HOMEWORK 2

M/M/2/2+5 Simulator

CS 555 - Fall 2017

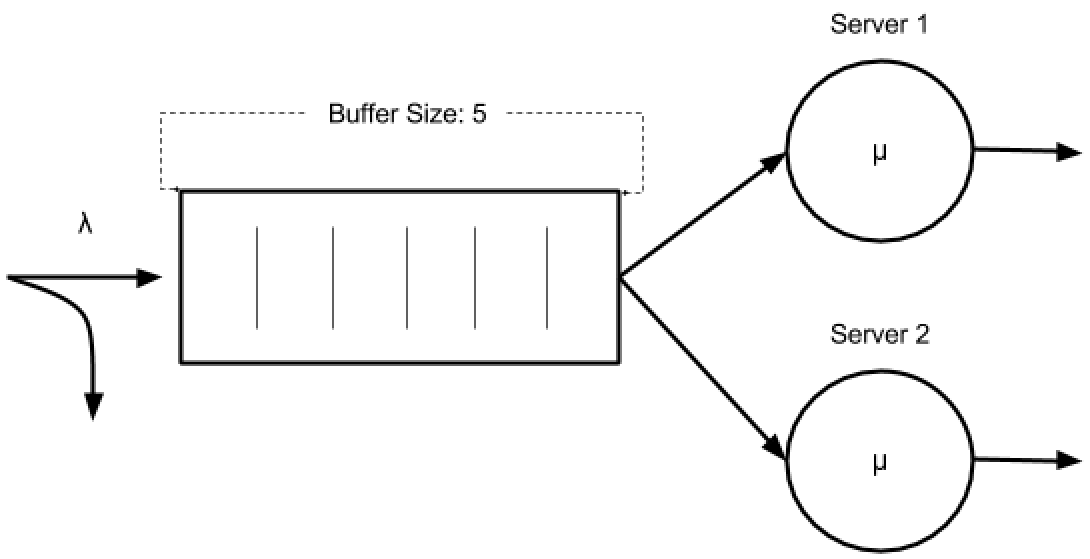

Diagram of an M/M/2/2+5 queue

Team Members:

Ver Hoef, Christopher John
A20364831
Section 02

Menk, Sufyan
A20306092
Section 02

Professor: Edward Chlebus

Section 1 - Problems

Problem #1

Methodology

To solve this problem we created a separate Java program that tested out RNG separate from any other queuing code. We are using the latest version at the time, which is Java 8. We use a combinations of arrays and arraylists to store our sequences of random numbers. We use Java 8's implementation of a pseudo random number generator located in the java.util.Random package. The source code of our RNG tester is shown in section 2.2. More information concerning individual questions are answered below.

1.1.1 Does your RNG generate random numbers?

Yes, we use Java's built it random number generator located in java.util.random package. According to the documentation the Random class generates a stream of pseudorandom numbers using a 48-bit seed, which is modified using a linear congruential formula[1].

To test this statement in the documentation we generated 1 million random numbers and counted how many times each number occurs. For example if the RNG generate the number 567.6785 we round down to 567 and increment the 567 counter by 1. Ideally every number between 0 to 1000 should occur 1000 times. This means that the probability of getting any number is equal, which results in randomness.

We then calculate how far away each counter is from 1000, below are the results:

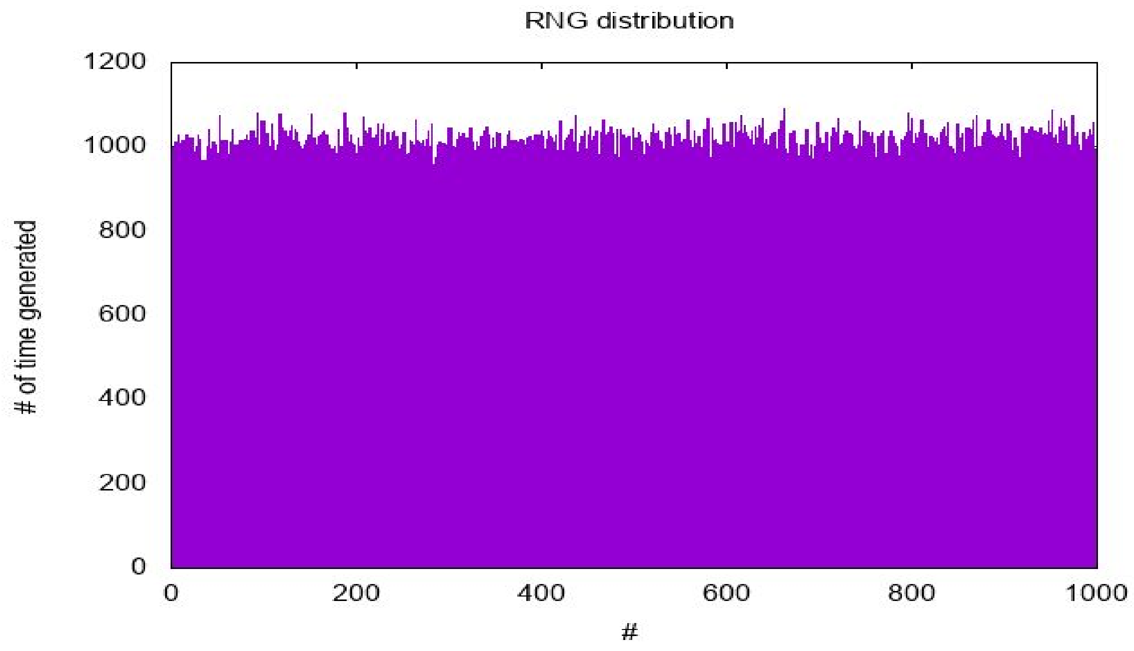

Figure 1: RNG distribution

The source code to test the RNG is located in section 2.1 line numbers 22-75.

1.1.2 How do you initialize the seed of your RNG?

To generate the seed for the random number generator (RNG) we use the current system time in milliseconds. It takes time for each replication to run and therefore every run will have a different seed for their RNG's. Below is the one line code to initialize the RNG in the system.

```
 private Random r = new Random(System.currentTimeMillis());
```

1.1.3 Generate two sequences of 1000000 numbers each, for every sequence use a different seed. Are these two sequences different? How do you know this?

