COSC122, 2020 Semester 2
INTRODUCTION TO COMPUTER SCIENCE

Assignment 2—Improved testing

7% of overall grade
Due 11:55pm on Friday 25 September 2020

1 Overview

1.1 Introduction

This assignment follows on from assignment one. The basic task will still be to generate a list of test results (or lack thereof) for a list of quarantined people. You will get to apply two different methods that improve on the best method in assignment one. In the first task you will finish the implementation of a hash table class and use it to provide very fast lookup of names in the tested data set. Then you will implement a *dual* index method that is based on the *merge* part of merge sort (and the common elements function you wrote in Lab 6.2).

1.2 Due date

The official due date is 11:55pm on Friday 25 September 2020.

Students can submit to the normal submission quiz up two days after this (ie, Sun 27 Sep) with no penalty (ie, normal submissions close on Sun 27 Sep). After this point submissions can be made until the drop dead date, which is Fri 2 October, via the *late* submission quiz. But *late* submissions incur a 15% absolute penalty, ie, if you submit to the *late* quiz then `your_mark = raw_mark - 15`. Please ask if you are unsure.

1.3 Submission

Submit via the quiz server. The submission page will be open about a week after the quiz is released—to emphasise the point that the quiz won't offer much help, your own testing and the provided unit tests are what you should be using to test your code. The submission quiz will not test your code properly until after submissions close, that is, it won't give you any more information than the provided tests so you can start work straight away! *This assignment is a step up from COSC121 so please make sure you start early so that you have time to understand the requirements and to debug your code.*

1.4 Implementation

All the files you need for this assignment can be found on the quiz server. Do 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 helper functions if you wish. All submitted code needs to pass *pylint* style checks—make sure you leave time for style adjustments when submitting.

1.5 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 or the lecturer 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, so don't cheat yourself of the learning by having someone else write material for you.

2 Looking up results

For each task of the assignment you will be required to write a function that takes a list of test results and a list of quarantined people, and returns a list containing the quarantined peoples' test results, or lack thereof. The functions will need to count the number of times `Name` objects are compared and return this count along with the list of results—as a measure of the work done.

You can assume that there are no duplicate names in either the *tested* list or the *quarantined* list, ie, within either list no name will appear more than once.

The file `tests.py` provides you with a suite of tests to help you check your implemented code before submission. These tests use the Python unit-test framework and *you are not expected to fully understand how they work*; you just need to know that the tests will check that your code generates the correct list of output, and that the number of comparisons that your code makes is appropriate. If you think that you have an implementation that is very close to the test range, and works reliably, you can ask to have the expected number of comparisons range reconsidered.

The quiz server tests that use for marking will mainly be based on the provided unit-tests, but there will be a number of other test cases. Therefore passing the unit tests is not a guarantee that full marks will be given (however it's a good indication your code is working as required).

[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.

3 Tasks

Your main tasks are listed in his section. More details on the provided files and tests are provided in subsequent sections. Please make sure you read everything so you are aware of the changes since assignment 1.

3.1 Finding results using a chaining hash table 50 Marks]

This task requires you to complete the `get_value` method in the `HashTable` class and the `hash_result_finder` function in `hash_module.py`.

For this task you cannot assume anything about the order of the items in the `tested` and `quarantined` lists. That is, some tests might use sorted lists but other will not.

3.1.1 get_value

You need to complete the `get_value` method in the `HashTable` class definition so that it returns the value associated with a given key. You can refer to the `store` method to see how `key`, `value` pairs are stored in the `HashTable`. Basically the `Hashtable` will store a linked list of `Node` objects in each slot. Your method will have to lookup the appropriate slot and then look through the linked list to get the value that is associated with the given key, then return the value (or None if the key cannot be found).

