them a space complexity of $O(1)$

In conclusion the space and time complexity of this function is.

$$O(1)$$

This function takes either an integer location, or a location by it's string address and always returns the former.

```
    def check_and_get_address(self, address) -> int:
        if type(address) is int:
            return address
        elif type(address) is str:
            return self.locations[address]
        else:
            raise Exception("A address must be either int or str.")
```

The run time of this function is not dependent on its inputs. It is just simple comparison and sometimes a dictionary lookup.

The simplified space and time complexity is

$$O(1)$$

. The function takes constant space and time.

The next funtction is.

```
1     def get_travel_time(self, address_one, address_two):
2         safe_access = self.check_and_get_address
3         index_one = min(safe_access(address_one), safe_access(address_two))
4         index_two = max(safe_access(address_one), safe_access(address_two))
5         if index_two is index_one:
6             return Decimal('0.0')
7         return self.travel_times[index_two][index_one]
```

lines 3-4, call a previously explained function, which was shown to have runtime $O(1)$. Min/Max were explained to have constant runtime. The branch and matrix lookup are constant. There are no objects that grow with the inputs.

The final space and time complexity is

$$O(1)$$

The next section of LoactionGraph is.

```
class _IndexIterator:
    def __init__(self, parent, id_code: int):
        self.index = 0
        self.id_code = id_code
        self.parent = parent

    def __iter__(self):
        return self

    def __next__(self):
        value = None
        index_one = None
        index_two = None
        if(self.index == self.id_code):
            self.index = self.index + 1
            return [self.id_code, self.id_code]

        if self.index > (len(self.parent.travel_times) - 1):
            raise StopIteration

        if self.index >= self.id_code:
            index_one = self.index
            index_two = self.id_code
            #value = self.parent.travel_times[self.index][self.id_code]
        else:
            index_one = self.id_code
            index_two = self.index
            #value = self.parent.travel_times[self.id_code][self.index]
        self.index = self.index + 1

        return [index_one, index_two]

def IndexIterator(self, id_code: int):
    return self._IndexIterator(self, id_code)    #    return self.travel_times[index_two][index_one
```

This class, nested in LocationGraph, is an iterator and has an accompanying function to return an instance of itself. The constructor only initializes default values and is therefore constant. When the method 'next()' is called, it takes constant time. For the entire control block of next, there are only branches, reads and writes. It should be noted that although each function in this iterator is constant, or $O(1)$, when traversing through an entire instance of this iterator it takes $O(V)$ time. Notice, that on line 19, the iterator does not stop until the index is greater than travel times, which has length V.

The final space and time complexity of each function is.

$$O(1)$$

.

The run-time of traversing the entire iterator is

$$O(V)$$

Traversing the iterator does not change the length of the iterator and so the size is constant.

The next section of LocationGraph is.

```
    class _AdjacencyIterator:
        def __init__(self, parent, id_code: int):
            self.index = 0
            self.id_code = id_code
            self.parent = parent

        def __iter__(self):
            return self

        def __next__(self):
            value = None
            if(self.index == self.id_code):
                self.index = self.index + 1
                return Decimal('0.0')

            if self.index > (len(self.parent.travel_times) - 1):
                raise StopIteration

            if self.index >= self.id_code:
                value = self.parent.travel_times[self.index][self.id_code]
            else:
                value = self.parent.travel_times[self.id_code][self.index]
            self.index = self.index + 1
            return value

    def AdjacencyIterator(self, id_code: int):
        return self._AdjacencyIterator(self, id_code)
```

Notice that the logic of this Iterator is exactly the same as the previous section, it shares the complexities of the previous section, $O(V)$ & $O(1)$.

package.py

This file takes no imports. It defines the Package object. The first function is the constructor.

```
class Package:
    def __init__(
            self,
            id_code=None,
            address=None,
            city=None,
            state=None,
            zip_code=None,
            deadline=None,
            mass=None,
            note=None,
            fields=None):

        if fields is not None:
            self.id_code = int(fields[0])
            self.address = fields[1]
            self.address = fields[1]
            self.address = self.address.replace("South", "S")
            self.address = self.address.replace("East", "E")
            self.address = self.address.replace("North", "N")
            self.address = self.address.replace("West", "W")
            self.city = fields[2]
            self.state = fields[3]
            self.zip_code = fields[4]
            self.deadline = fields[5]
            self.mass = fields[6]
            note = fields[7]
            if "Wrong" in note:
                self.note = "wrong_address"
            elif "Delayed" in note:
                self.note = "delayed"
            elif "Can" in note:
                self.note = "truck2"
```

```
34             elif "Must" in note:
35                 self.note = "hasfriends"
36                 self.friends = self.all_packages_in_string(note)
37             else:
38                 self.note = None
39         else:
40             self.id_code = id_code
41             self.address = address
42             self.city = city
43             self.state = state
44             self.zip_code = zip_code
45             self.deadline = deadline
46             self.mass = mass
47             self.note = note
48
49         self.status = None
```

The constructor is made up of only branches and writes with the exception of the call to all-packages-in-string(note), see line 36. This is shown in the next page to be $O(1)$. The members of Package do not change with any input. They are static descriptors of that package and all take $O(1)$ space.

The final space and time complexity is

$$O(1)$$

The next function is.

```python
def all_packages_in_string(self, string):
    packages = list()
    number = ""
    for character in str(string + "\n"):
        is_digit = character.isdigit()
        if character == "\n":
            break
        if is_digit:
            number = number + character
        if not is_digit and number != "":
            packages.append(int(number))
            number = ""
    return packages
```

The function is ran for as many characters as are in the special note section of the package data. This does not grow with the number of packages and is essentially constant. Every note is only a handful of characters.

The final space and time complexity is

$$O(1)$$

.

planner.py

This file contains functions for organizing the route taken by the trucks. Planner is the class that abstracts away these decisions.

```python
import copy
from decimal import Decimal
from functools import cmp_to_key
class Planner:
    def __init__(self, package_hashmap, location_graph):
        self.package_hashmap = package_hashmap
        self.location_graph = location_graph
```

It's constructor is made up of just assignments to reference variables, which are just numbers ultimately, and therefore has a space and time complexity $O(1)$

```python
    def compare_but_ignore_min_and_zero(self, i, j):
        if i is Decimal("0.0"):
            return 1
        if i is None:
            return 1
        if j is Decimal("0.0"):
            return -1
        if j is None:
            return -1
        return i - j
```

This function is just branches and comparisons. It's complexity does not grow with the graph.

Its final space and time complexity is

$$O(1)$$

.

```
    def compare_packages_by_distance_from_origin(self, i, j):
        address_i = self.package_hashmap.get(i).address
        address_j = self.package_hashmap.get(j).address
        distance_i = self.location_graph.get_travel_time(0, address_i)
        distance_j = self.location_graph.get_travel_time(0, address_j)
        return distance_i - distance_j

```

This function is just retrievals and comparisons. It's complexity does not grow with the graph.

Its final space and time complexity is

$$O(1)$$

```
1     def n_closest_packages_from_origin(self, n, package_set):
2         compare = self.compare_packages_by_distance_from_origin
3         sorted_package = sorted(list(package_set), key=cmp_to_key(compare))
4         return set(sorted_package[0:n])
```

The most costly line contains the call to sorted, see line 3. In python's standard library, sorting an unsorted list takes $O(n \cdot log(n))$ time. Where n is the size of the list. The set it returns takes $O(N)$ space where N is the variable 'n', which is the cutoff parameter. The auxiliary space is $O(P)$ where P is the length of the package set.