To generate two sequences of random numbers we first initialize our RNG with the current system time in milliseconds, generate 1,000,000 numbers, then sleep for a certain period to guarantee the seed for the second sequence is not equal to the first, and generate another 1,000,000 numbers. All numbers are between 0 and 1 and are stored in 2 fixed size arrays.

After generating two sequences of 1,000,000 numbers each with different seeds none of the values in sequence 1 were found in sequence 2. We come to this conclusion by sorting both sequences into ascending order and comparing each number and if they were equal we incremented a counter if not we check if the value in sequence 1 was smaller than sequence 2 and moved the head of sequence 1 to the next element. If the value at the head of sequence 2 was smaller than the value and sequence 1 then we incremented sequence 2. This process is repeated until one sequence is completed.

The source code for this is located in section 2.2 line number 128.

Problem #2

Methodology

To solve this problem we created a separate Java program that takes lambda, mu, initial amount of customers in the system, the warmup period to test, and the number replications, as input. Using these inputs we create *n* replications of our M/M/2/2+5 class and store the event list of each replication in a results array. Then we sort all the events in order of the time they happened. For each event, either increase or decrease the total number of items across replications depending on arrival or departure, then write average number of items to file. The file represents the time and the average number of customers at that time. We use GNUPlot to plot the graph. Then we analyze the graph to see at what time the average number of customers becomes constant. This time is the warmup period. If the test warm up time is too small we increase it until the graph is what we want. The source code of our Warmup Calculator is shown in section 2.3. More information concerning individual questions are answered below.

1.2.1 System A with the initial condition x(t=0)=0, i.e. the system is empty at t=0.

To use our Warmup Calculator to generate the results for this problem run the following command:

```
java WarmupCalculator 2 1 0 350 10000
```

The above command is calling the queue with parameters $\lambda = 2$, $\mu = 1$, $init = 0$, $t_{warm\ up\ test} = 350$, $\#\ replications = 10,000$.

To plot the resulting graph run the following command using GNUPlot:

```
plot 'data_file.txt' with linespoints ls 1 ps 0
```

Below is the plot. Your results may vary.

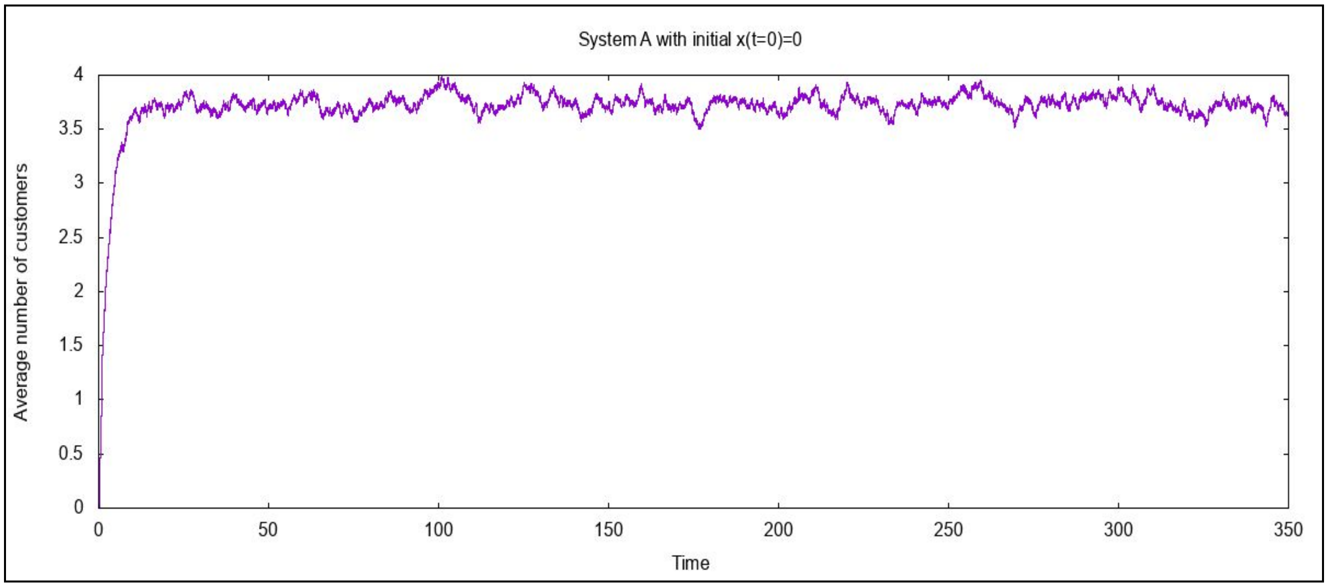

Figure 2: System A with initial x(t=0)=0

Analyzing the graph we can see that at around *t=150* the average number of customers levels out enough between 3.5 and 4. So, we conclude that the warmup time is 150 for this configuration of System A.

1.2.2 System A with the initial condition x(t=0)=7, i.e. the system is full at t=0.

To use our Warmup Calculator to generate the results for this problem run the following command:

```
java WarmupCalculator 2 1 7 350 10000
```

The above command is calling the simulation with parameters $\lambda = 2$, $\mu = 1$, $init = 7$, $t_{warm\ up\ test} = 500$, $\#\ replications = 10,000$.

To plot the resulting graph run the following command using GNUPlot:

```
plot 'data_file.txt' with linespoints ls 1 ps 0
```

Below is the plot. Your results may vary.

