PRNG Lab

AP Computer Science A

Name: ______________________________

Contents

Background

Within a computer system or program, a pseudo-random number generator (PRNG) is used to generate sequences of seemingly random numbers. Each PRNG is an algorithm to generate a sequence of numbers that mimic the properties of truly random numbers. They are called "pseudo"-random because these algorithms are actually deterministic; that is, if you knew the starting value (called a generator's *seed*) and the algorithm in use, you could easily predict all future numbers generated by that algorithm. Nonetheless, PRNGs in wide use today benefit from being completely mathematical in nature, meaning they are easily programmed in software, and generally very fast. Their deterministic nature can be combated by chosing different seed values and by choosing algorithms and seed values that result in long *periods*. A *period* of a PRNG is the length of the sequence of "random" numbers before repetitions occur.

This lab was designed to give you some exposure to the generation of random numbers. The following example shows you how a pseudo-random number generator works to generate a sequence of seemingly random numbers. The two activities that follow will ask you to implement your own pseudo-random number generators based on historic and well-known algorithms.

Example

A bit of chaos...

Work with basic chaos theory can provide for an interesting pseudo-random number generator.

Given $0 < x_n < 1$ and $r > 0$, the following generating formula can create a sequence of numbers that seem to jump randomly in some subinterval of $[0, 1]$ after the first few iterations.

$$x_{n+1} = rx_n(1 - x_n)$$

Here is a sequence of values for $r = 3.63$ and $x_0 = 0.47$:

$$\{0.470000000,\ 0.904233000,\ 0.314342325,\ 0.782378356,\ 0.618052744,\ 0.856910685,\ 0.445091590,\ 0.896555791,\ \ldots\}$$

Because of the way that this particular algorithm works, the seed value, x_n is not nearly as important as the choice in r. Each will generate pretty distinct sequences, regardless of the initial seed value, x_n.
For instance, here is a sequence of values for $r = 3.64$ and $x_0 = 0.47$:

$$\{0.470000000,\ 0.944089000,\ 0.200054999,\ 0.606525056,\ 0.904492644,\ 0.327401808,\ 0.834595385,\ 0.523194068,\ \ldots\}$$

There will be, however, som values for r which do not produce good sequences of seemingly random number. Here is a sequence of values for $r = 1.3$ with, again, $x_0 = 0.47$:

$$\{0.470000000,\ 0.323830000,\ 0.284653370,\ 0.264713578,\ 0.253032389,\ 0.245709099,\ 0.240936979,\ 0.237752257,\ 0.235593957,\ 0.234116278,\ \ldots\}$$

Notice that the random numbers begin exhibiting "stable" (non-changing) values in certain positions, with $0.23\ldots$ beginning each of the random numbers towards the end of the printed sequence. In fact, no future generated number will begin with anything else, with an increasing number of decimal positions becoming stable as the sequence progresses. In fact, spreadsheet software used to calculate these sequences shows complete repetition of the pseudo-random number by the 92nd iteration.

Applications

Question #1: Why are random numbers important to the study and application of computer science?

Question #2: Although hardware based “true” random number generators are available, software-based pseudo-random number generators still remain the predominant method for generating random numbers in use today. Why do you think developers continue to rely on pseudo-random number generators?

Question #3: Do you think a software based “true” random number generator will ever be created? Explain why or why not.

Question #4: Why is understanding the period of a sequence of random numbers important for the study, implementation, and use of a pseudo-random number generator?

Activity #1

Introduction

In 1949, John von Neumann proposed a method for generating pseudo-random numbers known as the "Middle-Square Method". This method works to generate numbers in the following way:

1. Beginning with an n-digit seed number, square the current number.
2. Select the middle n-digits of the resulting number. This is the new, pseudo-random number.
3. Repeat the process from Step #1, using the new number.

