Problem Sheet 2 Notes

Luke Jones

Candidate Number: 229538

1 Circle Class

```
//To compile I entered the following
g++ -o q1 Circle.cpp
```

For members of a class to be accessible outside of the class object you can declare the variables under the access modifier **public**. Without this, you need to utilise public member functions that have access to the private members. You could also use a **struct** rather than a **class** as the default access modifier is public whereas a class has the default as private.

Using the overloaded > operator, I can write the following bool expression to test for equality:

```
// Where both A and B are Circle objects
bool equalAB = !(A > B) && !(B > A);
```

Here we should see ***equalAB*** be set to false. If instead we use my copy constructor to duplicate an existing Circle object, my boolean expression will return true when they are compared.

```
// Copy Constructor
Circle D(A);
bool equalAD = !(A > D) && !(D > A);
```

2 Matrix Transpose

```
//To compile I entered the following
g++ -o q2 matrixMain.cpp matrixTranspose.cpp
```

The first idea I had was to use 2 ***for*** loops where for each element we perform the swap - this obviously lead to 'double' swapping. I then thought to make a copy of the matrix and use this as a template for transposition. As we hadn't been set any space complexity constraints and the matrix was very small, this was a reasonable solution. I wanted to see if this operation could be done 'in place' and thankfully, we had been given a square matrix so I could just flip the upper and lower triangles - I explored the algorithm for transposing non-square matrices and that appears to be considerably more involved.

3 Virus Outbreak

```
//To compile I entered the following
g++ -o q3 virusMain.cpp Person.cpp virusFunctions.cpp -std=c++11
```

I tried 2 different implementations of the Person class; my first class used bools to characterise the state of a Person which could have ***isInfected*** as true or false as well as ***hasRecovered*** as true or false. I thought this could be more streamlined so I encoded the state of a Person into integers using an **enum**.

My initial implementation of the simulation had a function that removed people that had died with the thought that I would not have to check if a person was alive before calling the **meeting()** function. Whilst this functioned as expected when I reduced the number of meetings each day (≤ 200 meetings per day appeared to work), when I set it to the specified 50,000, it led to a stack dump. I knew that **vector** re-allocation can be expensive so I also tried using a **deque** - here I swapped the target person with the final **deque** element to allow me to use the $O(1)$ **pop_back** functionality of deques. When this resulted in similar functionality, I briefly considered **map** and **multiset** but finally decided it would be easier to implement a function to check if a meeting is valid. This lead to me making the **selectRandom-**

Pair() function. This handles the selection of a pair to meet by selecting an index and using **while** loops to ensure that both people are alive and unique.

C++98 vs C++11

I did write the simulation in both C++ standards but I submitted the C++11 implementation because I am more confident in the randomness used because I can utilise the **random** header. This lets me use stronger pseudo-random generators such as the one I used, the Mersenne Twister 19937 generator. In the 98 standard, I used the **rand()** function again to produce a float in the range [0, 1] which I then shifted into the range [0, totalPopulation] with my additional **choose()** function. I was surprised at how similar my results were when using the C++98 **rand()** function despite its comparatively limited randomness - the main advantage of using Mersenne Twister is the increased period. **RAND_MAX** has a period of [0, 32767] whereas Mersenne Twister produces integers in the range $[0, 2^{w-1}]$ where in my implementation $w = 32$. This means I could in theory increase the length of my simulation far beyond 365 days.

```
//C++11 Random selection of a person
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_real_distribution<> select(0, popSize);
float RNG() { return dis(gen); }

//C++98 Random selection of a person
float RNG() { return static_cast<float>(rand()) / RAND_MAX; }
int choose(int &popSize) { return roundf(RNG() * popSize); }
```

I output my simulation to a .csv file so that I could manipulate the data in Python - the results of my simulation are plotted in Figure 1. At the end, as noted in the figure legend, we see that $\approx 6.7\%$ of the total population (13,393 people) died as a result of the virus. We see that at its peak, a total of 32,096 people were infected on the 64th day. The graph shows, however, that the majority of people recovered and developed immunity to the virus, making up 73.6% (147,189 people) of the total population. Interestingly, we see that 19.7% of the population (39,418 people) were still in the 'vulnerable' category at the end of the simulation. This will be as a result of us performing my **daycheck()** function from day 0 - if there was an incubation time

before people recovered or the virus killed its host, we would see a complete spread of the virus throughout the population. I did test this by putting my **daycheck()** function in a **if** block to only trigger after 45 days, however, due to the relatively high probability of recovery, I still observed similar recovery and mortality rates.