Its final run-time is is

$$O(P \cdot log(P))$$

The auxiliary space is $O(P + N)$ The returned object takes up $O(N)$ space

```
1         def path_from_nearest_neighbor(self, start, package_set):
2         unvisited = set()
3         for package_id in package_set:
4             address = self.package_hashmap.get(package_id).address
5             unvisited.add(self.location_graph.check_and_get_address(address))
6 
7         path = list()
8         path.append(start)
9         current_vertex = start
10 
11         while unvisited:
12             minimum_value = float("inf")
13             minimum_index = None
14             iterator = self.location_graph.AdjacencyIterator(current_vertex)
15             for index, edge_value in enumerate(iterator):
16                 if index not in unvisited:
17                     continue
18 
19                 if edge_value < minimum_value:
20                     minimum_value = edge_value
21                     minimum_index = index
22 
23             path.append(minimum_index)
24             unvisited.discard(minimum_index)
25             current_vertex = minimum_index
26         return path
```

And now the implementation of nearest neighbor.

Line 2

Initalizing an empty set takes $O(1)$

Lines 3-5

If P is the number of packages belonging to the package set then the complexity is $O(P)$.

All instructions within the loop take constant time including appending to a list/set, $O(1)$. The time and space complexity is $O(P)$.

Lines 7-9.

All operations are in constant time, $O(1)$ and the space is constant aswell

Lines 11-25.

The outer most loop continues until unvisited is the empty set. Because for each iteration of the loop we remove one member from unvisited, the outer loop will run once for each element of univisited. The carnality of unvisited is at most V because the set of vertices considered can not exceed the vertices that exist. To understand this, notice that although the package set on lines 3-5 might be larger than the number of vertices in the graph, the number of packages with unique locations can not exceed the number of locations themselves. The inner loop iterates V times because it must iterate over every edge to a neighboring location for the given. This complexity was shown in locations.py. All other instructions and function calls are $O(1)$. Multiplying the inner and outer loops yields $O(N \cdot N)$ or $O(N^2)$.

Its final runtime is

$$O(1) + O(P) + O(1) + O(V^2)$$

or

$$O(P) + O(V^2)$$

. Where V is the total number of locations.

The final space grows as $O(V)$ where V is the maximum number of unique locations belonging pointed to by the addresses in the package set.

simulation.py

Simulation contains Classes and functions that involve modeling the real world.

```python
def packages_to_location_verticies(packages_set, packages_hashmap, graph):
    sub_graph = set()
    for package_id in packages_set:
        package = packages_hashmap.get(package_id)
        sub_graph.add(graph.get(package.address))
    return sub_graph
```

This function loops for each package in package set giving it complexity $O(P)$. Getting a package from the hashmap is constant. Adding a location to the sub graph is constant. The subgraph can only be is big as the number of locations, L.

Its final run time grows as

$$O(P)$$

Its final space complexity is

$$O(V)$$

V is the number of locations and P is the number of packages.

```
class Clock:
    def __init__(self):
        self.current_time = datetime(2001, 3, 27, 8,0,0)

    def tick(self):
        self.current_time = self.current_time + timedelta(minutes=1)
```

This class has a constructor and space and time usage is not dependent on inputs. It's tick function is not dependent on inputs either. All functions in the class are constant time.

Its final space and time complexity is

$$O(1)$$

```python
class Truck:
    def __init__(self, id_code, graph, package_hashmap):
        self.is_moving = False
        self.distance_per_minute = Decimal('0.3')
        self.id_code = id_code
        self.source = 0
        self.odometer= 0
        self.distance_from_source = Decimal('0')
        self.packages = set()
        self.path = []
        self.history = []
        self.package_hashmap = package_hashmap
        self.global_packages = set()
        if (type(graph) != location.LocationGraph):
            raise Exception("Graph is not of type LocationGraph")
        self.map = graph
```

The constructor of truck simply initalizes variables. It has a Branch but all operations are in constant time.

Its final space and time complexity is

$$O(1)$$

```
    @property
    def distance_from_target(self):
        if self.target is None:
            raise Exception("There is no distance from target")
        return self.map.get_travel_time(self.source, self.target) - self.distance_from_source

    @property
    def target(self):
        if len(self.path) > 0:
            return self.path[0]
        return None

    def tick_n_times(self, n_ticks = 1):
        while n_ticks > 0:
            self.tick()
            n_ticks = n_ticks -1
```

Target and distance-from-target define a properties and their call has constant time. $O(1)$

Tick n times while loop loops once for the magnitude of n ticks. Notice that n ticks decreases once per iteration until it reaches 0. It has complexity $O(N_t)$ where N_t is the value of n ticks.

```python
    def deliver(self):
        to_discard = []
        for package_id in self.packages:
            address = self.package_hashmap.get(package_id).address
            package_location = self.map.check_and_get_address(address)
            if package_location == self.target:
                to_discard.append(package_id)

        for old in to_discard:
            self.packages.discard(old)
            self.global_packages.discard(old)
```

The first loop loops once for every package in the trucks inventory. This makes it have time $O(16)$ or $O(1)$ due to the physical constraints of the trucks outlined in the requirements. For each iteration there is, at most, one package added to the discarded list so the second loops runs 16 or less times times. Both loops effective run in constant time given the restraints of a truck.

Its final space and time complexity is

$$O(1)$$

.

```python
    def tick(self):
        if self.source is None:
            raise Exception("There is not source for truck", self.id_code)
            return

        if self.target is None:
            self.is_moving = False
            return

        self.odometer = self.distance_per_minute + self.odometer
        self.distance_from_source = self.distance_from_source + self.distance_per_minute
        if self.distance_from_target <= 0:
            self.deliver()
            self.distance_from_source = self.distance_from_target * -1
            self.source = self.target
            self.history.append(self.path.pop(0))
```

These operations are all constant even the call to self.deliver as shown before. There are no loops only branches.

Its final space and time complexity is

$$O(1)$$

package-hasher.py

```python
class PackageHashMap:
    def __init__(self, number_of_cells : int):
        self.number_of_cells = number_of_cells
        self.table_cells = []

        for value in range(0, number_of_cells):
            self.table_cells.append(list())
```

The constructor for this class just initializes values. It loops once for the number of cells specified, giving it a run-time of $O(N)$ where N is the number of cells passed to the constructor. The space of table-cells grows the same way.

Its final space and time complexity is

$$O(N)$$

.

```
1      def insert(self, key, package):
2          index = self.hash_function(key)
3          package_list = self.table_cells[index]
4
5          for entry in package_list:
6              if entry[0] == key:
7                  entry[1] = package
8                  return True
9
10         entry = [key, package]
11         package_list.append(entry)
12         return True
```

This the loop on line 5-9 runs E times where E is the number of entries in a given cell on the table. Every other operation is constant. Package list is only a subset of the total packages and so P is not considered, in other words indexing cuts down the considered search space to E.

Its final runtime is

$$O(E)$$

. The space taken up by a single call is $O(1)$. The space taken up by E calls to insert that happen to share a index is $O(E)$.

```
def get(self, key):
    index = self.hash_function(key)
    package_list = self.table_cells[index]
    for entry in package_list:
        if entry[0] == key:
            return entry[1]

    return None

def does_contain(self, key):
    index = self.hash_function(key)

    package_list = self.table_cells[index]
    for entry in package_list:
        if entry[0] == key:
            return True
    return False

def remove(self, key):
    index = self.hash_function(key)
    package_list = self.table_cells[index]
    for entry in package_list:
        if entry[0] == key:
            package_list.remove([entry[0], entry[1]])
```

These next three functions are similar to the previous block of code analyzed. The same reasoning gives in as before give their complexity.