Although this method does appear to generate sequences of pseudo-random numbers, it quickly fell to criticism due to relatively short periods. Nonetheless, it is an academically interesting way of generating random numbers. In this activity, you will be implementing a number of methods related to generating pseudo-random numbers using the middle-square method.

Example

Here is an example of using the middle-square method to generate a sequence of pseudo-random numbers. We will begin with the 4-digit seed value 5678.

$$\begin{aligned}(5678)^2 &= 32\underline{2396}84\\(2396)^2 &= 05\underline{7408}16\\(7408)^2 &= 54\underline{8784}64\\&\dots\end{aligned}$$

In this case, the sequence of pseudo-random numbers would be: $\{5678, 2396, 7408, 8784, \ldots\}$, continuing until a repetition of a four-digit number would cause the sequence, or a subsequence of it, to begin again.

Note that because a sequence of n-digits is not always exactly centered in the resultant square, a choice needs to be made as to which n-digits should be selected. This occurred with the selection of 7408 in the example above. Additionally, in order to form an n-digit number, numbers prefixed with as many as n zeroes can be considered as part of the sequence of n-digit numbers (e.g., 0032 can be considered a 4-digit number).

Exercises

1. Implement the `getMiddleSquare` method which takes as its parameters `x`, the current number in a sequence of pseudo-random numbers and `n`, the number of digits required for the generated number. This method should return the next number in the sequence using the middle-square method described above.
2. Overload `getMiddleSquare` to additional take as a parameter `N`, the number of pseudo-random numbers to generate. The method should return an array of integer values generated successively using the middle-square method.
3. Implement the `getMiddleSquarePeriod` method which will take as a parameter the beginning seed number, `x`, for a sequence of pseudo-random numbers generated using the middle-square method and `n`, the number of digits for each pseudo-random number in the sequence. This method should return the length of the sequence before any repetition occurs.

Questions

Question #5: Find a four-digit seed number with a period of less than 10. How did you conduct your search?

Question #6: Why does the existence of numbers such as those you discovered in Question #5 mean that the middle-square method for generating pseudo-random number generators is not of practical use?

Activity #2

Introduction

The PRNG built into many programming languages and/or their libraries (including those accessible by Java's `Random` and `Math`) use a method known as a *Linear Congruential Generator*. This technique generates the next pseudo-random number in a sequence of seemingly random numbers using the following formula:

$$x_{n+1} = (kx_n + c) \bmod m$$

This method requires the selection of three parameters: k, c, and m as well as an initial seed value, x_0. As with most pseudo-random number generators, the selection of these parameters becomes an important part of the algorithm implementation in order to generate a convincingly "random" sequence of numbers.

Example

Here is an example using a linear congruential generator with the following values:

$$k = 17 \quad c = 42 \quad m = 217 \quad x_0 = 72$$

These values yield the following sequence of numbers:

$$\{72,\ 181,\ 81,\ 117,\ 78,\ 66,\ 79,\ 83,\ 151,\ 5,\ 127,\ 31,\ 135,\ 167,\ 60,\ \ldots\}$$

Notice, however, what happens with a small change to the parameter m:

$$k = 17 \quad c = 42 \quad m = 216 \quad x_0 = 72$$

These values now yield the following sequence of numbers:

$$\{72,\ 186,\ 180,\ 78,\ 72,\ 186,\ 180,\ 78,\ \ldots\}$$

Note that once a repetition has occurred, the sequence will continue to repeat indefinitely.

Exercises

1. Implement the `getLCG` method that will take as parameters `k`, `c`, `m` and a current seed value, `x`, and returns the next value in the sequence generated by the linear congruential generator method described in the introduction.

2. Overload the `getLCG` method to additionally take as a parameter `N`, the number of pseudo-random numbers to generate. This method should return an array of `N` integer values.

3. Implement the `getLCGPeriod` method which will take as parameters `k`, `c`, `m` and an initial seed value, `x`, and returns the length of the sequence before any repetition occurs.

