CMPT 435 - Fall 2021 - Dr. Labouseur

Semester Project – Pooled Testing Simulation

C. Marcus DiMarco
C.DiMarco1@Marist.edu

December 14, 2021

1 Objectives

- Create a program that runs a pooled testing simulation on a given population (which must be divisible by 1000).
- Identify the 3 cases which may exist in the test results.
- Print the results of each case along with the percentage of occurrence in the population.

2 Conditions

- The program must accept the population size as a command line argument when launching the program.
- The program must not simply count the instances of each case. It must run an actual simulation of a pooled testing environment and increment for each test used and each case encountered.

3 Overview

Our pooled testing simulator is driven by two classes: MarcusTestCase and MarcusTester.

3.1 MarcusTestCase

This class is a simple container of two private fields which track the number of cases and tests for a given MarcusTestCase. In our full program, we will use an array of this class to represent each of the three possible cases. This class also contains a method to calculate the percentage of the pools that it represents.

```
1  /**
2   * A class to withhold counters of case instances and tests used,
3   * as well as methods to test an array of ints for positive cases.
4   */
5
6  public class MarcusTestCase {
7    private int cases;
8    private int tests;
9
10   public MarcusTestCase() {
11     this.cases = 0;
12     this.tests = 0;
13   }
14
15   public void addCase() {
16     this.cases++;
17   }
18
19   public void addTests(int quantity) {
20     this.tests += quantity;
21   }
22
23   public double getPercentage(int numOfGroups) {
24     return (cases * 1.0) / numOfGroups;
25   }
26
27   // Setters and getters for private fields
28   public void setCases(int cases) {
29     this.cases = cases;
30   }
31
32   public int getCases() {
33     return cases;
34   }
35
36   public void setTests(int tests) {
37     this.tests = tests;
38   }
39
40   public int getTests() {
```

```
41      return tests;
42    }
43  }
```

3.2 MarcusTester

MarcusTester has no fields of its own. It was necessary to abstract the methods of MarcusTester into their own class in order to accept an array of MarcusTestCase objects as a parameter. The only public method is test(), which conducts the pooled testing simulation using two private methods and a structure inspired by recursion and Merge Sort. By accepting arrays and indices as parameters, test() is able to call poolIsPositive() on the pools' subarrays, allowing for more user-friendliness. Since there is no testing to be done on groups of size 2, we cannot use a true recursive approach, and we instead call testIndividuals() on any subarrays of size 4 which return true for poolIsPositive().

```
1  /**
2   * An object designed to simulate pooled testing on a group.
3   * Conducts tests in the way we would have to in a real world scenario -
4   * abstracting the individual results and only returning the result of
5   * the pool until deeper testing is conducted.
6   */
7
8  public class MarcusTester {
9
10   // Default constructor
11   public MarcusTester() {
12
13   }
14
15   // Abstraction for user - test() takes the array of cases and sample of
16   // population, tests on group of size 8. If negative, case1 is
17   // incremented and test1 is incremented. If positive, tests on L/R
18   // subarrays. If one of these tests comes back as negative, increment
19   // case 2 and conduct tests on the positive subarray.
20   public void test(MarcusTestCase[] cases, int[] group, int leftIndex,
21           int rightIndex) {
22     int tests = 1;
23     int caseNumber = 0;
24
25     // Using boolean to represent pooled test, since we will never
26     // be able to determine individual positivity until conducting
27     // individual tests in real world scenario
28     if (poolIsPositive(group, leftIndex, rightIndex)) {
29       int midpoint = (leftIndex + rightIndex) / 2;
30       boolean leftIsPositive = poolIsPositive(group, leftIndex, midpoint);
```

```
      boolean rightIsPositive = poolIsPositive(group, midpoint + 1, rightIndex);
      tests += 2;

      if (leftIsPositive ^ rightIsPositive) {
        caseNumber = 1;
      } else {
        caseNumber = 2;
      }

      if (leftIsPositive) {
        testIndividuals(group, leftIndex, midpoint, cases[caseNumber]);
      }

      if (rightIsPositive) {
        testIndividuals(group, midpoint + 1, rightIndex, cases[caseNumber]);
      }

      cases[caseNumber].addCase();
      cases[caseNumber].addTests(tests);
    } else {
      cases[caseNumber].addCase();
      cases[caseNumber].addTests(tests);
    }
  }

  // Returns the result of the pooled test as a boolean without shortcircuiting
  // in order to represent real world conditions
  private boolean poolIsPositive(int[] group, int leftIndex, int rightIndex) {
    boolean result = false;

    for (int i = leftIndex; i <= rightIndex; i++) {
      if (group[i] != 0) {
        result = true;
      }
    }

    return result;
  }

  // Iterates through the array from @leftIndex to @rightIndex, incrementing
  // @instance's test count for each test conducted.
  private void testIndividuals(int[] group, int leftIndex, int rightIndex,
        MarcusTestCase instance) {
    for (int i = leftIndex; i <= rightIndex; i++) {
      instance.addTests(1);
    }
```

```
77    }
78  }
```

On line 34, we are using the bitwise operator ^ to represent logical exclusive-or, which will return true if and only if 1 of the 2 statements are true.

