ALGORITHMS AND DATA STRUCTURES II

Lecture 11

Random Number Generators.

Lecturer: K. Markov
markov@u-aizu.ac.jp

PSEUDORANDOM NUMBERS

- In many applications, we need a sequence of random numbers. In this lecture we will discuss the algorithms for generating random number sequences.

- Since the numbers generated depend on an algorithm, we can not get true random numbers. What algorithms generate are pseudo-random numbers which are numbers that appear to be random.

PSEUDORANDOM NUMBERS

- Properties of Pseudorandom Numbers
  - Uncorrelated Sequences - The sequences of random numbers should be serially **uncorrelated**.
  - Long Period - The generator should be of **long period** (ideally, the generator should not repeat; practically, the repetition should occur only after the generation of a very large set of random numbers).
  - Uniformity - The sequence of random numbers should be uniform, and unbiased. That is, equal fractions of random numbers should fall into equal ``areas'' in space.
  - Efficiency - The generator should be efficient.

NUMBERS CYCLE

- Almost all random number generators have as their basis a sequence of pseudorandom integers.
- The Nature of the cycle:
  - the sequence has a finite number of integers.
  - the sequence gets traversed in a particular order.
  - the sequence repeats if the period of the generator is exceeded.
  - the integers need not be distinct; that is, they may repeat.

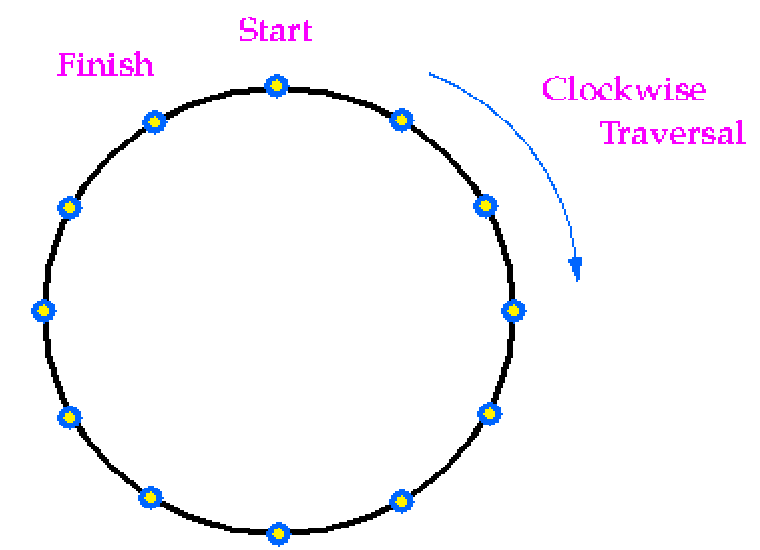


Figure 2: Illustration of Random Number Cycle

LINEAR CONGRUENTIAL METHOD

- Introduced by D. Lehmer in 1951, is the best-known method for the above purpose. In this method, numbers $x_1, x_2, \ldots$ are generated by

$$x_{i+1} = (\mathrm{A} \times x_i) \bmod \mathrm{M}.$$

- The value x_0, called **seed**, is needed to start the sequence. Notice that x_0 should not be 0 otherwise we will get a sequence of 0s.

LINEAR CONGRUENTIAL METHOD

- With x_i determined, we generate a corresponding **real** number as follows:

$$r_i = \frac{x_i}{M}$$

- When dividing by M, the values are then distributed on $(0,1)$.
- We desire uniformity, where any particular r_i is just as likely to appear as any other r_i, and the average of the r_i is very close to 0.5.