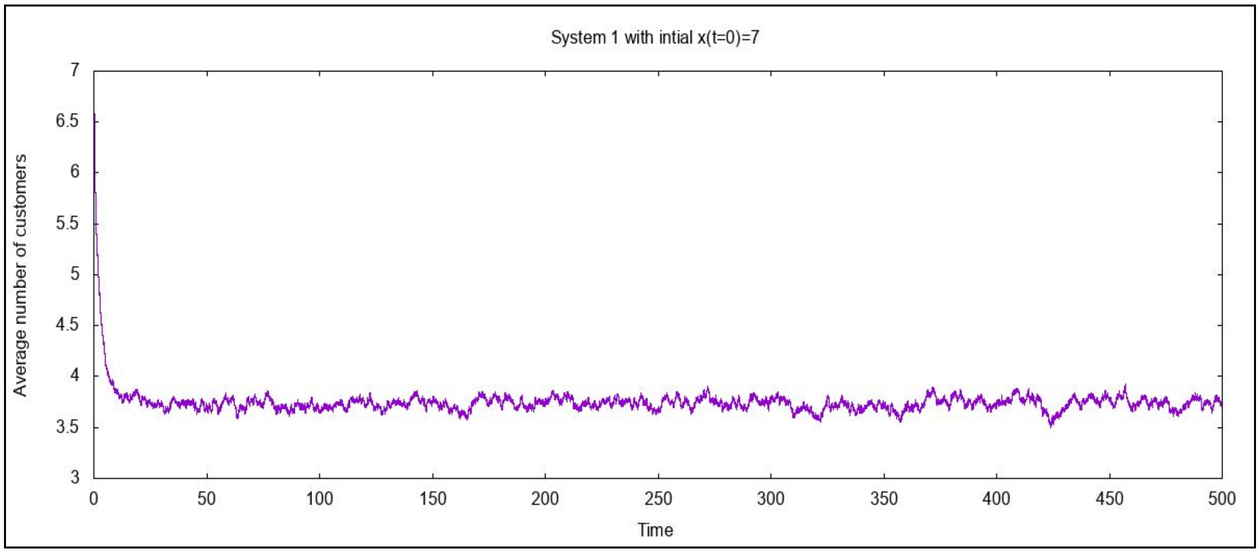

Figure 3: System A with initial x(t=0)=7

Analyzing the graph we can see that at around *t=100* the average number of customers levels out enough between 3.5 and 4. So, we conclude that the warmup time is 100 for this configuration of System A.

1.2.3 System B with the initial condition x(t=0)=0, i.e. the system is empty at t=0.

To use our Warmup Calculator to generate the results for this problem run the following command:

```
java WarmupCalculator 10 1 0 100 10000
```

The above command is calling the simulation with parameters $\lambda = 10$, $\mu = 1$, $init = 0$, $t_{warm\ up\ test} = 100$, $\#\ replications = 10,000$.

To plot the resulting graph run the following command using GNUPlot:

```
plot 'data_file.txt' with linespoints ls 1 ps 0
```

Below is the plot. Your results may vary.

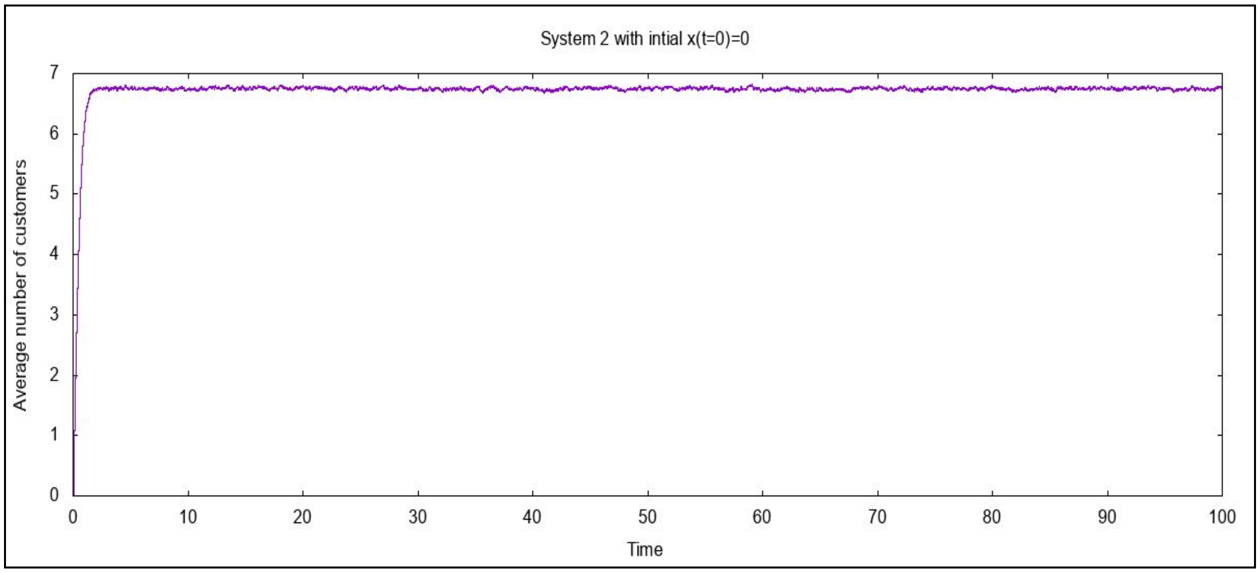

Figure 4: System B with initial x(t=0)=0

System B is an overloaded system and as a result the system fills up very quickly. Analyzing the graph we can see that at around *t=10* the average number of customers levels out enough between 6.5 and 7. So, we conclude that the warmup time is 10 for this configuration of System B.

1.2.4 System B with the initial condition x(t=0)=4, i.e. there are 4 customers in the system at t=0.

To use our Warmup Calculator to generate the results for this problem run the following command:

```
java WarmupCalculator 10 1 4 50 10000
```

The above command is calling the simulation with parameters $\lambda = 10$, $\mu = 1$, $init = 4$, $t_{warm\ up\ test} = 50$, $\#\ replications = 10,000$.

To plot the resulting graph run the following command using GNUPlot:

```
plot 'data_file.txt' with linespoints ls 1 ps 0
```

Below is the plot. Your results may vary.

