Application of Random Numbers

- Simulation
  - Simulate natural phenomena
- Sampling
  - It is often impractical to examine all possible cases, but a random sample will provide insight into what constitutes typical behavior
- Decision making
  - "Many executives make their decisions by flipping a coin…"
- Recreation
Random Numbers in Cryptography

- The keystream in the one-time pad
- The secret key in the DES encryption
- The prime numbers p, q in the RSA encryption
- The private key in DSA
- The initialization vectors (IVs) used in ciphers

Environmental Sources of Randomness

- Radio frequency noise [http://www.random.org](http://www.random.org)
- Noise generated by a resistor or diode.
  - Canada [http://www.tundra.com/](http://www.tundra.com/) (find the data encryption section, then look under RBG1210. My device is an NM810 which is 2?8? RBG1210s on a PC card)
  - Sweden [http://www.protego.se](http://www.protego.se)
Environmental Sources of Randomness (continued)

- Inter-keyboard timings (watch out for buffering)
- Inter-interrupt timings (for some interrupts)

Combining Sources of Randomness

- Suppose $r_1, r_2, \ldots, r_k$ are random numbers from different sources. E.g.,
  - $r_1$ = from JPEG file
  - $r_2$ = sample of hip-hop music on radio
  - $r_3$ = clock on computer
  - $b = r_1 \oplus r_2 \oplus \cdots \oplus r_k$
- If any one of $r_1, r_2, \ldots, r_k$ is truly random, then so is $b$. 
Random Number Generators

• Based upon specific mathematical algorithms
• Which are repeatable and sequential

Random

• Truly Random
  – Exhibiting true randomness
• Pseudorandom
  – Appearance of randomness but having a specific repeatable pattern
• Quasi-random
  – Having a set of non-random numbers in a randomized order
Problems

• Difficult to isolate
  – Often need to replace current generator
  – Require
    • Knowledge of current generator
    • Sometimes in-depth understanding of random number generators themselves
• Large scale tests cause most problems
  – Needing sometimes millions or billions of random numbers

Desirable Properties

• When performing Monte Carlo Simulations
  – Attributes of each particle should be independent of those attributes of any other particle
  – Fill the entire attribute space in a manner which is consistent with the physics
Random Number Cycle

- **Basis**
  - sequence of pseudorandom integers
    - Some exceptions
- **Integers ("Fixed")**
  - Manipulated arithmetically to yield floating point ("real")
- Can be presented in either Integer or Real numbers

**Cycle**

![Figure 2: Illustration of Random Number Cycle](image)
What Does This Show Us?

• Properties of pseudorandom sequences of integers
  – 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

"Anyone who considers arithmetic methods of producing random digits is, of course, is in a state of sin."

--John von Neumann
Pseudorandom Numbers

• Contrary to what we may think, clustering of data is entirely natural. Requiring some minimal spacing will make numbers less random.

Pseudorandom Numbers (continued)

• A sequence of numbers looks random if:
  1. the probability of x appearing is the same as any other number y
  2. the numbers are independent; e.g., 2 will not always be followed by 7.
• Condition (1) is easy. Condition (2) is never met.
Von Neumann's (Flawed) Method

• Square the number and clip out the middle digits:
  – $1234^2 = 01522756 \rightarrow 5227$
  – $5227^2 = 27321529 \rightarrow 3215$
  – $3215^2 = 10336225 \rightarrow 3362$
  – $3362^2 = 11303044 \rightarrow 3030$

Von Neumann's Method (continued)

• 50 trials later:
  – $4003^2 = 16024009 \rightarrow 0240$
  – $0240^2 = 00057600 \rightarrow 0576$
  – $0576^2 = 00331776 \rightarrow 3317$
  – $3317^2 = 11002489 \rightarrow 0024$
  – $0024^2 = 00000576 \rightarrow 0005$
  – $0005^2 = 00000025 \rightarrow 0000$
  – $0000^2 = 00000000 \rightarrow 0000$
Von Neumann's Method (continued)

- Choosing a starting value becomes *extremely* important.
- With a starting value of 1490, the sequences produces (after 15 cycles) 2100, 4100, 8100, 6100, 2100, ...
- Most middle square generators have short cycles.