LINEAR CONGRUENTIAL METHOD

- For correctly chosen A and M, any x_0 with $1 \leq x_0 < \mathrm{M}$ is equally valid. If M is prime then $1 \leq x_i \leq \mathrm{M}\text{-}1$.
- For example, if $\mathrm{M} = 11$, $\mathrm{A} = 6$, and $x_0 = 1$, then the sequence of numbers is

$$\underbrace{6,3,7,9,10,5,8,4,2,1}_{\text{Period of M-1 = 10 digits.}},6,3,7,9,10,5,8,\ldots$$

If we choose $\mathrm{A} = 5$ then the sequence is

5, 3, 4, 9, 1, 5, 3, 4, 9, 1, 5,...

LINEAR CONGRUENTIAL METHOD

- If M is prime then there are always choices of A that give a full-period of M-1. If M is large, e.g., a 31-bit prime, a full-period generator should satisfy most applications.
- Lehmer suggested $M = 2^{31}-1 = 2147483647$. For this prime, $A = 48271$ is one of the values that gives a full-period generator:
  - $x_0 \quad = 1$
  - $x_{i+1} = (x_i \times 48271) \bmod (2^{31}-1)$

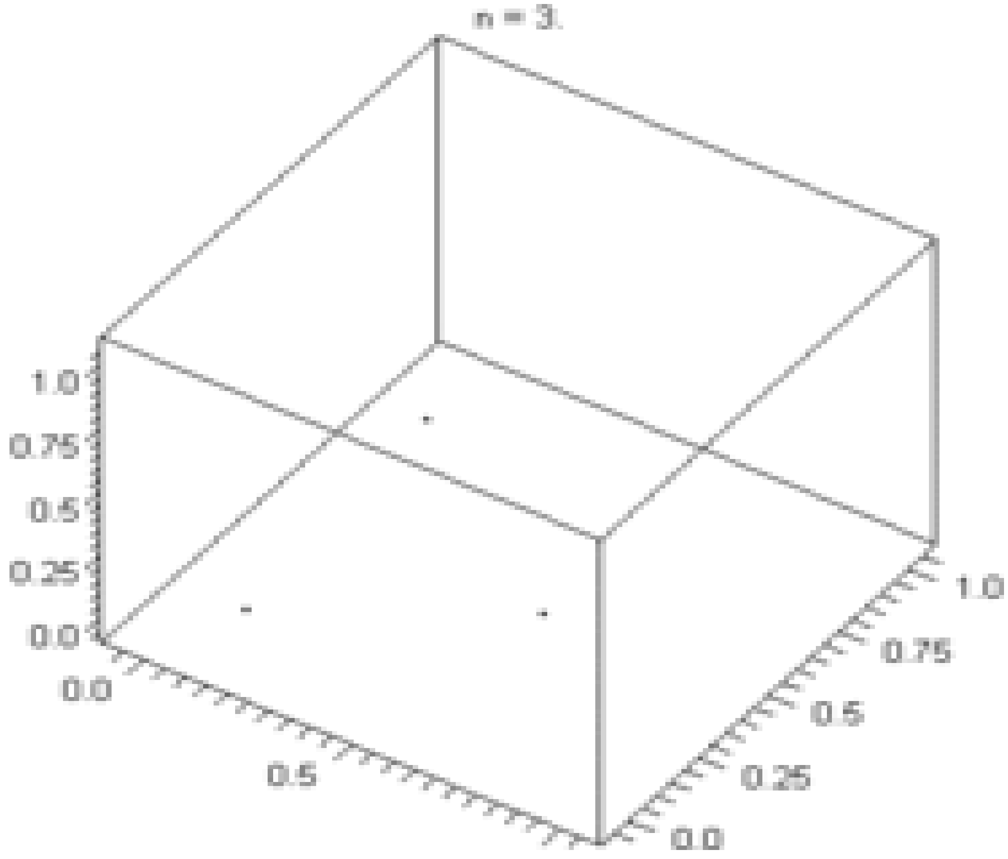
LINEAR CONGRUENTIAL METHOD

- A straightforward implementation would be:

```
def RAND(x_0)
    x = x_0                 // x_0 is the seed
    A = 48271
    M = 2147483647
    rand_seq = Ø    // Empty list
    for i = 1 to n:
        x = (Ax) % M
        rand_seq.append (x)
    return rand_seq
```

LINEAR CONGRUENTIAL METHOD

- Linear congruential random generator in three dimensions:



LINEAR CONGRUENTIAL METHOD

- However, the above implementation does not work well on most computers.

