Data Structures

(Feodor F. Dragan)
Department of Mathematics and Computer Science
Kent State University

Fall, 2000

CHAPTER 10
Randomization

Contents

Introduction

- Many applications require random numbers
  - Cryptography, simulation, even searching and sorting
  - Programming testing

Definition 1. A permutation of 1, 2, ... , *N* is a sequence on *N* integers that includes each of 1, 2, ... , *N* exactly once.

- How do we test a sorting algorithm
- generate test data; random permutations, sort them
- Simulation is another important application
- Gaming, Randomized Algorithms

Generating Pseudorandom Numbers Algorithms

- The following algorithms differ in
  - Technique
  - Speed of Generation
  - Length of the repeatability cycle
  - Ease of programming

Random Number Generators

- No true randomness
- *Pseudorandom numbers* appear to be random
  - Have many properties of random numbers
- Hard to find good number generator
- How to flip a coin? ➔ look for the last digit of the clock
  - How fast is the clock?
- *Uniform random numbers:* each number occurs with the same probability
- Typically a random sequence, rather than one random number, is required

The Mid-Square Method

- Developed in mid 1940's by John von Neumann
- It starts with some initial number (seed)
- The number is then squared
- The middle digits of this square are used as the next number in the sequence

- Example:

$$\begin{aligned} & x_0 = 3187 \\ (3187)^2 &= 10156969 \Rightarrow x_1 = 1569 \\ (1569)^2 &= 02461761 \Rightarrow x_2 = 4617 \\ (4617)^2 &= 21316689 \Rightarrow x_3 = 3166 \\ (3166)^2 &= 10023566 \Rightarrow x_4 = 0235 \\ (0235)^2 &= 00055225 \Rightarrow x_5 = 0552 \\ (0552)^2 &= 00304704 \Rightarrow x_6 = 3047 \\ (3047)^2 &= 09284209 \Rightarrow x_7 = 2842 \end{aligned}$$

- This process could continue to generate;
  0769, 5913, 9635, 8332, 4222, 8542

- This technique has a number of shortcomings:
  1. Short repeatability period
  2. Whenever zero is generated, all succeeding numbers will be zero
- The Problem with Mid-Square method led to development of alternate algorithms

$$\begin{aligned} & x_0 = 44 \\ (44)^2 &= 1936 \Rightarrow x_1 = 93 \\ (93)^2 &= 8649 \Rightarrow x_2 = 64 \\ (64)^2 &= 4096 \Rightarrow x_3 = 09 \\ (09)^2 &= 0081 \Rightarrow x_4 = 08 \\ (08)^2 &= 0064 \Rightarrow x_5 = 06 \\ (06)^2 &= 0036 \Rightarrow x_6 = 03 \\ (03)^2 &= 0009 \Rightarrow x_7 = 00 \\ (00)^2 &= 0000 \Rightarrow x_8 = 00 \end{aligned}$$

Linear Congruential Generator

- Random sequence $$X_{i+1} = A X_i (\bmod M) \qquad (1)$$
- X_0 is the *seed* ➔ carefully initialized
- If *A* and *M* are carefully chosen, any other seed satisfying $A \le X_0 \le M$ is equally valid
- If *M* is prime, X_i is never 0
- $M = 11,\ A = 7,\ X_0 = 1$ ➔ $\underbrace{7,5,2,3,10,4,6,9,8,1,}\underbrace{7,5,2,3,10,\ldots}$

- The sequence repeats after *M-1=10* numbers
- The length of a sequence until a number is repeated is called a *period*
- A *full period* can be obtained if *A* and *M* are carefully chosen
  ➔ *full-period linear congruential generator*
- If *M* is prime, several choices of *A* give a full period, *M-1*
- Suppose *M=11, A=5* ➔ $\underbrace{5,3,4,9,1,}5,3,4,9,1,\ 5,3,4,\ldots$
- If $M = 2^{31} - 1 = 2{,}147{,}483{,}647$ which is a prime number, *(1)* gives a large cycle
  - *A=48,271* is one of the many good choices

- In 32-bit machine multiplication of M will result overflow
- How to fix this?

