2017 Assignment—Part 1

CSC002

1 Overview

1.1 Introduction

This assignment is based around detecting voter fraud. The work required is designed to help you understand both the theoretical and practical concerns related to implementing a variety of algorithms. As a computer scientist, you will often encounter problems for which there is no pre-existing solution. Your knowledge of and experience with algorithms will directly affect the quality and efficiency of systems that you produce, especially when processing problems that involve large data-sets.

In an election voters can choose which voting booth they can use. Because the system is currently paper-based, if a voter goes to two voting booths they could vote twice, so it's important to check afterwards if someone has done that. In the scenario we are solving here, you are given the names of voters from two different voting booths, and your job is to find voters who voted in both booths. (Of course, in a realistic situation, there may be multiple booths, and the voter would need to be identified by more than just their name.)

1.2 Due date

Monday 3 April 2017, 11pm.

1.3 Submission

Submit via the quiz server. The submission page will be open closer to the due date.

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* program checking.

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. Remember that the main point of this assignment is for you to exercise what you have learnt 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 Detecting voter fraud

For each task of the assignment the functions you write will need to take two `VoterList` objects (described later) containing the voter names (as `VoterName` objects) from two voting booths. It can be assumed that there are no duplicate voter names in a single `VoterList` (i.e. each voter voting in a booth has a unique name). In each task you will need to use specific data structures (e.g. `VoterList`), which you will import into your program. You do not have to implement these data structures, you will just have to interact with them. They are provided in the files `classes.py`. Details of these data structures can be found in the relevant sections. For each task you are required to return a `VoterList` object containing the names of the voters who fraudulently voted in both voting booths, and in addition you will need to count and return the number of `VoterName` comparisons your code makes. Note, that we only count `VoterName` comparisons (that is, comparisons between `VoterName` objects), and not comparisons of indexes, counters and so on. The data structures we provide will also automatically count comparisons, which will allow you to check how many comparisons have actually been made, but this is intended for debugging only. If you use this comparison checking in any code you submit (rather than doing the comparison counting yourself) then your code will fail the coderunner tests used for submitted assignments.

You will be provided with offline tests in the file `tests.py` to check your implemented code before submission. These tests use the Python unittest framework. You are not expected to know how these tests work; you just need to know that the tests will check that your code finds the correct fraudulent voters, and that the number of comparisons that your code makes is appropriate. For the first algorithm you should be able to match the exact number of comparisons, but for binary search, merging and hashing the comparisons made are checked to be in the right ballpark for the expected number of comparisons. 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 tests used on the quiz server will be mainly based on these unittests with a small number of added secret test cases, so passing the unit test is not a guarantee that full marks will be given for that question (it is however a very good indication your code is correct).

2.1 Provided classes

The main class provided is `VoterList`, which is a list of `VoterName` objects. The `VoterList` object is similar to a normal Python list but has several limitations to make sure that you write the required algorithms yourself: you can only add `VoterName` objects to it, and the built in Python sorting and searching methods will not work with it. You can however access the `VoterName` objects in the `VoterList` by indexing into it as with a normal list i.e. `v[i]`. Each `VoterName` object consists of a string representing the name of a voter. You are not allowed to convert any `VoterList` to a regular Python list. In addition, you must not change any of the classes provided, and only access them in following ways:

- `my_list = VoterList()` creates a new `VoterList` that can be used for recording voter names. Available methods include `len`, `append` and `==`.
- `my_voter = VoterName(name)` creates a new `VoterName` object with the given name. This can be added to a `VoterList` using the `my_list.append()` method. This can be compared with other `VoterName` objects by using `<=`, `>=`, `<`, `>`, `==` and `!=`.
- `print(my_list)` will print your `VoterList` in the correct format. Similarly `print(my_voter)` will work correctly.
- `VoterList.get_comparisons()` gives the actual number of `VoterName` comparisons that have been performed. This is for debugging only and using this in submitted code will cause your code to fail the submission tests.

The following example code should help you understand how to use these classes:

```
>>> from classes import VoterName, VoterList
>>> my_list = VoterList()
>>> my_list.append(VoterName('Paul'))
>>> len(my_list)
1
>>> my_list.append(VoterName('Tim'))
>>> len(my_list)
2
>>> my_list[1] #VoterList can be randomly accessed.'
'Tim'
>>> for name in my_list: #VoterList objects are iterable.
...     print(name)
Paul
Tim
>>> print(my_list)
['Paul', 'Tim']
>>> VoterName('a') < VoterName('b')
True
>>> my_list[0] == VoterName('Paul')
True
>>> VoterList.get_comparisons() #This is allowed only for testing!
2
```

2.2 Provided tools and test data

The `tools.py` module contains functions for reading test data from test files. Test files are given to you in the folder `TestData` to make it easier to test your own code. The test data files are named `test_datai-j-k.txt` and contain a line with the number i followed by i lines each containing a name of a voter from voting booth 1, following that another number j is given followed by j lines containing the names of the voters from voting booth 2 and lastly a final number k is given followed by k lines containing the names of the voters which voted in both booths. Note lines starting with # are commented out and are not read.

The most useful function in this module is `tools.read_test_data(filename)`, which reads the contents of the test file and returns a triple containing the voters from booth 1, the voters from booth 2 and the fraudulent voters respectively.

