## A: ALGORITHM SELECTION

This program uses a Nearest Neighbor algorithm to schedule the package deliveries. This algorithm is applied to selected subsets of the input data to ensure that the scenario requirements are met.

## B1: LOGIC COMMENTS

The algorithm is applied during the loading of the trucks. The trucks themselves run on a queue to visit package destinations in order.

Bucket Sort (One bucket for each truck plus an extra)

For t = 1 to number\_of\_trucks:

Create bucket

Append to truck\_bucket

Create package\_bucket to hold non-truck specific packages

For each package:

If package.requested\_truck > 0:

Add package to requested truck\_bucket

Else:

Add package to package\_bucket

For each truck t:

Create list of early delivery packages (early\_packages)

For each package in truck\_bucket[t]:

If package has early delivery:

Add to early\_packages

For each package in package\_bucket:

If package has early delivery:

Add to early packages

For each package in early\_packages:

Remove package from truck\_bucket/package\_bucket

(continuing inside the For loop)

While there are early packages to deliver:

Create current\_earliest variable and initialize to datetime.max

Create current\_list and initialize to an empty list []

For each package in early\_packages:

If package deadline is earlier than current\_earliest:

Store deadline as new current\_earliest

Clear current\_list

Add package to current\_list

If package deadline is the same as current\_earliest:

Add package to current\_list

Run **load\_onto\_truck** method with current\_list.

Depth\_first\_searching is False and optional packages include the trucks truck\_bucket and the package\_bucket.

For each package in the truck’s delivery queue:

Remove package from early\_packages, package\_bucket,

and truck\_bucket[t].

Loop until no more early packages to load for this truck.

While there are packages remaining in truck\_bucket[t]:

Run **load\_onto\_truck** method with truck\_bucket[t].

Depth\_first\_searching is True and optional packages include the package\_bucket.

For each package in the truck’s delivery queue:

Remove package from package\_bucket and truck\_bucket[t].

While there are packages remaining in package\_bucket:

Run **load\_onto\_truck** method with package\_bucket[t].

Depth\_first\_searching is True.

For each package in the truck’s delivery queue:

Remove package from package\_bucket.

Once For loop is complete trucks are loaded and ready to be dispatched.

Method: **Load\_onto\_truck:**

Selected arguments:

Current\_location: Address of last package added to truck

Package\_list: Packages to be loaded

Optional\_packages: Packages that can be loaded if space allows

Depth\_first\_search: We are done with this truck’s early packages and want to try to drive away from the Hub until we reach the furthest package possible.

While len(package\_list) > 0 and truck is not at its load limit (specified by scenario):

If (load\_limit – truck.delivery\_queue) == 0:

No room on truck for optional packages, need to finish package\_list

Result = **get\_closest\_package**(package\_list)

Else if depth\_first == True:

Result = **get\_closest\_outward\_package**(package\_list +

optional\_packages)

If result[0 (found package)] == 0:

No outward package found, do nearest neighbor instead.

Result = **get\_closest\_package**(package\_list +

optional\_packages)

Else:

Check if there are any zero distance packages we can drop off while

taking care of package\_list.

Result = **get\_closest\_package**(package\_list +

optional\_packages)

(continuing the While loop)

If result[1 (distance)] > 0:

Result = **get\_closest\_package**(package\_list)

Next\_package = result[0]

New\_location = next\_package.get\_address()

Load next\_package onto truck (append to truck’s delivery queue)

If truck’s earliest departure is earlier than next\_package’s

earliest departure:

Update truck’s earliest departure to be next\_package’s

earliest departure.

Return location of truck’s most recently added package.

Method: **Get\_closest\_package:**

**This method implements a Nearest Neighbor greedy algorithm.**

Selected arguments:

Location: Source address

Package\_list: Packages to be searched for nearest neighbor

For each package in package\_list:

If distance to package is closest found so far:

Store distance and package

Return [closest package, distance]

Method: **Get\_closest\_outward\_package:**

**This method implements a modified Nearest Neighbor greedy algorithm. It finds the nearest neighbor that is farther from the Hub than the source address.**

Selected arguments:

Location: Source address

Package\_list: Packages to be searched for nearest neighbor

For each package in package\_list:

Measure distance from location to Hub

Measure distance from package to Hub

If distance to package is closest found so far AND

package is farther from hub than current location:

Store distance and package

Return [closest package, distance]

## B2: Development Environment

This program was developed on PyCharm Community 2020.3.2, running on a Windows 10 machine. No external modules were imported beyond the provided data tables.

