Semester Project - Infection Simulation

Joseph McDonough

Joseph.McDonough1@marist.edu

December 11, 2020

1 The Problem

In order to attempt to contain a widespread virus, individuals will needed to be tests. The larger the population is, the most tests that need to be performed. Eventually there comes a point where there are simply too many people to test and not enough resources to perform that many tests. One solution to screen a large population is via pooled testing. Pooled testing gathers multiple samples and combines them together. Through this process, multiple people can be tested simultaneously.

In the event at least one person in the pool was positive, a predefined procedure would take place that would either divide the group in half or just go straight to testing each individual in the pool separately. But if the infection rate is too high, or the pool sizes and divisions are selected poorly, there is a risk that this pooled testing process will require more tests than just testing everyone individually right from the beginning.

The purpose of this simulation is to try and identify the best method to performed pooled testing via trial and error. In this simulation, all the variables at play can be tweak according to the tester desire. The end goal being to find the most efficient and cheapest way to test a large population for an illness.

2 The Simulation

The simulation is programmed in the Java language with the main method calling all the necessary functions to perform the simulation from start to finish. The first task is for the user to end the data they want to be passed into the simulation. The data is requested in the order that follows: population size, infection rate, testing group size, group division amount, and test accuracy. Upon entering all this information, the user is asked to confirm the following values are correct prior to running. An option to print the results for each group is given too. If desired, the simulation will print the tests performed on each individual group. With this feature, the user can see for themselves the amount of positive tests per group and what group they appeared it. This information can be useful because if the user notices that almost all groups make its way to individual tests, some tweaking to group sizes might be required.

Regardless of the user wanting to see each individual group, the simulation will run and print a report in the end that contains the various testing scenarios, the amount of false positive and false negative tests, the total amount of infected people, and the amount of tests that were performed.

Upon completion, the user has the option to quickly re-run the simulation with the same input data, change the data, or end the simulation.

There is also a built-in help menu that can be accessed at any time during the user input stage by entering "-555".

2.1 The Assignment

The predetermined virus situation had a desired starting group of size eight, with one division before individual tests (i.e. one group of eight → two groups of four → four or eight individual tests). The scenarios in which no additional tests were required are denoted as "case one". Those in which only four individual tests were required (i.e. the only positive tests were in the same subgroup of four people) are marked as "case two". And lastly those groups that required everyone to have individual tests (i.e. at least one positive tests per subgroup of four people) are considered "case three".

There is also the assumption of 100% test accuracy so although the amount of false positive and false negative tests will print, it will always be zero.

2.2 Structure

The code is broken up into four subsections: user input, create population, perform tests, and print results.

Gathering user input requests five pieces of simulation data: population size, infection rate, testing group size, group division amount, and test accuracy. All of these elements have their own series of functions that are called to work with each other to make sure all the data is valid and error-checked prior to run-time.

Creating the population results in ensuring the population is of an appropriate size for the desired group, and if not, fixing it such that it will be. Then the members are given a number to determine if they are infected or not. Upon determining whether each individual is infected or virus-free, the false positive and false negative cases get determined. The end result is a Boolean array containing true for infected and false for clean patients.

To perform testing, the population is divided into the specified group sizes. If there are no positive tests in the initial group, the function moves onto the next group. If a positive test does occur in the first group, it delegates that subgroup to a function to handle sub-tests. If the maximum amount of group divisions have been done, then the function built to perform single tests handles that group. Upon determining the amount of positive cases as well as the amount of tests performed on the group, the data gets sent to the results functions to be printed.

Prior to the simulation being ran, the user had the option to print out the individual group results. If that was desired, the outcome of each group is printed as the simulation performs the tests on it. After the simulation is complete, a final report is presented. This report contains all the necessary information and samples can be seen in the test cases.

At the end, there is one more input line where the user determines if they want to run the simulation again with the same numbers, different numbers, or not at all and end the program.

