COSC122 2022 Semester 2
INTRODUCTION TO ALGORITHMS

Assignment 2: Tracking Flagged Cars

7% of overall grade
Due Friday 30 September (end of week 9), 11:55pm

1 Overview

In this assignment you will be writing a program to do tracking as in Assignment 1A and 1B, but this time using hash tables to do the searching, and with time stamps to keep track of as well. A key goal of this assignment is to give you the experience of implementing code to a specification, and gain direct experience with the relevant benefits of different algorithms for the same task.

1.1 Due date

Your assignment should be submitted by Friday 30 September (end of week 9), 11:55pm. It can be handed in up to 7 days later, but any assignment handed in during the 7 days will incur a 15% absolute penalty. That is, your mark will be the raw mark less 15 marks (given the assignment is marked out of 100).

1.2 Submission

Submit via the quiz server. The submission page will be open closer to the due date. As you will be aware from the first assignment, you should start early so that you have time to understand the requirements and to debug your code.

1.3 Implementation

All the files you need for this assignment are on the quiz server. You must not import any additional standard libraries unless explicitly given permission within the task outline. For each section there is a file with a function for you to fill in. You need to complete these functions, but you can also write other functions that these functions call if you wish. All submitted code needs to pass the PYLINT style check before it is tested on the quiz server. *Please allow extra time to meet the style requirements for this assignment.*

1.4 Getting help

The work in this assignment is to be carried out individually, and what you submit needs to be your own work. You must not discuss code-level details with anyone other than the course tutors and lecturers. You are, however, permitted to discuss high-level details of the program with other students. If you get stuck on a programming issue, you are encouraged to ask your tutor for help. You may not copy material from books, the internet, or other students. We will be checking carefully for copied work.

If you have a general assignment question or need clarification on how something should work, please use the class forums on Learn, as this enables everyone to see responses to common issues, but **never** post your own program code to Learn (or make it available publicly on the internet via sites like GitHub)[1]. Remember that the main point of this assignment is for you to exercise what you have learned from lectures and labs, so make sure you do the relevant labs first, and don't cheat yourself of the learning by having someone else write material for you.

[1]Check out section 3 of UC's *Academic Integrity Guidance* document. Making your code available to the whole class is certainly not discouraging others from copying!

2 Tracking Flagged cars

This assignment follows on from the first assignment and will be primarily focused on the use of hash tables to provide fast look-up and storage of data. You will again be working with the data from a roadside camera based Automatic Number Plate Recogniser (ANPR), but in this assignment each camera sighting will come with a time stamp and you will be aiming to build up a table containing the list of times that each *flagged* vehicle was seen by the camera.

2.1 General Overview

You will complete a program that:

- Accepts a provided database list of number plates, with *flags*:
  - You will need to store the database in a hash table for quick look-up.
  - Vehicles of interest will be flagged with descriptions such as `stolen`, `outstanding fines`, etc....
- Accepts a second list of number plates, with the *time* they were sighted.
- Using this, you will build up another hash table that maps *flagged* number plates to the time(s) they were sighted by the camera.

To do this you will need to:

- Build *Linear Probing* and *Chaining* hash tables that operate in a similar way to Python dictionaries. They would be used as follows:

  `my_table[plate] = flag`

  `flag = my_table[plate]`

  `plate in my_table`.
- Write code for storing the plate database, ie, storing flags for plates so that the flags can be accessed using `flag = database_table[plate]`
- Write code that builds a chaining hash table containing the time stamps for sightings of vehicles that have flags (ie, are interesting).
  - Using a chaining hash table will allow the table to handle as many plates as you want to store (without getting full and needing to be resized).
  - Once built looking up the list of sightings for a given number plate should be quick, eg, `sighting_list = results_table[NumberPlate]` will be fast.

2.2 Provided Skeleton Code

In this assignment you will be completing the implementation of two hash table classes and two functions (plus a little bit of counting in the `ListTable` class). The skeleton code for these is provided in the `hashing.py` module. More details on what you need to do is provided in the Task section below (and in the docstrings).

We also provide a class called `ListTable` that gives you an example of how your hash tables should be working. It uses a list to store items and searches the list with a sequential algorithm, rather than using hashing. This class will work for small data sets but it's main value is that it demonstrates the horrible slowness of such sequential/linear searching!

2.3 Changes to NumberPlate

You can now hash number plates using `hash(my_plate)`. The hash method that is provided will use a small fast hashing function that will give the same value across different runs of your program. Otherwise, `NumberPlate` behaves the same as before.

In this assignment you are expected to count both how many `NumberPlate` comparisons **and** hashes that your hash tables use. More information is in section 4.1 and in the tasks section below.

