## A: ALGORITHM SELECTION

Identify by name the self-adjusting algorithm used to create a program to deliver the packages and meet all requirements specified in the scenario.

*You must identify the self-adjusting part(s) of your algorithm by name. Though you may have written your algorithm using a variety of approaches, you should be able to connect or generalize part of your algorithm to a commonly recognizable method, e.g., a greedy algorithm, farthest neighbor algorithm, etc.*

*What is self-adjusting? Any code which adjusts dynamically according to input, i.e., you give the code package info, and the code makes decisions. For example, if you manually choose ten packages and then your code decides how to deliver those packages,*

* *The code choosing the packages is not self-adjusting.*
* *The code deciding how to deliver the packages is adjusting.*

*The identified (also called “chosen” in Part I1 of the rubric and “core” in Part I of the directions) algorithm must be self-adjusting. Using non-self-adjusting methods is acceptable, but at least one part must use a self-adjusting algorithm.*

## B1: LOGIC COMMENTS

The submission accurately explains the algorithm’s logic using pseudocode.

*Provide an outline of how your code finds a solution. You can do this in your document or code comments, though we recommend the former.*

*There is no formal way to write pseudocode (or it would not be pseudocode). Following the example found in the* [*sample algorithm document*](https://drive.google.com/open?id=1EMv2teFrmjC_5WZ23wxwSS3Te2l_h8Lm) *is one way to fulfill this requirement. Still, you certainly need a step-by-step explanation (via pseudocode or bullet points) outlining how your code finds a solution.*

## B2: DEVELOPMENT ENVIRONMENT

This program was written on an Intel i7, Lenovo Ideapad, laptop. Software used includes Windows 11, Microsoft Office, and PyCharm IDE. Version control was handled by Git, Git Bash, and Github. The program was written in Python 3.

## 

## B3: SPACE-TIME AND BIG-O

|  |  |  |
| --- | --- | --- |
| Method/Part name | Time Complexity | Space Complexity |
| 1. Beginning interface | O(1) | O(1) |
| 2. Package.get\_number\_of\_packages | O(n) | O(n) |
| 3. Package.create\_package\_objects | O(n) - O(n^2) | O(n) |
| 4. HashTable.put | O(1) - O(n) | O(n) |
| 5. Address.put\_addresses\_in\_city\_map\_matrix | O(n) | O(n) |
| 6. Address.put\_distances\_in\_array | O(n) | O(n) |
| 7. Address.put\_distances\_in\_city\_map\_matrix | O(n^2) | O(n) |
| 8. Update package #9 part | O(1) | O(1) |
| 9. Truck.update\_truck\_in\_hashmap | O(log(n)) | O(n) |
| 10. Truck.truck\_start\_time | O(n) | O(n) |
| 11. DeliveryAlgorithm.get\_ordered\_list | O(n log n^2) | O(n log n^3) |
| 12. DeliveryAlgorithm.get\_address1 | O(1) | O(1) |
| 13. DeliveryAlgorithm.get\_distance\_between\_addresses | O(1) | O(1) |
| 14. Time.get\_delivery\_times | O(log(n)) | O(log(n) |
| 15. Truck.truck3\_start\_time | O(1) | O(1) |
| 16. DeliveryAlgorithm.truck\_mileage | O(1) | O(1) |
| 17. Total Mileage calculation | O(1) | O(1) |
| 18. Ending interface | O(1) | O(n) |
| 19. Time.convert\_distance\_to\_time\_delta | O(1) | O(1) |
| 20. HashTable.update | O(n) | O(1) |
| 21. HashTable.get | O(1) | O(n) |
| 22. HashTable.hash\_function | O(1) | O(1) |
| 23. | O() | O() |
| 24. | O() | O() |
| 25. | O() | O() |
| 31. Overall | O() | O() |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

The (code) submission accurately describes the space-time complexity for each major block of code and the entire program using Big-O notation.

*Include this in code comments. “Major block” of codes and “accuracy” is not defined. Ask yourself, “if I was looking at this code for the first time, what would I want to be described? How much detail would I need?”*

*How accurately does the time complexity description need to be? Big-O classifications are sets of functions defined by an upper bound. e.g., a function that is in O(1) is also in O(n). However, it is reasonable for evaluators to expect better accuracy in obvious cases, e.g., if it’s constant -write O(1). For more complicated situations, less accuracy should be acceptable.*

*If you are uncertain about whether comments or accuracy is sufficient -submit it. At worst, the evaluator will ask for more.*

## B4: ADAPTABILITY

The submission application accurately explains the application’s capability to scale and adapt to an increasing number of packages.