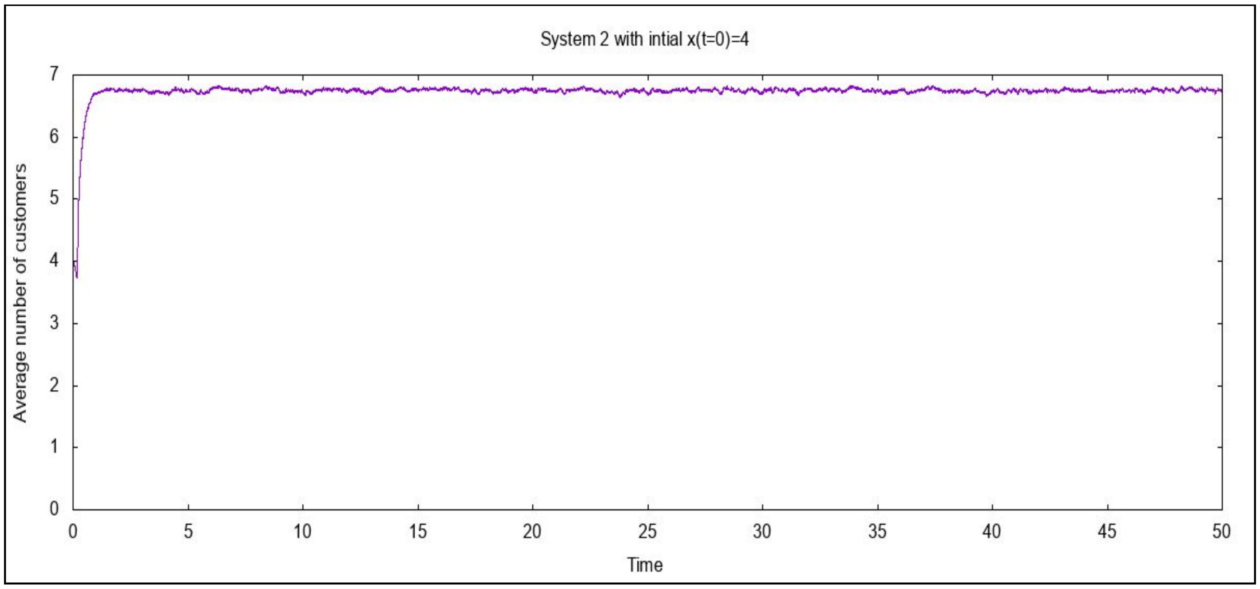

Figure 4: System B with initial x(t=0)=4

Analyzing the graph we can see that at around *t=10* the average number of customers levels out enough between 6.5 and 7. So, we conclude that the warmup time is 10 for this configuration of System B.

Problem #3

Methodology

Using the warmup times we discovered in the previous problem, to find the statistic below we use the method of batches. To combat the problem of choosing a large enough batch size we are listing the statistic using $\Delta t = 100$, $\Delta t = 1,000$, $\Delta t = 10,000$. We create 100 batches and calculate the average and standard deviation of each statistic mentioned in the problems below. All this is done using the *ProbFinder* program. *ProbFinder* take λ, μ, $initial\ customers$, $warmup\ time$, and Δt. The program returns the average, standard deviation, and 90% confidence interval the blocking probability, number of customers in the system, and the time spent in the system. The results may vary because we are simulating a stochastic system.

1.3.1 Blocking probability.

To calculate the blocking probability, we increment a statistical counter if a customer is rejected and the end of simulation we divide number of customers block with the total customers handled (customer received + customers blocked). This can be seen in our M/M/2/2+5 class on lines 130, and 162..

System Configuration	Java Command	Δt	Average	Standard Deviation	90% Confidence Interval

System A
$\lambda = 2,\ \mu = 1,$
$x(t = 0) = 0$
$t_{warmup} = 150$
	`java ProbFinder 2 1 0 150 100`
	`java ProbFinder 2 1 0 150 1000`
	`java ProbFinder 2 1 0 150 10000`
		100	0.1282555851776537	0.04250052968470239	**lower bound:** 0.12126424804452016 **upper bound:** 0.13524692231078725

		1,000	0.13119715764631806	0.04314495312618626	**lower bound:** 0.12409981285706043 **upper bound:** 0.1382945024355757
		10,000	0.1345835615331439	0.040293639184042836	**lower bound:** 0.12795525788736886 **upper bound:** 0.14121186517891895

System A
$\lambda = 2,\ \mu = 1,$
$x(t = 0) = 7$
$t_{warmup} = 100$
	`java ProbFinder 2 1 7 100 100`
	`java ProbFinder 2 1 7 100 1000`
	`java ProbFinder 2 1 7 100 10000`
		100	0.12813942819658217	0.052564276501865036	**lower bound:** 0.11949260471202537 **upper bound:** 0.13678625168113898

		1,000	0.12857818441326757	0.0525536040248066	**lower bound:** 0.11993311655118688 **upper bound:** 0.13722325227534826
		10,000	0.12883119874407462	0.05800068191365293	**lower bound:** 0.11929008656927871 **upper bound:** 0.13837231091887053

System B
$\lambda = 10,\ \mu = 1,$
$x(t = 0) = 0$
$t_{warmup} = 10$
	`java ProbFinder 10 1 0 10 100`
	`java ProbFinder 10 1 0 10 1000`
	`java ProbFinder 10 1 0 10 10000`
		100	0.7948359558388332	0.04771061129771444	**lower bound:** 0.7869875602803591 **upper bound:** 0.8026843513973072

		1,000	0.7993304747553563	0.05041536229622562	**lower bound:** 0.7910371476576272 **upper bound:** 0.8076238018530855
		10,000	0.7969935784003028	0.04726123788735533	**lower bound:** 0.7892191047678329 **upper bound:** 0.8047680520327728

System B
 	
$\lambda = 10$, $\mu = 1$,
$x(t=0) = 4$
 	
$t_{warmup} = 10$
	`java ProbFinder 10 1 4 10 100`
	
	`java ProbFinder 10 1 4 10 1000`
	
	`java ProbFinder 10 1 4 10 10000`
		100	0.796933636114769	0.04960365115707287	**lower bound:** 0.7887738354994305 **upper bound:** 0.8050934367301075

		1,000	0.801634752085013	0.051837978724898155	**lower bound:** 0.7931074045847673 **upper bound:** 0.8101620995852588
		10,000	0.795782938018011 5	0.048527018184983624	**lower bound:** 0.7878002435265816 **upper bound:** 0.8037656325094413

Table 1: Blocking Probability

1.3.2 Mean time a customer spends in the system.

To calculate the mean time a customer spends in the system whenever a customer departs the system we calculate the total wait time of each customer and sum them all up. Customers who enter the system before the time period being observed begins or depart after it ends are not included in this sum. To calculate the mean we divide the total wait time of all customers with total customer received by the system at that time. This excludes customers who arrived within the time period, but have not departed. This can be seen in our M/M/2/2+5 class on lines 163, 95, .