3 Tasks [100 Marks Total]

3.1 Linear Probing Hash Table [30 Marks]

Check out the `ListTable` class to get a feel for the worst case way of making a look-up table. The code is working but you **must** add in some code to keep track of any plate comparisons that are made (eg, insert some `self.n_plate_comparisons += 1` lines where appropriate).

Once warmed up, complete the `LinearHashTable` class:

- The data should be stored in the `table_list` instance variable, as set up in the provided `__init__` method.
  - When empty, the value of a slot is `None`.
  - When full, the value of a slot will be a `(key, value)` tuple.
- Interacting with this class will work like a Python dictionary, to do this implement the methods (see Example Usage section):
  - `__set_item__`: to set an item, ie, `table[plate] = flag`,
  - `__get_item__`: get a key's value ie, `table[plate]`,
  - `__contains__`: check if value in table, ie, `plate in table`.

3.1.1 General Notes

- Count all comparisons between two `NumberPlate` objects in the `n_plate_comparisons` instance variable (eg, do a `self.n_plate_comparisons += 1` each time number plates are compared.

- Also count the number of times number plates are hashed using the `n_plate_hashes` instance variable.
- It will be important to keep track of the number of items stored in the hash table (via `n_items`) – this instance variable can then be used to tell when the table is full.
- Some examples of usage are provided in the example functions in the `hashing` module. Feel free to play around with them to help you get a feel for how your tables are operating.
- Now might also be a good time to complete the code for the `generate_db_hash_table` function as it will help you test your hash table. See topic 3.3 below.

3.1.2 Example Usage

Note that the following example uses a hash table with 11 slots, with would be ridiculously small in practice, but useful for testing and debugging with small amounts of data.

```
plate = NumberPlate('ABC231')
flag = 'Overdue fines'
table = LinearHashTable(11)
table[plate] = flag      # translates to table.__set_item__(flag)
print(plate in table)    # translates to table.__contains__(flag)
print(table[plate])      # translates to table.__get_item__(plate)
table[plate] = 'new_flag' # translates to table.__set_item__(plate)
plate2 = NumberPlate('BBB222')
table[plate2] = 'BBB222-flag'
#
print(hash(plate)%11) # gives 7
print(hash(plate2)%11) # gives 9
# so we expect the plates to be in indexes/slots 7 and 9
print(table)
# Printing the table should give the following
Linear Hash Table:
0-> None
1-> None
2-> None
3-> None
4-> None
5-> None
6-> None
7-> ('ABC231', 'new_flag')
8-> None
9-> ('BBB222', 'BBB222-flag')
10-> None
```

3.2 Chaining Hash Table [30 Marks]

Complete the implementation of the `ChainingHashTable` class:

- To store the *chain* of items in each slot it uses a Python `list`.
  - The initialiser method will set up the `table_list` with an empty Python `list` in each slot (ie, [] in each slot).
  - To add an item, append its `(key, value)` tuple to the appropriate slot.
- The interface is same as the `LinearHashTable` class but under the bonnet it will use *Chaining* to resolve hash collisions. Implement the methods:

  `__set_item__`: to set an item, ie, `table[plate] = flag`

  `__get_item__`: get a values key ie, `table[plate]`

  `__contains__`: check if value in table, ie, `plate in table`

3.2.1 General Notes

- Count all comparisons between two `NumberPlate` objects in the `n_plate_comparisons` instance variable.
- Also count the number of times number plates are hashed using the `n_plate_hashes` instance variable.
- Because chaining is being used the table can contain more items than slots. That is, `n_items` can be greater than `n_slots` or in terms of the load factor, having $\lambda > 1$ is possible.
- Some examples of usage are provided in the example functions in the `hashing` module. Feel free to play around with them to help you get a feel for how your tables are operating.
- Now might also be a good time to complete the code for the `generate_db_hash_table` function as it will help you test your hash table. See topic 3.3 below.

3.2.2 Example Usage

The example below shows what a table will look like after adding a few flags.

```
plate1 = NumberPlate('BAA754')
plate2 = NumberPlate('MOO123')
plate3 = NumberPlate('WOF833')
plate4 = NumberPlate('EEK003')

table = ChainingHashTable(5)
table[plate1] = 'Sheep'
table[plate2] = 'Cow'
table[plate3] = 'Dog'
table[plate4] = 'Mouse'
```

```
print(table)
# gives
0-> [('WOF833', 'Dog')]
1-> [('BAA754', 'Sheep'), ('MOO123', 'Cow')]
2-> []
3-> []
4-> [('EEK003', 'Mouse')]
```

Note: Our testing will include using a `ChainingHashTable` to store database data, but the final usage of the `ChainingHashTable` will use `NumberPlates` as the key and lists of time stamps as the values. See the next section for details and examples of this.