*The application will have at least two scalable elements, the self-adjusting algorithm from* [*Part A*](#_6slq3j785okr) *and the self-adjusting data structure (hash-table) from* [*Part D*](#_e5lmjqcaaava)*. The discussion can also include shortcomings, e.g., “X will not scale well because…”*

## B5: SOFTWARE EFFICIENCY AND MAINTAINABILITY

The discussion addresses how the software is efficient and easy to maintain.

*By “efficiency,” they mean the Big-O time complexity of your entire program. Your program must run in polynomial time or better to be efficient.*

*For “maintainability,” describe how future developers can easily understand, repair, and enhance your code, i.e., -*[*software maintainability*](https://www.sealights.io/software-quality/software-maintainability-what-it-means-to-build-maintainable-software/)*—for example, code structure, comments, compartmentalization, etc.*

## B6: SELF-ADJUSTING DATA STRUCTURES

The discussion addresses the strengths and weaknesses of the self-adjusting data structures (including the hash-table).

Include all self-adjusting (non-builtin) data structures in the discussion. However, you are only required to have one such data structure -the hash-table from [Part D](#_e5lmjqcaaava). The rubric specifies “strength-s” and “weaknesses.” So include at least two for each.

## C: ORIGINAL CODE

All code was written by myself, with algorithm help referenced as appropriate in the reference section below. Only the standard python library was used. It runs without errors.

## C1: IDENTIFICATION INFORMATION

Rebecca McDaniel, Student ID: 001426057

See main.py, line 9.

## C2: PROCESS AND FLOW COMMENTS

Comments are scattered throughout the program.

## D: DATA STRUCTURE

The submission identifies a self-adjusting data structure that can store the package information and perform well with Part A’s algorithm.

*As with* [*Part A*](#_6slq3j785okr)*, self-adjusting means any code which adjusts dynamically according to input. This can include adjusting to size (lists are mutable in Python) or searches. For example, a hash table that can adapt to more packages without rewriting the code would be self-adjusting. This data structure must be the same hash-table in* [*Part E*](#_1l9uepa07fn2) *and* [*Part F*](#_t3406fuvv9c)*. Nothing else regarding the complexity of your hash table is required, e.g., it can be a 1-1 mapping, does not have to handle collisions, have chaining, etc.*

*The official task directions include a note:*

*“Note: Use only appropriate built-in data structures, except dictionaries.”*

*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 (a dictionary is a hash-table).*

*Per parts E and F, the hash table must have the following:*

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

*The ability to store and retrieve package info (via the package’s ID) is the only requirement. The information can be stored in an object and 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 how the data structure (hash-table) uses package IDs to store and retrieve package information.

*Provide an explanation that describes the hash table’s logic, i.e., how it stores and retrieves package information. You should include a description of why your hash-table retrieves information accurately and more efficiently than a simple linear search.*

## E: HASH TABLE

The hash table has an insertion function that stores all of the given components (listed in [Part D](#_e5lmjqcaaava)) using the package ID as the key.

## F: LOOK-UP FUNCTION

The provided hash table should include a look-up function that can use a package's ID to retrieve all of the same package’s components from the hash table (listed in [Part D](#_e5lmjqcaaava)).

*The package ID must be the key, and the components from* [*Part D*](#_e5lmjqcaaava) *must be the values. There are no other specifications, and you can choose how the values within the hash-table are stored. For example, the components could be in an object or a different data structure of your choosing.*

## G: INTERFACE

## Provide an intuitive interface for the user to view the status and information (as listed in [Part D](#_e5lmjqcaaava)) of any package at any time and the total mileage traveled by all trucks. The delivery status should report the package as “at the hub,” “en route,” or “delivered at \_\_\_\_.” Delivery status must include the time.

*This can be done through a simple console interface (it might be preferable). The evaluators need an easy way to check that all packages were delivered according to their constraints using under 140 miles. “Intuitive” means that evaluators can do this through the interface without reading your code.*

*To check package statuses, you should be able to enter a time and check all packages’ statuses* (delivered at X, at the hub, or en route). For example, in the console, it might look like:

Enter a time after 08:00:00 (in military time, i.e., 0900) to check package status or 'X' to exit:

0950

Package ID: 1, Address: 195 W Oakland Ave, ….Delivered at 08:46:20

Package ID: 2, Address: 2530 S 500 E At Hub, …. At the hub.

