Semester Project

Harrison Zheng
harrison.zheng1@marist.edu

December 15, 2021

1 Assignment Description

The semester project asked students to develop a simulation of group/pooled testing. The testing protocol works by dividing a population into smaller groups and collecting a sample (i.e. saliva) from each person in a group. The samples are then combined into one and tested as one. If the test result is negative, then no one in the group is infected. However, if the test result is positive, then the group is divided into smaller groups with a test performed on each subgroup and, if necessary, every individual in each subgroup until it can be determined who the infected person(s) is. If the infection rate in the population is low enough, this protocol can significantly reduce the number of tests needed to keep the population safe. Marist College utilized group testing throughout the 2020-2021 school year to track and limit the spread of COVID-19. For their projects, students ran their testing simulation on populations of 1,000, 10,000, 100,000, and 1,000,000 people with an infection rate of 2% and put them into groups of 8. Then, they needed to output the results of their simulation.

2 My Implementation

In the first run of my simulation, I start with a population size of 1000 and I create an instance of the Population class with this number (Listing 2.1). The Population class holds an array of instances of the Person class (Listing 2.3). An instance of the Person class represents a person that will be screened for COVID-19 and currently, it only describes whether the person is infected (Listing 2.2). The program then calls the infectPeople function, which infects a number of people at the front of the population array and then uses the Knuth shuffle from previous assignments to spread out the infected people. This ensures that multiple groups will have at least one infected individual when we do testing. Next, the program calls the testPeople function, which tests the population in groups of 8 at a time (case 1). If it finds an infection in the first test, it divides the group into two subgroups of 4 and performs a test on each group (case 2). If it finds that only one subgroup has an infected individual, then it only individually tests the people in that subgroup. However, if it finds that both subgroups have an infected individual, it individually tests all 8 people (case 3). The program keeps track of the number of tests performed and the number of occurences of each case as it goes through the testPeople function. Finally, the program prints out the total number of tests needed to screen the population and some other information. It repeats these steps for population sizes 10,000, 100,000, and 1,000,000.