## B3: SPACE-TIME AND BIG-O

1. Decode input tables (runs in O(*n2*) where *n* is the number of addresses in data table)
   1. Importing the table of packages runs in O(*n*) time.
   2. Importing the table of distances runs in O(*n2*) time and will run slowly for a sufficiently large number of addresses. For the constraints of this project (40 addresses) this is acceptable.
2. Load trucks (runs in O(*n2*) time for large *n* where *n* is the number of packages)
   1. Each package may be compared to all remaining packages to find its nearest neighbor.
3. Dispatch trucks (runs in O(*n*) time for large *n* where *n* is the number of packages)
   1. Each package is touched once as it is delivered.

Overall, the program runs in O(*n2*) for large *n* where *n* is the number of packages. An input could be constructed to improve this down to O(*n*) but this would be an unrealistic input with each package having manually tuned deadlines and specifically requested trucks for loading.

## B4: ADAPTABILITY

This application will scale well within the given scenario. While it runs in O(*n2*) time based on the number of packages provided, this number would not realistically grow into the thousands for a given hub of this small company.

For larger companies, this application will not scale well. As the number of addresses and packages grow this application will not be able to break the problem down into manageable pieces. The application would certainly be able to function and produce a result, but the time required could grow to be excessive. The largest shortcoming would be in the handling of the distance table. Unless the table is pruned to only include addresses pertaining to packages it would quickly become too large to store and process.

## B5: SOFTWARE EFFICIENCY AND MAINTAINABILITY

This application is both efficient and maintainable. It runs in polynomial time and, for this scenario, would be able to run most of its functions overnight before trucks were loaded. The hash table used for storing packages scales in size to match the number of packages, conserving memory space by only using the minimum required.

For maintainability, the program’s functions are split into separate modules that can be replaced or upgraded as needed. Data import, package loading, and truck dispatching are all separated. The underlying algorithm can also be replaced without rewriting the entire application.

## B6: SELF-ADJUSTING DATA STRUCTURES

The PackageHash class found in packagehash.py is a self-adjusting data structure. It initializes to a list of size 20 and resizes in O(1) time if a key exceeds its size. The table uses direct addressing, mapping the package id to the table index. The strengths of this data structure are that it prevents collisions (assuming package ids are unique) and resizes as needed. The weaknesses of this data structure are that it inefficiently handles non-consecutive package ids by creating empty table space in the missing ids and it does not scale itself back down if packages are removed from the table.

## C: ORIGINAL CODE

The application delivers all packages on time with a total mileage of 121.4 and a finish time of 12:10 PM.

## D: DATA STRUCTURE

*Identify a data structure written by the student using only primitive data structures, lists, tuples, or sets, used by your program to store and retrieve package data.*

This data structure must be same the hash table in parts E and F. The official task directions include a note:

“Do NOT use any existing data structures...”

This should read, “use only built-in data structures.” Submitted code may use anything from [Python’s standard library](https://docs.python.org/3/library/), including the built-in data structures (e.g., lists, tuples, sets, and dictionaries). The only exception is the hash table, where the use of dictionaries is prohibited.

Per parts E and F, the hash table is required to have the following:

* E: an insertion function that includes as input all a package’s info (see below).
* F: a look-up function that includes as input a package’s ID and returns the corresponding package’s info (see below).

The abilities to store and retrieve package info (via the package’s ID) are the only requirements. The information can be stored in an object and can include additional parameters, e.g., special notes, time the package left the hub, etc.

The insert function (Part E) and look-up function (Part F) must respectively store and retrieve the following information:

* package ID number
* delivery address
* delivery deadline
* delivery city
* delivery zip code
* package weight
* delivery status (at the hub, en route, or delivery time)

## D1: EXPLANATION OF DATA STRUCTURE

*The submission accurately explains the data structure and how that data structure accounts for the relationship between the data points to be stored.*

Provide an explanation that describes the logic of the hash table and how it is used in the context of solving the problem.

## E: HASH TABLE

*The hash table has an insertion function that includes, as input, all of the given components.*

## The provided hash table must include an insertion function which can insert all the packages info (see part D) into the hash table.

NOTE: Submitted code may use anything from [Python’s standard library](https://docs.python.org/3/library/), including the builtin data structures (e.g., lists, tuples, sets, and dictionaries), except for the hash table where only the use of dictionaries is prohibited.

## F: LOOK-UP FUNCTION

*The look-up function includes all of the given data elements, completes searches via package ID, returns the data corresponding to the provided ID including the package’ status (at the hub, en route, or delivery time).*

The provided hash table should include a look-up function which can use a package's ID to retrieve all the packages info (see part D) from the hash table.

NOTE: Submitted code may use anything from [Python’s standard library](https://docs.python.org/3/library/), including the builtin data structures (e.g., lists, tuples, sets, and dictionaries); except for the hash table where only the use of dictionaries is prohibited.

## G: INTERFACE

## *Provide an interface for the user to view the status and info of any package at any time.*

## G1-G3: 1st, 2nd, and 3rd status checks.

*Provide screenshots showing the info (outlined in part F) and statuses at a time between:*

* *8:35 a.m. and 9:25 a.m.*
* *9:35 a.m. and 10:25 a.m.*
* *12:03 p.m. and 1:12 p.m.*

Provide one screenshot within each of the time intervals above. The screenshot can be included anywhere in your submission, e.g., the document, separately, in the project folder.

