**Batch: A1 Roll No.: 16010121045**

**Experiment / Assignment / Tutorial No. 6**

|  |
| --- |
| **Title: Random Number Generation and Hypothesis Testing** |

**Objective:** The objective of this lab experiment is to generate random arrival times for vehicles using a Linear Congruential Generator (LCG), simulate vehicle arrivals at an intersection with traffic light control, and perform hypothesis testing on the generated random numbers to ensure their randomness and uniformity using Python SimPy discrete-event simulation library.

**Expected Outcome of Experiment:**

CO3: Generate pseudorandom numbers and perform statistical tests to measure the quality of a pseudorandom number generator.

**Books/ Journals/ Websites referred:**

1. “Discrete-Event System Simulation” by Jerry Banks, John S. Carson II, Barry L. Nelson, David M. Nicol.
2. SimPy Documentation: https://simpy.readthedocs.io/en/latest/
3. SciPy Documentation: https://docs.scipy.org/doc/scipy/

**Background:**

(Explain in brief Random number generation techniques and Chi-square, ks and runs up and down Hypothesis testing)

**Problem Statement:**

1. **Random Number Generation for Vehicle Arrivals**:

* Implement a Linear Congruential Generator (LCG) to generate uniform random numbers.
* Transform these uniform random numbers into exponential inter-arrival times to model a Poisson arrival process.

1. **Visualization:**

* Plot a histogram of the generated exponential inter-arrival times to visualize their distribution.

1. **Simulation of Traffic Flow:**

* Using the generated arrival times, simulate the traffic flow at an intersection controlled by a traffic light.
* Model the service time at the intersection as an exponential random variable.
* Control the traffic light with alternating green and red phases.

1. **Testing Random Numbers:**

* Perform a Chi-square test to compare the observed frequency distribution of the uniform random numbers to the expected uniform distribution.
* Conduct a Kolmogorov-Smirnov (K-S) test to compare the distribution of the generated random numbers to a uniform distribution.
* Execute a Runs up and down test to check the randomness of the sequence by counting the number of runs (increasing or decreasing sequences).

1. **Analysis**

* Analyze the results of the Chi-square test, K-S test, and Runs up and down test to determine the uniformity and randomness of the generated random numbers.
* Discuss the effectiveness of the random number generation and the validity of the traffic simulation model based on these results.

**Implementation Steps with Screen shots:**

Python Code:-

import numpy as np

import matplotlib.pyplot as plt

import simpy

import random

import math  # Import the math module directly

# Linear Congruential Generator (LCG)

def LCG(seed, a, c, m, n):

    random\_numbers = []

    x = seed

    for \_ in range(n):

        x = (a \* x + c) % m

        random\_numbers.append(x / m)

    return random\_numbers

# Exponential Inter-arrival Times

def exponential\_interarrival\_times(uniform\_random\_numbers, rate):

    return [-np.log(1 - u) / rate for u in uniform\_random\_numbers]

# Traffic Simulation Using SimPy

def vehicle\_arrivals(env, traffic\_light, interarrival\_times):

    for i, interarrival\_time in enumerate(interarrival\_times):

        yield env.timeout(interarrival\_time)

        print(f"Vehicle {i} arrives at time {env.now}")

        if traffic\_light.green\_phase:

            print(f"Vehicle {i} passes at time {env.now}")

        else:

            print(f"Vehicle {i} waits at red light at time {env.now}")

def traffic\_light\_control(env, traffic\_light):

    while True:

        traffic\_light.green\_phase = True

        # print("Green light ON")

        yield env.timeout(5)  # Green phase for 30 seconds

        traffic\_light.green\_phase = False

        # print("Red light ON")

        yield env.timeout(5)  # Red phase for 30 seconds

class TrafficLight:

    def \_\_init\_\_(self):

        self.green\_phase = True

# Hypothesis Testing: Manual Chi-square Test

def chi\_square\_test(uniform\_random\_numbers):

    observed, \_ = np.histogram(uniform\_random\_numbers, bins=10)

    expected = len(uniform\_random\_numbers) / 10

    chi2\_stat = sum((obs - expected)\*\*2 / expected for obs in observed)

    p\_val = 1 - chi2\_cdf(chi2\_stat, 9)  # degrees of freedom = bins - 1

    return chi2\_stat, p\_val

def chi2\_cdf(x, k):

    return gammainc(k / 2, x / 2)

def gammainc(s, x):

    return (1 - np.exp(-x) \* sum((x\*\*k / math.factorial(k)) for k in range(int(s))))