We have provided some simple tests in the `my_tests` function (similar shell commands shown below). You should add more, in your quest to confirm your method is working.

```
>>> from hash_module import HashTable
>>> h = HashTable(11)
>>> h.store_pair(Name('Lee'), (123,True))
>>> h.store_pair(Name('Bee'), (234,True))
>>> h.store_pair(Name('Gee'), (567,True))
>>> h.store_pair(Name('Fee'), (235,True))
>>> # Bee had another test and is now clear :)
>>> h.store_pair(Name('Bee'), (234,False))
>>> print(h)
HashTable:
     0: None
     1: <Gee>:(567, True) -> None
     2: None
     3: <Bee>:(234, False) -> <Fee>:(235, True) -> <Bee>:(234, True) -> None
     4: None
     5: None
     6: <Lee>:(123, True) -> None
     7: None
     8: None
     9: None
    10: None
Num of items = 5
Num of slots = 11
Load factor = 0.45
>>> print(h.get_value(Name('Bee')))
(234, False)
```

3.1.2 hash_result_finder

This task is similar to the linear and binary result finders in assignment 1. Your function should start with an empty list representing the results for quarantined people. It should then, if needed, create a hash table by storing `name, (nhi, result)` pairs from the `tested` list. In terms of the hash table, the name will be the key and the value you need to store will be a tuple with the nhi and result. Once your function has setup the hash table, it will need to go through each `Name` in `quarantined` and get the associated value from in the hash table. If a match is found, append a tuple containing `(Name, nhi, result)` to the results list. If a match isn't found then add a tuple with `(Name, None, None)` to the results list as you don't have a NHI number or test result for that person. The returned list should be given in the same order that the names appear in the `quarantined` list.

Your `hash_result_finder` should return the results list and the number of comparisons as a tuple, that is, in the form `(your_results_list, comparisons_used)`.

3.1.3 Notes

- We recommend testing your function with very small lists first, eg, start out with one item in each list, then try with one in the first list and two in the second, etc, ...
- Make sure your `hash_result_finder` function deals with edge cases, such as empty `tested` and/or `quarantined` lists, properly. For example, you don't need to make a hash table if no people have been tested.
- The provided tests are really just a final check—they won't help you very much when debugging your code.
- Your function shouldn't mutate the lists it is given in any way, eg, don't pop anything off the `quarantined` or `tested` list and don't insert anything into them either. You will, of course, need to append things to the results and this is fine.

3.2 Finding results using a dual index method [50 Marks]

This task requires you to complete the `dual_result_finder` function in `dual_module.py` using a dual index based method to determine the results for each quarantined person (or the lack of results). Your algorithm needs to utilise the fact that the `quarantined` and `tested` are both sorted in increasing order by name. The *dual* is referring to the use of two indices for the current position in each list, in a way that is similar to the *merge* operation in merge sort and the method for finding common elements in Lab 6.2.

You can assume that `tested` and `quarantined` will be in alphabetical order by name (ie, the `<=` operator is true between any successive names in `tested`). Therefore you will need to make sure you only test your `dual_result_finder` function with lists that are sorted.

You should start with an empty list representing the results found so far and work through the input lists adding a result tuple for each quarantined person when appropriate. If a match is found add a (Name, nhi, result) tuple to the `results` list. If test results cannot be found for a given quarantined person then you should add a tuple with `(the_name, None, None)` to the results list as you don't have a NHI number or result for that person. The returned list should be given in the same

order that the names appear in the `quarantined` list. Your function should return a tuple containing the results list and the number of `Name` comparisons the function made, ie, a tuple in the form `(your_results_list, comparisons_used)`.

There are various different possible orders for doing comparisons in when using the dual index method. Our testing is set-up to accept a specific ordering and the `HelpfulDualTests` test cases will help you work out which one. This doesn't mean this is always the best order for doing comparisons, we just chose one in order to help you examine more closely what is going on and to help you practice your debugging skills.

```
>>> import tools
>>> from dual_module import dual_result_finder
>>> filename = "test_data/test_data-2n-2n-1-a.txt"
>>> tested, quarantined, expected_results = tools.read_test_data(filename)
>>> tested
[(2120001, Name('Filippa Mau'), False), (2120000, Name('Sabina Starkes'), False)]
>>> quarantined
[Name('Albert Willison'), Name('Sabina Starkes')]
>>> expected_results
[(Name('Albert Willison'), None, None), (Name('Sabina Starkes'), 2120000, False)]
>>> results, comparisons = dual_result_finder(tested, quarantined)
>>> results
[(Name('Albert Willison'), None, None), (Name('Sabina Starkes'), 2120000, False)]
>>> # Think about how to use 5 comparisons
>>> comparisons
5
```

3.2.1 Notes

- We recommend testing your function with very small lists first, eg, start out with one item in each list, then try with one in the first list and two in the second, etc, ... Remember both lists must be sorted by name!
- Make sure your `hash_result_finder` function deals with edge cases, such as empty `tested` and/or `quarantined` lists, properly.
- The provided tests are really just a final check — they won't help you very much when debugging your code.
- Your function shouldn't mutate the lists it is given in any way, eg, don't pop anything off the `quarantined` or `tested` list and don't insert anything into them either. You will, of course, need to append things to the results and this is fine.