The following example code should help you understand how to use the module `tools`:

```
>>> import tools
>>> filename = "TestData/test_data2-2-1.txt"
>>> booth1, booth2, fraud = tools.read_test_data(filename)
>>> booth1
['Athene Debord', 'Audi Laskin']
>>> booth2
['Athene Debord', 'Blinny Forgeron']
>>> fraud
['Athene Debord']
>>> type(booth1)
<class 'classes.VoterList'>
>>> type(booth1[0])
<class 'classes.VoterName'>
```

2.3 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; uncomment these lines out as you progress through the assignment tasks to run subsequent tests.

In a 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.

3 Tasks

3.1 Detecting voter fraud using sequential search [15 Marks]

This task requires you to complete the file `fraud_detector_sequential.py`. This first method isn't going to be very efficient, but it's a starting point! You should start with an empty `VoterList` representing the fraudulent voters found so far. Go through each `VoterName` in `first_booth_voters` and squentially search `second_booth_voters` to try find that `VoterName`. If a match is found add that name to the fraudulent voters found so far and stop searching the rest of `second_booth_voters` for that `VoterName` (as it can be assumed that no one can vote twice in a single booth). You cannot assume that the `VoterNames` will be in any particular order for either booth. The returned `VoterList` should be given in the same order that the fraudulent voters appear in `first_booth_voters`. You should return the populated `VoterList` containing the fraudulent voters and the number of `VoterName` comparisons the function made.

3.2 Detecting voter fraud using binary search [35 Marks]

This task requires you to complete the file `fraud_detector_binary.py`. This method should perform considerably better than sequential search. Again, you should start with an empty `VoterList` representing the fraudulent voters found so far. Go through each `VoterName` in `first_booth_voters` and perform binary search on `second_booth_voters` to try find that `VoterName`. If a match is found, add that name to the fraudulent voters found so far. The list `second_booth_voters` will be in order (i.e., the `<=` operator is true between any successive `VoterName` objects in `second_booth_voters`) but you may not assume anything about the order of `first_booth_voters`. The returned `VoterList` should be given in the same order that the fraudulent voters appear in `first_booth_voters`.

To get full marks in this question you must minimise the average number of comparisons made. Therefore, you will need to implement binary search in such a way that only uses one `VoterName` comparison per halving of the search area.

Warning: fast binary search is difficult! Set aside a lot of time to get this right, and don't be put off if it doesn't work correctly straight away.

3.3 Detecting voter fraud using merging [25 Marks]

This task requires you to complete the file `fraud_detector_merge.py`. In the binary search question one of the two `VoterList` parameters was given in order. Can we make a more efficient algorithm for the voter fraud detection task if we know both booth `VoterList` parameters are given in order? For this task you are required to design and implement a method that finds the `VoterList` of fraudulent voters using just one simultaneous pass through both `VoterList` parameters. The returned `VoterList` should be given in the same order that the fraudulent voters appear in `first_booth_voters`.

Tip1: try using two integers to keep track of which index you are at in both `VoterList` parameters. Increment the index integers in such a way that any voter fraud is guaranteed to be found.

Tip2: if you are stuck look up the merge operation from merge sort (this is taught in next terms material) and try implement a modified version for this task (you are not supposed to store or keep track of the merged list; see tip1). Note you are not expected to understand merge sort just the merge operation.

3.4 Detecting voter fraud using a hash table [25 Marks]

This task requires you to complete the file `fraud_detector_hash.py`. The performance of the merging solution can be matched even when the `VoterList` parameters are not given in order. In this variation, you will use a hash table (implemented in the `classes.py` module as `VoterHashTable`) to store `VoterName` objects and check for duplicates. Collisions will be dealt with by having each entry in the hash table being a `VoterList` object and appending to the end of the corresponding `VoterList` object when a `VoterName` is added to the hash table (i.e., the hash table is using a `VoterList` for chaining in each slot). When you create a `VoterHashTable` you have to give it a capacity indicating how many slots you wish the table to have. This value cannot be changed and rehashing all the entries is time consuming, so a suitable value based on the size of the parameters is needed. To make marking consistent, you are required to use the `VoterName` hash function for hashing and a table capacity of `len(second_booth_voters)* 2`. The returned `VoterList` should be given in the same order that the fraudulent voters appear in `first_booth_voters`.

The following example code should help you understand how to interact with the given objects:

```
>>> from classes import VoterName, VoterList, VoterHashTable
>>> table_capacity = 10
>>> my_table = VoterHashTable(table_capacity)
>>> my_table
[[], [], [], [], [], [], [], [], [], []]
>>> type(my_table[0])
<class 'classes.VoterList'>
>>> voter_gen = VoterName("gen")
>>> hash(voter_gen)
118508637
>>> hash(voter_gen) % table_capacity
7
>>> for voter_name in my_table[7]:#No output as Gen is not in my_table.
...     if voter_name == voter_gen:
...         print("'Voter gen is in the hash table.'")
>>> voter_tim = VoterName("Tim")
>>> tim_index = hash(voter_tim) % table_capacity
>>> my_table[tim_index].append(voter_tim)
>>> voter_anna = VoterName("Anna")
>>> anna_index = hash(voter_anna) % table_capacity
>>> my_table[anna_index].append(voter_anna)
>>> my_table
[[], [], [], [], [], [], ['Anna'], [], [], ['Tim']]
>>> for voter_name in my_table[hash(voter_anna) % table_capacity]:
...     if voter_name == voter_anna:#Anna is in my_table.
...         print("'Voter Anna is in the hash table.'")
'Voter Anna is in the hash table.'
```