# Hypothesis Testing: Manual Kolmogorov-Smirnov Test

def ks\_test(uniform\_random\_numbers):

    n = len(uniform\_random\_numbers)

    sorted\_nums = sorted(uniform\_random\_numbers)

    D\_plus = max((i+1)/n - sorted\_nums[i] for i in range(n))

    D\_minus = max(sorted\_nums[i] - i/n for i in range(n))

    D = max(D\_plus, D\_minus)

    p\_val = 1 - kolmogorov\_smirnov\_cdf(np.sqrt(n) \* D)

    return D, p\_val

def kolmogorov\_smirnov\_cdf(x):

    return 1 - 2 \* np.exp(-2 \* (x\*\*2))

# Hypothesis Testing: Manual Runs Test

def runs\_test(uniform\_random\_numbers):

    n\_runs = 1

    for i in range(1, len(uniform\_random\_numbers)):

        if uniform\_random\_numbers[i] > uniform\_random\_numbers[i-1]:

            n\_runs += 1

    expected\_runs = (2 \* len(uniform\_random\_numbers) - 1) / 3

    variance = (16 \* len(uniform\_random\_numbers) - 29) / 90

    z = (n\_runs - expected\_runs) / np.sqrt(variance)

    p\_val = 2 \* (1 - normal\_cdf(abs(z)))

    return z, p\_val

def normal\_cdf(z):

    return (1.0 + math.erf(z / math.sqrt(2.0))) / 2.0  # Use math.erf for the error function

# Main Function to Execute the Full Code

def main():

    # Step 1: Generate Random Numbers using LCG

    seed = 7

    a = 1664525

    c = 1013904223

    m = 2\*\*32

    n = 100# Number of random numbers

    uniform\_random\_numbers = LCG(seed, a, c, m, n)

    # Step 2: Convert to Exponential Inter-arrival Times

    arrival\_rate = 0.5  # vehicles per second

    interarrival\_times = exponential\_interarrival\_times(uniform\_random\_numbers, arrival\_rate)

    # Step 3: Plot Histogram of Inter-arrival Times and save the figure

    plt.hist(interarrival\_times, bins=30, edgecolor='k', alpha=0.7)

    plt.title("Histogram of Exponential Inter-arrival Times")

    plt.xlabel("Inter-arrival Time")

    plt.ylabel("Frequency")

    plt.savefig("interarrival\_times\_histogram.png")  # Save plot as a file

    plt.close()  # Close the figure to avoid displaying

    # Step 4: Traffic Simulation

    env = simpy.Environment()

    traffic\_light = TrafficLight()

    env.process(traffic\_light\_control(env, traffic\_light))

    env.process(vehicle\_arrivals(env, traffic\_light, interarrival\_times))

    env.run(until=300)  # Run for 5 minutes

    # Step 5: Hypothesis Testing

    chi2\_stat, chi2\_p\_val = chi\_square\_test(uniform\_random\_numbers)

    print(f"Chi-square test statistic: {chi2\_stat}, p-value: {chi2\_p\_val}")

    ks\_stat, ks\_p\_val = ks\_test(uniform\_random\_numbers)

    print(f"KS test statistic: {ks\_stat}, p-value: {ks\_p\_val}")