All of their time complexities are $O(E)$. Where E is the number of entries in a cell. All of there space complexities are $O(1)$ because the space they take up does not grow with any input.

```
    class _PackageIterator():
        def __init__(self, parent):
            self.parent = parent
            self.table = parent.table_cells
            self.cell_index = 0
            self.cell_deepness = 0

        def __iter__(self):
            return self

        def __next__(self):
            if self.cell_index >= len(self.table):
                raise StopIteration

            current_cell = self.table[self.cell_index]
            if len(current_cell) > self.cell_deepness:
                return_deepness = self.cell_deepness
                self.cell_deepness = self.cell_deepness + 1
                return current_cell[return_deepness][1]

            self.cell_index = self.cell_index + 1
            self.cell_deepness = 0
            return self.__next__()

    def PackageIterator(self):
        return self._PackageIterator(self)
```

Every function in this iterator does not loop and consequently does not grow with any potential input, which is equivalent to $O(1)$. But iterating through the entire table takes $O(P)$ time where P is the number of packages in the table. Notice that the iterator only stops when the index exceeds the total number of cells in the table.The time spent traversing all cells and chain of packages within them grows linearly with the number of packages.

The run time of each function and a full iteration is

$$O(1) \ \ \& \ \ O(P)$$

. The space taken up by each function is $O(1)$.

```python
    def hash_function(self, key :int):
        string = str(key)
        sum_of = 0
        limit = 4
        for character in string:
            sum_of = sum_of + int(character)
            if limit == 0: #Keep loop at constant time
                break
            limit = limit - 1
        return (sum_of + key)  % self.number_of_cells
```

This function has constant operations and one loop. The loop is limited to 4 iterations or $O(4) \equiv O(1)$. It does not grow with inputs.

Its final space and time complexity is

$$O(1)$$

.

package-reader.py

```
1  from package_hasher import PackageHashMap
2  from package import Package
3  import csv
4
5  def get_packages():
6      packages = list()
7      with open('deliveries-from-scratch/resources/rawpackage.csv', newline='') as f:
8          rows = csv.reader(f)
9          for data in rows:
10             packages.append(Package(fields=data))
11         hashmap = PackageHashMap(int(len(packages)*(3/4))
12     for package in packages:
13         hashmap.insert(package.id_code, package)
14     return hashmap
```

Package reader has one function. This function first iterates through every row and column of rawpackage.csv. The rows are each a package, P, and the Columns are attributes or members of the package. The number of attributes does not change so searching through the csv document takes $O(P)$ time. Creating a package on line 10 was shown to take constant time earlier in this document. Creating a hashmap was also show to be linear earlier (see line 11). Looping through all packages in the list takes P iterations (for lines 12-13). Inserting package into the hashmap takes $O(E)$.

It is helpful to relate E to P; we force the number of cells in this hashmap to be linearly related to the number of packages, this makes $C = P \cdot F$, where F is some constant factor, in this case $F = 3/4$ (see line 11). To increase E two packages must share a hash value, which for two arbitrary keys the likely hood of this is $\frac{1}{C}$, assuming the keys are chosen completely randomly. For the majority of inserts the time will be constant because the inserted package will be the first package in the cell. Even if every package happens to be inserted into the same cell, the maximum number of collisions, and consequently loops through that cell, will be at most P per insertions. Each previous loop would iterate $P - i$ times, where i is the difference between the very last iteration and the current preceding iteration.

$$P + (P - 1) + (P - 2) + ... + (P - P)$$

$(P - P)$ is zero. This leaves us with P total terms in parenthesis which will be pertinent later. $[P - (P - 1)]$ is the second to last term, $[P - (P - 2)]$ is the third to last.

$$P + (P - 1) + (P - 2) + ... + [P - (P - 2)] + [P - (P - 1)]$$

Notice the parity between the addition of $P - 1$ on the left side and the subtraction of the same term on the right. The same goes for $(P - 2)$. We can take out every similar term from both sides leaving a $P/2$ number of P terms all added together.

$$P + P + P + ... + P = \frac{P}{2} \cdot P = \frac{1}{2}P^2$$

Another way to show this with the expression from before.

$$P + (P - 1) + (P - 2) + ... + [P - (P - 2)] + [P - (P - 1)]$$

$(P - (P - j)) = j$ for any value of j

$$P + (P - 1) + (P - 2) + ... + 2 + 1$$

Now remove parenthesis and zero out like numbers.

$$P + P - 1 + P - 2 + ... + 2 + 1$$

$$P + P + P + ... + (2 - 2) + (1 - 1)$$

This halves our total number of terms from P to $p/2$

$$P + P + P + ... + P = \frac{1}{2}P^2$$

In Big O.

$$O(\frac{P^2}{2}) \equiv O(P^2)$$

The worst case run time is

$$O(P^2)$$

It is extremely unlikely for the worst case to happen. The likelihood of this for, say, 40 packages is $\frac{1}{30^{40}}$; a likelihood that can be ignored. For more packages this becomes even less likely. With the given hash function, and a chronological group of keys it will not happen for the first 40. This gives it an an effective run time of

$$O(P)$$

Spatially the hash table grows with the number of packages added or

$$O(P)$$

map-reader.py

```python
import csv
from location import LocationGraph

def get_graph():
    graph = LocationGraph()
    packages = list()
    with open('deliveries-from-scratch/resources/rawdistance.csv', newline='') as f:
        rows = csv.reader(f)
        locations = list()
        for row in rows:
            address = row[0][1:-8]
            address = address.replace("South", "S")
            address = address.replace("East", "E")
            address = address.replace("North", "N")
            address = address.replace("West", "W")
            locations.append(address)
            graph.add_location(address)
            for index in range(1, len(row[1:])):
                if row[index] == "0.0":
                    break
                horizantal_location = locations[index-1]
                vertical_location = address
                graph.set_travel_time(horizantal_location, vertical_location, row[index])
    return graph
```

This file has one function. It starts by creating an empty graph. Then it opens a csv file with L Rows and L Columns where L is the number of locations in the csv. Replacing one part of a string in python is $O(N)$ but since the size of the string does not grow with the number of locations, seeing as how addresses can only get so long, in this case the function call is constant. Because the edges don't have directions we only traverse through half of the L by L matrix and we need half as many spaces to yield $L^2/2$. So the loops give it a complexity of $O(\frac{L^2}{2}) \equiv O(L^2)$.