2.1 Simulation Class

```
import java.text.DecimalFormat;

public class Simulation {
    public static int POPULATION_SIZE = 0;
    public static int GROUP_SIZE = 8;
    public static int GROUP_COUNT = 0;
    public static double INFECTION_RATE = .02;
    public static int testCount = 0;
    public static int case1Count = 0;
    public static int case2Count = 0;
    public static int case3Count = 0;

    public static void main(String[] args) {
        //Run my pooled testing simulation on populations
        //of various sizes ranging from 1000-1M people
        runSimulation(1000);
        runSimulation(10000);
        runSimulation(100000);
        runSimulation(1000000);
    }

    //Run simulation with given population size
    public static void runSimulation(int popSize) {
        POPULATION_SIZE = popSize;

        Population simulatedPop = new Population(POPULATION_SIZE);
        simulatedPop.wholePop = infectPeople(simulatedPop.wholePop);
        testPeople(simulatedPop.wholePop);
        printOutput();

        testCount = 0;
        case1Count = 0;
        case2Count = 0;
        case3Count = 0;
    }

    //Infect people in the population based on the infection rate
    public static Person[] infectPeople(Person[] population) {
        //Infect the first 20 people in the population array
        int peopleToInfect = (int)Math.round(INFECTION_RATE * population.length);
        for (int i = 0; i < peopleToInfect; i++) {
            population[i].isInfected = 1;
        }

        //Randomly distribute infected people throughout
        //the population with a Knuth shuffle
        for (int i = 0; i < population.length; i++) {
            //Create a "random" value
            int rand = i + (int) (Math.random() * (population.length - i));

            //Swap the value at this index with the value at
            //the index of the randomly generated number
            Person temp = population[rand];
            population[rand] = population[i];
            population[i] = temp;
        }
        return population;
    }

    public static void testPeople(Person[] population) {
        //Place people into groups of 8 (125 groups in our case)
        GROUP_COUNT = POPULATION_SIZE / GROUP_SIZE;

        //Perform necessary tests on each group
        for (int i = 0; i < GROUP_COUNT; i++) {
            //Create a new testing pool
            Person[] currentPool = new Person[GROUP_SIZE];

            for (int j = i * GROUP_SIZE; j < ((i * GROUP_SIZE) + GROUP_SIZE); j++) {
                currentPool[j%GROUP_SIZE] = population[j];
            }

            //Test the intial group of 8
            testCount++;
            if (checkForInfection(currentPool)) {
                //If an infection was found,
                //divide pool into two equal size groups
                Person[] newPool1 = new Person[GROUP_SIZE / 2];
                for (int j = 0; j < (currentPool.length / 2); j++) {
                    newPool1[j] = currentPool[j];
                }
                Person[] newPool2 = new Person[GROUP_SIZE / 2];
                for (int j = (currentPool.length / 2); j < currentPool.length; j++) {
                    newPool2[j%(GROUP_SIZE / 2)] = currentPool[j];
                }

                //Test the two subgroups
                if ((checkForInfection(newPool1) && !checkForInfection(newPool2)) || (!
                    ↪ checkForInfection(newPool1) && checkForInfection(newPool2))) {
                    //Case 2: If only one subgroup showed infection,
                    //then we used 6 more tests (1 per subgroup + 4 for people in infected
                        ↪ group)
                    testCount += 6;
                    case2Count++;
                }
                else {
                    //Case 3: If both subgroups showed infection,
                    //then we used 10 more tests (1 per subgroup + 8 for people in infected
                        ↪ groups)
                    testCount += 10;
                    case3Count++;
                }
```

```java
            }
            else {
                //Only 1 test was needed because nobody was infected
                case1Count++;
            }
        }
    }

    public static boolean checkForInfection(Person[] pool) {
        int giantTube = 0;

        //Combine the isInfected value of everyone in the pool
        //as you would combine the biomarkers in real life
        for (int i = 0; i < pool.length; i++) {
            giantTube += pool[i].isInfected;
        }

        //If the combined value is still 0, then nobody was infected
        if (giantTube == 0) {
            return false;
        }
        //If the combined value is > 0, then at least one person was infected
        else {
            return true;
        }
    }

    public static void printOutput() {
        DecimalFormat df = new DecimalFormat("#.##");
        //Print the expected instances and tests
        System.out.println("Expected values for a population of size " + POPULATION_SIZE);
        System.out.println("Case (1): " + GROUP_COUNT + "x0.8500 = " + Double.valueOf(df.format(GROUP_COUNT * 0.8500)) + " instances requiring " + Math.round(GROUP_COUNT * 0.8500) + " tests");
        System.out.println("Case (2): " + GROUP_COUNT + " x 0.1496 = " + Double.valueOf(df.format(GROUP_COUNT * 0.1496)) + " instances requiring " + Math.round((GROUP_COUNT * 0.1496) * 7) + " tests");
        System.out.println("Case (3): " + GROUP_COUNT + " x 0.0004 = " + Double.valueOf(df.format(GROUP_COUNT * 0.0004)) + " instances requiring " + Math.round((GROUP_COUNT * 0.0004) * 11) + " tests");
        System.out.println("----------------------------------------------------------------------------");
        int expectedTests = (int) (Math.round(GROUP_COUNT * 0.8500) + Math.round((GROUP_COUNT * 0.1496) * 7) + Math.round( (GROUP_COUNT * 0.0004) * 11));
        System.out.println(expectedTests + " tests are expected to screen a population of " + POPULATION_SIZE + " people for a disease with 2% infection rate.\n");

        //Print the actual numbers generated by my simulation
        System.out.println("Actual results for a population of size " + POPULATION_SIZE);
        System.out.println("Case (1): " + case1Count + " instances requiring " + case1Count + " tests");
        System.out.println("Case (2): " + case2Count + " instances requiring " + Math.
            ↪ round(case2Count * 7) + " tests");
        System.out.println("Case (3): " + case3Count + " instances requiring " + Math.
            ↪ round(case3Count * 11) + " tests");
        System.out.println
            ↪ ("-------------------------------------------------------------------------------")
            ↪ ;
        System.out.println(testCount + " tests were actually used to screen a population
            ↪ of " + POPULATION_SIZE + " people for a disease with 2% infection rate.\n");
    }
}
```