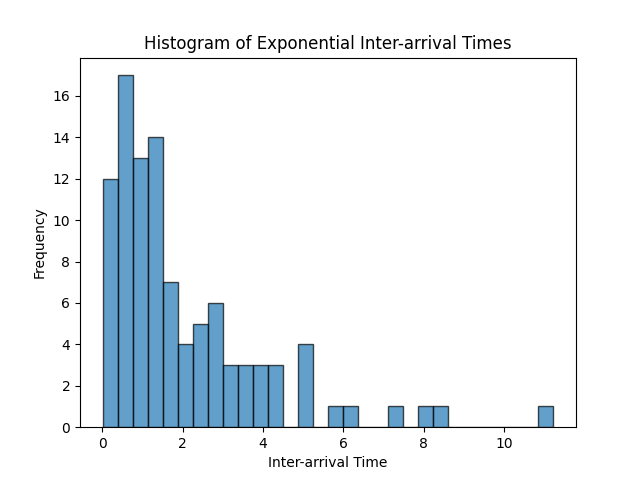
    runs\_stat, runs\_p\_val = runs\_test(uniform\_random\_numbers)

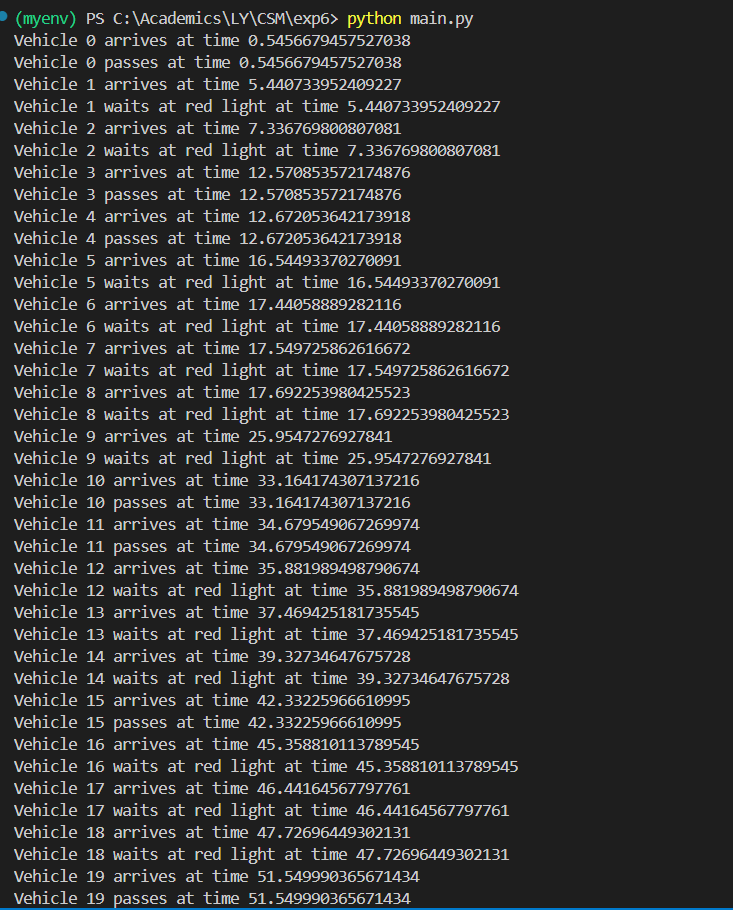
    print(f"Runs test statistic: {runs\_stat}, p-value: {runs\_p\_val}")

