Chapter Two:
Test Plan Foundations

Preface to Method Selection

The STEM engine manipulates inputted data to produce spatiotemporal models, therefore it must contain a plethora of mathematical functions needed to compute related statistical values for modeling. Thus, most of the methods that we are interested in testing are going to be math related methods.

The Selected Initial Method

```java
/**
 * compute the log(n!)
 * @param n
 * @return log(n!)
 */
 static double lnFactorial(int n) {
    double retVal = 0.0;

    for (int i = 1; i <= n; i++) {
      retVal += Math.log((double)i);
    }

    return retVal;
 }
```

This genre of method is relatively easy to test because we can compute these values outside of the method, and produce an oracle based off that information. Because of the nature of the method, it is easy to recognize what sort of input *should* be invalid, and which input is reasonable. This nicely segues us into the next section of what kinds of tests we can conduct.

Method Context

The lnFactorial() method is used in the calculation of binomial distributions within the StochasticPoissonSIDiseaseModelImpl class, for modeling purposes.

```
// The effective Infectious population  is a dimensionles number normalize by total
// population used in teh computation of bets*S*i where i = Ieffective/Pop.
// This includes a correction to the current
// infectious population (Ieffective) based on the conserved exchange of people (circulation)
// between regions. Note that this is no the "arrivals" and "departures" which are
// a different process.
final double effectiveInfectious = getNormalizedEffectiveInfectious(diseaseLabel.getNode(), diseaseLabel, currentSI.getI());

int S = (int)currentSI.getS();
double prob = 0.0;
if(getNonLinearityCoefficient() != 1.0 && effectiveInfectious >= 0.0)
    prob = transmissionRate * Math.pow(effectiveInfectious, getNonLinearityCoefficient());
else
    prob = transmissionRate * effectiveInfectious;
double rndVar = rand.nextDouble();
int pickN = 0;
pickN = BinomialDistributionUtil.fastPickFromBinomialDist(prob, S, rndVar);

// Move pickK from S to I;

double numberOfSusceptibleToInfected = pickN;

double numberOfInfectedToSusceptible = getAdjustedRecoveryRate(timeDelta)
        * currentSI.getI();
```

Test Cases

Firstly, we know that this method should not take anything other than an integer, as is specified by its parameter. Second, this method should not compute the value of a negative integer, due to the nature of factorials. Third, the method should not compute the value of an out of bounds integer; thus -2,147,483,648 <= n <= 2,147,483,647. Fourth, the method should not produce valid output if there is white space between an integer, for example, passing in 1 2, should not return valid output. Lastly, the method should produce a valid computation of ln(n!), given valid input; a positive integer, in the range of $2^{32} - 1$.

Therefore, the test cases are:

Test ID	Input (n)	Expected Output (The Driver Does Not Know These Values)
#1	0	0
#2	-5	Error: Domain
#3	2,000,000,000	4.083282604664613E10
#4	1	0
#5	3	1.791759469228055

System Architecture Relating to Method Behavior

The system architecture of the machine that might run this method will have no bearing on how the method runs. This is because the method is written in a high-level language, and by definition, this means that system architecture has no effect on the behavior of the method, so long as the system has Java installed on it.

Test Case 1:

```
{
  "id": 1,
  "requirement": "Method computes the log(n!)",
  "component": "../project/BinomialDistributionUtil.java",
  "method": "lnFactorial",
  "driver": "../testCasesExecutables/lnFactorial/testCase1.java",
  "input": "0",
  "output": "0"
}
```

Test Case 2:

```
{
  "id": 2,
  "requirement": "Method computes the log(n!)",
  "component": "../project/BinomialDistributionUtil.java",
  "method": "lnFactorial",
  "driver": "../testCasesExecutables/lnFactorial/testCase2.java",
  "input": "-5",
  "output": "Error"
}
```

Test Case 3:

```
{
  "id": 3,
  "requirement": "Method computes the log(n!)",
  "component": "../project/BinomialDistributionUtil.java",
  "method": "lnFactorial",
  "driver": "../testCasesExecutables/lnFactorial/testCase3.java",
  "input": "2000000000",
  "output": "4.083282604664613E10"
}
```

Test Case 4:

```
{
  "id": 4,
  "requirement": "Method computes the log(n!)",
  "component": "../project/BinomialDistributionUtil.java",
  "method": "lnFactorial",
  "driver": "../testCasesExecutables/lnFactorial/testCase4.java",
  "input": "1",
  "output": "0"
}
```

Test Case 5:

```
{
  "id": 5,
  "requirement": "Method computes the log(n!)",
  "component": "../project/BinomialDistributionUtil.java",
  "method": "lnFactorial",
  "driver": "../testCasesExecutables/lnFactorial/testCase5.java",
  "input": "3",
  "output": "1.791759469228055"
}
```