Questions

Question #7: Language implementations of a linear congruential generator use large powers of 2 as the m parameter (Java, for instance, uses 2^{48}). This allows for the generation of numbers with certain precision (say, 32-bit integers) by truncating the generated number to the first, for instance, 32-bits. Which part of the LCG formula does this make simpler? Explain why this works.

Question #8: Explain how you would generate random, floating-point numbers using a linear congruential generator.

Final Analysis

Question #9: Many language-implemented pseudo-random number generators will use a different seed value each time they are called. This seed value is usually derived from some system factors, such as the current system time. What are the benefits and drawbacks to this approach?

Question #10: Languages that use the linear congruential generator method for generating pseudo-random numbers, such as Java, usually do not vary the parameters other than the initial seed value. Java, for instance, has values: $k = 25214903917$, $c = 11$, and $m = 2^{48}$. Why do you think these languages use established parameters for their LGC rather than generate them at the time of use?

Question #11: What part of implementing either the middle-square method or the linear congruential generator method for generating pseudo-random numbers did you find most challenging? How did you overcome this challenge?

Question #12: What new programming techniques or knowledge did you learn as a result of this lab?

Template Class & Test Cases

```java
/**
 * PRNG Lab (Template Class and Test Cases)
 * This is the template class and test cases for the PRNG lab.
 * Written for the Woodstock School in Mussoorie, Uttarakhand, India.
 *
 * @author Jeffrey Santos
 * @version 1.0
 */

public class PRNG {
  public static void main(String[] args) {
    // Tests for getMiddleSquare:
    System.out.println(getMiddleSquare(48, 2));       // Output: 30
    System.out.println(getMiddleSquare(2015, 4));     // Output: 602
    System.out.println(getMiddleSquare(577324, 6));   // Output: 303000

    // Tests for getMiddleSquare (Overloaded):
    //  Output: [48, 30, 90, 10, 10]
    printArray(getMiddleSquare(48, 2, 5));
    //  Output: [2015, 602, 6240, 9376, 9093, 6826, 5942, 3073, 4433, 6514]
    printArray(getMiddleSquare(2015, 4, 10));
    //  Output: [577324, 303000, 809000]
    printArray(getMiddleSquare(577324, 6, 3));

    // Tests for getMiddleSquarePeriod:
    System.out.println(getMiddleSquarePeriod(48, 2));      // Output: 4
    System.out.println(getMiddleSquarePeriod(2015, 4));    // Output: 21
    System.out.println(getMiddleSquarePeriod(577324, 6)); // Output: 10

    // Tests for getLCG:
    System.out.println(getLCG(13, 47, 103, 10));    // Output: 74
    System.out.println(getLCG(7, 12, 217, 55));     // Output: 180
    System.out.println(getLCG(73, 1, 1024, 250));   // Output: 843

    // Tests for getLCG (Overloaded):
    //  Output: [10, 74, 82, 83, 96, 59, 93, 20, 101, 21]
    printArray(getLCG(13, 47, 103, 10, 10));
    //  Output: [55, 180, 187, 19, 145]
    printArray(getLCG(7, 12, 217, 55, 5));
    //  Output: [250, 843, 100, 133, 494, 223, 920, 601, 866, 755, 844, ...]
    printArray(getLCG(73, 1, 1024, 250, 100));

    // Tests for getLCGPeriod:
    System.out.println(getLCGPeriod(13, 47, 103, 10));  // Output: 17
    System.out.println(getLCGPeriod(7, 12, 217, 55));   // Output: 16
    System.out.println(getLCGPeriod(73, 1, 1024, 250)); // Output: 1024
  }

  /**
   * Calculates the next in a sequence of pseudo-random numbers using the
   * middle-square method.
   *
   * @param x The current seed number in a sequence of pseudo-random numbers.
   *  Precondition: x >= 0
   * @param n The number of digits for each number in the sequence.
   *  Precondition: n > 0
   * @return  The next number generated as the middle-square of the given value.
   */
  public static int getMiddleSquare(int x, int n) {