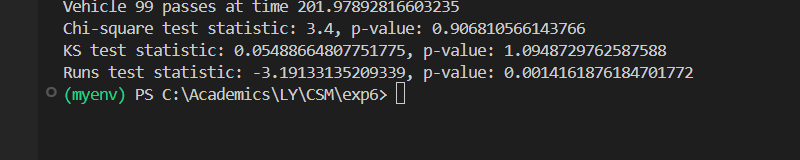
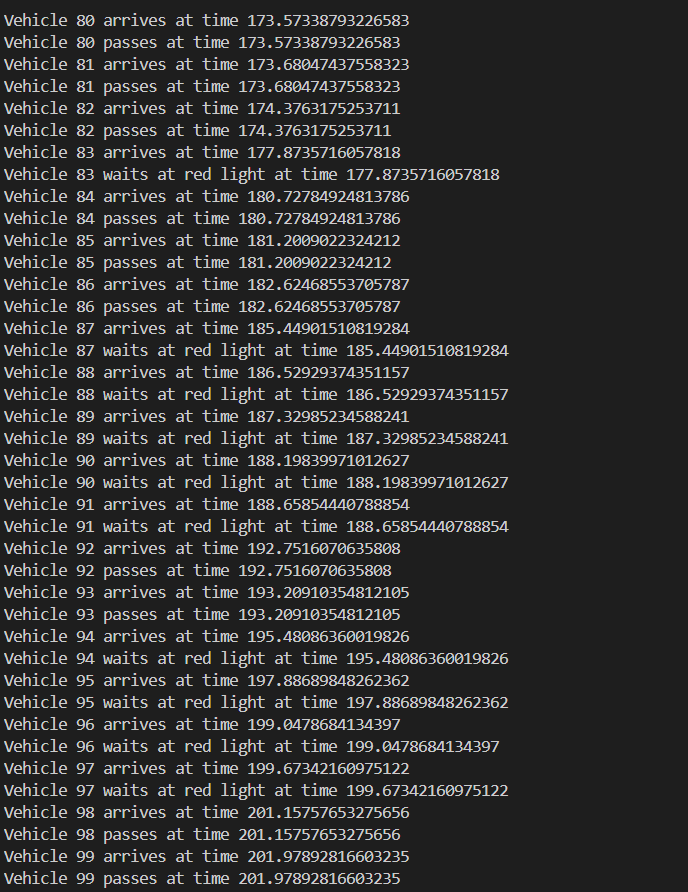
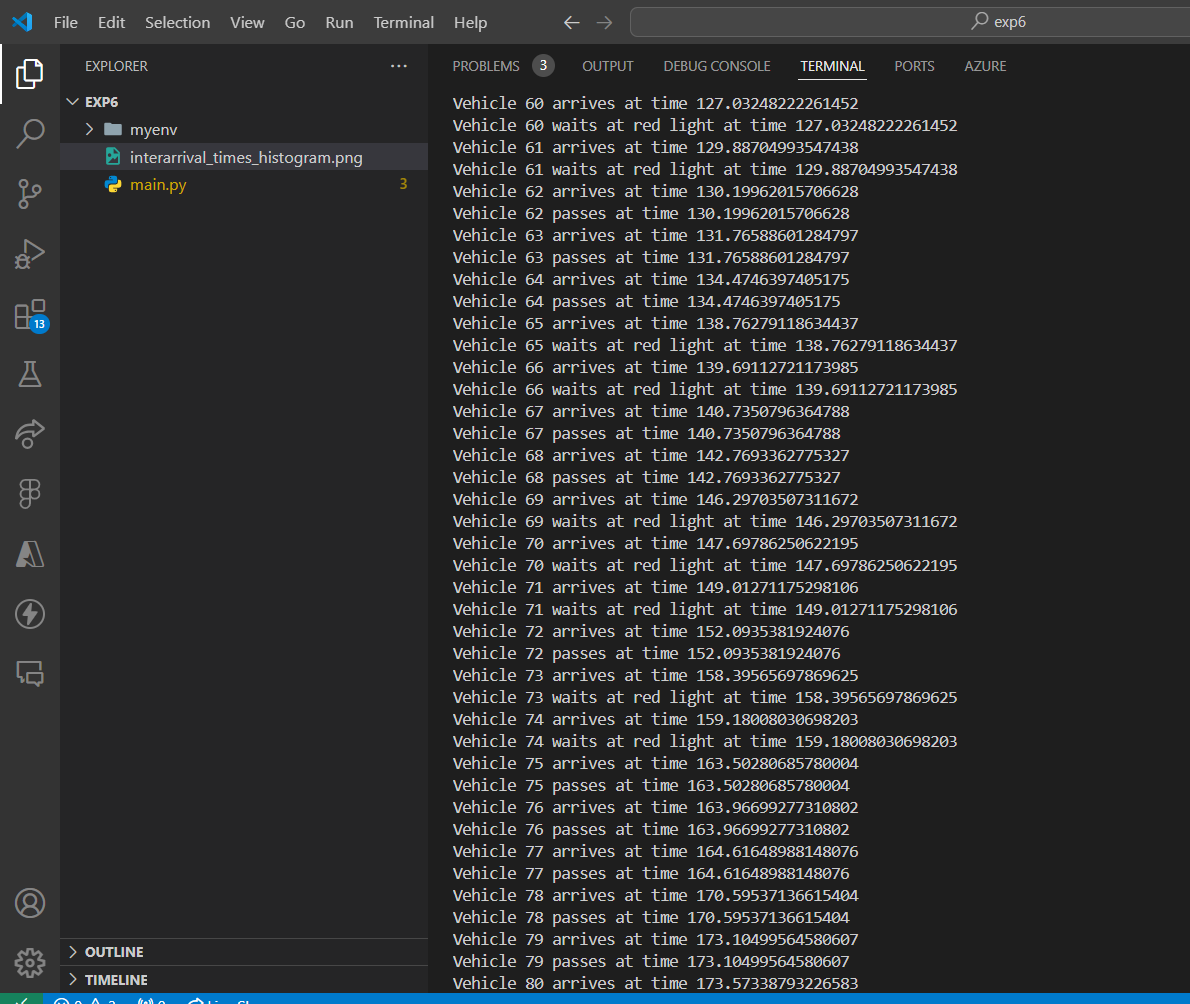