Its final run time is

$$O(L^2)$$

. For reasons explained in the pseudo code section of the project, to be specific the adjacency matrix, the worst case space used grows wit the number of locations at rate of $\frac{L^2}{2}$ or in Big O

$$O(L^2)$$

main.py

```
#First Name: George
#Last Name: Allen
#Student ID: 010189261

from datetime import datetime
import math
import os
import package_reader
import map_reader
import planner
import simulation

graph = map_reader.get_graph() # Load Objects from External Files
packages = package_reader.get_packages()
wgu_planner = planner.Planner(packages, graph)
clock = simulation.Clock()

truck_one = simulation.Truck(1, graph, packages) #  Instatiate trucks
truck_two = simulation.Truck(2, graph, packages)
truck_three = simulation.Truck(3, graph, packages)

undelivered = set()  #Instantiate all sets
undelivered_reference = set()
ready_packages = set()
normal_packages = set()
grouped_packages = set()
closet_packages = set()
truck_two_only = set()
delayed_packages = set()
wrong_address = set()
has_deadline = set()

for package in packages.PackageIterator(): #  All packages marked undelivered and put
↪  in hub
    undelivered.add(package.id_code)
    undelivered_reference.add(package.id_code)
```

```
    package.status = "hub"

packages.get(9).status = "Wrong Address"

for package_id in undelivered: #  Packages are put into groups based on their
↪  attributes.
    note = packages.get(package_id).note
    deadline = packages.get(package_id).deadline
    if note is None:
        normal_packages.add(package_id)
    elif note == "hasfriends":
        friends = packages.get(package_id).friends
        for friend in friends:
            grouped_packages.add(friend)
        grouped_packages.add(package_id)
    elif note == "truck2":
        truck_two_only.add(package_id)
    elif note == "delayed":
        delayed_packages.add(package_id)
        packages.get(package_id).status = "Delayed"
    elif note == "wrong_address":
        wrong_address.add(package_id)
    if deadline != "EOD":
        has_deadline.add(package_id)

normal_packages = normal_packages.difference(grouped_packages) # remove groups packages
↪  from normal
normal_packages = normal_packages.difference(delayed_packages) # remove groups packages
↪  from normal

for index, package_id in enumerate(closet_packages): #
    package_address = packages.get(package_id).address

truck_one.is_moving = True #  Strategy is to get the simple stuff
truck_two.is_moving = False #  Strategy is to get more complicated stuff
truck_one.global_packages = undelivered
truck_two.global_packages = undelivered

def safe_load(working_set, potential_set, max_load):  #Loads without overloading
```

```python
    closest_of = wgu_planner.n_closest_packages_from_origin
    filtered_set = potential_set.difference(working_set)
    working_set = working_set.union(closest_of(min(max_load, max_load-len(working_set)), filtered_set))
    return working_set

def load_truck_one():  #Loads truck one
    working = set()

    grouped = undelivered.intersection(grouped_packages).difference(truck_two.packages)
    time_bound = undelivered.intersection(has_deadline).difference(truck_two.packages).difference(delayed_packag
    normal = undelivered.intersection(normal_packages).difference(truck_two.packages)

    working = safe_load(working, grouped, 16)
    working = safe_load(working, time_bound, 16)
    working = safe_load(working, normal, 16)

    truck_one.packages = working
    for package in truck_one.packages:
        packages.get(package).status = "en route"

def load_truck_two():  #Loads truck two and performs nearest neighbor.
    working = set()

    delay = undelivered.intersection(delayed_packages).difference(truck_one.packages)
    two_only = undelivered.intersection(truck_two_only).difference(truck_one.packages)
    normal = undelivered.intersection(normal_packages).difference(truck_one.packages)

    working = safe_load(working, delay, 16)
    working = safe_load(working, two_only, 16)
    working = safe_load(working, normal, 16)

    truck_two.packages = working
    for package in truck_two.packages:
        packages.get(package).status = "en route"

running = True
```

```python
def prompt_integer():  #Does not stop until it gets an integer
    print("Response must be an integer")
    try:
        return int(input())
    except:
        return prompt_integer()

breakpoints = list()

while running: #  Loops until all breakpoints are set
    print("Options\n1. Add breakpoint\n2. Start Simulation")
    response = input()
    if response == '2':
        running = False
        continue

    if response == '1':
        running_breakpoint = True
        while running_breakpoint:
            print("Hour of day to see status of delivery? 8-21")
            hour = prompt_integer()
            if hour < 8 or hour >= 22:
                print("Hour must be after 8 or before 10")
                running_breakpoint = False
                continue

            print("What minute of the hour?")
            minute = prompt_integer()
            if minute < 0 or minute >= 60:
                print("Minute must be between 00 and 59")
                running_breakpoint = False
                continue

            breakpoints.append(datetime(2001, 3, 27, hour, minute,0))
            running_breakpoint = False

debug = False
if debug:  #  When debugging intervals of 15 minutes are used.
    for index in range(0,30):
```

```python
        minute = (index % 4) * 15
        hour = 8 + math.floor(index/4)
        breakpoints.append(datetime(2001, 3, 27, hour, minute,0))

def compare_route(route):  #  Shows a route in it's entiretly
    source = route[0]
    target = None
    for index, destination in enumerate(route[1:]):
        target = destination
        print(index, source, destination, graph.get_travel_time(source, destination))
        source = destination
    print("\n\n")

while undelivered or clock.current_time.hour < 14:  #  Does not stop untill everything is underlivered.
    if clock.current_time.hour == 10 and clock.current_time.minute == 20:
        packages.get(9).address="410 S State St" #  We know the address at this time
        packages.get(9).status="hub" #  We know the address at this time
        normal_packages.add(9)

    if clock.current_time.hour == 9 and clock.current_time.minute == 5:
        for package_id in delayed_packages:
            packages.get(package_id).status = "hub"

    if len(truck_one.path) == 0 and truck_one.source == 0:  #Checks if truck can be loaded.
        load_truck_one()
        if len(truck_one.packages) != 0:
            truck_one.path = wgu_planner.path_from_nearest_neighbor(0, truck_one.packages)
            truck_one.path.append(0)
            truck_one.is_moving = True
            print("One Pack:", truck_one.packages)
            print("One Path:", truck_one.path)

    if len(truck_two.path) == 0 and truck_two.source == 0 and clock.current_time.hour >= 9 and clock.current_time.minute >= 5:  #  Checks if truck can be loaded.
        load_truck_two()
        if len(truck_two.packages) != 0:
```

```
            truck_two.path = wgu_planner.path_from_nearest_neighbor(0,
            ↪  truck_two.packages)
            truck_two.path.append(0)
            truck_two.is_moving = True
            print("Two Pack", truck_two.packages)
            print("Two Path", truck_two.path)

    for time in breakpoints:  #  handles breakpoints
        this_hour = clock.current_time.hour
        this_minute = clock.current_time.minute

        break_hour = time.hour
        break_minute = time.minute

        if this_hour == break_hour and this_minute == break_minute:
            print("What package would you like to see? For all write -1")
            if not debug:
                id_code = prompt_integer()
            else:
                id_code = -1
            if id_code == -1:
                string = str()
                delivered = set()
                en_route = set()
                hub = set()
                for package in packages.PackageIterator():
                    string = ""
                    string += "[ID: "
                    string += str(package.id_code)
                    string += "] [Address: "
                    string += package.address

                    if package.deadline is not None:
                        string+= "] [Deadline: "
                        string+= package.deadline
                    string += "] [City: "+ package.city
                    string += "] [Weight: "+ package.mass
                    string += "] [ZipCode: "+ package.zip_code
                    string += "] [Status: " + package.status + "]"
```

```python
                    print(string)
            else:
                package = packages.get(id_code)
                print("ID:", package.id_code)
                print("Address:", package.address)
                if package.deadline is not None:
                    print("Deadline:", package.deadline)
                print("City:", package.city)
                print("Weight:", package.mass)
                print("ZipCode:", package.zip_code)
                print(id_code, "status is ", packages.get(id_code).status)
            print("Truck One Miles: ->", truck_one.odometer)
            print("Truck Two Miles: ->", truck_two.odometer)
            print("Truck Three Miles: ->", truck_three.odometer)
            print("Total Miles: ->", truck_three.odometer + truck_two.odometer + truck_one.odometer)
            print("Current Time ->", clock.current_time)
            input()
            print("\n\n")

    for recent_dropoff in undelivered_reference.difference(undelivered):  # Compares previous package states to current
        undelivered_reference.discard(recent_dropoff)
        minute_as_string = str(clock.current_time.minute)
        if len(minute_as_string) == 1:
            minute_as_string = "0" + minute_as_string
        packages.get(recent_dropoff).status = "Delivered " + str(clock.current_time.hour) + ":" + minute_as_string
        print("package", recent_dropoff, "has been delivered at", clock.current_time)

    clock.tick()
    if undelivered:
        truck_one.tick()
        truck_two.tick()
print("Total Miles: ->", truck_three.odometer + truck_two.odometer + truck_one.odometer)
```

L: Line

S: space

T: Time

I: System I/O input

W_s: Working set carnality

P_s: Potential set carnality

M_l: Max load value

R: Length of route

P_m: Max packages in truck

If two growth functions are given the smaller function is the practical or likely growth.