2.2 Person Class

```java
public class Person {
    int isInfected;

    //Initialize each Person as not infected
    public Person() {
        this.isInfected = 0;
    }
}
```

2.3 Population Class

```java
public class Population {
    //Will consist of 0s and 1s
    Person[] wholePop;

    //Initialize population array
    public Population(int populationSize) {
        this.wholePop = new Person[populationSize];
        for (int i = 0; i < populationSize; i++) {
            this.wholePop[i] = new Person();
        }
    }
}
```

3 Results of Simulation on Various Population Sizes

The results for all runs of the simulation are shown below. The decimal numbers used to calculate the expected instances of each case are based on the likelihood of each scenario occurring with a 2% infection rate and they come from Dr. Labouseur's article on pooled testing. The results suggest that there will most often be a difference in the number of tests we expect to use and the number that we actually use due to randomizing the infections. The actual and expected results are much closer for populations of 1,000 and 10,000 than they are for populations of 100,000 and 1,000,000 since the randomness of infections will have a greater effect as the population gets bigger. One should also note that each time population size increases by a factor of 10, so does the total number of tests needed (approximately).

3.1 Results for a Population of 1,000

Expected Results
Case (1): 125 x 0.8500 = 106.25 instances requiring 106 tests
Case (2): 125 x 0.1496 = 18.7 instances requiring 131 tests
Case (3): 125 x 0.0004 = 0.05 instances requiring 1 tests

---

238 tests are expected to screen a population of 1000 people for a disease with 2% infection rate.

Actual Results
Case (1): 105 instances requiring 105 tests
Case (2): 20 instances requiring 140 tests
Case (3): 0 instances requiring 0 tests

---

245 tests were actually used to screen a population of 1000 people for a disease with 2% infection rate.

3.2 Results for a Population of 10,000:

Expected Results
Case (1): 1250 x 0.8500 = 1062.5 instances requiring 1063 tests
Case (2): 1250 x 0.1496 = 187.0 instances requiring 1309 tests
Case (3): 1250 x 0.0004 = 0.5 instances requiring 6 tests

2378 tests are expected to screen a population of 10000 people for a disease with 2% infection rate.

Actual Results
Case (1): 1069 instances requiring 1069 tests
Case (2): 169 instances requiring 1183 tests
Case (3): 12 instances requiring 132 tests

2384 tests were actually used to screen a population of 10000 people for a disease with 2% infection rate.

3.3 Results for a Population of 100,000

Expected Results
Case (1): 12500 x 0.8500 = 10625.0 instances requiring 10625 tests
Case (2): 12500 x 0.1496 = 1870.0 instances requiring 13090 tests
Case (3): 12500 x 0.0004 = 5.0 instances requiring 55 tests

23770 tests are expected to screen a population of 100000 people for a disease with 2% infection rate.

Actual Results
Case (1): 10627 instances requiring 10627 tests
Case (2): 1801 instances requiring 12607 tests
Case (3): 72 instances requiring 792 tests

24026 tests were actually used to screen a population of 100000 people for a disease with 2% infection rate.

3.4 Results for a Population of 1,000,000

Expected Results
Case (1): 125000 x 0.8500 = 106250.0 instances requiring 106250 tests
Case (2): 125000 x 0.1496 = 18700.0 instances requiring 130900 tests
Case (3): 125000 x 0.0004 = 50.0 instances requiring 550 tests

---

237700 tests are expected to screen a population of 1000000 people for a disease with 2% infection rate.

Actual Results
Case (1): 106334 instances requiring 106334 tests
Case (2): 17924 instances requiring 125468 tests
Case (3): 742 instances requiring 8162 tests

239964 tests were actually used to screen a population of 1000000 people for a disease with 2% infection rate.

4 Binomial vs. Hypergeometric Distribution

In the field of statistics, a binomial distribution is a probability distribution of the number of "successes" over a sequence of trials. To calculate the binomial probability of successes at a certain point in the distribution, use the formula

$$P(x) = \binom{n}{x} p^x q^{n-x}$$

where x is the total number of successes, p^x is the probability of a success on a single trial, q^{n-x} is the probability of a failure on a single trial, and n is the number of trials.