System Configuration	Java Command	Δt	Average	Standard Deviation	90% Confidence Interval

System A
 	
$\lambda = 2$, $\mu = 1$,
$x(t=0) = 0$
 	
$t_{warmup} = 150$
	`java ProbFinder 2 1 0 150 100`
	
	`java ProbFinder 2 1 0 150 1000`
	
	`java ProbFinder 2 1 0 150 10000`
		100	2.151904239596478	0.25409396899738773	**lower bound:** 2.1101057816964075 **upper bound:** 2.193702697496548

		1,000	2.1354940708716392	0.2937939317069326	**lower bound:** 2.087164969105849 **upper bound:** 2.1838231726374295
		10,000	2.1594306442369047	0.2643891729119356	**lower bound:** 2.115938625292891 **upper bound:** 2.2029226631809182
System A	`java ProbFinder 2 1 7 100 100`	100	2.15116855710283	0.363583086782496	**lower bound:** 2.091359139327113

$\lambda = 2$, $\mu = 1$, $x(t=0) = 7$ $t_{warmup} = 100$	`java ProbFinder 2 1 7 100 1000` `java ProbFinder 2 1 7 100 10000`		4		**upper bound:** 2.2109779748785545
		1,000	2.1290275808393426	0.32572842990917433	**lower bound:** 2.0754452541192836 **upper bound:** 2.1826099075594017
		10,000	2.131399700867364	0.3570506522269013	**lower bound:** 2.0726648685760387 **upper bound:** 2.1901345331586897
System B $\lambda = 10$, $\mu = 1$, $x(t=0) = 0$ $t_{warmup} = 10$	`java ProbFinder 10 1 0 10 100` `java ProbFinder 10 1 0 10 1000` `java ProbFinder 10 1 0 10 10000`	100	3.281589371742584	0.8275147019758037	**lower bound:** 3.1454632032675645 **upper bound:** 3.417715540217604
		1,000	3.41305023511197 36	0.9149144411478831	**lower bound:** 3.2625468095431467 **upper bound:** 3.5635536606808005
		10,000	3.386112770937579	0.8857306504186984	**lower bound:** 3.240410078943703 **upper bound:** 3.531815462931455
System B $\lambda = 10$, $\mu = 1$, $x(t=0) = 4$ $t_{warmup} = 10$	`java ProbFinder 10 1 4 10 100` `java ProbFinder 10 1 4 10 1000` `java ProbFinder 10 1 4 10 10000`	100	3.382209187586442	0.8892348882609208	**lower bound:** 3.2359300484675204 **upper bound:** 3.5284883267053635
		1,000	3.4222805127578164	0.94485023155157	**lower bound:** 3.266852649667583 **upper bound:** 3.57770837584805
		10,000	3.4787993339377414	1.0246163649378937	**lower bound:** 3.310249941905458 **upper bound:** 3.647348725970025

Table 2: Mean Time a customer spends in the system

1.3.3 Mean number of customers in the system.

To calculate the mean number of customers in the system we keep track of when events occur in the system. When an event occurs to change the number of customers in the system, we find the area under the curve from the previous event to the current event and add that area to the total area under the curve so far. To calculate the mean, once the observation has been completed, we divide the total area by the total observation time. This can be seen in our M/M/2/2+5 class on lines 161, 109, and 143.

System Configuration	Java Command	Δt	Average	Standard Deviation	Confidence Interval

System A
 	
$\lambda = 2$, $\mu = 1$, $x(t = 0) = 0$
 	
$t_{warmup} = 150$
	`java ProbFinder 2 1 0 150 100`
	
	`java ProbFinder 2 1 0 150 1000`
	
	`java ProbFinder 2 1 0 150 10000`
		100	3.7380849670275857	0.3989550289334571	**upper bound:** 3.8037130692871393 **lower bound:** 3.6724568647680322

		1,000	3.684166484023728	0.4648685366221381	**lower bound:** 3.607695609749386 **upper bound:** 3.76063735829807
		10,000	3.669867629177346	0.4056188012940064	lower bound: 3.67844124770716 upper bound: 3.818624152499943

System A
 	
$\lambda = 2$, $\mu = 1$, $x(t = 0) = 7$
 	
$t_{warmup} = 100$
	`java ProbFinder 2 1 7 100 100`
	
	`java ProbFinder 2 1 7 100 1000`
	
	`java ProbFinder 2 1 7 100 10000`
		100	3.761096678634176	0.5449956712749241	**lower bound:** 3.6714448907094512 **upper bound:** 3.850748466558901

		1,000	3.6657435104114438	0.49691923443103414	**lower bound:** 3.5840002963475386 **upper bound:** 3.747486724475349
		10,000	3.7099739240971332	0.5617113249822472	**lower bound:** 3.6175724111375533 **upper bound:** 3.802375437056713

System B
 	
$\lambda = 10$, $\mu = 1$, $x(t = 0) = 0$
 	
$t_{warmup} = 10$
	`java ProbFinder 10 1 0 10 100`
	
	`java ProbFinder 10 1 0 10 1000`
	
	`java ProbFinder 10 1 0 10 10000`
		100	6.4390788634952365	0.36773862713047123	**lower bound:** 6.378585859332274 **upper bound:** 6.499571867658199

		1,000	6.452970918390862	0.32212979189958174	**lower bound:** 6.3999805676233805 **upper bound:** 6.505961269158343
		10,000	6.463112	0.34538574	**lower bound:**

			870536229	72469246	6.406296915114109 **upper bound:** 6.519928825958348

System B
 	
$\lambda = 10$, $\mu = 1$, $x(t = 0) = 4$
 	
$t_{warmup} = 10$
	`java ProbFinder 10 1 4 10 100`
	
	`java ProbFinder 10 1 4 10 1000`
	
	`java ProbFinder 10 1 4 10 10000`
		100	6.531509026480769	0.2755080873223051	**lower bound:** 6.48618794611625 **upper bound:** 6.576830106845288

		1,000	6.4930135119736905	0.3352704266683207	**lower bound:** 6.437861526786752 **upper bound:** 6.548165497160629
		10,000	6.465718539838695	0.40667402281633563	**lower bound:** 6.3988206630854085 **upper bound:** 6.532616416591982

Table 3: Mean number of customers in the system

Comparing the averages the differences between $\Delta t's$ are negligible, therefore $\Delta t = 100$ is a large enough batch size to get an accurate result for every statistic.