- The problem is that $(A \times x)$ could **overflow**. When a overflow occurs, $(A \times x)$ becomes the value of $(A \times x)\%2^b$, where b is the number of bits in the computer's integer type variables.

LINEAR CONGRUENTIAL METHOD

- This means that $(A \times x)\%\mathrm{M}$ becomes $((A \times x)\%2^b)\%M)$ if $(A \times x)$ overflows. This changes the results of the generated sequence and thus, affects the **pseudorandomness** of the sequence.
- Notice that if we take $M = 2^b$ then a overflow is an operation of $(A \times x)\%M$.
- In practice, many generators are based on the function

$$x_{i+1} = (A \times x_i + c) \bmod 2^b$$

LINEAR CONGRUENTIAL METHOD

- Now we give an algorithm for generating random numbers based on the above formula in a computer with 32-bit integer.
- For a 32-bit integer, 1-bit is used for the sign and the other 31-bits are used for the absolute value of the integer, and thus the largest integer that can be expressed is

$$2^{31}-1 = 2147483647.$$

LINEAR CONGRUENTIAL METHOD

- We choose $b = 31$ and $M = 2^b$. Since M can not be expressed by a 32-bit integer, we express M by (M - 1) + 1.

```
def RAND1(n)    // n - Number of random integers
    x = 53402397    // Seed
    rand_seq = Ø   // Empty list
    for i = 1 to n:
        x = 65539x +125654
        if x < 0:       // Check for overflow
            x += 2147483647  //  +(M-1)
            x += 1
        rand_seq.append (x)
    return rand_seq
```

LINEAR CONGRUENTIAL METHOD

- In the above algorithm, if c is odd then the values of x_i alternate between even and odd, and if c is even then the values of x_i are all even. This may not be a nice property for a sequence of random numbers.
- If we want to use $x_{i+1} = (\mathrm{A} \times x_i) \bmod \mathrm{M}$ to generate better random sequences then the overflow problem must be solved.

LINEAR CONGRUENTIAL METHOD

- Given M and A, let $Q = \lfloor M / A \rfloor$ and $R = M \% A$. Then it can be shown that $x_{i+1} = (A \times x_i) \bmod M$ is equivalent to

$$x_{i+1} = A(x_i \% Q) - R \lfloor x_i / Q \rfloor + M(\lfloor x_i / Q \rfloor - \lfloor A x_i / M \rfloor).$$

- For $M = 2^{31}-1$ and $A = 48271$, it is easy to check that there will be no overflow in calculating the above formula on a computer with 32-bit integer.

LINEAR CONGRUENTIAL METHOD

- Also $(\lfloor x_i / \mathrm{Q} \rfloor - \lfloor \mathrm{A}x_i / \mathrm{M} \rfloor)$ is either 0 or 1 and $(\lfloor x_i / \mathrm{Q} \rfloor - \lfloor \mathrm{A}x_i / \mathrm{M} \rfloor)$ is 1 if and only if
$$A(x_i \% Q) - R\lfloor x_i / Q \rfloor < 0.$$

- Thus, we can calculate x_{i+1} as follows:
  1. We first compute
  $$y = \mathrm{A}(x_i \% \mathrm{Q}) - \mathrm{R}\lfloor x_i / \mathrm{Q} \rfloor$$
  2. Then, if $y \geq 0$, $x_{i+1} = y$, otherwise $x_{i+1} = y + \mathrm{M}$.