A hypergeometric distribution is very similar to a binomial distribution, except it describes the number of successes over a sequence of trials from a finite population without replacement. This means every time a trial is performed on an individual from the population, that individual is removed from the population for the next trial. To calculate the hypergeometric probability of successes, use the formula

$$P(x) = \frac{\binom{k}{n}\binom{N-k}{n-x}}{\binom{N}{n}}$$

where x is the number of successes, k is the number of failures, n is the sample size, and N is the total number of individuals in the population.

Although we used binomial probabilities to calculate the expected number of instances of each case in our simulations, hypergeometric probabilities actually would have provided us with better estimates, especially for the larger populations. The table in section 4.1 shows the results I found using hypergeometric probabilities. For each population size, I took the number of testing groups (ex: 1250) and multiplied it by the hypergeometric probability (ex: 0.8507) of finding x infected people in a group (x is 0 for case 1, 1 for case 2, and 2+ for case 3), which I found using the formula above. This produced estimates of the number of instances we would get for each case in a simulation and upon comparison, I found that many of them are closer to the actual results from the simulation runs than our original estimates. For example, using hypergeometric probabilities produced an estimate of 12.87 instances for case 3 in a population of 10,000, which is a lot closer to the actual result of 12 than our original estimate of 0.5. This is likely due to the fact that a binomial distribution has a fixed number of independent trials and thus does not model our simulations as well as a hypergeometric distribution, which has a set number of dependent trials. When the simulation is testing a group and it finds that somebody is infected, it divides the group into 2 subgroups and then conducts a pooled test on each one. If it finds that there is an infection in one subgroup but not the other, it removes that half of the group from any further testing. Additionally, once the simulation is done testing one group, it moves on to the next eight individuals in the array that holds the population and does not go back so any individuals that have been tested are effectively removed from the screening population. This way of sampling without replacing the contents of the sample makes a hypergeometric distribution the best way to model the simulation. When the population is small (i.e. 1000 people), which distribution we use to make estimates does not matter much as the numbers come out roughly the same. As the population gets bigger, though, the effect of removing individuals from the screening population after each pooled test becomes more pronounced and estimates from a hypergeometric distribution become more accurate than those from a binomial distribution.

4.1 Expected Results Based On Hypergeometric Distribution

	1,000	**10,000**	**100,000**	**1,000,000**
Case (1)	125 x 0.8503 = 106.28	1250 x 0.8507 = 1063.29	12500 x 0.8508 = 10634.48	125000 x 0.8508 = 106345.32
Case (2)	125 x 0.1398 = 17.48	1250 x 0.1390 = 173.74	12500 x 0.1390 = 1736.37	125000 x 0.1390 = 17362.62
Case (3)	125 x 0.0099 = 1.24	1250 x 0.0103 = 12.87	12500 x 0.0103 = 129.16	125000 x 0.0103 = 1292.06

5 Potential Ways to Improve Simulation

One way that I would improve my simulation is adding a mechanism that creates false positives and false negatives in the test results. For our simulations, we were asked to assume that the tests are 100% accurate. In reality, however, no test used to screen for a disease is 100% accurate. I would implement testing inaccuracies in my simulation by having it give each person in the population two attributes; their test result and their actual infection status. First, it would change 2% of the population's infection status to 1, as usual. Then during the testing phase, in random fashion, it would give the majority of the healthy people a test result of negative and a small percentage (based on real-world false positive rates) of them a test result of positive to serve as false positives. It would also give, in random fashion, most infected people a test result of positive and a small amount (based on real-world false negative rates) of them a test result of negative to serve as false negatives. After testing, my program would go through the population and compare each person's test result with their actual infection status while keeping track of disparities. This is similar to how testing centers need to perform a second test on people to confirm whether their previous result was erroneous. My program would then print out the number of false positives and false negatives in the tests along with the other results from the simulation.

Another way that I would improve my simulation is by making it so that the user can enter values for population size, group size, and infection rate when the program starts up. The program would then check to make sure that the user entered values that are within an acceptable range and run the simulation as well as output results based on those values. Allowing for user input would make my simulation more dynamic and useful for approximating test results in a real-world community where it is unlikely that the population size is an exact power of 10.