I have also included an animated .gif version of my plot as an extra because I think it provides a nice illustration of the simulation and how the population health changes over time.

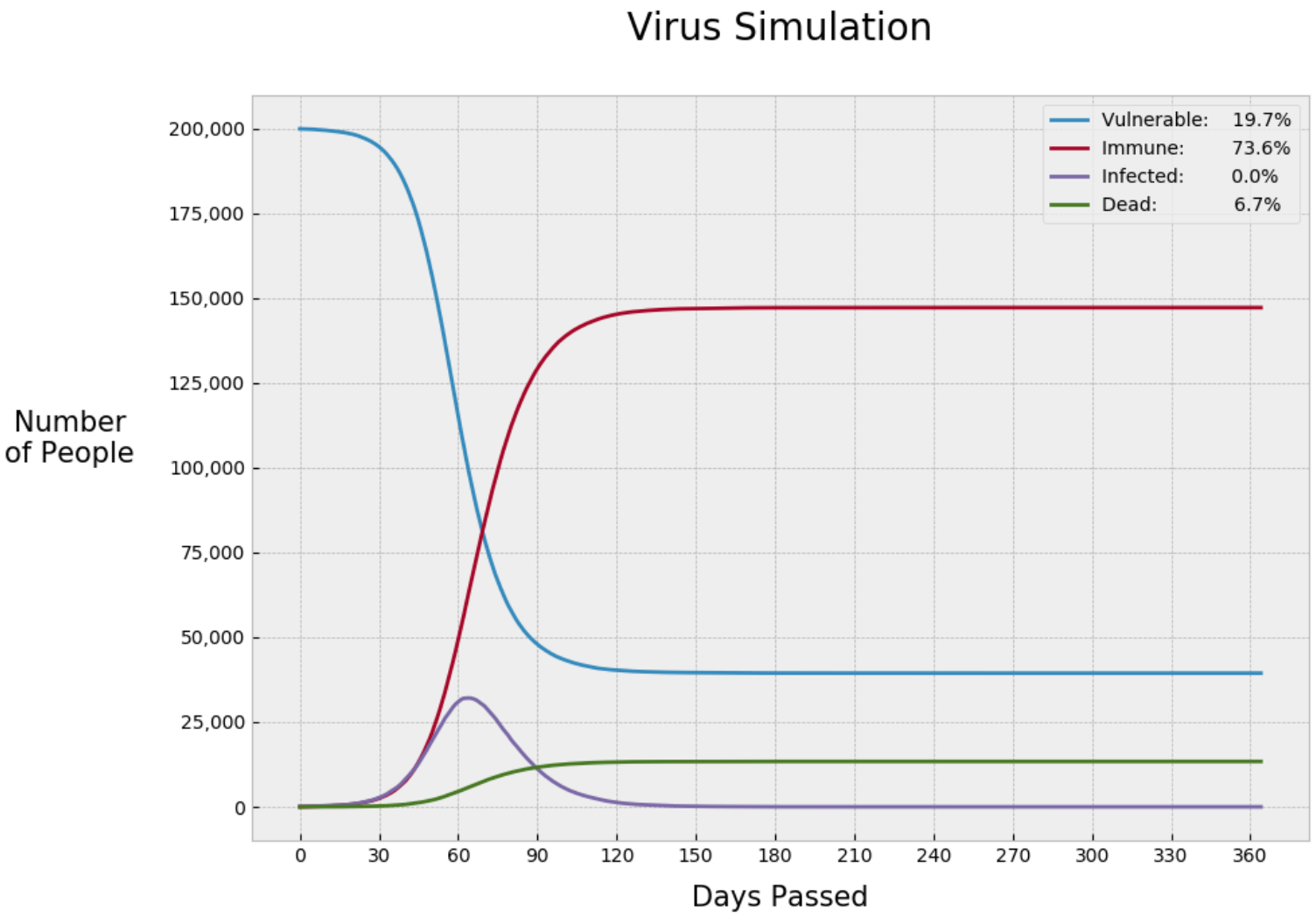

Figure 1: A plot illustrating the current state of the population over time. We see that the population develop a 'herd immunity' resulting in a very low probability for vulnerable people to meet an infected person. This slows the spread considerably and eventually leads to the elimination of the virus because it is unable to spread faster than people are able to recover. These are results from my C++11 implementation.