COSC122 2017
Introduction to Algorithms

Assignment 3: Prioritising Patients

7% of overall grade
Due Friday 20 October 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 20 October 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. The taks relate to two files that contain classes for which you should complete the implementation. You need to complete the methods marked as such, but you can also write other methods that these methods 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 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.

The four tasks for this part of the assignment all build towards a common goal: implement a binary heap that is capable of handling patients with various priorities, and adding and removing them from the heap as required. As in previous parts of this assignment, we will provide some supporting classes for you to use when you are designing and building the heap. The first heap you will be writing, `PatientHeapQueue`, is a subclass of `PriorityQueue`. The `PriorityQueue` serves as the abstract data type that we are adhering to, and being a queue, requires all subclasses to provide at minimum `enqueue` and `dequeue` methods. These heaps 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 be determining the order that patients should be treated. Unfortunately, it is a reality of medicine that people will need to be removed from the queue despite not being at the front—if their priority was incorrect, or due to unforseen circumstances, a patient will not make it to treatment. To account for this, we will be creating an `EditablePatientHeapQueue` that has the property of $O(\log n)$ remove from anywhere in the heap, not just the top.

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 correctly adds and removes patients to/from the queue, and that your code counts the number of priority comparisons correctly. We will be checking that your comparisons are 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 th expected number of comparisons range considered. 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).

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.

The `utilities.py` module contains another 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.

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

`PriorityQueue` The `PriorityQueue` class is 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.

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-k-l.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 type of heap created depends on the class name given as a parameter to `run_tests`. If the parameter `fast` is set to `False`, the data is put in the heap using the `_heapify` method. But if the `fast` parameter is `True`, the data is put in the heap using the `_fast_heapify` method.

- There are then `j + k + l` lines, each of which starts with one of the following words: `enqueue`, `dequeue`, or `remove`. Of these, `j` lines will start with `enqueue`, `k` lines will start with `dequeue`, and `l` lines will start with `remove`. The order of these lines is *almost* random: we will not ask you to remove or dequeue from an empty queue, but otherwise any ordering is valid.
  - If the line starts with an `enqueue`, the rest of the line will contain patient data as for imports. This will trigger our testing code to put that patient into the priority queue.
  - If the line starts with a `dequeue`, the rest of the line will contain a patient name. We will check that this name matches the name of the `Patient` object dequeued from the patient queue.
  - If the line starts with a `remove`, the rest of the line will contain a patient name. We will ask you to remove th patient with this name from the queue, and then we check that this was done successfully.

There is a function in `utilities.py` called `verify_heapness`, which traverses a heap to check that the heap invariant is maintained. This might be useful for your own testing.

2.3 Provided tests

The `tests.py` module 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 the method names it is testing. In the case of a test case failing, the test will print which assertion failed or what exception was thrown. The `all_tests_suite()` function has a number of lines commented out indicating that these tests will be skipped; uncomment these lines as your progress through the assignment tasks to run further 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]

For tasks 1 through 3, you will be working in the `PatientHeapQueue` class. For task 4, you will be working in the `EditablePatientHeapQueue` class. Do not forget to increment the `self.comparisons` counter when you perform a priority comparison.

3.1 Sifting up and enqueueing patients [30 Marks]

A heap has two fundamental operations: sift-up, and sift-down. For this task, you are to implement the sift-up operation, and use to it enqueue patients. A sift-up operation procedes 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"). Use this description to complete the method `_sift_up`.

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.) We recommend you use these methods, even if you could write them yourself—they are used in later tasks in more interesting ways.

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 utilities import *
>>> from patient_queue import *
>>> heap = run_tests("TestData/test_data-0-1-0-0.txt", PatientHeapQueue)
Enqueueing Patient(Nancy Toney, 41500)
>>> print(heap)
PQ[Patient(Nancy Toney, 41500)]
>>> verify_heapness(heap)
True
>>> print(heap.dequeue())
Patient(Nancy Toney, 41500)
>>> print(heap)
PQ[]
>>> verify_heapness(heap)
True
```

3.2 Sifting down and dequeueing patients [30 Marks]

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)` (note that this method does perform bounds checking), consider whether the data at position `i` should swap with that at `c`—that is, does the data at position `c` have a higher priority than that at position `i`. Repeat this procedure until the heap invariant is restored.

As before, use the provided methods, for example `self._child_indices(index)`. While it also does not perform any bounds checking, we recommend using it (and `self._swap(i, j)`) as we will make use of it later.

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`.

3.3 Building the queue in $O(n)$ time [15 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*. For this task, you are expected to implement the `fast_heapify` method in the `PatientHeapQueue` 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 very 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.

3.4 Removing patients from any position in the queue [25 Marks]

An unfortunate reality of medicine is that, sometimes, the treatment cannot come fast enough. In this case, it is useful to be able to remove patients from the queue that are no longer waiting. However, if we were to use our current priority queue implementation, a naïve approach would require us to dequeue patients until the person we want is dequeued, and then re-enqueue all the people we just dequeued. This would take $O(n \log n)$ time which, while not terrible, is certainly not as efficient as it could be.

A better algorithm would to be to perform a linear search through the underlying array, and when the person is found, move the last person into their position and sift up/down, as appropriate. The final sifting is $O(\log n)$, but the initial search is $O(n)$—hence, this algorithm runs in $O(n)$ time. Better, but not the best it could be.

For this task, you will be implementing an `EditablePatientHeapQueue`, which enables arbitrary patient deletion (by name) in $O(\log n)$ time using a new `remove` method. As with most efficiency improvements, we have traded space for time: we will store the index of every patient in a Python dictionary `self.indices`, keyed by their name. This way, finding a patient is $O(1)$ time, so the dominant operation is the subsequent sifting of the replacement. The wrinkle in the plan is that we need to keep track of where a patient is stored, even when sifting. For this reason, you should provide a new version of the `enqueue`, `dequeue`, and `_swap` methods which take the new `self.indices` dictionary into account.

⚠ **Important:** Do not reinvent the wheel. Use `super()` to your advantage so that you can reuse code you have already written in Tasks 3.1–3.3.

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

```
>>> from utilities import *
>>> from editable_patient_queue import *
>>> heap = run_tests("TestData/test_data-0-1-0-1.txt",
    EditablePatientHeapQueue)
Enqueueing Patient(Juan Daudelin, 20169)
Removing Juan Daudelin
>>> print(heap)
PQ[]
```

4 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[2], 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.

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