4 Provided classes = classes2.py

4.1 classes2.py

We use `classes2.py` to provide the basic classes you will need for assignment 2. This ensures you don't accidentally import something old from assignment 1. Please make sure you download/extract all the files for assignment 2 into a new folder so nothing gets mixed up with assignment 1 (eg, you could extract the files into a folder named *assignment2*).

4.2 Node class

You will need to use Nodes to make up the linked lists used for the chains in the chaining hash table class. Nodes are pretty simple, they contain a `key`, a `value` and a `next_node` pointer. `Nodes` also contain a `__repr__`, method which returns a string representation of the list starting at the giving node, and `__len__` method that will return the length of the linked list starting at the given node.

4.3 Name class

For this assignment the main class you will need to know about is the `Name` class. A `Name` object is basically a repackaged `str` object, that updates an internal counter every time it is compared with another `Name` object. You will be required to count each `Name` comparison made by your code and our tests will use the internal counter to check that you got it right.

Further information on `Name` objects is given below.

- `my_name = Name(name)` creates a new `Name` object with the given name.
- `print(my_name)` will print the Name object, in the form `<name>` so you don't confuse it with a simple `str` object.
- Every time two `Name` objects are compared the name comparison counter in the StatCounter class will be updated. For example, whenever a comparison such as `name1 < name2` is evaluated the internal counter is incremented. You will see that all the comparison operators work like this, ie, <, >, >=, <=, ==, !=.
- You shouldn't be using any of the double under-scored methods for `Name` objects directly. You should use the normal comparison operators, eg, name1 < name2 will be automatically translated into `name1.__lt__(name2)` in the background so you don't need to call `name1.__lt__(name2)` directly!
- `StatCounter.get_count(NAME_COMPS)` gives the actual number of `Name` comparisons that have been performed. If you read the definition for the `Name` class you will see that `Name` objects update the name comparisons counter each time they are compared. This is for your debugging only and using this in submitted code will cause your code to fail the submission tests. You must use the line `from stats import StatCounter, NAME_COMPS` if you want to check this counter.

The following example code should help you further understand the Name class:

```
>>> from classes2 import Name
>>> from stats import StatCounter, NAME_COMPS
>>> my_list = []
>>> name1 = Name('Paul')
>>> name1
Name('Paul')
>>> print(name1)
<Paul>
>>> my_list.append(name1)
>>> my_list.append(Name('Tim'))
>>> my_list
[Name('Paul'), Name('Tim')]
>>> name3 = Name('Zheng')
>>> name1 < name3
True
>>> my_list[0] == Name('Paul')
True
>>> StatCounter.get_count(NAME_COMPS) #This is allowed only for testing!
```

```
2
>>> my_list[1] >= my_list[0]
True
>>> StatCounter.get_count(NAME_COMPS) #This is allowed only for testing!
3
```

5 Provided tools and test data

Important Note: The contents of some test files are slightly different from those used in assignment 1 so make sure you extract all your assignment 2 files into a different folder to the one you used for assignment 1.

Test files are given to you in the folder `test_data` to make it easier to test your own code. The test data files are named in the form `test_data-{i}{x}-{j}{y}-{k}-{m}.txt` and contain three lists of data. The first list, referred to as *tested*, contains (NHI, name, result) tuples for people that have been tested for the virus. The second list, referred to as *quarantined*, contains the names of people who are quarantined. The third list, referred to as *results*, contains (name, NHI, result) tuples for all the quarantined people, in the same order as they appear in the *quarantined* list. If a quarantined person isn't in the *tested* list then their NHI and result will be None.

5.1 Decoding data file names

The *tested* and *quarantined* data sections start with a line containing the number or records in that section. The third data section doesn't have this number as it will have the same number of records as the *quarantined* list. Note that lines starting with # are commented out and are not read.

- i = number of records in *tested* list
- x = 'i' if *tested* is sorted by NHI or 'n' if *tested* is sorted by name.
- j = number of records in *quarantined*
- y = 'n' if *quarantined* is sorted by name or 'r' if it's unsorted/randomly ordered.
- k = number of *quarantined* people that have a result, ie, can be found in the *tested* list.
- m = the random seed used to generate the test file. We may generate different random files for the quiz server tests.