.

.

.

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

*Provide screenshots showing the information (as listed in* [*Part D*](#_e5lmjqcaaava)*) and statuses (see* [*Part G*](#_7mhkwjcp0dwg)*) 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. If everything does not fit on a single screenshot, you can use multiple images; label them, so it’s clear which time interval they cover, e.g., “G1-a.jpg, G1-b.jpg,...” The screenshots can be included anywhere in your submission, e.g., the document, separately, in the project folder -but make them easy to find.

## H: SCREENSHOTS OF CODE EXECUTION

Provide a screenshot or screenshots showing the total delivery mileage and successful completion of the code free from runtime errors or warnings.

*Provide a screenshot or screenshots so that the evaluator can check the mileage and that your code ran successfully to completion. The screenshot(s) should include a view of the console output, the project files, etc. The screenshot(s) can be added anywhere in your submission, e.g., the document, separately, in the project folder -but make it easy to find.*

## I1: STRENGTHS OF THE CHOSEN ALGORITHM

Identify at least two specific strengths of the algorithm from [Part A](#_6slq3j785okr) relevant to finding a solution.

*Strengths can include anything, e.g., efficiency, simplicity, adaptability, etc. You also need to justify that these are strengths relative to the project.*

## I2: VERIFICATION OF ALGORITHM

Verify that the algorithm used in the application meets all the requirements by:

* Provide the total combined miles traveled by all trucks. It must be less than 140.
* State that all packages were delivered on time.
* State that all packages were delivered according to their delivery specifications.
* Describe how all the above points are verifiable through the user interface.

*By “requirements,” they mean requirements of the solution -not requirements of the algorithm listed in Part A. Evaluators should be able to verify the mileage and deliveries via the user interface. So this section may seem a little odd as it equates verification with “stating,” but that is what they ask you to do.*

## I3: OTHER POSSIBLE ALGORITHMS

The submission identifies two algorithms different from the one provided in [Part A](#_6slq3j785okr) that could meet the scenario’s requirements.

*The two alternative algorithms only need to be different from the algorithm identified in Part A; they do not need to be equitable or better in completely-known. Furthermore, these two algorithms could apply to any portion of the application. The problem of finding a delivery route is known as the “Traveling Salesperson Problem” or TSP. An old and well-known problem, there are many, many approaches to this problem.*

## I3A: ALGORITHM DIFFERENCES

*The description includes attributes of each algorithm identified in* [*Part I3*](#_6m6o2ryhezd0) *and how the identified attributes compare to the algorithm’s attributes from* [*Part A*](#_6slq3j785okr)*.*

Compare the two alternative algorithms to the algorithm identified in [Part A](#_6slq3j785okr). Attributes, and the comparison can include almost anything, e.g., time-complexity, advantages, disadvantages, etc. The rubric writes “attribute-s.” So you should list at least two attributes per algorithm list in [Part I3](#_6m6o2ryhezd0).

## J: DIFFERENT APPROACH

This program could be improved in the future by making the date changeable. Currently, the time is given by the user, but it is only running for one date: 1/22/2022. In theory, this program could accept different dates and port in information from different CSV files as appropriate. See main.py line 20.

*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

Verify that the data structure used in the application meets all the requirements by:

* Provide the total combined miles traveled by all trucks. It must be less than 140.
* State that all packages were delivered on time.
* State that all packages were delivered according to their delivery specifications.
* State that an “efficient” hash-table (Part D) with a look-up function (Part F) is present.
* State that the “reporting” (package statuses and information) can be verified through the user interface and that all information is accurate (Part G).

*This section looks to have been cut and pasted from* [*Part I2*](#_l8kk3ejnpg8n)*. It is confusing that you are asked to re-verify mileage and delivery statuses, particularly as these items are not relevant to the hash table. However, these items are specified in the rubric. When the rubric is redundant -be redundant.*

*By “requirements,” they mean requirements of the solution -not requirements of the data structure listed in* [*Part D*](#_e5lmjqcaaava)*. Evaluators should be able to verify the mileage and deliveries via the user interface. Other parts are addressed in previous sections. So this section is a little odd as it equates verification with “stating,” but that is what they ask you to do.*

## K1A: EFFICIENCY

The discussion accurately explains how adding packages directly affects the time needed to complete the look-up function.