Let $M=AQ+R$ (2)

Then, $X_{i+1} = AX_i (\bmod M) = AX_i - M\left\lfloor \frac{AX_i}{M} \right\rfloor = AX_i - M\left\lfloor \frac{X_i}{Q} \right\rfloor + M\left\lfloor \frac{X_i}{Q} \right\rfloor - M\left\lfloor \frac{AX_i}{M} \right\rfloor$

$$= AX_i - M\left\lfloor \frac{X_i}{Q} \right\rfloor + M(\left\lfloor \frac{X_i}{Q} \right\rfloor - \left\lfloor \frac{AX_i}{M} \right\rfloor)$$

Since $X_i = \underbrace{Q\left\lfloor \frac{X_i}{Q} \right\rfloor}_{quotient} + \underbrace{X_i \bmod Q}_{remainder}$

$$X_{i+1} = A(Q\left\lfloor \frac{X_i}{Q} \right\rfloor + X_i \bmod Q) - M\left\lfloor \frac{X_i}{Q} \right\rfloor + M(\left\lfloor \frac{X_i}{Q} \right\rfloor - \left\lfloor \frac{AX_i}{M} \right\rfloor)$$

$$= (AQ - M)\left\lfloor \frac{X_i}{Q} \right\rfloor + A(X_i \bmod Q) + M(\left\lfloor \frac{X_i}{Q} \right\rfloor - \left\lfloor \frac{AX_i}{M} \right\rfloor)$$

Since by eqn (2) $AQ - M = -R$,

$$X_{i+1} = A(X_i \bmod Q) - R\left\lfloor \frac{X_i}{Q} \right\rfloor + M(\left\lfloor \frac{X_i}{Q} \right\rfloor - \left\lfloor \frac{AX_i}{M} \right\rfloor)$$

The term $\delta(X_i) = \left\lfloor \frac{X_i}{Q} \right\rfloor - \left\lfloor \frac{AX_i}{M} \right\rfloor$ is either 0 or 1, because both terms are integers and their difference lies between 0 and 1, thus $X_{i+1} = A(X_i \bmod Q) - R\left\lfloor \frac{X_i}{Q} \right\rfloor + M\delta(X_i)$


```
// Random class interface
// CONSTRUCTION: with (a) no initializer or (b) an integer
//     that specifies the initial state of the generator
// ******************PUBLIC OPERATIONS*********************
//     Return a random number according to some distribution
// int randomInt( )                  --> Uniform, 1 to 2^31-1
// double randomReal( )              --> Uniform, 0..1
// int randomInt( int low, int high )  --> Uniform Low..High
// int poisson( double expectedVal )   --> Poisson
// double negExp( double expectedVal )  --> Negative exponential
class Random
{  public:
    Random( int initVal = 1 ) : state ( initVal ) { }
// Uniform distributions
    int randomInt( );          // 1 to 2^31 - 1
    double randomReal( );        // >0.0 to <1.0
    int randomInt( int low, int high );
// Non-uniform distribution
    int poisson( double expectedVal );
    double negExp( double expectedVal );

    private:
    int state;     static const int A;     static const int M;     static const int Q;     static const int R;
};
```

Interface for the ***Random*** class

Implementation of some of the ***Random*** class

```
const int Random::A = 48271;
const int Random::M = 2147483647;
const int Random::Q = M / A;
const int Random::R = M % A;

// randInt returns the next random number and updates state.
int Random::randomInt( )
{
    int tmpState = A * ( state % Q ) - R * ( state / Q );
    if( tmpState >= 0 )
        state = tmpState;
    else
        state = tmpState + M;
    return state;
}

// Return a random double in 0..1.
double Random::randomReal( )
{  return randomInt( ) / 2147483647.0;  }

// Return random int in low..high
int Random::randomInt( int low, int high )
{   return randomInt( ) % ( high - low + 1 ) + low;  }
```


Linear Congruential Generator Improved

- Many Libraries have generators based on the functions

$$X_{i+1} = (A\,X_i + C)(\bmod 2^B)$$

where B is the machine word size and C is odd.