For example, a file named `test_data-5i-5n-2-a.txt` will contain: 5 records for tested people that are sorted by NHI, 5 names of quarantined people that are sorted by name, 5 results records for the quarantined people, and two of the quarantined people will have have results in the tested list (in the example below Cissiee and Jon are the two).

```
# Number tested and their details:
5
2120000,Zorina Latin,True
2120002,Selime Sziladi,False
2120003,Vallipuram Kortekaas,True
2120005,Jon Klaudt,False
2120007,Cissiee Bednar,True
# Number quarantined and their names:
5
Andrzej Challice
Cissiee Bednar
Jon Klaudt
```

```
Meris Dendi
Rheal Wolfenbarger
# Expected result details for quarantined people:
Andrzej Challice,None,None
Cissiee Bednar,2120007,True
Jon Klaudt,2120005,False
Meris Dendi,None,None
Rheal Wolfenbarger,None,None
```

When tested records are sorted by NHI the names will effectively be in random order (as seen in the example above).

5.2 Reading data files

The `tools.py` module contains functions for reading data from test files and for making simple lists of `Name` objects, etc...The most useful function in the `tools` module is `read_test_data(filename)`, which reads the contents of the test file and returns a tuple containing the *tested, quarantined* and expected *results* lists respectively.

Suppose `test_data-2n-2n-1-a.txt` contains the following data, then the shell code below should give you an idea of how the data is read.

```
# Number tested and their details:
2
2120001,Filippa Mau,False
2120000,Sabina Starkes,False
# Number quarantined and their names:
2
Albert Willison
Sabina Starkes
# Expected result details for quarantined people:
Albert Willison,None,None
Sabina Starkes,2120000,False
```

Example data loading using `read_test_data`:

```
>>> import tools
>>> filename = "test_data/test_data-2n-2n-1-a.txt"
>>> tested, quarantined, expected_results = tools.read_test_data(filename)
>>> tested
[(2120001, Name('Filippa Mau'), False), (2120000, Name('Sabina Starkes'), False)]
>>> quarantined
[Name('Albert Willison'), Name('Sabina Starkes')]
>>> expected_results
[(Name('Albert Willison'), None, None), (Name('Sabina Starkes'), 2120000, False)]
>>> print(quarantined[0])
<Albert Willison>
```

5.2.1 Making your own tested/quarantined lists

The `make_name_list` function in the `tools` module can be used to make lists of `Name` objects from simple lists of strings or using the letters in a string. The `make_tested_list` function will make a list of `(nhi, Name, result)` tuples from a simple list of strings, or using the letters in a string. `make_tested_list` will generate the NHI numbers and test results for you and there are options for list sorting and the result you want everyone to have. Check out the function definitions and doc-strings for more details. An example of each is given below.

```
>>> my_quarantined = tools.make_name_list(['Bob', 'Abba', 'Faba'], sort_order='name')
>>> my_quarantined
[Name('Abba'), Name('Bob'), Name('Faba')]
>>> my_tested = tools.make_tested_list(['Bingle', 'Zabba', 'Faba'], sort_order='name')
>>> my_tested
[(1, Name('Bingle'), True), (3, Name('Faba'), True), (2, Name('Zabba'), True)]
```

You will be able to use these functions to quickly generate simple lists that you can send to your functions for processing. This will help you verify and debug your code by allowing you to compare your hand-cranked results with the results returned by your function. For example, you could generate the lists above and then call your function to see if it returns the result you expect. If not then you need to work out whether your hand-cranking was wrong or your function was wrong...

6 Provided tests

The `tests.py` provides a number of tests to perform on your code. Running the file will cause all 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. The `all_tests_suite()` function has a number of lines commented out indicating that the commented out tests will be skipped; un-comment these lines out as you progress through the assignment tasks to run subsequent tests.

To get all the test results in a more manageable form in Wing101 you should make sure that *Enable debugging* is turned off in the options for the Python shell. You can get to the options by clicking on the Options drop down at the top right of the Shell window.

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 parameters and parameters of differing sizes. You will, of course, want to start by testing with very small lists, eg, with zero, one or two items in each list. This will allow you to check your answers by hand.

NOTE: *The tests that are provided in `tests.py` aren't good for debugging! You should be using your own simple tests to help you debug*