## H: SCREENSHOTS OF CODE EXECUTION

*Provide a screenshot or screenshots showing successful completion of the code free from runtime errors or warnings.*

Provide a screenshot or screenshots so that the evaluator can check that your code ran on your machine successfully to completion. The screenshot(s) should include a view of the console output, the project files, etc. The screenshot can be added anywhere in your submission, e.g., the document, separately, in the project folder.

## I1: STRENGTHS OF THE CHOSEN ALGORITHM

*The description includes at least two specific strengths of the chosen algorithm as they apply to the scenario.*

“Chosen algorithm” refers to the core algorithm identified in Part A.

## I2: VERIFICATION OF ALGORITHM

*The verification includes the total miles added to all trucks, and it states that all packages were delivered on time.*

You must verify the total miles traveled by all the trucks is under 145 miles and state that all packages were delivered on time and according to their constraints. Evaluators should be able to verify the mileage and deliveries via the user interface. So the total mileage must be provided via the user interface or console output.

## I3: OTHER POSSIBLE ALGORITHMS

*The submission identifies two other algorithms that could meet the requirements of the scenario.*

The two alternative algorithms should be different than the algorithm identified in Part A

## I3A: ALGORITHM DIFFERENCES

*The description includes attributes of each algorithm identified in part I3 and how the identified attributes compare to the attributes of the algorithm used in the solution.*

The two alternative algorithms should be compared to the algorithm identified in Part A.

## J: DIFFERENT APPROACH

*The description includes at least one aspect of the process that the candidate would do differently and includes how the candidate would modify the process.*

## K1: VERIFICATION OF DATA STRUCTURE

*The verification shows all the criteria have been met: the least number of total miles added to all trucks, all packages were delivered on time, the hash table with look-up function is present, and the reporting needed is accurate and efficient.*

This section should be titled “Verification of Data Structure and Solution.” Provide evidence that the hash table (identified in Part D) and delivery solution meets the following criteria:

* Completes in 145 miles or less.
* Delivers all packages on time and according to their constraints outlined in the package notes.
* Uses a hash table with a look-up function as described in Part F.
* Reports on package statuses and mileage accurately and in a user-friendly manner.

This requirement is redundant to previous requirements. Reciting evidence or citing previously written sections is acceptable.

## K1A: EFFICIENCY

*The description of the efficiency of the data structure (hash table) used in the solution includes what type of data is being used and how that data is being used.*

The “data structure used in the solution” refers to the hash table identified in Part D. Describe the type of data used and how your program uses that data. Provide a justification that the hash table makes the program more efficient.

## K1B: OVERHEAD

*The explanation of the data structure (hash table) includes the computational time, memory, and bandwidth aspects when handling data in this program.*

Computational time means time-complexity for handling data. This will include reading, storing, and finding package info from your hash-table (there will be some redundancy here from Part K1A). Bandwidth and memory aren’t much of a concern as everything is run from a local machine. However, just as with “communication protocol” in Part B2, you need to mention it and explain why as it’s in the rubric.

## K1C: IMPLICATIONS

*In regards to the data structure (hash table), describe the implications when more packages are added to the system or other changes in scale occur.*

Describe how adding more packages, cities, or trucks impacts your hash table performance (scaling). The discussion can include possible shortcomings.

## K2: OTHER DATA STRUCTURES

*The submission identifies two data structures other than the one used in the solution that meets the criteria and requirements in the scenario.*

“one used in the solution” refers to the data structure in Part D. Identify two alternative data structures and justify that they could be used for the project.

## K2A: DATA STRUCTURES DIFFERENCES

*The description includes the attributes of each data structure identified in part K2 and compares these attributes to the attributes of the data structure used in the solution.*

## L: SOURCES

*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.*

You must follow ([APA standards](https://wgu.mindedgeonline.com/content.php?cid=48184)). Contact the writing center for questions or help with this portion. If no sources were used, include a sources section in your documentation explaining that no sources were used. Every source listed must have a matching in-text citation; sources not warranting an in-text citation should be excluded. Code sources should be included in code comments near their citation.

## M: PROFESSIONAL COMMUNICATION

*The content reflects an attention to detail, is organized, and focuses on the main ideas as prescribed in the task or chosen by the candidate. Terminology is pertinent, is used correctly, and effectively conveys the intended meaning. Mechanics, usage, and grammar promote accurate interpretation and understanding.*

The submitted document should be grammatically correct and easy to read. Make use of one of the many freely available [grammar checkers](https://app.grammarly.com/) and/or the [writing center](https://my.wgu.edu/success-centers/writing-center).