Lines	Space	Time	Reason
1→11	NA	NA	imports
13→16	$O(P+V^2)$	$O(P+V^2) - O(P^2+V^2)$	See Explanation A
18→20	$O(1)$	$O(1)$	Constructors are constant
22→31	$O(1)$	$O(1)$	Creating a set is constant
33→36	$O(P)$	$O(P)$	All Packages duplicated in undelivered
40→58	$O(P)$	$O(1) - O(P)$	Loops through undelivered
60	$O(P)$	$O(P)$	See Explanation B
62→65	$O(1)$	$O(1)$	Only Reference Copy
67→71	$O(M_l)$	$O(W_l + P_s + M_l)$	Max load cuts off any growth
73→101	$O(1)$	$O(P) - O(P^2)$	Set difference can take $O(P^2)$
103	$O(1)$	$O(1)$	write is constant
105→110	$O(I)$	$O(I)$	Depends on input
112	$O(1)$	$O(1)$	Creating Object is constant
114→139	$O(I)$	$O(I)$	Depends on input
141→146	NA	NA	Debug
148→155	$O(1)$	$O(R)$	Iterates over entire loop
157	$O(1)$	$O(1)$	See Explanation C
158→165	$O(1)$	$O(1)$	Does not grow
167→184	$O(P)$	$O(P^2+V^2)$	Calls nearest neighbor and Picks Packages
185→234	$O(1)$	$O(I)$	Depends on inputs
236→242	$O(1)$	$O(1)$	Effectively constant
244→247	$O(1)$	$O(1)$	Tick functions all constant

Explanation A

Because these functions both construct and fill out both package and location data structures, they have a space-time complexity equivalent to the function that performs the reading of that data. The package reader is not at it's worse case because id's 1-40 don't hash to the same cell.

Explanation B

A difference of two sets, of carnality m and n respectively, has a runtime $O(m+n)$ in the very unlikely but worst case. Because in this case $p=m>n$ this runtime is $\leq O(p+p) \leq (2p) \leq O(p)$

Explanation C

There are a constant amount of minutes in a day. Simulating the day is the same as unrolling the loop a constant number of times. One unroll for each minute yields a runtime of $O(740) = O(1)$

Overall Time Complexity

Let's start by going through the entire programs execution in main.py and then simplifying form there. Starting out should be an amalgamation of the entire table above.

$$O(P^2 + V^2 + 1 + P + W_t + P_m + M_t + I)$$

Remove constant inputs, values dependent on other inputs, and lower order powers.

$$O(P^2 + V^2 + I)$$

If N is the largest input then the final time complexity is

$$O(N^2)$$

Let's start by going through the entire programs execution in main.py and then simplifying form there. Starting out should be an amalgamation of the entire table above.

$$O(P^2 + V^2 + 1 + P + W_t + P_m + M_t + I)$$

Remove constant values and inputs dependent on other inputs.

$$O(P^2 + V^2 + I)$$

If N is the largest input then the final time complexity is

$$O(N^2)$$

Overall Space Complexity

Again we start with the entire execution complexities.

$$O(P + V^2 + I + M_t)$$

Remove dependent variables and constants.

$$O(P + V^2 + I)$$

If N is the largest input then our worst case space complexity is.

$$O(N^2)$$

B.4

This software does not have any hard limitations. It can take a completely dynamic amount of locations and packages and grows according to the analysis in section B.3. For example, both the map and location reader only stop reading data when the file is complete. Adding an arbitrary amount of packages/locations is completely supported because for the nature of the for loop that reads the file.

Another way this software allows for a growing number of packages is the implementation of loading packages into trucks. It uses a simple algorithm to grab preferred packages from the entire set of undelivered packages, and does not stop delivering until all packages are undelivered.

The algorithm has a space complexity of $O(N)$ where N is the number of packages. It has a run time of $O(N^2)$.

B.5

This software is efficient in the way it loads locations and distances into the LocationGraph. Instead of storing each edge twice for a location pair. It uses logic to simulate a full matrix which is flipped diagonally. This halves the space required because undirected graphs don't need two edges per pair.

All distances are stored as fixed point numbers. This means there is no loss of value when simulating the trucks throughout the day. Every minute the precise Odometer reading can be understood.

A way the program is maintainable is the way it groups packages in by their ID in sets. Then the hash map is utilized to actually retrieve the information. This makes classifying packages at a high level very intuitive for future modifications to the software. Each ID is only a numeric value but using the hashmap we quickly categorize and utilize packages.

B.6

A hash table has many strengths. It makes storing and retrieving data take effectively constant time in situations where, without a hashmap, the time for retrieving data would grow with the amount of data that is stored. This is done by translating variables to values that can be indexed instead of searched for.

It also can be implemented a variety of ways. You can pass an entire object as the hash key or only an identifying value of the object. In this softwares case the object is stored as a reference into the hash map, but one might store a reference to a deep copy of the object depending on the particular need.

These things are nifty, but a hashmap is not the end all be all of data structures. Consider the fact that moving, or sorting, data in a hashmap can only be done my transferring that data to outside the hashmap. Internal moving breaks the data structure, because storing and retrieving needs to be deterministicly dependent on the key. The cells the variables are stored in are not interchangeable. You can not move an object to a cell whose value does not correspond to its key.

The other disadvantage pertains to resources, if the developer sets the number of cells too low, the runtime grows too fast with each variable/object being stored. If the developer sets the number of cells to big, then more space will be used than necessary.

C

C.1 and C.2

See Attached zip file for complete and commented source.

D

The data structure chosen, a hash table, uses the id-code as a key, to determine what cell the reference to the package goes into. Because a reference is passed into the cell and only the key is used to determine the index of the cell. The data being stored can be changed without destroying the integrity of the data structure. This is because, although the attributes change, the id-code of the package and reference to the package never actually change.

This system is it's way of accounting for the attributes. It puts data points behind a references in the python language. When one wants to retrieve the data an id is provided the hashmap and a reference is returned.

E and F

See source file package-hasher.py for the table and lookup function.