2.2.1 User Input

1. Population Size
    - This is the amount of people that will be tested.
    - Input must be a whole number that is greater than zero.
2. Infection Rate

- This is the percentage of people that will be infected on average.
- Input is a whole or decimal number between 0.0 and 100.0 (inclusive)

3. Testing Group Size
    - This is the amount of people that will be tested in the initial testing group
    - Input must be a whole number greater than zero but less than or equal to the total population size (e.g. cannot have group size of six on a population of size five).
4. Group Division Amount
    - This is the amount of times the groups will be split in half if a positive case is found up until individual tests are required.
    - Input must be a whole number between zero and the whole number that is $(\log_2 groupSize)$ - 1 (inclusive). This is done so groups can be divided evenly and each time. Further examples in Appendix 3.1 - 1.
5. Test Accuracy
    - This is the percentage that represents how accurate the tests are, such that 0% means all tests are inaccurate and 100% means all tests are accurate.
    - Input is a whole or decimal number between 0.0 and 100.0 (inclusive)

2.2.2 Create Population

The creation of the population begins with making sure the desired group size is divisible by the population size. This must be true as each group must have the exact same number of people. In the event the numbers do not divide evenly, people are added to the population until they are (e.g. request a population of fifteen but a group size of eight. One additionally person would be added to the population so two even groups of eight people would be achieved) (Appendix 3.2 - 1).

Once the population size is acquired, each person is assigned a random number between one and 100 (inclusive of both)(Appendix 3.2 - 2). Then the population gets "infected". That is if the person's number is less than or equal to the infection rate, they are deemed infected and marked as true, while if the number is greater than the infection rate, they are clean and marked "false". This is done such that an infection rate of zero would infect zero people since a number equal to or greater than one would always be higher than zero. Likewise, for an infection rate of 100, every number would be equal to or less than 100, resulting in everyone being infected (Appendix 3.2 - 3).

To account for inaccuracy in the tests, the infected population is ran through another function. This function generates a random decimal between 0.0 (inclusive) and 100.0 (exclusive) for each individual. If that individuals number is greater than the testing accuracy, then their infection status is flipped (i.e. test accuracy is 50% and the individual was originally marked infected (true) and their decimal number is 87.3. Their number is greater than the testing accuracy so their status would now appear as clean (false) at the time of test). A test with 100% would not have the ability to flip the status of any patient just like a test with 0% accuracy would flip every patient. There is a counter that keeps track of the amount of false positives and negatives that occur (Appendix 3.2 - 4).

2.2.3 Perform Tests

To perform the tests, there are three functions that need to be called. The first is what checks the original group. It takes the current individual in the population and takes that individual and the next members to make up the current group. These individuals are all checked to see if any of them are marked "true". If

they are all clean and therefore marked "false", there are no need for individual tests and the test count gets incremented by only one. This instance is also a case one scenario, so that counter is increased by one. If there are at least one "true" value in this group, the group gets send to the function to handle sub-tests. This outermost function will only work on groups whose size is that of the user inputted group size. Regardless of if there are any positives tests, the yet-to-be-evaluated population list is modified to excluded the individuals just tests and the next set of individuals get tested (Appendix 3.3 - 1).

The function to complete the sub-tests will first check to make sure that the required divisions have not yet been made before individual tests. If the current level exceeds the allotted divide level, that means no more group tests are requested and the simulation must go right to individual tests. That work gets delegated to another function.

In the perform sub-tests function, the inputted group gets split in half. This is expected as the purpose of this function is to identify which half (or possibly both) is the one with the infected person. Both subgroups are analyzed for the infected ("true") individual. If found, that group gets recursively calls the sub-tests function for further analysis. If not found, that means the infected individual must be in the other group and not everyone needs an individual test. This is a case two scenario and that counter is incremented. If both groups find an infected individual, that meets the criteria of a case three scenario, in accordance with the assignment directions, and the case three counter reflects that. Regardless of if an infection is found or not, a line in the group report will print the status of said group (Appendix 3.3 - 2).

