

# Assignment 1: Finding Genetic Markers

---

5% of overall grade

Due Part A: 11:55pm Fri 11 August 2023, Part B: 11:55pm Fri 25 August 2023

## 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 representing 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`<sup>1</sup> 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 dates

Ideally your assignment code should be submitted by:

- Part A is due by Friday 11 August 2023 (ie, end of Week 4)
- Part B is due by Friday 25 August 2023 (ie, end of Week 6)

---

<sup>1</sup>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>.

Part A has no extension. However, Part B may be submitted up to one week late—you will incur a 15% absolute penalty (in other words, your mark will be the raw mark less 15 marks, given the assignment is marked out of 100). Assignments cannot be submitted more than one week after the deadline unless you’ve organised an extension with the course supervisor because of illness etc.

### 1.3 Submission

Submit via the quiz server. There are two submission pages: one for Part A, and another for Part B. Each submission page will be open about a week before its due date. This emphasises the point that the quiz won’t offer much help, as 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, so submission won’t give you any more information than the provided tests! This means you can start on the assignment straight away.

*This assignment is a step up from COSC121/131 so please make sure you start early so that you have time to understand the requirements and to debug your code. The first part uses a very simple algorithm, but you’ll need time to understand the structure you’re working in.*

### 1.4 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 of the assignment there is a file with a function for you to fill in. You need to complete these functions, but you can also write new 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.*

### 1.5 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)<sup>2</sup>. 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.

---

<sup>2</sup>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 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.

### 2.1 Task Overview

This assignment takes two different approaches to solving the same problem. Each task requires you to make a function that:

- Takes two Genome objects of the same length containing **unique** Gene objects:
  1. `first_genome` – the genome of individual one.
  2. `second_genome` – the genome of individual two.
- Goes through each Gene of *individual one's* Genome and searches for that Gene in *individual two's* genome, recording the number of comparisons made between Gene objects.
- Returns a GeneList object containing the genes that occur in both genomes, alongside the number of comparisons used.

You will need to use specific data structures (e.g. GeneList), located in `classes.py`. You do not have to implement these data structures, you will just have to interact with them. Details of these data structures are in the [Provided Classes](#) section below.

You are expected to test your code using **both** your own tests and the provided tests. More information about testing your code is in the [Testing your Code](#) section.

### 2.2 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.

**Gene** The most fundamental class is the Gene. For now, you may consider a Gene to be a [str](#). 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]`.

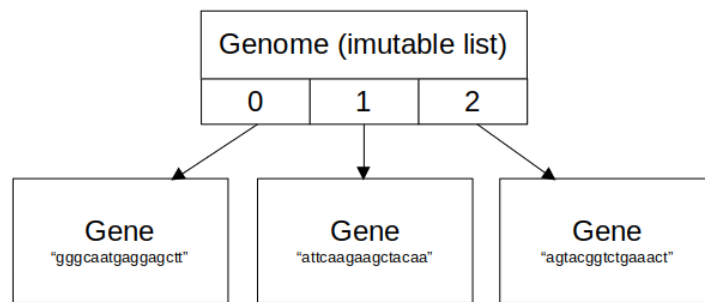


Figure 1: A Genome is an immutable list of strings.

**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**.

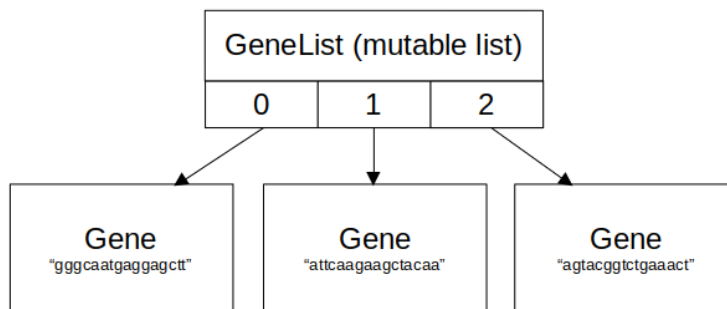


Figure 2: A **GeneList** is like a mutable list of strings.

The `stats.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.

**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`.



**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.

### 3 Tasks [100 Marks Total]

#### 3.1 Part A: Finding common genes using sequential search [40 Marks]

This task requires you to complete the file `sequential_gene_match.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` (it can be assumed that no gene occurs twice in a single genome).
- You should return the populated `GeneList` containing the common genes and the number of `Gene` comparisons the function made (see the return line in the supplied code).

