COSC122 2023 Semester 2
INTRODUCTION TO COMPUTER SCIENCE

Assignment 3: Prioritising Patients

5% of overall grade
Due Friday 20 October 2023, 11:55pm

1 Overview

Note: this section has the same information as Section 1 in Assignment 1, except for the due date; the new material starts in Section 2.

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` 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

Assignment 3 has one due date: Friday 20 October 2023, 11:55pm.

Note: You can submit it up to one week after the due date, but you will incur a 15% absolute penalty (in other words, your mark will be the raw mark less 15 marks (given the assign-

ment 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. The submission page will be open closer to the due date.

1.4 Implementation

All the files you need for this assignment are on the quiz server. Do not import any additional libraries unless explicitly given permission within the task outline. There is a file with methods that you should complete. 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 Moodle, 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, and don't cheat yourself of the learning by having someone else write material for you.

2 Prioritising patients for treatment

Diagnosis is only half the battle. The algorithms we have built over the last two assignments have proven so successful, we now have a large queue of patients that must be treated. However, a basic queue will not do: some of the genetic disorders diagnosed are more serious than others, and should be treated accordingly. For Assignment 3, you will be tasked with designing and implementing a patient queue that is able to deal with the influx of new patients, and ensuring that they receive treatment in an appropriate order. This will be a Priority Queue, implemented as a binary heap.

> ⚠ **Important:** The heap will be stored in a Python list, and have the root at index 0 (unlike the labs, which had the root at index 1).

Your heap should contain `Patient` objects, which store the name, severity of diagnosed disease, how long they have been waiting for diagnosis, and priority of each patient in the queue.

The priority is how you will determine the treatment order of patients, and is stored in a `Priority` object. In this assignment, we will be counting comparisons between two `Priority` objects.

You will need to use specific data structures (e.g. `Patient` and `Priority`), located in `classes3.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 later.

2.1 Provided classes

The `classes.py` module contains three useful classes for you to work with: `Priority`, `Patient`, and `PriorityQueue`. These are described in more detail below.

`Priority` A wrapper around an `int`, and as such you can use it in the same way. All the common comparison operations are available, but we count these and store them in the `StatCounter`.

`Patient` A very simple class that just maintains a few properties about a person. The most important are their `name`, and their `priority`.

Example: interactive Python Tutor demo.

`PriorityQueue` An *abstract base class*, which all of the classes written in this assignment inherit from. It defines the methods a priority queue must implement: `enqueue` an item that can be compared in some way (in this case, we use patients that have a `priority` attribute for you to use), and `dequeue` items from the queue when they reach the front.

A `PriorityQueue` object has a `comparisons` instance variable (i.e., `self.comparisons`) that you should increment whenever you make a priority comparison.

The `stats.py` module contains a familiar class that you might 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 on the quiz server at the time of submission, so do not use it in your final code.

3 Tasks [100 Marks Total]

For Tasks 1 through 3, you will be working in the `PatientQueueHeap` class. Your code must keep track of comparisons made between two Priority objects. Your code should increment the `PatientQueueHeap`'s comparisons attribute whenever such comparisons are made with a line like `self.comparisons += 1`.

3.1 Sifting up and enqueueing patients [40 Marks]

A heap has two fundamental operations: sift-up, and sift-down. For this task, you are to implement the sift-up operation.

PatientQueueHeap._sift_up A sift-up operation proceeds by determining the parent index `p` of the sifted index `i`. If the data stored at `i` has a higher priority than the data stored at `p`, then they need to be swapped. Otherwise, the sifting up is complete. Repeat this process until the data that started in position `i` is in the right place (this is called "restoring the heap invariant").

Note: To help you, we have provided implementations of `self._parent_index(i)` and `self._swap(i, j)` (be aware that neither do any bounds checking).

PatientQueueHeap.enqueue Once you have implemented `_sift_up`, implement the `enqueue` method. This should be a very short method, relying almost entirely on `_sift_up`.

After completing this task, you should be able to use the heap as follows:

```
>>> from classes3 import Patient
>>> from utilities import verify_heapness
>>> patient = Patient("Nancy Toney", 50, 31)
>>> my_heap = PatientQueueHeap()
>>> print(my_heap)
PQ[]
>>> my_heap.enqueue(patient)
>>> print(my_heap)
PQ[Patient('Nancy Toney', 5600)]
>>> verify_heapness(my_heap)
True
>>> # To properly test your code, enqueue more than one patient!
```

3.2 Sifting down and dequeueing patients [40 Marks]

PatientQueueHeap._sift_down The `_sift_down` method is the dual of the `_sift_up` method. Rather than comparing the data at index `i` with the parent, we compare the data with that at the positions of the children c_1 and c_2. After selecting the appropriate child `c` using `self._max_child_priority_index(child_indices)` (this method does perform bounds checking), consider whether the data at position `i` should swap with that at `c`.

Does the data at position `c` have a higher priority than that at position `i`? Repeat this procedure until the heap invariant is restored.

Note: As before, use the provided methods.

PatientQueueHeap.dequeue Once you have implemented `_sift_down`, please implement the `dequeue` method. This should also be a very short method, leaving most of the work to `_sift_down`.