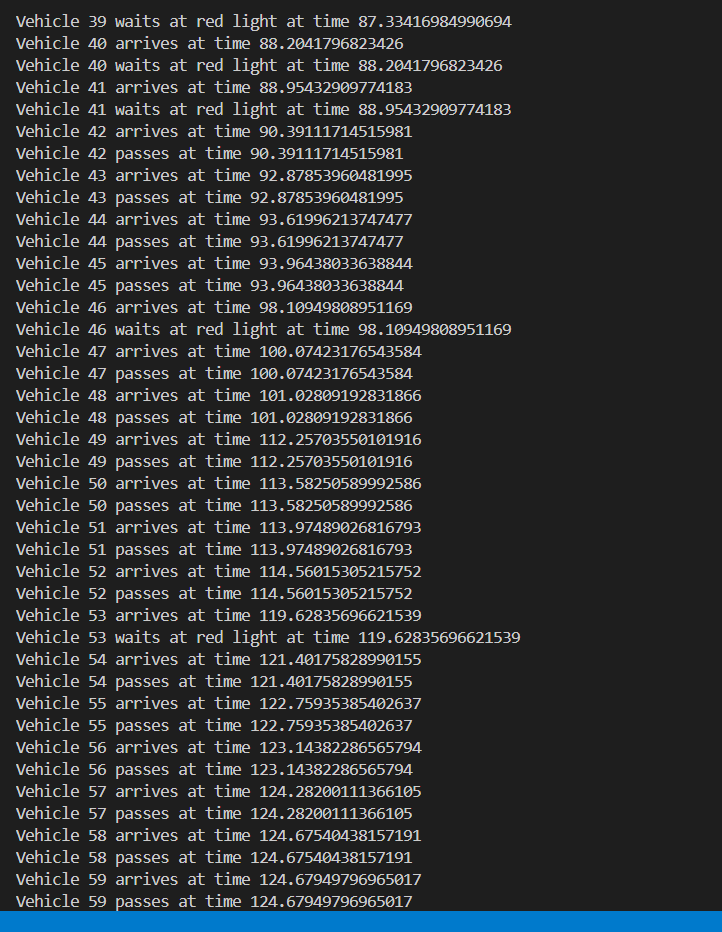
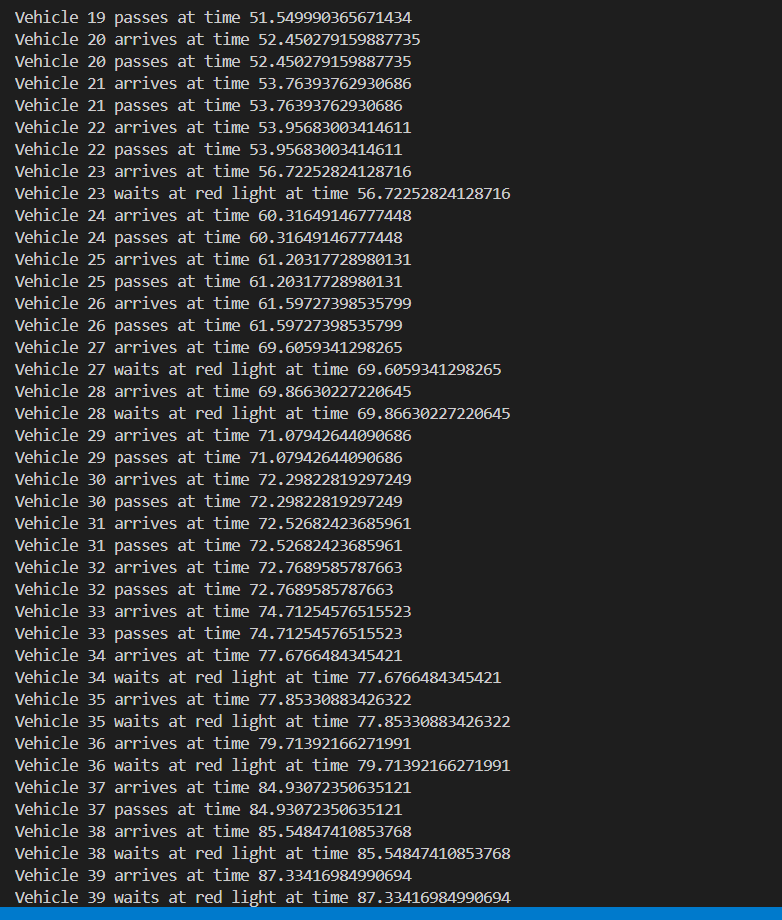
# Call the main function