    // To be implemented in Activity #1, Exercise 1
  }

  /**
 * Calculates a sequence of pseudo-random numbers using the middle-square
 * method.
 *
 * @param x The initial seed number in a sequence of pseudo-random numbers.
 *  Precondition: x >= 0
 * @param n The number of digits for each number in the sequence.
 *  Precondition: n > 0
 * @param N The number of random numbers that should be generated.
 *  Precondition: N > 0
 * @return  An array of pseudo-random numbers generated from the given value
 *          using the middle-square method.
 */
public static int[] getMiddleSquare(int x, int n, int N) {
  //  To be implemented in Activity #1, Exercise 2
}

/**
 * Calculates the period of a sequence of pseudo-random numbers generated
 * using the middle-square method.
 *
 * @param x The initial seed number for the sequence of pseudo-random numbers.
 *  Precondition: x >= 0
 * @param n The number of digits for each number in the sequence.
 * @return  The length of the sequence of pseudo-random numbers generated from
 *          the given value using the middle-square method prior to a
 *          repetition occuring.
 *          Note: Repetition does not need to begin with the intial seed value.
 */
public static int getMiddleSquarePeriod(int x, int n) {
  //  To be implemented in Activity #1, Exercise 3
}

/**
 * Calculates the next in a sequence of pseudo-random numbers using a
 * linear congruential generator.
 *
 * @param k The multiplier for the LCG.
 *  Precondition: k > 0
 * @param c The constant offset for the LCG.
 *  Precondition: c >= 0
 * @param m The modulus for the LCG.
 *  Precondition: m > 1
 * @param x The current seed value for the sequence of pseudo-random numbers.
 *  Precondition: x >= 0
 * @return  The next number generated using a linear congruential generator
 *          with the given parameters.
 */
public static int getLCG(int k, int c, int m, int x) {
  //  To be implemented in Activity #2, Exercise 1
}

/**
 * Calculates a sequence of pseudo-random numbers generated using a linear
 * congruential generator.
 *
 * @param k The multiplier for the LCG.
 *  Precondition: k > 0
 * @param c The constant offset for the LCG.
 *  Precondition: c >= 0
 * @param m The modulus for the LCG.
 *  Precondition: m > 1
 * @param x The initial seed value for the sequence of pseudo-random numbers.
 *  Precondition: x >= 0
 * @param N The number of random numbers that should be generated.
 *  Precondition: N > 0
 * @return  An array of psedo-random numbers generated using a linear
```

```
   *          congruential generator with the given parameters.
   */
  public static int[] getLCG(int k, int c, int m, int x, int N) {
    // To be implemented in Activity #2, Exercise 2
  }

  /**
   * Calculates the period of a sequence of pseudo-random numbers generated
   * using a linear congruential generator.
   *
   * @param k The multiplier for the LCG.
   *  Precondition: k > 0
   * @param c The constant offset for the LCG.
   *  Precondition: c >= 0
   * @param m The modulus for the LCG.
   *  Precondition: m > 1
   * @param x The initial seed value for the sequence of pseudo-random numbers.
   *  Precondition: x >= 0
   * @return  The length of the sequence of pseudo-random numbers generated from
   *          the given value using the linear congruential generator with the
   *          given parameters prior to a repetition occuring.
   *          Note: Repetition does not need to begin with the intial seed value.
   */
  public static int getLCGPeriod(int k, int c, int m, int x) {
    // To be implemented in Activity #2, Exercise 3
  }

  /**
   * A helper method to print all values in a given integer array.
   *
   * @param a The integer array to be printed.
   */
  public static void printArray(int[] a) {
    System.out.print("[" + a[0]);
    for (int i = 1; i < a.length; i++)
      System.out.print(", " + a[i]);
    System.out.println("]");
  }
}
```