*The “look-up function” refers to the hash table’s look-up function identified in Part D. Describe how adding more packages affects the time it takes to retrieve package information. The effect could be nil, as in the case of a direct (1-1) mapping. Whatever the case, provide a brief explanation.*

## K1B: OVERHEAD

My hash table implementation’s size, at least in regards to the length of the lists, is constant from its creation. When the program is run, the program uses CSV reader to count the number of the packages in the provided CSV and creates a hash table that size. See Packages.py, get\_number\_of\_packages(), and how that argument is returned and used in the creation of “package\_hashtable = HashTable(number\_packages)”. The only space added as packages are added, is the package data itself. The package data itself is a list of 1 key and 10 items. See Package.py: “hashmap.put(i, [line[1], line[2], line[3], line[4], line[5], line[6], line[7], truck\_num, time\_stamp, status])”. The space taken is the base empty list + 11\*n for each package inserted.

## K1C: IMPLICATIONS

The discussion accurately explains adding trucks or cities would affect look-up time and space usage.

*Look-up and space usage are both referring to the hash-table. Depending on your code, additional cities or trucks may not affect hash-table performance. In which case, you should explain why.*

## K2: OTHER DATA STRUCTURES

The submission identifies two data structures other than the one used in [Part D](#_e5lmjqcaaava) to meet the requirements in the scenario.

*Identify two alternative data structures and justify why they could have been used as your hash-table. The alternatives can include modifications of the hash table listed in* [*Part D*](#_e5lmjqcaaava)*.*

## K2A: DATA STRUCTURES DIFFERENCES

The submission accurately describes attributes of both data structures identified in [part K2](#_g9al3fuucalf) and compares these to [Part D](#_e5lmjqcaaava)’s data structure attributes.

*Attributes and the comparison can include almost anything, e.g., mappings, structure, advantages, disadvantages, etc. The rubric writes “attribute-s.” So you should list at least two attributes per data structure from* [part K2](#_g9al3fuucalf)*.*

## L: SOURCES

AskPython. (2021, June 25). *3 ways to sort a dictionary by value in python*. AskPython. Retrieved January 26, 2022, from https://www.askpython.com/python/dictionary/sort-a-dictionary-by-value-in-python

Drowell, E. (n.d.). *Know thy complexities!* Big-O Cheat Sheet. Retrieved January 26, 2022, from https://www.bigocheatsheet.com/

GeeksforGeeks. (2020, July 28). *String alignment in Python f-string*. GeeksforGeeks. Retrieved January 26, 2022, from https://www.geeksforgeeks.org/string-alignment-in-python-f-string/

Hellman, D. (2011, July 11). *PYMOTW*. datetime – Date/time value manipulation - Python Module of the Week. Retrieved January 26, 2022, from http://pymotw.com/2/datetime/

James, J. (2016). *Python: Creating a HASHMAP using Lists*. Youtube. Retrieved January 26, 2022, from https://www.youtube.com/watch?v=9HFbhPscPU0.

James, J. (2019). *Python: Dates, Times & Timestamps Part-1 | datetime, time libraries*. Youtube. Retrieved January 26, 2022, from <https://www.youtube.com/watch?v=hUkDhkrUNys&t=459s>.

Lusby, J. (2022, January 14). C950 Outreach... Email with overall program structure tips.

Lysecky, R. (n.d.). *ZyBooks My library > C950: Data Structures and Algorithms II home > 6.6: Python: Graphs*. Building the Graph and Vertex classes Retrieved January 26, 2022, from https://learn.zybooks.com/zybook/WGUC950AY20182019/chapter/6/section/6?content\_resource\_id=29588064.

phihagphihag 257k6666 gold badges427427 silver badges455455 bronze badges, & martineaumartineau 108k2323 gold badges148148 silver badges263263 bronze badges. (1959, February 1). *How to sum all the values in a dictionary?* Stack Overflow. Retrieved January 26, 2022, from https://stackoverflow.com/questions/4880960/how-to-sum-all-the-values-in-a-dictionary

WGU C950 course documents for handling time steps and the greedy algorithm. https://srm--c.na127.visual.force.com/apex/coursearticle?Id=kA03x000001DbBGCA0. (n.d.). C950 WGUPS Project Implementation Steps - Example. https://srm--c.na127.visual.force.com/apex/coursearticle?Id=kA03x000001DbBGCA0.