LINEAR CONGRUENTIAL METHOD

- No overflow algorithm:

```
def RAND2(n)    // n - Number of random integers
    x = 1               // Seed
    A = 48271, M = 2147483647
    Q = M / A, R = M % A
    rand_seq = Ø   // Empty list
    for i = 1 to n:
        x = A (x % Q) – R (x / Q)
        if x < 0:
            x += M
        rand_seq.append (x)
    return rand_seq
```

SUBTRACTIVE METHOD

- The above algorithm has the period of $M - 1$.
- If we want to generate a random sequence with longer period, the subtractive method introduced below can be used.

SUBTRACTIVE METHOD

- Let M be an even integer and

$$x_0, x_1, \ldots, x_{54}$$

be a sequence of integers such that at least one of them is odd.

- Then the numbers generated by

$$x_n = (x_{n-24} - x_{n-55}) \bmod M$$

will have a period length of at least

$$2^{55}-1.$$

SUBTRACTIVE METHOD

- To initialize the sequence

$$x_0,\ x_1,\ldots,x_{54}$$

we can use the previous linear congruential algorithm, i.e.

$$\{x_0,\ x_1,\ldots,x_{54}\} = \mathrm{RAND2}\ (55)$$

SUBTRACTIVE METHOD

- The subtractive method:

```
def RAND3(n)   // n - Number of random integers
    x = 1, next = 0
    A = RAND2 (55)
    rand_seq = Ø   // Empty list
    for i = 1 to n:
        j = (next + 31) % 55
        x = A[j] – A[next]
        if x < 0:
            x += 2147483647,  x += 1
        A[next] = x
        next = (next + 1) % 55
        rand_seq.append (x)
    return rand_seq
```

RULES DEVELOPED BY KNUTH

- M should be large: it can be the computer word size. It will normally be convenient to make M a power of 10 or 2.
- A should not be too large or too small: a safe choice is to use a number with one digit less than M. A should be an arbitrary constant with no particular pattern in its digits, except that it should end with $\ldots x21$, with x even.

TESTING THE RANDOMNESS

- Many tests have been developed for determining whether a sequence shares various properties with a truly random sequence

- One statistical test, the X^2 (chi-square) test, is fundamental in nature and quite easy to implement.

TESTING THE RANDOMNESS

- The idea is to check whether or not the produced numbers are spread out reasonably. If we generate N positive integers less than r, then we would expect to get about N/r numbers of each value.
- But the frequencies of the occurrences of all the values should not be exactly the same: that would not be random.

TESTING THE RANDOMNESS

- The X^2 test:

$$D = \frac{\left(\sum_{i=0}^{r}(o_i - e)^2\right)}{e} \lesseqgtr X^2_{[1-\alpha, r-1]}$$

where o_i is the frequency of occurrence of value i, and $e = N/r$ is the expected frequency.

- If $D = 0$ there is an exact fit.
- If $D \leq X^2_{[1-\alpha, r-1]}$ test is passed with confidence α.

TESTING THE RANDOMNESS

- Example: $x_i = (125x_{i-1} + 1)\ mod\ (2^{12})$
- 1000 numbers with $x_0 = 1$
- $X^2_{[0.9,9]} = 14.68$
- Observed

Difference 10.38

- Observed is

less => Accept

Cell	Obsrvd	Exptd	$\frac{(o-e)^2}{e}$
1	100	100.0	0.000
2	96	100.0	0.160
3	98	100.0	0.040
4	85	100.0	2.250
5	105	100.0	0.250
6	93	100.0	0.490
7	97	100.0	0.090
8	125	100.0	6.250
9	107	100.0	0.490
10	94	100.0	0.360
Total	1000	1000.0	10.380

DISCUSSION

- Random numbers are the basis for many cryptographic applications.
- There is no reliable "independent" function to generate random numbers.
- Present day computers can only app-roximate random numbers, using pseudo-random numbers generated by Pseudo Random Number Generators (PRNG)s.

That's All for Today!