Lehmer's Method

- Also known as the Linear Congruential Method is a method of choice.
- It uses three integer constants:
  - $a$, the multiplier
  - $m$, the modulus
  - $c$, the increment (sometimes set to 0)
- We generate the next number:
  \[ x_{n+1} = (ax_n + c) \mod m \]
Linear Congruential Method

- We can rewrite
  \[ x_{n+1} = (ax_n + c) \mod m \]
as a linear congruence. It can only be true if
\[ ax_n + c = qm + x_{n+1}, \text{ where } q \text{ is an integer} \]

First try – `rand()`

```cpp
// rand() - Random Number Generator
// First try
void rand(int &x) {
    // Or some other suitable values
    const int m = 32;
    const int a = 25;
    const int c = 7;

    x = (x*a + c) % m;
}
```
• The random number from one run becomes the seed for the next run.
• Procedures like randomize() use the clock and calendar to produce a seed based on data and time – far more likely to be unique.
• The success of this method is entirely dependent on finding the right values of $m$, $a$ and $c$.

• The period is at most $m$. If we pick the wrong $a$, the period may be less than $m$.
• Knuth points out that $a = c = 1$ will produce a sequence with a period of $m$ which is anything but random.
• Knuth gives the following conditions for a period of $m$:
  – $c$ must be relatively prime to $m$
  – $(a-1)$ must be divisible by every prime factor of $m$
    E.g., if $m$ is a multiple of 4, $(a-1)$ must also be a multiple of 4.

• Park and Miller suggest:
  – $m = 2^{31} - 1 = 2, 147, 483, 647$
  – $a = 16, 807$ (or 48, 271)
  – $c = 0$
Implementation

• Since the multiplier and intermediate results are large, we need an oversized data type (such as `long int`) and we have to recognize that with overflow, our result may be negative.

• We need to be able to save the seed between calls. A `static` type as in C or FORTRAN should be used to store the seed.

• For a portable generator to avoid overflows, there must be a way to save intermediate results within the `int` data type.

Implementation (continued)

• Schrage's method based on Park and Miller starts with two numbers p and q such that
  \[ p = m \ \text{div} \ a \quad \text{and} \quad q = m \ \text{mod} \ a \]

• If we choose a suitable m and a, we can guarantee that q < p.
Proof Of Our Computation

\[ x_{n+1} = ax_n \mod m \]
\[ = ax_n - m(ax_n / m) \]

*Calculating mod on some systems*

Let \( p = m / a \) & \( q = m \% a \) \( \Rightarrow \) \( m = ap + q \)

---

Proof Of Our Computation (continued)

\[ x_{n+1} = a (x_n \% p) = q(x_n / p) \]
\[ + m[(x_n/p) - ax_n / m] \]

*We can prove this by*

\[ a (x_n \% p) - q(x_n / p) = ax_n - ap(x_n/p) - q(x_n / p) \]
\[ = ax_n - (ap+q) (x_n / p) \]
\[ = ax_n - m (x_n / p) \]
Proof Of Our Computation (continued)

Substitution gives:

\[ x_{n+1} = ax_n - m(x_n / p) + m(x_n / p) - m(ax_n / m) \]

\[ x_{n+1} = f(x_n) + mg(x_n) \]

where \( f(x_n) = a(x_n \% p) - q(x_n / p) \)
\[ g(x_n) = (x_n / p) - (ax_n / m) \]

\( f \) and \( g \) cannot overflow!!

```
parkm.cc

#include <iostream>

using namespace std;

long p, q; // Two values that we will need

// Initialized the random number generator's values
void randinit(void);

// The random number generator
void rand(int &x);
```
int main(void) {
    // 91331 is a large prime
    int i, x = 91331;

    // Initialize the random number generator
    randinit();

    // Start calculating and printing random numbers
    cout << "x  = " << x << endl;
    for (i = 0; i < 100; i++) {
        rand(x);
        cout << "x  = " << x << endl;
    }
    return(0);
}

const long m = 65536L*65536L-1L; // 2^31-1
const long a = 18397L; // A large prime number

// randinit() - Initialize p snd q
void randinit(void) {
    p = m / a;
    q = m % a;
}
// rand() - We do the calculation in stages to avoid overflow

void rand(int &x) {
    long d, e, f;
    
    d = x / p;
    e = x % q;
    f = a * e - q * d;
    if (f > 0)
        x = f;
    else
        x = f + m;
}

---

Lattice Problem

- If m = 32, c = 7 and a = 25, we will get:
  - 7, 22, 13, 12, 19, 2, 25, 24, 31, 14, 5, 4, 11, 26, 17, 16, 23, 6, 29, 28, 3, 18, 9, 8, 15, 30, 21, 20, 27, 10, 11, 0, 7
- While it may LOOK random, it REALLY isn't.
- This becomes apparent when you graph $x_{n+1}$ vs $x_n$
frand()

// frand() - A random floating point generator
float frand(int &x) {
    long d, e, f;

    d = x / p;
    e = x % q;
    f = a * e - q * d;
    if (f > 0)
        x = f;
    else
        x = f + m;
    return ((float)x / m);