G

G.1

```
-1
[ID: 29] [Address: 1330 2100 S] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 2] [ZipCode: 84106] [Status: en route]
[ID: 15] [Address: 4580 S 2300 E] [Deadline: 9:00 AM] [City: Holladay] [Weight: 4] [ZipCode: 84117] [Status: Delivered 8:13]
[ID: 34] [Address: 4580 S 2300 E] [Deadline: 10:30 AM] [City: Holladay] [Weight: 2] [ZipCode: 84117] [Status: Delivered 8:13]
[ID: 1] [Address: 195 W Oakland Ave] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 21] [ZipCode: 84115] [Status: en route]
[ID: 20] [Address: 3595 Main St] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 37] [ZipCode: 84115] [Status: Delivered 8:38]
[ID: 16] [Address: 4580 S 2300 E] [Deadline: 10:30 AM] [City: Holladay] [Weight: 88] [ZipCode: 84117] [Status: Delivered 8:13]
[ID: 35] [Address: 1060 Dalton Ave S] [Deadline: EOD] [City: Salt Lake City] [Weight: 88] [ZipCode: 84104] [Status: hub]
[ID: 2] [Address: 2530 S 500 E] [Deadline: EOD] [City: Salt Lake City] [Weight: 44] [ZipCode: 84106] [Status: hub]
[ID: 21] [Address: 3595 Main St] [Deadline: EOD] [City: Salt Lake City] [Weight: 3] [ZipCode: 84115] [Status: Delivered 8:38]
[ID: 40] [Address: 380 W 2880 S] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 45] [ZipCode: 84115] [Status: en route]
[ID: 17] [Address: 3148 S 1100 W] [Deadline: EOD] [City: Salt Lake City] [Weight: 2] [ZipCode: 84119] [Status: hub]
[ID: 36] [Address: 2300 Parkway Blvd] [Deadline: EOD] [City: West Valley City] [Weight: 88] [ZipCode: 84119] [Status: hub]
[ID: 3] [Address: 233 Canyon Rd] [Deadline: EOD] [City: Salt Lake City] [Weight: 2] [ZipCode: 84103] [Status: hub]
[ID: 22] [Address: 6351 S 900 E] [Deadline: EOD] [City: Murray] [Weight: 2] [ZipCode: 84121] [Status: hub]
[ID: 18] [Address: 1488 4800 S] [Deadline: EOD] [City: Salt Lake City] [Weight: 6] [ZipCode: 84123] [Status: hub]
[ID: 37] [Address: 410 S State St] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 2] [ZipCode: 84111] [Status: en route]
[ID: 4] [Address: 380 W 2880 S] [Deadline: EOD] [City: Salt Lake City] [Weight: 4] [ZipCode: 84115] [Status: hub]
[ID: 23] [Address: 5100 S 2700 W] [Deadline: EOD] [City: Salt Lake City] [Weight: 5] [ZipCode: 84118] [Status: hub]
[ID: 19] [Address: 177 W Price Ave] [Deadline: EOD] [City: Salt Lake City] [Weight: 37] [ZipCode: 84115] [Status: en route]
[ID: 38] [Address: 410 S State St] [Deadline: EOD] [City: Salt Lake City] [Weight: 9] [ZipCode: 84111] [Status: hub]
[ID: 5] [Address: 410 S State St] [Deadline: EOD] [City: Salt Lake City] [Weight: 5] [ZipCode: 84111] [Status: hub]
[ID: 24] [Address: 5025 State St] [Deadline: EOD] [City: Murray] [Weight: 7] [ZipCode: 84107] [Status: Delivered 8:30]
[ID: 10] [Address: 600 E 900 S] [Deadline: EOD] [City: Salt Lake City] [Weight: 1] [ZipCode: 84105] [Status: hub]
[ID: 39] [Address: 2010 W 500 S] [Deadline: EOD] [City: Salt Lake City] [Weight: 9] [ZipCode: 84104] [Status: hub]
[ID: 6] [Address: 3060 Lester St] [Deadline: 10:30 AM] [City: West Valley City] [Weight: 88] [ZipCode: 84119] [Status: Delayed]
[ID: 25] [Address: 5383 S 900 E #104] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 7] [ZipCode: 84117] [Status: Delayed]
[ID: 11] [Address: 2600 Taylorsville Blvd] [Deadline: EOD] [City: Salt Lake City] [Weight: 1] [ZipCode: 84118] [Status: hub]
[ID: 30] [Address: 300 State St] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 1] [ZipCode: 84103] [Status: en route]
[ID: 7] [Address: 1330 2100 S] [Deadline: EOD] [City: Salt Lake City] [Weight: 8] [ZipCode: 84106] [Status: hub]
[ID: 26] [Address: 5383 S 900 E #104] [Deadline: EOD] [City: Salt Lake City] [Weight: 25] [ZipCode: 84117] [Status: Delivered 8:25]
[ID: 12] [Address: 3575 W Valley Central Station bus Loop] [Deadline: EOD] [City: West Valley City] [Weight: 1] [ZipCode: 84119] [Status: hub]
[ID: 31] [Address: 3365 S 900 W] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 1] [ZipCode: 84119] [Status: en route]
[ID: 8] [Address: 300 State St] [Deadline: EOD] [City: Salt Lake City] [Weight: 9] [ZipCode: 84103] [Status: hub]
[ID: 27] [Address: 1060 Dalton Ave S] [Deadline: EOD] [City: Salt Lake City] [Weight: 5] [ZipCode: 84104] [Status: hub]
[ID: 13] [Address: 2010 W 500 S] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 2] [ZipCode: 84104] [Status: en route]
[ID: 32] [Address: 3365 S 900 W] [Deadline: EOD] [City: Salt Lake City] [Weight: 1] [ZipCode: 84119] [Status: Delayed]
[ID: 9] [Address: 300 State St] [Deadline: EOD] [City: Salt Lake City] [Weight: 2] [ZipCode: 84103] [Status: Wrong Address]
[ID: 28] [Address: 2835 Main St] [Deadline: EOD] [City: Salt Lake City] [Weight: 7] [ZipCode: 84115] [Status: Delayed]
[ID: 14] [Address: 4300 S 1300 E] [Deadline: 10:30 AM] [City: Millcreek] [Weight: 88] [ZipCode: 84117] [Status: Delivered 8:07]
[ID: 33] [Address: 2530 S 500 E] [Deadline: EOD] [City: Salt Lake City] [Weight: 1] [ZipCode: 84106] [Status: hub]
Truck One Miles: -> 12.0
Truck Two Miles: -> 0
Truck Three Miles: -> 0
Total Miles: -> 12.0
Current Time -> 2001-03-27 08:40:00
```