In order to most accurately represent the idea of pooled testing, it was important to not use a return statement on line 63, which would allow for more efficient code. In a real world environment, we are testing the entire pool at once, and using a return statement here would violate the condition of using a single test for the pool since we would be effectively analyzing individual results to determine the positivity of the pool.

3.3 SemesterProject

SemesterProject takes a command line argument of an integer divisible by 1000 and subsequently infects 2% of a population of that size. It then shuffles the array of the population to represent random distribution of infections and proceeds to test pools of size 8, determining which of three cases the pool belongs to, incrementing the appropriate case and adding the tests to it. Once the whole population has been pool-tested, the results are printed along with the occurrence percentages of each case. The three possible cases are:

- Case 1: 0 infections present in pool
- Case 2: Infections present in 1/2 subpools
- Case 3: Infections present in 2/2 subpools

```
1  /**
2   * A program which simulates group/pooled testing on a population of 1000,
3   * 10,000, 100,000 and 1,000,000 people for a disease with a 2% infection
4   * rate and a test accuracy of 100%. Testing is in groups of 8 and should
5   * account for 3 possible cases: 0 infections, 1 infection, or 2+ infections
6   * in the group.
7   */
8
9  public class SemesterProject {
10   // A modified implementation of notRosannaShuffle() to shuffle
11   // an array of integers instead of Strings
12   public static void notRosannaShuffle(int[] integers) {
13
14     for (int i = integers.length - 1; i >= 0; i--) {
15       // Get random index in the remaining length
16       int randomIndex = (int) Math.floor(Math.random() * i);
17       if (randomIndex == i) {
18         continue;
19       } else {
```

```
20          // If the random index is not i, swap their values
21          int tempInt = integers[i];
22          integers[i] = integers[randomIndex];
23          integers[randomIndex] = tempInt;
24        }
25      }
26    }
27
28    public static void main(String[] args) {
29      try {
30        // Initialize primary variables/data structures
31        final int POPULATION = Integer.parseInt(args[0]);
32          // Population size as a constant
33        if (POPULATION % 1000 == 0) {
34          final double INFECTION_RATE = 0.02;
35            // Infection rate as a constant
36          final int POOL_SIZE = 8;
37            // Pool size to be tested as a constant
38          final int NUM_POOLS = POPULATION / POOL_SIZE;
39            // Number of pools as a constant
40          int[] populationArray = new int[POPULATION];
41            // Array of population size
42          MarcusTester tester = new MarcusTester();
43            // Tester object
44
45          // Infect 2% of array
46          for (int i = 0; i < (POPULATION * INFECTION_RATE); i++) {
47            populationArray[i] = 1;
48          }
49
50          // Shuffle array
51          notRosannaShuffle(populationArray);
52
53          // Case 1: Pool of 8 tests negative
54          MarcusTestCase case1 = new MarcusTestCase();
55          // Case 2: Pool of 8 positive, 1/2 subarrays infected
56          MarcusTestCase case2 = new MarcusTestCase();
57          // Case 3: Pool of 8 positive, 2/2 subarrays infected
58          MarcusTestCase case3 = new MarcusTestCase();
59
60          MarcusTestCase[] cases = {
61            case1, case2, case3
62          };
63
64          // Test
65          for (int i = 0; i < populationArray.length; i += POOL_SIZE) {
```

```
66            tester.test(cases, populationArray, i, i + POOL_SIZE - 1);
67          }
68
69          // Having counted instances of each case, print results
70          int caseNumber = 1;
71          int sumTests = 0;
72          String resultString;
73            // Using String concatenation to print results in order to
74            // track max String length for better formatting of
75            // dashed line separator
76          int stringMaxLength = 0;
77          for (int i = 0; i < cases.length; i++) {
78            resultString = "";
79            resultString += "Case (" + caseNumber + "): ";
80            resultString += NUM_POOLS + " x ";
81            resultString += String.format("%.4f",
82                    cases[i].getPercentage(NUM_POOLS));
83            resultString += " = ";
84            resultString += String.format("%3s", cases[i].getCases());
85            resultString += " instances requiring " + cases[i].getTests() +
86                    " tests";
87            System.out.println(resultString);
88            if (resultString.length() > stringMaxLength) {
89              stringMaxLength = resultString.length();
90            }
91            caseNumber++;
92            sumTests += cases[i].getTests();
93          }
94
95          // Print dashed line separator based on max String length above
96          for (int i = 0; i < stringMaxLength; i++) {
97            System.out.print("-");
98          }
99          System.out.println();
100
101         // Print conclusion
102         System.out.println(sumTests + " tests to screen a population of " +
103                 POPULATION +
104                 " people for a disease with an infection rate of " +
105                 ((int) (INFECTION_RATE * 100)) +"%");
106       } else {
107         throw new IllegalArgumentException("Population not divisible " +
108                         "by 1000");
109       }
110     } catch (IndexOutOfBoundsException e) {
111       System.out.println("Oops! Please pass a population divisible by 1000 " +
```

```
                    "as an argument to the program!");
            e.printStackTrace();
        } catch (IllegalArgumentException e) {
            System.out.println("Oops! Please pass a population divisible by 1000 " +
                    "as an argument to the program!");
            e.printStackTrace();
        }
    }
}
```