- Consider a sequence generated by equation

$$X_{i+1} = (A\,X_i + C)(\bmod M)$$

  - if $C=0$, then the method is called a ***multiplicative congruential technique***
  - if $C \neq 0$, then the method is called a ***mixed congruential technique***
- The selection of values for A, C, M, X_0 has a dramatic effect on the length of the period of the generated sequence
- Example: Let $A=2, C=3, M=10, X_0=0$, then

$x_0 = 0$

$x_1 = (2\times 0+3) \bmod 10 = 3$	$x_5 = (2\times 5+3) \bmod 10 = 3$
$x_2 = (2\times 3+3) \bmod 10 = 9$	$x_6 = (2\times 3+3) \bmod 10 = 9$
$x_3 = (2\times 9+3) \bmod 10 = 1$	$x_7 = (2\times 9+3) \bmod 10 = 1$
$x_4 = (2\times 1+3) \bmod 10 = 5$	$x_8 = (2\times 1+3) \bmod 10 = 5$

Additive Congruential Generator

- Its seed is a sequence of n numbers $X_1, X_2, ..., X_n$
  - This sequence can be generated using some other technique
- Then, $X_j = (X_{j-1} + X_{j-n}) \bmod M$
- The main advantage is its speed
- It can yield periods greater than M
- The theoretical behavior of this technique is not well understood ➔ careful validation is necessary
- Example: Let M=10, and extend the sequence 1, 2, 4, 8, 6

$x_1 = 1$	$x_6 = (x_5 + x_1) \bmod 10 = (6+1) \bmod 10 = 7$
$x_2 = 2$	$x_7 = (x_6 + x_2) \bmod 10 = (7+2) \bmod 10 = 9$
$x_3 = 4$	$x_8 = (x_7 + x_3) \bmod 10 = (9+4) \bmod 10 = 3$
$x_4 = 8$	$x_9 = (x_8 + x_4) \bmod 10 = (3+8) \bmod 10 = 1$
$x_5 = 6$	$x_{10} = (x_9 + x_5) \bmod 10 = (1+6) \bmod 10 = 7$
	$x_{11} = (x_{10} + x_6) \bmod 10 = (7+7) \bmod 10 = 4$
	$x_{12} = (x_{11} + x_7) \bmod 10 = (4+9) \bmod 10 = 3$
	$x_{13} = (x_{12} + x_8) \bmod 10 = (3+3) \bmod 10 = 6$
	$x_{14} = (x_{13} + x_9) \bmod 10 = (6+1) \bmod 10 = 7$

Quadratic Congruential Generator

- Is used when M is a power of 2
- It has a longer period than Mid-Square
- The equation is

$$X_{i+1} = (X_i(X_i + 1)) \bmod M$$

- The seed X_0 must satisfy the relation X_0 *mod 4 = 2*
- Example: Let $M=16$ and $X_0=2$,

$$\begin{aligned}
x_0 &\quad = 2\\
x_1 &= (2(3)) \bmod 16 \quad = 6\\
x_2 &= (6(7)) \bmod 16 \quad = 10\\
x_3 &= (10(11)) \bmod 16 = 14\\
x_4 &= (14(15)) \bmod 16 = 2\\
x_5 &= (2(3)) \bmod 16 \quad = 6\\
x_6 &= (6(7)) \bmod 16 \quad = 10\\
x_7 &\quad = 14
\end{aligned}$$

Nonuniform Random Numbers

- Many events do not happen in uniform fashion
  - recall, in *uniform distribution* each number occurs with the same probability
- A uniform random number generator can be used to generate random numbers that satisfy other distributions.