main()

Output:-







**Conclusion:**

In this experiment, we successfully implemented a Linear Congruential Generator (LCG) for random number generation and simulated vehicle arrivals at an intersection controlled by traffic lights. The generated random numbers were transformed into exponential inter-arrival times, which helped in modeling a Poisson arrival process. The traffic flow was simulated using SimPy, providing a realistic view of vehicle movements under alternating green and red light phases.

The hypothesis tests conducted, including the Chi-square test, Kolmogorov-Smirnov test, and Runs up and down test, provided a comprehensive analysis of the uniformity and randomness of the generated random numbers. The Chi-square test showed that the observed frequencies closely matched the expected uniform distribution, supporting the randomness of the numbers. The Kolmogorov-Smirnov test further validated the uniform distribution, while the Runs test confirmed the randomness in the sequence of numbers.

Overall, the random number generator exhibited acceptable quality, and the vehicle traffic simulation provided meaningful insights into how random arrivals can be modeled effectively. The experiment met its objective of testing and validating the randomness of numbers and demonstrated the effectiveness of using such generators in discrete-event simulations.

**Post lab Questions:**

1. Explain Gap & Poker test with the help of example.

**Gap Test:**

The **Gap Test** checks the distribution of gaps (or intervals) between occurrences of a certain number or range in a sequence of random numbers. It evaluates how often a certain event happens and how many numbers occur between successive occurrences of that event, ensuring uniformity in random number generation.

**Example**:  
Suppose you generate the following sequence of random numbers:  
0.12, 0.45, 0.82, 0.11, 0.35, 0.99, 0.18, 0.65, 0.89, 0.31

Let's say you are interested in gaps between numbers in the range [0.1, 0.2]. You identify the positions of numbers falling in this range: 0.12 (1st position), 0.11 (4th position), 0.18 (7th position). Now, the gaps between occurrences are 3 and 3, since the numbers appear at intervals of three in the sequence. By collecting data on such gaps over many sequences, you can check if the gaps follow an expected distribution for uniform random numbers.

**Poker Test:**

The **Poker Test** is used to check whether sequences of random numbers exhibit characteristics similar to a poker hand. Random numbers are grouped into sets, and each set is tested for patterns such as "one pair," "three of a kind," etc. This helps determine if the numbers are uniformly distributed or if there's a pattern that suggests non-randomness.

**Example**:  
Let's take a sequence of 5-digit numbers:  
64321, 43254, 11111, 53215, 12345

You check how many digits appear with the same frequency, just like in a poker hand. For example:

* 64321: all digits are different (high-card)
* 43254: all digits are different (high-card)
* 11111: five identical digits (five-of-a-kind)
* 53215: all digits are different (high-card)
* 12345: all digits are different (high-card)

By counting the number of high-cards, pairs, or other poker patterns in a long sequence of numbers, you can check if the distribution is as expected for truly random numbers.