3.3 Processing Camera Sightings [30 Marks]

3.3.1 Building the Database Table

Complete the `generate_db_hash_table` function :

- The database table is generated by going through all the `(plate,flag)` pairs in the database list and adding/storing them to the database table.
- You should test building database tables with both Linear Probing and Chaining hash tables.
- Expected output for each is given in the files prefixed with `expected_db_linear_` and `expected_db_chaining_`. Have a look at a few of those files, in the `test_data` folder, to see what the resulting database tables look like.

3.3.2 Building the Results Table

Complete the `process_camera_sightings` function:

- The `process_camera_sightings` function takes a *database* list and a *sighted* list, followed by the size of the table to use for the database and the size of the table to use for storing the timestamps of sighted flagged vehicles.
  Note: The timestamps/dates are Python strings (`str`). You do not need to modify these for storage.
- The results table is generated by going through all the plates in the sighted list and recording the time stamp for each plate that has a flag in the database.
  Note: do not include plates with an empty tag.
- If you think of the results table as a dictionary: the keys are plates and the value associated with each key will be a list of time stamps.
- The `process_camera_sightings` function should return a tuple containing the resulting database table and the resulting sighted flagged vehicles table.

- The database table should be a `LinearHashTable` and the results table should be a `ChainingHashTable`.
- The example results table output files should give you a good idea of what the final results tables should look like. The contents of one of these files is given below.

3.3.3 Expected output files

For this task the expected results table files have the obvious prefix (`expected_results_table`) followed by the following values:

`{n_db}[s]-{n_sighted}-{n_flagged_stamps}-{seed}-{db_table_size}-{results_table_size}`

For example, `expected_results_table_10s-10000-10-a-20-5.txt` gives the expected results table when using the input from the 10s-10000-10 file with a database table of size 20 and a results table of size 5. Notice that each slot contains a list of tuples of the form `(plate, time_stamp_list)`.

```
Chaining Hash Table:
0-> [('BI7323', ['2017-05-22T16:54:23', '2017-05-23T10:33:45',
    '2017-05-25T12:34:24']), ('TQ5014', ['2017-05-23T11:03:17']),
    ('JI6816', ['2017-05-24T16:00:52'])]
1-> []
2-> []
3-> []
4-> [('KA2801', ['2017-05-22T15:24:38', '2017-05-23T06:13:42']),
    ('HT9954', ['2017-05-22T19:51:16', '2017-05-23T21:16:52',
    '2017-05-25T00:39:27'])]
```

3.4 General Hash Table Questions [10 marks]

Once opened, the submission quiz will ask you to submit answers to four questions (about hash-tables), each worth 2.5 marks. Make sure to allocate enough time to complete them. Like the previous assignment, there will be no check button–be sure that you are confident in your working.

4 Classes, Tools and Tests

4.1 Provided classes

Note: Time stamps will be represented using Python `str` (in ISO8601 format), eg, `date = '2022-05-22T19:51:16'`.

4.1.1 NumberPlate

The `NumberPlate` class, located in `classes2.py`, has **changed** since the last assignment:

- You can now hash number plates using `hash(my_plate)`. The hash method that is provided will use a small fast hashing function that will give the same value across different runs of your program.
- You should be counting how many `NumberPlate` comparisons **and** hashes that your hash tables use — via their `self.n_plate_comparisons` and `self.n_plate_hashes` instance variables. Note, this means your functions don't need to return the counts.
- `StatCounter.get_comparisons(HASHES)` returns how many hashes have been performed on `NumberPlate` objects.
- `StatCounter.get_comparisons(COMPS)` returns how many comparisons have been made between `NumberPlate` objects.

As per the previous assignment:

- `NumberPlate` is a repackaged `str` class, but with the added benefit of being able to keep track of every `NumberPlate` comparison that is made.
- All the comparison operators from `str` (==, <, >, <=, >=, `!=`) are available
  - if `a` and `b` are `NumberPlates`
  - expressions like `a < b` and `a == b`, etc, will work as expected.
- Every time a comparison occurs between two `NumberPlate` objects, a counter in the `StatCounter` class is incremented.

4.1.2 Important Notes on the StatCounter Class

- Note that because of the way it is implemented, it will not work on the Quiz Server!
- Only use the `StatCounter.get_comparisons` method for debugging! Remove all calls to it from the code you submit to the Quiz Server.
- Remove any imports related to `stats` from your code—otherwise *pylint* might get annoyed with the unused import.