The last function that can be reached in the perform tests category is the one that handles individual tests. This function takes in the for-sure infected subgroup and searches it to identify how many sick individuals are here. The total infection count gets increased by the number found in this function. Upon testing each individual in the inputted subgroup, the function sends the data to the results function that handle it and will print the amount of infected individuals in the group report (Appendix 3.3 - 3).

2.3 The Results

Below are some sample results and analysis for the tests cases laid out in the assignments directions.

2.3.1 Test Cases

For all test cases that will be examined here, the population size varies, but the rest of the data will remain consistent throughout. The infection rate will remain at 2%. The group size is eight tests and will be divided one time, such that sub-tests will be done on two groups of four samples, followed by individual tests on the group of four that need it ($8 \rightarrow 4 \rightarrow 1$). The test is also believed to be 100% accurate such that no false positives or false negatives could possibly occur.

There are three possible cases for each group.

Case one is when the group tests negative and no additional tests are required.

Case two is when the initial group tests positive and the only one of the two groups of four people test positive. This case results in seven total tests, meaning six additional tests were required.

Case three is when the initial group tests positive and both subgroups of four people also test positive. This means that every individual in this group ends up being tested, resulting in ten additional tests were performed.

With an infection rate of 2%, around 85% of the pooled tests are expected to fall under case one, 14.96% are expected to be of case two, and the last 0.04% would be case three.

2.3.2 Population of 1,000 People

With a population of 1,000 people and testing group sizes of eight people, there would be 125 groups made. Therefore, about 106 instances of case one are expected to occur, about 18 instances of case two, and one instance of case three.

As case two results in six additional tests, and case three results in ten additional tests, around 243 total tests are the expected amount.

After running the simulation hundreds of times, these numbers were achieved more often than not. A few instances are shown below

Instance where expected results were achieved

Case (1): 106 - instances requiring no partial tests
Case (2): 18 - instances requiring six additional tests
Case (3): 1 - instances requiring ten additional tests

---

0 false positive tests were identified due to the 100.0% test accuracy rate
0 false negative tests were identified due to the 100.0% test accuracy rate

---

Out of a population of 1000 people, 21 tested positive for the infection.
243 tests to screen a population of 1000 people for a disease with 2.0% infection rate.

Instance where more positive tests than expected

Case (1): 97 - instances requiring no partial tests
Case (2): 26 - instances requiring six additional tests
Case (3): 2 - instances requiring ten additional tests

---

0 false positive tests were identified due to the 100.0% test accuracy rate
0 false negative tests were identified due to the 100.0% test accuracy rate

---

Out of a population of 1000 people, 31 tested positive for the infection.
301 tests to screen a population of 1000 people for a disease with 2.0% infection rate.

Instance where less positive tests than expected

Case (1): 110 - instances requiring no partial tests
Case (2): 15 - instances requiring six additional tests
Case (3): 0 - instances requiring ten additional tests

---

0 false positive tests were identified due to the 100.0% test accuracy rate
0 false negative tests were identified due to the 100.0% test accuracy rate

---

Out of a population of 1000 people, 15 tested positive for the infection.
215 tests to screen a population of 1000 people for a disease with 2.0% infection rate.

2.3.3 Population of 10,000 People

With a population of 10,000 people and testing group sizes of eight people, there would be 1,250 groups made. Therefore, about 1,062 instances of case one are expected to occur, about 187 instances of case two, and one instance of case three.

As case two results in six additional tests, and case three results in ten additional tests, around 2,382 total tests are the expected amount.

After running the simulation hundreds of times, these numbers were achieved more often than not. A few instances are shown below

Instance where expected results were achieved

Case (1): 1060 - instances requiring no partial tests
Case (2): 189 - instances requiring six additional tests
Case (3): 1 - instances requiring ten additional tests

---

0 false positive tests were identified due to the 100.0% test accuracy rate
0 false negative tests were identified due to the 100.0% test accuracy rate

---

Out of a population of 10000 people, 193 tested positive for the infection.
2394 tests to screen a population of 10000 people for a disease with 2.0% infection rate.