G.2

```
Response must be an integer
-1
[ID: 29] [Address: 1330 2100 S] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 2] [ZipCode: 84106] [Status: Delivered 8:58]
[ID: 15] [Address: 4580 S 2300 E] [Deadline: 9:00 AM] [City: Holladay] [Weight: 4] [ZipCode: 84117] [Status: Delivered 8:13]
[ID: 34] [Address: 4580 S 2300 E] [Deadline: 10:30 AM] [City: Holladay] [Weight: 2] [ZipCode: 84117] [Status: Delivered 8:13]
[ID: 1] [Address: 195 W Oakland Ave] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 21] [ZipCode: 84115] [Status: Delivered 8:49]
[ID: 20] [Address: 3595 Main St] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 37] [ZipCode: 84115] [Status: Delivered 8:38]
[ID: 16] [Address: 4580 S 2300 E] [Deadline: 10:30 AM] [City: Holladay] [Weight: 88] [ZipCode: 84117] [Status: Delivered 8:13]
[ID: 35] [Address: 1060 Dalton Ave S] [Deadline: EOD] [City: Salt Lake City] [Weight: 88] [ZipCode: 84104] [Status: hub]
[ID: 2] [Address: 2530 S 500 E] [Deadline: EOD] [City: Salt Lake City] [Weight: 44] [ZipCode: 84106] [Status: Delivered 9:38]
[ID: 21] [Address: 3595 Main St] [Deadline: EOD] [City: Salt Lake City] [Weight: 3] [ZipCode: 84115] [Status: Delivered 8:38]
[ID: 40] [Address: 380 W 2880 S] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 45] [ZipCode: 84115] [Status: Delivered 8:45]
[ID: 17] [Address: 3148 S 1100 W] [Deadline: EOD] [City: Salt Lake City] [Weight: 2] [ZipCode: 84119] [Status: en route]
[ID: 36] [Address: 2300 Parkway Blvd] [Deadline: EOD] [City: West Valley City] [Weight: 88] [ZipCode: 84119] [Status: en route]
[ID: 3] [Address: 233 Canyon Rd] [Deadline: EOD] [City: Salt Lake City] [Weight: 2] [ZipCode: 84103] [Status: en route]
[ID: 22] [Address: 6351 S 900 E] [Deadline: EOD] [City: Murray] [Weight: 2] [ZipCode: 84121] [Status: Delivered 9:18]
[ID: 18] [Address: 1488 4800 S] [Deadline: EOD] [City: Salt Lake City] [Weight: 6] [ZipCode: 84123] [Status: en route]
[ID: 37] [Address: 410 S State St] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 2] [ZipCode: 84111] [Status: Delivered 9:13]
[ID: 4] [Address: 380 W 2880 S] [Deadline: EOD] [City: Salt Lake City] [Weight: 4] [ZipCode: 84115] [Status: en route]
[ID: 23] [Address: 5100 S 2700 W] [Deadline: EOD] [City: Salt Lake City] [Weight: 5] [ZipCode: 84118] [Status: hub]
[ID: 19] [Address: 177 W Price Ave] [Deadline: EOD] [City: Salt Lake City] [Weight: 37] [ZipCode: 84115] [Status: Delivered 8:40]
[ID: 38] [Address: 410 S State St] [Deadline: EOD] [City: Salt Lake City] [Weight: 9] [ZipCode: 84111] [Status: en route]
[ID: 5] [Address: 410 S State St] [Deadline: EOD] [City: Salt Lake City] [Weight: 5] [ZipCode: 84111] [Status: hub]
[ID: 24] [Address: 5025 State St] [Deadline: EOD] [City: Murray] [Weight: 7] [ZipCode: 84107] [Status: Delivered 8:30]
[ID: 10] [Address: 600 E 900 S] [Deadline: EOD] [City: Salt Lake City] [Weight: 1] [ZipCode: 84105] [Status: en route]
[ID: 39] [Address: 2010 W 500 S] [Deadline: EOD] [City: Salt Lake City] [Weight: 9] [ZipCode: 84104] [Status: hub]
[ID: 6] [Address: 3060 Lester St] [Deadline: 10:30 AM] [City: West Valley City] [Weight: 88] [ZipCode: 84119] [Status: en route]
[ID: 25] [Address: 5383 S 900 E #104] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 7] [ZipCode: 84117] [Status: Delivered 9:13]
[ID: 11] [Address: 2600 Taylorsville Blvd] [Deadline: EOD] [City: Salt Lake City] [Weight: 1] [ZipCode: 84118] [Status: en route]
[ID: 30] [Address: 300 State St] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 1] [ZipCode: 84103] [Status: Delivered 9:16]
[ID: 7] [Address: 1330 2100 S] [Deadline: EOD] [City: Salt Lake City] [Weight: 8] [ZipCode: 84106] [Status: en route]
[ID: 26] [Address: 5383 S 900 E #104] [Deadline: EOD] [City: Salt Lake City] [Weight: 25] [ZipCode: 84117] [Status: Delivered 8:25]
[ID: 12] [Address: 3575 W Valley Central Station bus Loop] [Deadline: EOD] [City: West Valley City] [Weight: 1] [ZipCode: 84119] [Status: hub]
[ID: 31] [Address: 3365 S 900 W] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 1] [ZipCode: 84119] [Status: en route]
[ID: 8] [Address: 300 State St] [Deadline: EOD] [City: Salt Lake City] [Weight: 9] [ZipCode: 84103] [Status: hub]
[ID: 27] [Address: 1060 Dalton Ave S] [Deadline: EOD] [City: Salt Lake City] [Weight: 5] [ZipCode: 84104] [Status: hub]
[ID: 13] [Address: 2010 W 500 S] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 2] [ZipCode: 84104] [Status: Delivered 9:30]
[ID: 32] [Address: 3365 S 900 W] [Deadline: EOD] [City: Salt Lake City] [Weight: 1] [ZipCode: 84119] [Status: en route]
[ID: 9] [Address: 300 State St] [Deadline: EOD] [City: Salt Lake City] [Weight: 2] [ZipCode: 84103] [Status: Wrong Address]
[ID: 28] [Address: 2835 Main St] [Deadline: EOD] [City: Salt Lake City] [Weight: 7] [ZipCode: 84115] [Status: en route]
[ID: 14] [Address: 4300 S 1300 E] [Deadline: 10:30 AM] [City: Millcreek] [Weight: 88] [ZipCode: 84117] [Status: Delivered 8:07]
[ID: 33] [Address: 2530 S 500 E] [Deadline: EOD] [City: Salt Lake City] [Weight: 1] [ZipCode: 84106] [Status: Delivered 9:38]
Truck One Miles: -> 30.0
Truck Two Miles: -> 10.5
Truck Three Miles: -> 0
Total Miles: -> 40.5
Current Time -> 2001-03-27 09:40:00
```