⚠ **Important:** `StatCounter` will not work on the quiz server! You must do your own comparison counting to get marks.

4.2 Testing your code

There are two ways to test your code for this assignment (you are required to do **both**):

- Writing your own tests – useful for debugging and testing edge cases
- Using the provided tests in `tests.py` as a final check before you submit

The tests used on the quiz server will be mainly based on the tests in `tests.py`, but a small number secret test cases will be added, so passing the unit tests is not a guarantee that full marks will be given for that question.

Be sure to test various edge cases such as when an input list is empty or when no number plates are sighted.

4.2.1 Provided Data for Testing

The `utilities.py` module contains functions for reading test data from test files. *Test* files and *expected output* files are in the folder `test_data` to make it easier to test your own code.

The *expected output* files contain string representations of the hash tables expected from various test files. The specifics of these expected files is explained in Section 3.3.3 above.

The *test* data files are named `db[s]-sighted-matches-seed.txt`, where:

- *db* = the number of plates in the database
- *s* is optional and indicates they are in sorted order;
- *sighted* = the number of plate sightings by the camera (note each plate may be sighted more than once);
- *matches* = the number of sightings that were flagged plates (for you to check that you get the right ones);
- *seed* = the seed value that was used when generating the test file.

For example `10-20-5-a.txt` was generated with `a` as the random key and has a database of 10 plates with 20 plates being sighted by the camera and 5 of the plate sightings were flagged vehicles (note there may have been less than 5 vehicles sighted as this is the number of time stamped sightings and each vehicle can be sighted more than once). The list of sighted flagged plates is sorted by plate and then by time stamp. If the file name was `10s-20-5-a.txt` then the database would be provided in sorted order — most of the test data files will use sorted database entries (as this is likely the way that you would receive a database file).

You should open some of the test data files and make sure you understand the format—remember to open them in Wing or a suitably smart text editor. If you are using windows then try Notepad++ because the normal Notepad is pretty horrible. If you are using Linux (or a Mac) then almost any text editor will do.

The test files are generated in a quasi-random fashion and the seed suffix indicates the random seed that was used when generating the data—you don't need to worry about this. But, you do need to worry about the fact that we can easily generate different random files to test with.

4.2.2 Provided Tools for Testing

The `read_dataset` function in `utilities.py` reads the contents of a test file and returns three lists: a *database* list, a *sighted* list and a list of *matches* – the entries in both lists.

The following example shows how to use the `read_dataset` function from the `utilities` module:

```
# from utilities import read_dataset # if you need it for the example
# note you can ignore the indent below they are just typesetting in this document
filename = './test_data/5s-5-2-a.txt'
db_list, sighted_list, matches_list = read_dataset(filename)
print(db_list)
[('DZ4997', ''), ('HQ9423', ''), ('LA5930', 'Crusaders supporter'), ('RC3494', ''),
    ('SH5242', '')]
print(sighted_list)
[('SH5242', '2017-01-01T00:03:17'), ('HQ9423', '2017-01-01T00:03:52'), ('LA5930',
    '2017-01-01T00:04:11'), ('HN8360', '2017-01-01T00:04:34'), ('LA5930',
    '2017-01-01T00:04:42')]
print(matches_list)
[('LA5930', '2017-01-01T00:04:11'), ('LA5930', '2017-01-01T00:04:42')]
print(type(db_list[0]))
<class 'classes.NumberPlate'>
```

4.3 Provided tests

Off-line tests are available in the `tests.py` module:

- These tests are **not** very helpful for debuging - do your own smaller tests using hand workable examples when debugging!
- You should use them to check your implemented code before submission: the submission quiz *won't* fully mark your code until after submissions close.
- The provided tests use the Python `unittest` framework. You are **not** expected to know how these tests work, or to write your own `unittests`.

4.3.1 Running the Provided Tests

- The `all_tests_suite()` function has a number of lines commented out:
  - The commented out tests will be skipped.
  - Uncomment these lines out as you progress through the assignment tasks to run subsequent tests.
- Running the file will cause all uncommented tests to be carried out.
- Each test has a name indicating what test data it is using, what it is testing for, and a contained class indicating which algorithm is being tested.

- In the case of a test case failing, the test case will print which assertion failed or what exception was thrown.

⚠ **Important:** In addition to the provided tests you are expected to do your own testing of your code. This should include testing the trivial cases such as empty lists and lists of differing sizes.

5 Updates and clarifications

Please keep an eye on the new forums for any updates or clarifications that might come out during the course of the assignment.

— Have fun —