Instance where more positive tests than expected

Case (1): 1054 - instances requiring no partial tests
Case (2): 186 - instances requiring six additional tests
Case (3): 10 - instances requiring ten additional tests

---

0 false positive tests were identified due to the 100.0% test accuracy rate
0 false negative tests were identified due to the 100.0% test accuracy rate

---

Out of a population of 10000 people, 219 tested positive for the infection.
2466 tests to screen a population of 10000 people for a disease with 2.0% infection rate.

Instance where expected positive, but high case three

Case (1): 1066 - instances requiring no partial tests
Case (2): 176 - instances requiring six additional tests
Case (3): 8 - instances requiring ten additional tests

---

0 false positive tests were identified due to the 100.0% test accuracy rate
0 false negative tests were identified due to the 100.0% test accuracy rate

---

Out of a population of 10000 people, 200 tested positive for the infection.
2386 tests to screen a population of 10000 people for a disease with 2.0% infection rate.

2.3.4 Population of 100,000 People

With a population of 100,000 people and testing group sizes of eight people, there would be 12,500 groups made. Therefore, about 10,625 instances of case one are expected to occur, about 1870 instances of case two, and five instance of case three.

As case two results in six additional tests, and case three results in ten additional tests, around 23,770 total tests are the expected amount.

After running the simulation hundreds of times, these numbers were achieved more often than not. A few instances are shown below

Instance where expected results were achieved

Case (1): 10652 - instances requiring no partial tests
Case (2): 1760 - instances requiring six additional tests
Case (3): 88 - instances requiring ten additional tests

---

0 false positive tests were identified due to the 100.0% test accuracy rate
0 false negative tests were identified due to the 100.0% test accuracy rate

---

Out of a population of 100000 people, 2000 tested positive for the infection.
23940 tests to screen a population of 100000 people for a disease with 2.0% infection rate.

Instance where more positive tests than expected

Case (1): 10608 - instances requiring no partial tests
Case (2): 1802 - instances requiring six additional tests
Case (3): 90 - instances requiring ten additional tests

---

0 false positive tests were identified due to the 100.0% test accuracy rate
0 false negative tests were identified due to the 100.0% test accuracy rate

---

Out of a population of 100000 people, 2032 tested positive for the infection.
24212 tests to screen a population of 100000 people for a disease with 2.0% infection rate.

Instance where less positive tests than expected

Case (1): 10655 - instances requiring no partial tests
Case (2): 1768 - instances requiring six additional tests
Case (3): 77 - instances requiring ten additional tests

---

0 false positive tests were identified due to the 100.0% test accuracy rate
0 false negative tests were identified due to the 100.0% test accuracy rate

---

Out of a population of 100000 people, 1987 tested positive for the infection.
23878 tests to screen a population of 100000 people for a disease with 2.0% infection rate.

2.3.5 Population of 1,000,000 People

With a population of 1,000,000 people and testing group sizes of eight people, there would be 125,000 groups made. Therefore, about 106,250 instances of case one are expected to occur, about 18,700 instances of case two, and 50 instance of case three.

As case two results in six additional tests, and case three results in ten additional tests, around 237,700 total tests are the expected amount.

After running the simulation hundreds of times, these numbers were achieved more often than not. A few instances are shown below

Instance where expected results were achieved

Case (1): 106342 - instances requiring no partial tests
Case (2): 17899 - instances requiring six additional tests
Case (3): 759 - instances requiring ten additional tests

---

0 false positive tests were identified due to the 100.0% test accuracy rate
0 false negative tests were identified due to the 100.0% test accuracy rate

---

Out of a population of 1000000 people, 20001 tested positive for the infection.
239984 tests to screen a population of 1000000 people for a disease with 2.0% infection rate.