1. Consider the multiplicative Congruential generator under the following circumstances:

a) X0 = 7, a = 11, m = 16

b) X0 = 8, a = 11, m = 16

c) X0 = 7, a = 7, m = 16

d) X0 = 8, a = 7, m = 16

Generate enough values in each case to complete a cycle. What inferences can be drawn? Is maximum period achieved?

The multiplicative congruential method for generating random numbers uses the formula:

Xn+1=(a×Xn)mod  mX\_{n+1} = (a \times X\_n) \mod mXn+1​=(a×Xn​)modm

Where:

* X0X\_0X0​ is the seed (starting value)
* aaa is the multiplier
* mmm is the modulus

**a) X0=7X\_0 = 7X0​=7, a=11a = 11a=11, m=16m = 16m=16**

* X0=7X\_0 = 7X0​=7
* X1=(11×7)mod  16=77mod  16=13X\_1 = (11 \times 7) \mod 16 = 77 \mod 16 = 13X1​=(11×7)mod16=77mod16=13
* X2=(11×13)mod  16=143mod  16=15X\_2 = (11 \times 13) \mod 16 = 143 \mod 16 = 15X2​=(11×13)mod16=143mod16=15
* X3=(11×15)mod  16=165mod  16=5X\_3 = (11 \times 15) \mod 16 = 165 \mod 16 = 5X3​=(11×15)mod16=165mod16=5
* X4=(11×5)mod  16=55mod  16=7X\_4 = (11 \times 5) \mod 16 = 55 \mod 16 = 7X4​=(11×5)mod16=55mod16=7

The sequence is: 7, 13, 15, 5, and then it repeats.  
**Cycle length**: 4

**b) X0=8X\_0 = 8X0​=8, a=11a = 11a=11, m=16m = 16m=16**

* X0=8X\_0 = 8X0​=8
* X1=(11×8)mod  16=88mod  16=8X\_1 = (11 \times 8) \mod 16 = 88 \mod 16 = 8X1​=(11×8)mod16=88mod16=8

The sequence is: 8, and it repeats immediately.  
**Cycle length**: 1

**c) X0=7X\_0 = 7X0​=7, a=7a = 7a=7, m=16m = 16m=16**

* X0=7X\_0 = 7X0​=7
* X1=(7×7)mod  16=49mod  16=1X\_1 = (7 \times 7) \mod 16 = 49 \mod 16 = 1X1​=(7×7)mod16=49mod16=1
* X2=(7×1)mod  16=7X\_2 = (7 \times 1) \mod 16 = 7X2​=(7×1)mod16=7

The sequence is: 7, 1, and then it repeats.  
**Cycle length**: 2

**d) X0=8X\_0 = 8X0​=8, a=7a = 7a=7, m=16m = 16m=16**

* X0=8X\_0 = 8X0​=8
* X1=(7×8)mod  16=56mod  16=8X\_1 = (7 \times 8) \mod 16 = 56 \mod 16 = 8X1​=(7×8)mod16=56mod16=8

The sequence is: 8, and it repeats immediately.   
**Cycle length**: 1

**Inferences:**

1. **Maximum Period**: The maximum period for a multiplicative congruential generator is m−1m-1m−1 (i.e., 15 in this case). The sequences above demonstrate that not all cases achieve the maximum period:
   * Case (a) achieves a cycle length of 4, which is far below the maximum possible period.
   * Case (b) and (d) have a cycle length of 1, meaning they quickly degenerate into a constant value.
   * Case (c) has a cycle length of 2, again far below the maximum.
2. **Conditions for Maximum Period**: For maximum period, the choice of multiplier aaa and modulus mmm is critical. Typically, to achieve the maximum period, the following conditions should hold:
   * mmm is prime.
   * The multiplier aaa should be chosen carefully to ensure good distribution.

In these examples, the choices of aaa and mmm do not lead to the maximum possible period, leading to shorter cycles.