Section 2 - Source Code (Java 8)

2.1) M/M/2/2+5 Queue Class

Source file name: MM2Queue.java

```java
import java.util.LinkedList;
import java.util.ArrayList;
import java.util.Random;

public class MM2Queue {

    //Initialization variables
    private double lambda;
    private double mu;
    private double blockProb;   //blocking probability
    private double meanCusTime; //average amount of time a customer spends in the system
    private double meanCusNum;  //average number of customers in the system
    private LinkedList<Double> queue;
    private Random r;

    private double nextArrival;
    private double nextDeparture1;
    private double nextDeparture2;

    public MM2Queue(double lambda, double mu, int startCustNum) {
        //Initialize
        this.lambda = lambda;
        this.mu = mu;
        blockProb = 0;
        meanCusTime = 0;
        meanCusNum = 0;
        queue = new LinkedList<Double>();
        //Add items into the queue
        for(int i = 0; i < startCustNum; i++) {
            queue.add(-0.1);
        }
        //Initialize random variable
        r = new Random(System.currentTimeMillis());

        //Set the next arrival time
        nextArrival = exp(lambda);

        //If there is someone in the queue, set the next departure time for server 1;
        //otherwise, mark server as empty
        if (queue.size() >= 1)
            nextDeparture1 = exp(mu);
        else
            nextDeparture1 = Double.POSITIVE_INFINITY;

        //If there is someone in the queue not being worked on by server 1,
        //set the next departure time for server 2;
        //otherwise, mark server as empty
        if (queue.size() >= 2)
            nextDeparture2 = exp(mu);
        else
        nextDeparture2 = Double.POSITIVE_INFINITY;
}

//Run the queue for timeLen, set the state variables for that interval,
//and return a list of the times something entered or left the queue
//and which it was
public ArrayList<double[]> runFor(double timeLen) {
    //List of times and state changes
    //A {(time), +1} array has time of arrival into buffer; a {(time), -1} array has
    //time of departure from server
    ArrayList<double[]> toRet = new ArrayList<double[]>();

    //Total number of customers who entered the queue and who were blocked
    double custReceived = 0.0;
    double custBlocked = 0.0;

    //The total wait time for customers
    double totalWait = 0.0;

    //The previous time someone entered/left the queue and the mean customer numbers
    double prevEvent = 0.0;
    meanCusNum = 0.0;

    //Simulate an M/M/2/2+5 queue as long as at least one event will still happen in the time interval
    while (!(nextArrival > timeLen && nextDeparture1 > timeLen && nextDeparture2 > timeLen)) {

        //If the next arrival time happens after a departure time,
        //it's a departure from...
        if (nextDeparture1 < nextArrival || nextDeparture2 < nextArrival) {
            //...the first server
            if (nextDeparture1 < nextDeparture2) {
                //Add in the area under the curve from the previous event
                //until now; mark now as the next event to look back to
                meanCusNum += queue.size() * (nextDeparture1 - prevEvent);
                prevEvent = nextDeparture1;
                //Into the "what happened?" list, add in departure time
                //and mark that event was departure
                toRet.add(new double[]{nextDeparture1, -1});

                //If the item arrived within the interval, add its
                //wait time to the total wait time
                double arriveTime = queue.remove();
                if(arriveTime >= 0)
                    totalWait += (nextDeparture1 - arriveTime);

                //If queue size is 0, buffer empty, no one being serviced; mark server empty
                //If queue size is 1, buffer empty, someone being serviced in server 2
                //Otherwise, buffer has at least one item; begin servicing it in this server
                if (queue.size() <= 1)
                    nextDeparture1 = Double.POSITIVE_INFINITY;
                else
                    nextDeparture1 += exp(mu);

            }
            //... the second server; logic identical to first server, except
            //changing nextDeparture1 to nextDeparture2
        else {
            meanCusNum += queue.size() * (nextDeparture2 - prevEvent);
            prevEvent = nextDeparture2;

            toRet.add(new double[]{nextDeparture2, -1});

            double arriveTime = queue.remove();
            if(arriveTime >= 0)
                totalWait += (nextDeparture2 - arriveTime);

            if (queue.size() <= 1)
                nextDeparture2 = Double.POSITIVE_INFINITY;
            else
                nextDeparture2 += exp(mu);
        }
    }
    //If next arrival time happens before either departure time, next event is arrival
    else {
        //If queue size is 7, buffer is full and both servers are busy;
        //mark blocked customer and call next arrival time
        if (queue.size() >= 7) {
            custBlocked += 1;
            nextArrival += exp(lambda);
        }
        //Otherwise, stick item into queue
        else {
            //Is a server empty? Use it to begin servicing product immediately
            if (nextDeparture1 == Double.POSITIVE_INFINITY)
                nextDeparture1 = nextArrival + exp(mu);
            else if (nextDeparture2 == Double.POSITIVE_INFINITY)
                nextDeparture2 = nextArrival + exp(mu);
            //Add in the area under the curve from the previous event
            //until now; mark now as the next event to look back to
            meanCusNum += queue.size() * (nextArrival - prevEvent);
            prevEvent = nextArrival;
            //Add arrival time into queue; in "what happened?" list, add in departure time
            //and mark that event was departure
            queue.add(nextArrival);
            toRet.add(new double[]{nextArrival, 1});
            //Calculate next arrival, increment customers received
            nextArrival += exp(lambda);
            custReceived += 1;
        }
    }
}
//Calculate average customer number in queue, blocking probability, and average amount of time
//customer spent in system; we subtract the size of the queue from total customers received
//because those customers are still in the system, so we don't count them when calculating
//customer wait time
meanCusNum = meanCusNum / timeLen;
blockProb = custBlocked / (custBlocked + custReceived);
meanCusTime = totalWait / (custReceived - queue.size());
//Get arrival/departure times ready for next interval
nextArrival = nextArrival - timeLen;
nextDeparture1 = nextDeparture1 - timeLen;
nextDeparture2 = nextDeparture2 - timeLen;
```

```
        //Mark arrival times as happening before next interval
        for (int i = 0; i < queue.size(); i++) {
            queue.set(i, queue.get(i) - timeLen);
        }
        //Return event list
        return toRet;
    }
    //Calculate new exponential variable
    private double exp(double lambdaOrMu) {
        return -Math.log(r.nextDouble()) / lambdaOrMu;
    }
    //Get state variables from last run
    public double getBlockProb() {
        return blockProb;
    }
    public double getMeanCusTime() {
        return meanCusTime;
    }
    public double getMeanCusNum() {
        return meanCusNum;
    }
}
```

