COSC122 2017 Semester 2
INTRODUCTION TO ALGORITHMS

Assignment 2: Diagnosing Genetic Diseases

7% of overall grade
Due Friday 22 September 2017, 11:55pm

1 Overview

1.1 Introduction

Modern medicine is increasingly moving towards personalised diagnosis and treatments. A large part of this is based on DNA sequencing, which has seen significant growth over the past two decades. What was once a prohibitively expensive task, sequencing the entirety of your genome is now available as an affordable online service.

In these assignments we will be using algorithms to learn about, and help treat, patients based on their genes. While short DNA sequences such as that of a fly are so small that almost any algorithm will be sufficient on a modern machine, humans are orders of magnitude more complicated. Thus the algorithms we use need to be able to scale to look at hundreds of millions of base pairs, and as computer science students you will be the ones designing the tools for the next generation of medical researchers.

Throughout these assignments, when we represent DNA, we only ever store one half of the strand—the first half always unambiguously implies the other. Indeed, this is exactly what a cell does when it is replicating. For example, from `atcgta`[1] we can determine the pair sequence is `tagcat`. For our purposes, we will consider a gene to be a sequence of 16 bases/letters—this is about a thousand times shorter than genes usually are, but still gives us $4^{16} \approx 4.3$ billion different genes to work with.

1.2 Due date

Friday 22 September 2017, 11:55pm; drop dead date is 7 days later, with a 15% absolute penalty.

1.3 Submission

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

[1] DNA is made from four bases: adenine (`a`), guanine (`g`), cytosine (`c`), and thymine (`t`). Adenine always pairs with thymine, and guanine always pairs with cytosine. For more details, see `https://simple.wikipedia.org/wiki/DNA`.

1.4 Implementation

All the files you need for this assignment are 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 functions that these functions call 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 Diagnosing patients via their genome

A person's DNA can tell us a lot about what is happening inside their body, and more importantly what may one day happen. By determining which genes a person carries, we can determine which diseases they are predisposed to. In Assignment 1, we were able to work out which genes relate to certain diseases by examining the genomes of people who are known to carry them. In Assignment 2 we will be working with people that do not yet show symptoms of any disease, but will be using their genomes to provide early detection.

This assignment revolves around hash tables, and how they can be used to efficiently search large amounts of data. The result will be a system that can read in a file containing genetic disorder data, and then be queried to find out if a patient is suffering from a genetic disorder recorded in the table.

For each of the first two tasks of this assignment, you will be completing the definitions of hash table classes that stores `(Gene, str)` pairs so we can match genes to diseases. Both of these hash tables are subclasses of `BaseGeneHashTable`. If you need to use other data structures, please use those in `classes.py`. To complete the definition of a hash table, you must fill in the `__init__`, `__getitem__`, and `insert` methods. For each class you should maintain two counters as instance attributes (i.e. `self.hashes`): `hashes`, which counts the number of times a `Gene` object was hashed; and `comparisons`, which counts how many times two `Gene` objects are compared against one another. As with last time, we do not need to count comparisons of indexes, counters, and so on. We will be checking your counts against the results we count ourselves. You must not rely on our counters being available to you on the quiz server: they are for debugging purposes only.

Offline tests are available 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 stores and retrieves the correct diseases, and that the number of hashes and comparisons that your code makes is appropriate. We will be checking that your comparisons are in the right ballpark for the expected number of comparisons, while your number of hashes is expected to be exactly as we find it. 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 tests is not a guarantee that full marks will be given for that question (it is however a strong indication your code is correct).

The final part of the assignment asks you to answer some short questions about the two types of hash tables you have implemented. These are to be submitted into text boxes on the quiz server when submissions have opened, and will be marked by hand by your lecturers/tutors. Consider writing these paragraphs in a plain text file *before* you submit it to the quiz server.

2.1 Provided classes

The `classes.py` module contains six useful classes for you to work with: three as from the first assignment (`Gene`, `Genome`, and `GeneList`), and three new classes: `BaseGeneHashTable`, `GeneLinkedList`, and `GeneLink`. The old classes are described in the handout from Assignment 1, while the new classes are described in more detail below.

The `utilities.py` module contains another class that you may find useful: `StatCounter`. This class holds all the comparison information that you might need to check your code against. We also use it to verify that you do perform the correct number of comparisons.

⚠ **Important:** You cannot rely on `StatCounter` being available to you on the quiz server at the time of submission, so do not use it in your final code.

`BaseGeneHashTable` The `BaseGeneHashTable` class provides an *abstract base class* for both of the hash tables you are asked to implement. This means that you need to implement the same methods with the same number of arguments. For example, `BaseGeneHashTable` defines a method `__getitem__(self, gene)`, so your class implementation will also need to define a method `__getitem__(self, gene)`.

`GeneLinkedList` A `GeneLinkedList` is, as the name would suggest, a linked list. It is purposefully handicapped to ensure you understand the internal mechanism of a linked list, and do not just treat it as any other list type. It forms the chains in a `ChainingGeneHashTable`. Each link in the chain is made of a `GeneLink`, starting at the `head` (if the list is empty, `head` is `None`).