4 Results

Average Percentage of Cases in Population (5 executions)				
Population Size	Case 1	Case2	Case3	Total Tests
1,000	85.6	13.6	0.8	237
	84.8	14.4	0.8	243
	84.8	14.4	0.8	243
	86.4	12.0	1.6	235
	86.4	12.0	1.6	235
Average	85.60	13.28	1.12	238.6
10,000	84.88	14.56	0.56	2,412
	85.12	14.24	0.64	2,398
	85.28	13.76	0.96	2,402
	85.44	13.52	1.04	2,394
	85.12	14.48	0.40	2,386
Average	85.168	14.112	0.72	2,398.4
100,000	85.040	14.392	0.568	24,004
	85.040	14.392	0.568	24,004
	85.016	14.344	0.640	24,058
	85.160	14.208	0.632	23,946
	85.096	14.296	0.608	23,982
Average	85.0704	14.3264	0.6032	23,998.8
1,000,000	85.0808	14.3024	0.6168	239,978
	85.0568	14.3672	0.5760	239,954
	85.0592	14.3712	0.5696	239,904
	85.0568	14.3520	0.5912	240,030
	85.0856	14.2792	0.6352	240,034
Rounded Average	85.0678	14.3344	0.5978	239,980

5 Analysis

Binomial distribution suggests that since our infection rate is 2%, 98% of our population is uninfected and therefore, the probability of choosing a sample of size 8 with no infections will be 0.98^8, which is 0.85 or 85%. It would follow that case 2 would have a projected probability of 14.96% and that case 3 would have a probability of 0.04%. Our results are shuffled at random, so exact replication of these percentages is not expected, but as we increase the number of tests, our averages should trend towards these percentages.

However, binomial distribution is based on replacement in the population set. Hypergeometric distribution, which is based on removal of assessed elements in the population set, paints a more accurate picture of what to expect. While the same percentages will initially be true, hypergeometric distribution provides that as we remove elements of a given type (infected or uninfected, in this case), the probability of encountering the opposite type of element rises. This occurs since we are effectively reducing the sample size without reducing the number of possible elements in the complementary set.

What this also means is that while the probabilities calculated using binomial distribution will not changed based on the population size, the probabilities calculated using hypergeometric distribution will vary slightly as the population size changes. In a population of 1000, our odds of encountering an infection at index 0 are 20/1000. If we do encounter this infection, the odds of the next element being infected are 19/999, or 0.01901902%. This same scenario in a population of 1,000,000 uses the respective probabilities of 20,000/1,000,000 and 19,999/999,999, the latter being roughly equal to 0.01999902%. These are notably small differences, resulting in a change of a thousandth of 1 percent when increasing the scale by a thousandfold. For this reason, the results will still appear to be in line with the projections made by the binomial distribution, but as the number of tests increases, the average will trend toward each population size's respective hypergeometric distribution.

The primary improvement I would make to this project is to introduce a test accuracy of less than 100% to better reflect real world conditions. My preliminary thought is to set a variable for testAccuracy equal to a percentage - let's say 0.98. Then for each test, a decimal from 0-1 would be randomly chosen, and if the random number is equal to or greater than testAccuracy, the test inverts its result (displaying FALSE when the pool should have returned TRUE or vice versa). The redundancy would be to do a second round of testing and to compare the two rounds. If both rounds provide the same result, then the tests are accepted as accurate. If the two rounds provide different results, a third test is conducted and whichever result matches two of three tests is selected as the correct result.