2.2) Random Number Generator Tester

Source file name: RandomGeneratorTester.java
Command line Arguments: none
Output: Outputs to terminal with results mentioned in section 1.1.*
Example Call: java RandomGeneratorTester

```java
import java.util.Random;
import java.util.ArrayList;
import java.util.Arrays;
public class RandomGeneratorTester {
    public static void main(String[] args) {
        System.out.println("Performing distance test...");
        distTest();
        System.out.println();
        System.out.println("Performing comparison test...");
        sameNumTest();
    }

    public static void distTest() {
        //Make array of 1000 integers; these will be the counts of doubles
        //generated in the relevant ranges; each value automatically initialized
        //to 0
        int[] arr = new int[1000];
        //Make new random, seed it
        Random r = new Random(System.currentTimeMillis());
        //The counter works like this: a random double between 0 and 1 is generated and multiplied by 1000
        //It is then cast to an int, rounding down in the process; so, say, 954.9356 will be rounded to 954
        //The index of the cast number in the array has one added to it
        //
        //In this way, the amount of numbers generated in a range is counted; when a number between 0 and 0.001
        //is generated, multiplied (for a range between 0 and 1), and cast, it will be rounded down to 0
        //A similar method goes for numbers between 0.001 and 0.002 (between 1 and 2 after multiplying, rounded
        //to 1), between 0.002 and 0.003 (between 2 and 3 after multiplying, rounded to 2), all the way up to the
        //0.999 to 1 range
        //Do this for one million numbers
        for(int i = 0; i < 1000000; i++) {
            arr[(int)(1000*r.nextDouble())]++;
        }
        //Initialize variables to count the number of ranges that have the amount of
        //numbers generated outside 10, 25, 50, and 100, respectively
        int ov10 = 0;
        int ov25 = 0;
        int ov50 = 0;
        int ov100 = 0;
        //For each count in the array, calculate the difference between 1000 (the ideal count) and
        //the actual count; if this difference lies outside a certain threshold, increment the
        //relevant variable
        for(int i : arr) {
            int dif = Math.abs(1000 - i);
            if(dif > 10)
                ov10++;
            if(dif > 25)
                ov25++;
            if(dif > 50)
            ov50++;
        if(dif > 100)
            ov100++;
    }
    //Print results
    System.out.println("We generated one million random doubles between 0 and 1, with the "+
                       "range subdivided into one thousand equal-sized intervals. Ideally, "+
                       "the count of the numbers in each range should be one thousand.");
    System.out.println("Count difference more than 10: "+ov10);
    System.out.println("Count difference more than 25: "+ov25);
    System.out.println("Count difference more than 50: "+ov50);
    System.out.println("Count difference more than 100: "+ov100);
}

public static void sameNumTest() {
    //Create array of doubles, initialize random generator
    double[] arr1 = new double[1000000];
    Random r = new Random(System.currentTimeMillis());
    //Fill array w/ one million random numbers
    for(int i = 0; i < 1000000; i++) {
        arr1[i] = r.nextDouble();
    }
    //Sort the array
    Arrays.sort(arr1);
    //Sleep for a certain period of time, just to make the times a bit more separate
    try {
        Thread.sleep(394);
    } catch(InterruptedException ex) {
        Thread.currentThread().interrupt();
    }
    //Create another array of doubles, initialize new random generator w/ different seed
    double[] arr2 = new double[1000000];
    Random r2 = new Random(System.currentTimeMillis());
    //Fill array w/ one million random numbers
    for(int i = 0; i < 1000000; i++) {
        arr2[i] = r2.nextDouble();
    }
    //Sort array
    Arrays.sort(arr2);
    //Create variables to hold the index number for arrays 1 and 2,
    //as well as a variable to hold number of shared numbers
    int ind1 = 0;
    int ind2 = 0;
    int share = 0;
    //With the arrays sorted, we can efficiently compare numbers
    //We start at the lowest number in each array and compare; if
    //array 1 is smaller, go to the next largest number in that, but
    //if array 2 is smaller, go to the next largest number in that
    //If the numbers in the arrays are the same, make a note of that
    //If the sequences are not the same, they will share as few numbers
    //as possible
    //As long as there are still numbers to compare...
    while(ind1 < 1000000 && ind2 < 1000000) {
        //Increment array 1 if it's lower
        if(arr1[ind1] < arr2[ind2]) {
            ind1++;
        }
        //Increment array 2 if it's lower
        else if(arr1[ind1] > arr2[ind2]) {
            ind2++;
        }
        //If neither is lower, they share number
        //Increment array 1 and shared number variable
        else {
            ind1++;
            share++;
        }
    }
    //Print results
    System.out.println("We generated two sets of random doubles of one million numbers each, "+
                       "with two different seeds for the generator, and compared them. If the "+
                       "sets of numbers are different, they should share as few numbers as possible.");
    System.out.println("Number of items shared: "+share);
}}
```

2.3) Warmup-time Calculator

Source file name: WarmupCalculator.java
Command line Arguments: λ μ initial_Customer test_warmup_period num_replications
Output: 'data-file.txt' which stores (x,y) pairs that correspond with (time, average number of customers in the system). These results are plotted in graphs show in section 2.2.*.
Example Call: java WarmupCalculator 2 1 0 100 10000