##### 3.1.1 General Notes

- You cannot assume that either `Genome` will be in any particular order. Both `Genomes` are the same length.
- The returned `GeneList` must be in the same order that the genes appear in `first_genome`.
- Remember, you can compare `Genes` using all the normal comparison operators, eg, `>`, `>=`, `==`, etc all work so you can do comparisons like `gene1 > gene2`, etc...
- We recommend writing your own tests with very small lists first, eg, start out with one gene in each list, then try using one in the first list and two in the second list, etc, ...
- The [provided tests](#) are really just a final check — they won't help you very much when debugging your code. Initially, you can do some initial tests with small [lists](#) of strings ([str](#)). Later, you can swap the [lists](#) for `Genomes` and [strs](#) for `Genes`.

 <b>Important:</b> Part A of this assignment is due: 11:55pm, Fri 11 August 2023
---

#### 3.2 Part A: Theory Questions [6 Marks]

1. What is the worst case big-O complexity for your `sequential_gene_match` function, given there are `n` items in both the first genome and second genome. Explain how this would come about and give a small example of worst case input.
2. What is the best case big-O complexity for your `sequential_gene_match` function, given there are `n` items in both the first genome and second genome. Explain how this would come about and give a small example of best case input.

3. Give the equation for the number of comparisons used by the `sequential_gene_match` function, given there are  $n$  items in the `first_genome` and  $n$  items in the `second_genome` list AND that all the items in the `first_genome` are also in the `second_genome`. NOTE: you are giving an equation for the exact number of comparisons made, NOT the big-O complexity.

### 3.3 Part B: Finding common genes using binary search [50 Marks]

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 one of the `Genome` parameters are in lexicographic order?

This task requires you to complete the file `binary_gene_match.py`. You are to implement a function that finds the `GeneList` of common genes using binary search to find each gene in `first_genome` in the `second_genome`.

- Again, start with an empty `GeneList` representing the common genes found so far.
- Go through each `Gene` in `first_genome` and perform a binary search, trying to find that `Gene` in `second_genome`.
- If a match is found, add that gene to the common genes found so far.
- A helper function that does a binary search for an item in a sorted list and returns `True`/`False` along with the number of comparisons used could be helpful here.
- Again, you should return the populated `GeneList` containing the common genes and the number of `Gene` comparisons the function made.

#### 3.3.1 General Notes

- Again, the returned `GeneList` should be given in the same order that the genes appear in `first_genome`.
- You can assume that `second_genome` will be in order (ie, the genes contained in it will be sorted from smallest to largest). Both `Genomes` are the same length.
- Remember, you can compare `Genes` using all the normal comparison operators, eg, `>`, `>=`, `==`, etc all work so you can do comparisons like `gene1 > gene2`, etc...
- **Important:** Your binary search must only do one comparison per loop when it is searching the second genome and should only compare genes for equality once. This approach will narrow the search down to one gene and then check if that is the one being searched for. This method will be discussed in lectures and labs; it is *not* the commonly given form of a binary search algorithm (which makes two comparisons per loop).



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

### 3.4 Part B: Theory Questions [4 Marks]

1. What is the best case big-O complexity for your `binary_gene_match` function, given there are  $n$  items in both the first genome and second genome? Explain how this would come about and give a small example of best case input.
2. What is the worst case big-O complexity for your `binary_gene_match` function, given there are  $n$  items in both the first genome and second genome? Explain how this would come about and give a small example of worst case input.

## 4 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` – 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 of 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; and importantly, failing the test indicates that something fundamental may be wrong). Be sure to test various edge cases such as when an input list is empty or when no number genes are common etc. . .

### 4.1 Writing your Own Tests

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 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.

#### 4.1.1 Provided Data for Testing

The `utilities.py` module contains functions for reading test data from test files.

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-s.txt` where:

- Each file contains three DNA sequences, each gene is 16 base pairs long:
  - The first two DNA sequences are genomes, belonging to two patients with a common disease.
  - The third sequence represents all the genes they have in common.
- $i$  = number of genes (or  $16i$  DNA bases) in both patient's genomes.
- $j$  = number of genes common to both genomes.

- `s` (optional) = the second genome is sorted if present.

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 (**not** Notepad).

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.

#### 4.1.2 Provided Tools for Testing

The `utilities.py` module contains functions for reading test data from test files. 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, alongside the genes they have in common as a `GeneList`.

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

---

```
>>> import utilities
>>> filename = "TestData/test_data-2-1.txt"
>>> genome_a, genome_b, common_genes = utilities.read_test_data(filename)
>>> genome_a
agtacgggtctgaaact 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'>
```

---

## 4.2 Provided tests

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


- These tests are **not** very helpful for debugging—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 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`
  - You just need to know that the tests will check that your code finds the list of common genes, and that the number of comparisons that your code makes is appropriate.



### 4.2.1 Running the Provided Tests

The `tests.py` provides a number of tests to perform on your code:

- 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—a value your code produced was not the expected value for that test;
  - which *exception* was thrown—your code produced a Python exception when run.

 **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

- A log of updates will go here.

— Have fun —