G.3

```
Response must be an integer
-1
[ID: 29] [Address: 1330 2100 S] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 2] [ZipCode: 84106] [Status: Delivered 8:58]
[ID: 15] [Address: 4580 S 2300 E] [Deadline: 9:00 AM] [City: Holladay] [Weight: 4] [ZipCode: 84117] [Status: Delivered 8:13]
[ID: 34] [Address: 4580 S 2300 E] [Deadline: 10:30 AM] [City: Holladay] [Weight: 2] [ZipCode: 84117] [Status: Delivered 8:13]
[ID: 1] [Address: 195 W Oakland Ave] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 21] [ZipCode: 84115] [Status: Delivered 8:49]
[ID: 20] [Address: 3595 Main St] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 37] [ZipCode: 84115] [Status: Delivered 8:38]
[ID: 16] [Address: 4580 S 2300 E] [Deadline: 10:30 AM] [City: Holladay] [Weight: 88] [ZipCode: 84117] [Status: Delivered 8:13]
[ID: 35] [Address: 1060 Dalton Ave S] [Deadline: EOD] [City: Salt Lake City] [Weight: 88] [ZipCode: 84104] [Status: Delivered 10:46]
[ID: 2] [Address: 2530 S 500 E] [Deadline: EOD] [City: Salt Lake City] [Weight: 44] [ZipCode: 84106] [Status: Delivered 9:38]
[ID: 21] [Address: 3595 Main St] [Deadline: EOD] [City: Salt Lake City] [Weight: 3] [ZipCode: 84115] [Status: Delivered 8:38]
[ID: 40] [Address: 380 W 2880 S] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 45] [ZipCode: 84115] [Status: Delivered 8:45]
[ID: 17] [Address: 3148 S 1100 W] [Deadline: EOD] [City: Salt Lake City] [Weight: 2] [ZipCode: 84119] [Status: Delivered 9:52]
[ID: 36] [Address: 2300 Parkway Blvd] [Deadline: EOD] [City: West Valley City] [Weight: 88] [ZipCode: 84119] [Status: Delivered 10:02]
[ID: 3] [Address: 233 Canyon Rd] [Deadline: EOD] [City: Salt Lake City] [Weight: 2] [ZipCode: 84103] [Status: Delivered 11:12]
[ID: 22] [Address: 6351 S 900 E] [Deadline: EOD] [City: Murray] [Weight: 2] [ZipCode: 84121] [Status: Delivered 9:18]
[ID: 18] [Address: 1488 4800 S] [Deadline: EOD] [City: Salt Lake City] [Weight: 6] [ZipCode: 84123] [Status: Delivered 10:15]
[ID: 37] [Address: 410 S State St] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 2] [ZipCode: 84111] [Status: Delivered 9:13]
[ID: 4] [Address: 380 W 2880 S] [Deadline: EOD] [City: Salt Lake City] [Weight: 4] [ZipCode: 84115] [Status: Delivered 9:45]
[ID: 23] [Address: 5100 S 2700 W] [Deadline: EOD] [City: Salt Lake City] [Weight: 5] [ZipCode: 84118] [Status: Delivered 10:23]
[ID: 19] [Address: 177 W Price Ave] [Deadline: EOD] [City: Salt Lake City] [Weight: 37] [ZipCode: 84115] [Status: Delivered 8:40]
[ID: 38] [Address: 410 S State St] [Deadline: EOD] [City: Salt Lake City] [Weight: 9] [ZipCode: 84111] [Status: Delivered 11:09]
[ID: 5] [Address: 410 S State St] [Deadline: EOD] [City: Salt Lake City] [Weight: 5] [ZipCode: 84111] [Status: Delivered 11:02]
[ID: 24] [Address: 5025 State St] [Deadline: EOD] [City: Murray] [Weight: 7] [ZipCode: 84107] [Status: Delivered 8:30]
[ID: 10] [Address: 600 E 900 S] [Deadline: EOD] [City: Salt Lake City] [Weight: 1] [ZipCode: 84105] [Status: Delivered 11:03]
[ID: 39] [Address: 2010 W 500 S] [Deadline: EOD] [City: Salt Lake City] [Weight: 9] [ZipCode: 84104] [Status: Delivered 10:51]
[ID: 6] [Address: 3060 Lester St] [Deadline: 10:30 AM] [City: West Valley City] [Weight: 88] [ZipCode: 84119] [Status: Delivered 9:57]
[ID: 25] [Address: 5383 S 900 E #104] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 7] [ZipCode: 84117] [Status: Delivered 9:13]
[ID: 11] [Address: 2600 Taylorsville Blvd] [Deadline: EOD] [City: Salt Lake City] [Weight: 1] [ZipCode: 84118] [Status: Delivered 10:19]
[ID: 30] [Address: 300 State St] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 1] [ZipCode: 84103] [Status: Delivered 9:16]
[ID: 7] [Address: 1330 2100 S] [Deadline: EOD] [City: Salt Lake City] [Weight: 8] [ZipCode: 84106] [Status: Delivered 10:53]
[ID: 26] [Address: 5383 S 900 E #104] [Deadline: EOD] [City: Salt Lake City] [Weight: 25] [ZipCode: 84117] [Status: Delivered 8:25]
[ID: 12] [Address: 3575 W Valley Central Station bus Loop] [Deadline: EOD] [City: West Valley City] [Weight: 1] [ZipCode: 84119] [Status: Delivered 11:29]
[ID: 31] [Address: 3365 S 900 W] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 1] [ZipCode: 84119] [Status: Delivered 9:49]
[ID: 8] [Address: 300 State St] [Deadline: EOD] [City: Salt Lake City] [Weight: 9] [ZipCode: 84103] [Status: Delivered 11:05]
[ID: 27] [Address: 1060 Dalton Ave S] [Deadline: EOD] [City: Salt Lake City] [Weight: 5] [ZipCode: 84104] [Status: Delivered 10:46]
[ID: 13] [Address: 2010 W 500 S] [Deadline: 10:30 AM] [City: Salt Lake City] [Weight: 2] [ZipCode: 84104] [Status: Delivered 9:30]
[ID: 32] [Address: 3365 S 900 W] [Deadline: EOD] [City: Salt Lake City] [Weight: 1] [ZipCode: 84119] [Status: Delivered 9:50]
[ID: 9] [Address: 410 S State St] [Deadline: EOD] [City: Salt Lake City] [Weight: 2] [ZipCode: 84103] [Status: Delivered 11:59]
[ID: 28] [Address: 2835 Main St] [Deadline: EOD] [City: Salt Lake City] [Weight: 7] [ZipCode: 84115] [Status: Delivered 9:41]
[ID: 14] [Address: 4300 S 1300 E] [Deadline: 10:30 AM] [City: Millcreek] [Weight: 88] [ZipCode: 84117] [Status: Delivered 8:07]
[ID: 33] [Address: 2530 S 500 E] [Deadline: EOD] [City: Salt Lake City] [Weight: 1] [ZipCode: 84106] [Status: Delivered 9:38]
Truck One Miles: -> 70.5
Truck Two Miles: -> 52.2
Truck Three Miles: -> 0
Total Miles: -> 122.7
Current Time -> 2001-03-27 12:05:00
```

H

```
Total Miles: -> 122.7
```

I

I.1

A strength of nearest neighbor is it's simplicity. It's procedure can be explained in a handful of steps and it's name begs it's structure. The number of lines to implement nearest neighbor is similarly quaint.

Another strength of nearest neighbor is the fact it runs in polynomial time.

I.2

The algorithm chosen is self adjusting and delivers all packages in an optimal number of miles.

I.3

An alternative algorithm that meets the requirements is Christofides algorithm which has more shorter average and worst case path. It uses a minimum spanning tree as a starting point and then creates a possible path with the caveat that some vertices may be used twice. It then takes these duplicate vertices and removes them from that path giving a Short path where vertices are only visited once. Nearest neighbor simply marks vertecies visited and unvisited and does not use a MSP at all.

A third algorithm different from NN and Christofides is The Greedy method. This differs from NN because it starts by considering all edges instead of all vertices and their immediate neighbor. It uses the smallest edges to create a Forrest of trees until all trees are connected into one large tree that forms a short path.

J

If I were to do this project again I would use Dikstras algorithm between every pari of vertices to make sure the edge was in fact the shortest edge and that there were not shorter intermediary paths between any pairs.

K

K.1.a

The average lookup time for a hash function with P packages and C cells would take $O(\frac{P}{C})$ time and a worst lookup of $O(P)$ in the case every package is put in the same cell.

K.1.b

The data structure uses more space the more cells that are needed. In this particular implementation that is carried out in package-reader.py, the pace is is proportional to the number of packages by a constant factor.

K.1.c

In this particular implementation the trucks have no bearing on the lookup time of each package. The trucks store only the package ID's and have no real stake in the hash table. The number of locations equally has no bearing on the hash table. Because the hash stores a reference to the packages and nothing else it is exonerated of the burden of the locations.

K.2

A dynamic array could also be used to satisfy the requirements. Simply insert a package as an element of the array and then search by it's id to retrieve the attributes from the array. It has a slower lookup time on average, but potentially faster insertion. It could also be sorted.

Once could also use a linked list to store the packages and their attributes. A linked list could be organized and rearanged whereas the hash map identified in part D has deterministic storage.

L

No context is quoted, paraphrased, or summarized.