Instance where more positive tests than expected

Case (1): 106128 - instances requiring no partial tests
Case (2): 18078 - instances requiring six additional tests
Case (3): 794 - instances requiring ten additional tests

---

0 false positive tests were identified due to the 100.0% test accuracy rate
0 false negative tests were identified due to the 100.0% test accuracy rate

---

Out of a population of 1000000 people, 20251 tested positive for the infection.
241408 tests to screen a population of 1000000 people for a disease with 2.0% infection rate.

Instance where less positive tests than expected

Case (1): 106361 - instances requiring no partial tests
Case (2): 17910 - instances requiring six additional tests
Case (3): 729 - instances requiring ten additional tests

---

0 false positive tests were identified due to the 100.0% test accuracy rate
0 false negative tests were identified due to the 100.0% test accuracy rate

---

Out of a population of 1000000 people, 19951 tested positive for the infection.
239750 tests to screen a population of 1000000 people for a disease with 2.0% infection rate.

2.3.6 Result Analysis

All in all, pooled testing is a good way to save the amount of tests needed, so long as the infection rate is low enough and group size and division are optimized in such a way. For an infection rate of 2%, which at the time of writing, is higher than the current Covid-19 infection rate of .6% on a world-wide scale, but less than the United States infection rate of about 3%. Regardless, an infection rate of 2% is on the higher than most transmittable viruses. Therefore the method of initial group sizes of eight that go down to four before individual tests would work on a worldwide scale. And for infections with a lower infection rate, the initial group size could be raised, and this simulation has the ability to test those variations.

With the numbers used in these test cases, in almost every simulation, the pooled testing approach saved about 75% of total tests in comparison to only performing individual tests. To test a million people individually, it would take a million tests, but with the pooled approach, 237,700 was the expected amount of tests needed. That saves around 763,300 tests on average. It becomes obvious that pooled test save a great amount of resources and time in a real-world scenario.

3 Appendix

For further comments on the lines of code, see the original code itself.

3.1 User Input

1

Group Division Amount Examples:
Group size 16 with division amount of 3: 16→ 8 → 4 → 2 → 1
Group size 16 with division amount of 2: 16 → 8 → 4 → 1
Group size 16 with division amount of 1: 16 → 8 → 1
Group size 16 with division amount of 0: 16 → 1
Group size 16 with division amount of 4: 16 → 8 → 4 → 2 → 1 → .5

- ERROR because can't go any further than group size 1

Group size 15 with division amount of 2: 15 → 7.5

- ERROR because 15 does not divide evenly

3.2 Create Population

1

```
327  public static void divideIntoGroups()
328  {
329      int difference = popSize MODULUS groupSize;
330      if(difference != 0 )
331      {
332          int adjustment = groupSize - difference;
333          popSize += adjustment;
334          System.out.println("\n*NOTE*\n In order for the population to
335          be divided into even groups, " + "the population size has been
336          increased by " + adjustment + " to a " + "total of " + popSize);
337      }
338  }
```

2

```
342  public static boolean[] createPopulation()
343  {
344      Random rand = new Random();
345      int[] cleanPop = new int[popSize];
346
347      for(int i = 0; i < cleanPop.length; i++)
348      {
349          cleanPop[i] = rand.nextInt(100) + 1;
350      }
351
352      return infectPopulation(cleanPop);
353  }
```

3

```
357  public static boolean[] infectPopulation(int[] cleanPop)
358  {
359      boolean[] infectedPop = new boolean[popSize];
360
361      for(int i = 0; i < infectedPop.length; i++)
362      {
363          if(cleanPop[i] <= infectionRate)
364              infectedPop[i] = true;
365          else
366              infectedPop[i] = false;
367      }
368
369      return identifyFalseTests(infectedPop);
370  }
```

