COSC122 2017 Semester 2
Introduction to Algorithms

Assignment 1: Finding Genetic Markers

6% of overall grade
Due Friday 25 August 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 25 August 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 Finding genetic markers of diseases

Genetic diseases occur as a result of abnormal DNA sequences, resulting in mutated genes. Many forms of cancer stem from genetic abnormalities, as well as conditions such as Huntington's Disease or even Asthma. Given the DNA of individuals known to carry a particular disease, we can find all the genes in common, and thus determine which genes are potentially linked to the disease in question. In Assignment 1, we will be focussing on the first part of the problem, finding common genes.

For each task of the assignment the functions you write will need to take two `Genome` objects containing the sequence of `Gene` objects from two individuals. You may assume that there are no duplicate genes in a single `Genome` (i.e., each gene occurs only once and is described by a unique DNA sequence). In each task you will need to use specific data structures (e.g. `GeneList`), 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 in the file `classes.py`. Details of these data structures are in the relevant section below. For each task you must return a `GeneList` object containing the genes that occur in both genomes, and in addition you will need to count and return the number of `Gene` comparisons your code makes. Note, that we only count `Gene` comparisons—that is, comparisons between `Gene` objects, because this is an expensive operation, particularly if the `Gene` objects are large—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 occurred, 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.

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 finds the correct genes in common, 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 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, 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 (it is however a strong indication your code is correct).

2.1 Provided classes

The `classes.py` module contains three useful classes for you to work with: `Gene`, `Genome`, and `GeneList`. They are primarily wrappers around existing Python types, but we can add or remove features as we need them.

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.

`Gene` The most fundamental class is the `Gene`. For now, you may consider a `Gene` to be a `str`, in that all the common comparisons (`==`, `<`, `>`, `<=`, `>=`, `!=`) are available. The difference is that we count these comparisons, and store the result in a `StatCounter`.

`Genome` A `Genome` is an immutable `list`-like structure that can only hold genes. You cannot edit a `Genome`, nor can you sort it nor query for the index of certain elements. You are able to iterate over a `Genome`, that is, you can use it in a `for` loop. You can also access specific `Gene` objects in the `Genome` by using indexing, that is `g[i]`.

`GeneList` Similar to a `Genome`, a `GeneList` is a `list`-like structure that can only contain `Gene` objects. The difference is that you can `append` onto a `GeneList`, which you cannot do with a `Genome`.

`StatCounter` In order to count how many comparisons your code makes, we provide a `StatCounter` class. Note that because of the way it is implemented, it will not behave like a regular class. This class is only for testing purposes, and not in your final code. You should only use the `StatCounter.get(counter_name)` method, providing `"comparisons"` as the `counter_name`.

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-i-j.txt` and contain two sequences of $16i$ DNA bases, plus another DNA sequence that is no longer than the first two (and also has a length which is a multiple of 16). The number j represents how many genes occur in both genomes. The first two DNA sequences belong to two patients with a common disease, and the third sequence represents all the genes they have in common.

The most useful function in this module is `utilities.read_test_data(filename)`, which reads the contents of the test file and returns both genomes in the datafile, as well as the genes they have in common as a `GeneList`.

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

```
>>> import utilities
>>> filename = "TestData/test_data-2-1.txt"
>>> genome_a, genome_b, common_genes =
    utilities.read_test_data(filename)
>>> genome_a
agtacggtctgaaact gggcaatgaggagctt
>>> genome_b
gggcaatgaggagctt attcaagaagctacaa
>>> common_genes
gggcaatgaggagctt
>>> type(genome_a)
<class 'classes.Genome'>
>>> type(genome_a[0])
<class 'classes.Gene'>
>>> type(common_genes)
<class 'classes.GeneList'>
```

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 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 Finding common genes using sequential search [40 Marks]

This task requires you to complete the file `genetic_similarity_sequential.py`. This first method isn't going to be very efficient, but it's a starting point! You should start with an empty `GeneList` representing the common genes found so far. Go through each `Gene` in `first_genome` and sequentially search `second_genome` to try find that `Gene`. If a match is found add that gene to the common genes found so far and stop searching the rest of `second_genome` for that `Gene` (as it can be assumed that no gene occurs twice in a single genome). You cannot assume that either `Genome` will be in any particular order. The returned `GeneList` should be in the same order that the genes appear in `first_genome`. You should return the populated `GeneList` containing the common genes and the number of `Gene` comparisons the function made.

3.2 Finding common genes using binary search [60 Marks]

This task requires you to complete the file `genetic_similarity_binary.py`. In the first task, we made no guarantees about the order of the genes. Can we make a more efficient algorithm for finding common genes if we know one of the `Genome` parameters (in particular, `second_genome`) are given in lexicographic order? For this task you are required to design and implement a method that finds the `GeneList` of common genes using binary search to find each gene in `first_genome` in the `second_genome`. The returned `GeneList` should be given in the same order that the genes appear in `first_genome`.

> ⚠ **Important:** Binary search can be difficult to implement correctly. Be sure to leave sufficient time to solve this task.