After completing this task, you should be able to use the heap as follows:

```
>>> from classes3 import Patient
>>> my_heap = PatientQueueHeap()
>>> patient = Patient("Nancy Toney", 50, 31)
>>> my_heap.enqueue(patient)
>>> print(my_heap)
PQ[Patient('Nancy Toney', 5600)]
>>> my_heap.dequeue()
Patient('Nancy Toney', 5600)
>>> print(my_heap)
PQ[]
>>> # To properly test your code, enqueue & dequeue more than one patient!
```

3.3 Building the queue in $O(n)$ time [20 Marks]

By this point, you should have a fully functioning priority queue, based around the heap data structure. If you look at how we build the heap currently, you will see that any existing data imported is simply enqueued one by one. Because when importing information we have the data all at once from the beginning, it turns out we are able to build the heap in less than $O(n \log n)$ time—we can build the heap in $O(n)$ time, in fact, using a technique called *fast heapify*.

PatientQueueHeap.fast_heapify For this task, you are expected to implement the `fast_heapify` method in the `PatientQueueHeap` class. It should not take more than a few lines.

As a hint, consider that the "slow" heapify uses enqueue, which in turn relies on the `_sift_up` method. Why might this not be the best option? What other options are there?

> ⚠ **Important:** This task appears simple, but is conceptually difficult. You do not need to rigorously prove that your implementation runs in $O(n)$ time, but you should try to convince yourself why it does. Allow sufficient time to solve this problem.

After completing this task, you should be able to use the heap as follows:

```
>>> from classes3 import Patient
>>> from utilities import verify_heapness, read_test_data
>>> patients, _ = read_test_data("test_data/test_data-5-5-0.txt")
```

```
>>> my_heap = PatientQueueHeap(start_data=patients, fast=True)
>>> verify_heapness(my_heap)
True
>>> # As before, you should test your code using other inputs too!
```

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 debuging and testing edge cases
- Using the provided tests – 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. 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 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.

4.2 Provided tools and test data

The `utilities.py` module contains functions for reading test data from test files. Test files are in the folder `test_data` to make it easier to test your own code.

The test data files are named `test_data-i-j-k.txt` and contain:

- A single number in the first line, equal to `i`, which is the number of existing records that we are going to import into our patient queue, followed by `i` lines of patient data. Patient data is stored as

  ```
  name, disease_severity, days_since_diagnosis
  ```

  The provided `run_tests` function can read these and turn them into a list of `Patients`, which is then passed to the heap as imported `start_data`. The data is put in the heap using the `_heapify` method (or `_fast_heapify` when testing Task Three).
- There are then `j + k` lines, each of which starts with either `enqueue`, `dequeue`. Of these, `j` lines will start with `enqueue` and `k` lines will start with `dequeue`. (The order of these lines is *almost* random: we will not ask you to dequeue from an empty queue.)
  - `enqueue` lines contain patient data (like imports). Our testing code will enqueue that patient.

- `dequeue` lines contain a patient name. Our testing code will check that this patient is dequeued from the patient queue.

The function `verify_heapness` traverses a heap to check that the heap invariant is maintained. This might be useful for your own testing.

4.3 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 correctly follows the heap enqueue, dequeue & import instructions in the file.

4.3.1 Running the Provided Tests

The `tests.py` module 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 code testing. This should include testing the trivial cases such as empty parameters and parameters of differing sizes.

5 Afterword

Throughout these assignments, you have been working with DNA. The work we have done here is not unlike real research going on around the world at this moment, although admittedly we are working on a much smaller scale. The most prominent example of computer science and DNA mixing is the story of Prof. Matthew Might of the University of Duke in the United States. While the full story is worth reading[1], a short summary follows.

Prof. Matthew Might's first son, Bertrand, was born with no apparent abnormalities. After six months, it became clear that his development was not proceeding correctly. After multiple mis-diagnoses, a genetic disorder was suspected. None matched the symptoms. Those that almost matched the symptoms were not found in Bertrand's genome.

Genome sequencing, at the time, was very expensive, however exome sequencing was more viable. The exome is a small part of the genome responsible for the vast majority of genetic disorders. Sequencing DNA is still not error-free, but even so, no notable common inconsistencies occurred between Prof. Might and his wife. Their different ethnic backgrounds all but precluded common ancestors in the past few thousand years. The chances of them having the same genetic mutation to pass onto Bertrand were vanishingly small.

They did not have the same "errors" in their genome. Two different mutations between them had combined: each genetic abnormality occurs in at most one person per thousand. Two people with those abnormalities results in a one-in-a-million chance of both parents having the appropriate abnormalities. The child will only exhibit symptoms if they inherit both faulty genes, a one-in-four chance. Thus the chance of Bertrand's disease occuring is—at most—one in four million. Bertrand was patient 0, the very first person to be diagnosed with what is now called NGLY1.

Since Bertrand's diagnosis, at least 15 other cases have been diagnosed. Prof. Might now works for the School of Biomedical Informatics at the Harvard Medical School, helping use computer science techniques to develop personalised medical treatment.

[1] http://matt.might.net/articles/my-sons-killer/