`GeneLink` A `GeneLink` is a single node in a `GeneLinkedList`. Each has some `data` (probably a `(gene, disease)` pair) and a pointer to the `next` node. If there is no next link, the value of `next` is `None`.

2.2 Provided tools and test data

The `utilities.py` module contains functions for reading test data from test files, and also our own utility that we use to count your comparisons. Test files are in the folder `TestData` to make it easier to test your own code. The test data files are named `test_data-k-i-j.txt` and contain:

- a line containing the numbers k, i, and j, where k is the size of the hash table we will be filling,
- i lines containing a gene (represented by 16 DNA bases) and a disease it is associated with,
- one line consisting of a patient's genome (length is variable, but always a multiple of 16), and
- j lines of disease names that the patient actually suffers from.

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

```
>>> import utilities
>>> filename = "TestData/test_data-1-1-1.txt"
>>> table_size, diseases, patient, present =
    utilities.read_test_data(filename)
>>> diseases
[('cgggtgtcacatgcga', 'leukemia')]
>>> patient
tggcccatcctaacat tccaaggtgcgccgta gcatgcaaagtggttc cgggtgtcacatgcga
>>> present
{'leukemia'}
>>> table_size
1
>>> bad_gene, disease = diseases[0]
>>> patient[3] == bad_gene
True
```

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.

⚠ **Important:** 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 [100 Marks Total]

3.1 Performing diagnoses with chaining hash tables [50 Marks]

This task requires you to complete the file `chaining_gene_hash_table.py`. To initialize a chaining hash table, create a python list of the correct size, where each index is filled with an empty `GeneLinkedList`. This is the basis of the hash table, where we will be storing all the disease information for quick lookup. Be sure that all hash and comparison counters in your class are reset during initialization.

When a new disease is to be added to the hash table through `insert(gene, disease)`, first we must find where it lives by `hash`ing the gene, and making sure this hash value is in the right range for our table (*hint:* modulo). Once the correct index is found, we add the disease to the head of the `GeneLinkedList` by placing a `(gene, disease)` tuple in a `GeneLink`, and then inserting the `GeneLink` into the `GeneLinkedList`.

To find whether a gene is associated with a disease with `__getitem__`, we have to search the hash table for a disease. We do this by hashing the value, and searching the linked list to determine if the gene we are looking for is in there. If the gene is in the hash table, return the disease; if the gene is not in the hash table, return `None`. Do *not* add this value to the hash table.

We implement the method `__getitem__` because it allows us to use Python's square bracket notation. That is, we can search for a gene with `hash_table[gene]`.

```
>>> table = ChainingGeneHashTable(5)
>>> table.insert(Gene('atcg'), 'Asthma')
>>> table.insert(Gene('tcga'), 'Leukemia')
>>> print(table)
ChainingGeneHashTable[
  0: None
  1: (('atcg', 'Asthma')) -> None
  2: (('tcga', 'Leukemia')) -> None
  3: None
  4: None
]
>>> print(table[Gene('atcg')])
Asthma
>>> print(table[Gene('cgta')])
None
```

3.2 Performing diagnoses with linear probing hash tables [40 Marks]

This task requires you to complete the file `linear_probing_gene_hash_table.py`. To initialize a linear probing hash table, create a python list of the correct size, where each index is filled with `None`. This is the basis of the hash table, where we will be storing all the disease information for quick lookup. Be sure that all hash and comparison counters in your class are reset during initialization.

As before, when a new disease is to be added to the hash table, first we must find where it lives by `hash`ing the gene, and making sure this hash value is in the right range for our table. If the hash index is already filled, we must probe ahead to find the next free location. Finally, when a free space is found, store the disease information as a `(gene, disease)` tuple at that index. In the case that the table is full (that is, there is no free space to fill), raise an `IndexError` with an appropriate error message[2].

To find whether a gene is associated with a disease, we have to search the hash table for that gene (again in `__getitem__`). This is very similar to inserting values into a hash table, except when probing the table, we check each entry to see if the value matches the gene we are searching for. If the gene is in the table, return the associated disease; if the gene is not in the hash table, return `None`. Do *not* add this value to the hash table.

3.3 Relative benefits of each approach [10 Marks]

Choosing an algorithm or data structure is an important part of solving any problem, and the choices made can have a dramatic impact on the performance of the resulting program. For this task, we want you to consider different aspects of the hash tables you have constructed, and provide short (one-or-two-sentence) statements about each factor. To help you understand some of the properties of your hash tables, we have provided a `runtimes.py` file that plots graphs of comparisons and runtimes.

- The hash function we have been using meets the requirements of a "good" hash function. What are these requirements?
- A linear hash table seems to perform poorly when the table has a high load factor. Why might this be?
- What happens when the linear probing hash table has a load factor greater than one? And what happens when the chaining hash table has a load factor greater than one?
- When deciding on the size of a hash table, what trade-offs are we forced to make?
- Chaining and linear probing are both methods to handle collisions in a hash table. What other methods are there?

[2] Our tests will check that you raise the correct exception, but not the message associated with it. As such, the actual message you include is entirely up to you.