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.io.*;

public class WarmupCalculator {
    public static void main(String args[]) throws IOException {
        //Initialization variables; in order: lambda, mu, the amount of items in the queue to begin
        //with, the warmup period to test, the number of replications
        double lambda = Double.parseDouble(args[0]);
        double mu = Double.parseDouble(args[1]);
        double initSize;
        double warmupTime;
        double n;
        //These try/catch blocks allow you to only specify lambda and mu,
        //taking default values otherwise
        try {
            initSize = Double.parseDouble(args[2]);
        } catch (IndexOutOfBoundsException e) {
            initSize = 0.0;
        }
        try {
            warmupTime = Double.parseDouble(args[3]);
        } catch (IndexOutOfBoundsException e) {
            warmupTime = 100;
        }
        try {
            n = Double.parseDouble(args[4]);
        } catch (IndexOutOfBoundsException e) {
            n = 100;
        }
        if(initSize > 8){
            System.out.println("Invalid init customer value. must be < 8");
            System.exit(0);
        }
        //Initialize list to store events
        ArrayList<double[]> results = new ArrayList<double[]>();
        //Run warmup period n times, gather events
        for(int i = 0; i < n; i++) {
            MM2Queue sysA = new MM2Queue(lambda, mu, (int) initSize);
            results.addAll(sysA.runFor(warmupTime));
        }
        System.out.println("Results calculated...");
        //Sort events in order of time they happened
        Collections.sort(results, new Comparator<double[]>() {
            public int compare(double[] d1, double[] d2) {
```

```java
                return Double.compare(d1[0], d2[0]);
            }
        });
        System.out.println("Results sorted...");
        //Total number of items across all replications, plus a variable to keep simultaneous
        //events across multiple replications from all being written
        double totalSize = n*initSize;
        double prevEvent = 0;
        //Initialize file, write first line
        PrintWriter writer = new PrintWriter("data_file.txt", "UTF-8");
        writer.println("0.0 "+String.format("%.4f", totalSize/n));

        //For each event, either increase or decrease the total number of items
        //across replications depending on arrival or departure, then write
        //average number of items to file; the if statement keeps events that may
        //happen to occur at the same time in multiple replications from all being
        //written to the file and only writes the net result, keeping the file
        //size down
        for(double[] arr : results) {
            if(prevEvent != arr[0])
                writer.println(prevEvent+" "+String.format("%.4f", totalSize/n));
            totalSize += arr[1];
            prevEvent = arr[0];
        }
        writer.println(prevEvent+" "+String.format("%.4f", totalSize/n));
        //Close file to prevent memory leaks
        writer.close();
        System.out.println("File written!");
    }

}
```

2.4) Probability Finder

Source file name: ProbFinder.java
Command line Arguments: λ μ initial_Customer test_warmup_period num_replications
Output: Outputs to the terminal the average, standard deviation, upper and lower bound 90% confidence intervals for the block probability, number of customers in the system, and the time in the system.
Example Call: java ProbFinder 2 1 0 100 10000

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.io.*;

public class ProbFinder {

    public static void main(String args[]) throws IOException {
        //Initialization variables; in order: lambda, mu, the amount of items in the
queue to begin
        //with, the warmup period, the length of time each period counted lasts
        double lambda = Double.parseDouble(args[0]);
        double mu = Double.parseDouble(args[1]);
        double initSize;
        double warmup;
        double deltaT;

        //These try/catch blocks allow you to only specify lambda and mu,
        //taking default values otherwise

        try {
            initSize = Double.parseDouble(args[2]);
        } catch (IndexOutOfBoundsException e) {
            initSize = 0;
        }

        try {
            warmup = Double.parseDouble(args[2]);
        } catch (IndexOutOfBoundsException e) {
            warmup = 100;
        }
```

```java
try {
    deltaT = Double.parseDouble(args[3]);
} catch (IndexOutOfBoundsException e) {
    deltaT = 100;
}

//Initialize lists to hold the state variables
ArrayList<Double> blockProb = new ArrayList<Double>();
ArrayList<Double> meanCusNum = new ArrayList<Double>();
ArrayList<Double> meanCusTime = new ArrayList<Double>();

//Create a queue and warm it up
MM2Queue system = new MM2Queue(lambda, mu, (int) initSize);
system.runFor(warmup);

//For 100 times...
for(int i = 0; i < 100; i++) {
    //run the queue for deltaT and add the state variables
    //to their respective lists
    system.runFor(deltaT);
    blockProb.add(system.getBlockProb());
    meanCusNum.add(system.getMeanCusNum());
    meanCusTime.add(system.getMeanCusTime());
}

//Get the average, standard deviation, and lower and upper bounds for
//each state variable
double[] blockAvg = avgAndStdDev(blockProb);
double[] numAvg = avgAndStdDev(meanCusNum);
double[] timeAvg = avgAndStdDev(meanCusTime);

//Print results
System.out.println("Blocking probability");
System.out.println("--Average: "+blockAvg[0]);
System.out.println("--Standard deviation: "+blockAvg[1]);
System.out.println("--Lower bound (90% confidence): "+blockAvg[2]);
System.out.println("--Upper bound (90% confidence): "+blockAvg[3]);
System.out.println("--Range Error: "+(blockAvg[3]-blockAvg[2]));
System.out.println();

System.out.println("Number of customers in the system");
System.out.println("--Average: "+numAvg[0]);
```

```
        System.out.println("--Standard deviation: "+numAvg[1]);
        System.out.println("--Lower bound (90% confidence): "+numAvg[2]);
        System.out.println("--Upper bound (90% confidence): "+numAvg[3]);
        System.out.println("--Range Error: "+(numAvg[3]-numAvg[2]));
        System.out.println();

        System.out.println("Time spent in the system");
        System.out.println("--Average: "+timeAvg[0]);
        System.out.println("--Standard deviation: "+timeAvg[1]);
        System.out.println("--Lower bound (90% confidence): "+timeAvg[2]);
        System.out.println("--Upper bound (90% confidence): "+timeAvg[3]);
        System.out.println("--Range Error: "+(timeAvg[3]-timeAvg[2]));
    }

    public static double[] avgAndStdDev(ArrayList<Double> list) {
        //Initialize array to hold results
        double[] values = new double[4];

        //Calculate average
        double avg = 0.0;
        for (Double d : list)
            avg += d;
        avg = avg/list.size();
        values[0] = avg;

        //Calculate standard deviation
        double stdDev = 0;
        for (Double d : list)
            stdDev += Math.pow((avg - d), 2);
        stdDev = Math.sqrt(stdDev/(list.size() - 1));
        values[1] = stdDev;

        //Calculate lower and upper bounds, respectively, given 90% confidence
interval
        values[2] = avg - 1.645*stdDev/(Math.sqrt(list.size()));
        values[3] = avg + 1.645*stdDev/(Math.sqrt(list.size()));

        //Return
        return values;
    }
}
```

Works Cited

1. https://docs.oracle.com/javase/8/docs/api/java/util/Random.html