4

```
377 static int falseNegative = 0;
378 static int falsePositive = 0;
379
380 public static boolean[] identifyFalseTests(boolean[] falsePop)
381 {
382     Random rand = new Random();
383
384     for(int i = 0; i < popSize; i++)
385     {
386         int wholeRand = rand.nextInt(100);
387         double decimalRand = rand.nextDouble();
388         double falseNum = wholeRand + decimalRand;
389
390         if(falsePop[i] && falseNum > testAccuracy)
391         {
392             falsePop[i] = false;
393             falseNegative++;
394         }
395         else if(!falsePop[i] && falseNum > testAccuracy)
396         {
397             falsePop[i] = true;
398             falsePositive++;
399         }
400     }
401
402     return falsePop;
403 }
```

3.3 Perform Tests

1

```
428  public static void performTests(boolean[] pop)
429 {
430     boolean infectionFound = false;
431     int whileIndex = 0;
432     int divideLevel = divideCount;
433     boolean[] evalPop = Arrays.copyOfRange(pop, 0, groupSize);
434     int currLevel = 0;
435
436     while(whileIndex < popSize)
437     {
438
439         testCount++;
440
441         if(userWantsCaseResults)
442             System.out.println("\n" + divider + "\nConducting tests on patients " + (while
443
444         for(int i = 0; i < groupSize; i++)
445         {
446             if(evalPop[i])
447             {
448                 infectionFound = true;
449             }
450         }
451
452         testResults(currLevel, infectionFound);
453
454         if(infectionFound && currLevel  < divideCount)                    performSubTests(
455         else if(infectionFound && currLevel == divideCount)                performSingleTe
456         else
457         {
458             currLevel = 0;
459             caseOne++;
460         }
461
462         whileIndex += groupSize;
463         evalPop = Arrays.copyOfRange(pop, 0 + whileIndex, groupSize + whileIndex);
464         infectionFound = false;
465
466     }
467 }
```

2

```
474 public static void performSubTests(boolean[] subgroup, int currLevel, int currGroupSize)
475 {
476     if(currLevel > divideCount)
477         performSingleTests(subgroup, currLevel, currGroupSize);
478     else
479     {
480         boolean[] subGroup1 = Arrays.copyOfRange(subgroup, 0, currGroupSize / 2 );
481         boolean[] subGroup2 = Arrays.copyOfRange(subgroup, currGroupSize / 2, currGroupSiz
482         boolean subGroup1Infect = false;
483         boolean subGroup2Infect = false;
484 
485         testCount++;
486         int i = 0;
487         while(!subGroup1Infect && i < subGroup1.length)
488         {
489             if(subGroup1[i])
490                 subGroup1Infect = true;
491             else
492                 i++;
493         }
494 
495         if(!subGroup1Infect)
496             testResults(currLevel, false);
497         else
498         {
499             testResults(currLevel, true);
500             performSubTests(subGroup1, currLevel + 1, currGroupSize / 2);
501         }
502 
503         testCount++;
504         int j = 0;
505         while(!subGroup2Infect && j < subGroup2.length)
506         {
507             if(subGroup2[j])
508                 subGroup2Infect = true;
509             else
510                 j++;
511         }
512 
513         if(!subGroup2Infect)
514             testResults(currLevel, false);
515         else
516         {
517             testResults(currLevel, true);
518             performSubTests(subGroup2, currLevel + 1, currGroupSize / 2);
519         }
520 
521         if(subGroup1Infect && subGroup2Infect)
522             caseThree++;
523         else if(subGroup1Infect ^ subGroup2Infect)
524             caseTwo++;
525     }
526 }
```

```
529 public static void performSingleTests(boolean[] subgroup, int currLevel, int currGroupSize
530 {
531     int infectPrevious = infectionCount;
532     int newInfect = 0;
533     ArrayList<Boolean> singleTest = new ArrayList<Boolean>();
534 
535     for(boolean b : subgroup)
536         singleTest.add(b);
537 
538     for(int i = 0; i < singleTest.size(); i++)
539     {
540         if(singleTest.get(i))
541             infectionCount++;
542     }
543 
544     newInfect = infectionCount - infectPrevious;
545     singleTestResults(currLevel, newInfect);
546 
547     testCount += singleTest.size();
548 }
```