Poisson Distribution

- Occurs in simulations: used as the arrival process in most models
- k= # of packets (customers) arrived, λ = average arrival rate

$$P(k) = \frac{\lambda^k}{k!} e^{-\lambda}, \quad k = 0,1,...$$

- The Poisson random number can be generated by
  - Starting with *n=0*, generate random number u_n, $0 \le u_n \le 1$, and compute $\prod_{i=0}^{n} u_i$, such that $\prod_{i=0}^{n} u_i \le e^{-\lambda} < \prod_{i=0}^{n-1} u_i$
  - Return *n* as the *Poisson(* λ *).*

Negative Exponential Distribution

- Interarrival times

$$P[t = x] = \lambda e^{-\lambda x},$$

- For Poisson process with mean $1/\lambda$, the interarrival time can be computed as

$$\delta t = -\frac{\ln(u)}{\lambda}.$$

Poisson distribution

```
// Return random number according to Poisson distribution.
int Random::poisson( double expectedValue )
{  double limit = -expectedValue;
   double sum = log( randomReal( ) );
   int count;
   for( count = 0; sum > limit; count++ )
       sum += log( randomReal( ) );
   return count;
}
```

Negative exponential distribution

```
// Return random number according to negative exponential distribution.
double Random::negExp( double expectedValue )
{       return - expectedValue * log( randomReal( ) ); }
```


using the *Random* class

```
int main( )
{
   Random r;
   const int SIZE = 15;
   vector<int> a( SIZE );
   int i;

   cout << "Random ints between " << SIZE << " and " << (SIZE + 5) << endl;
   for( i = 0; i < SIZE; i++ )
      cout << r.randomInt( SIZE, SIZE + 5 ) << endl;

   for( i = 0; i < SIZE; i++ )
      a[ i ] = 0;

   for( i = 0; i < 100000; i++ )
      a[ r.poisson( 1.6 ) ]++;

   cout << endl << "Poisson distribution with expected balue of 1.6." << endl;
   for( i = 0; i < a.size( ); i++ )
      cout << a[ i ] << endl;

   return 0;
}
```


Generating a Random Permutation

- How to generate a random permutation?

  The problem is to generate a random permutation of 1, 2, ..., N with all permutations being equally likely.

- Example: (1,8,4,6,2,3,7,5) is a random permutation of (1,2,3,4,5,6,7,8)
- One way: for *a[i]* take randomly a number from 1, 2, ..., N which is not equal to *a[j]* for all *j<i*. ➔ is not a good idea (?) (what is the time bound ?, improvements ?)
- A better way: Randomly swap elements in (1,2,...,N-1,N)
  - swap *a[i]* with *a[j]* where *j* is a random integer from *(0,i)*
  - A random permutation can be generated in linear time, using only one random number per item
  - Are all permutations equally likely? (Yes and No)

Random Permutation

```
// Generate a random permutation of 1..n.
void permute( vector<int> & a )
{
   Random r( (int) time( 0 ) ); // Random object; seed set by time
   const int n = a.size( );

   for( int i = 0; i < n; i++ )
      a[ i ] = i + 1;

   for( int j = 1; j < n; j++ )
   {
      // Swap with random object
      int pos = r.randomInt( 0, j );

      int tmp = a[ j ];
      a[ j ] = a[ pos ];
      a[ pos ] = tmp;
   }
}
```


Randomized Algorithms

- A randomized algorithm uses random numbers rather than deterministic decisions to control branching.
- The running time depends on the random numbers that occur, as well as the particular input.
- Randomized Quick_Select (Median) is statistically guaranteed to work in linear time.
  - Quick_Select we considered before had time bounds quadratic in worst case and linear in average case.
  - Try to make expected running time as an average case running time:
    1. make from your input a random input ➔ permute
    2. choose the pivot randomly instead of deterministically
    3. take a random item in the array and swap with the item in position low; take a random item in the array and swap with the item in position high; take a random item in the array and swap with the item in the middle position; then continue as usual.