Performance Modeling of Computer Systems and Networks

Prof. Vittoria de Nitto Personè

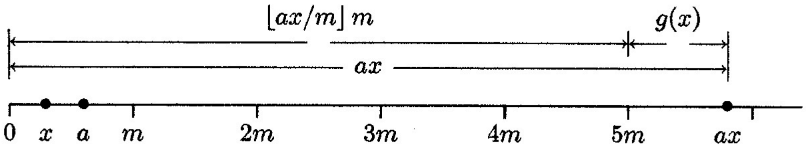
Lehmer Generators
Implementation

Università degli studi di Roma Tor Vergata
Department of Civil Engineering and Computer Science Engineering

1

Overflow Is Possible

- Recall that $g(x) = ax \bmod m$
- The *ax* product can be as big as *a*(*m*-1)



2

- If integers > m cannot be represented, integer overflow is possible!
- Not possible to evaluate g(x) in "obvious" way

Example 1: *m* decomposition

- consider $(a, m)=(48271, 2^{31}-1)$

  $q=\lfloor m/a \rfloor=44488$ $\quad r=m \bmod a=3399$ $\quad < 44488 = q$

- consider $(a, m)=(16807, 2^{31}-1)$

  $q=\lfloor m/a \rfloor=127773$ $\quad r=m \bmod a=2836$ $\quad < 127773 = q$

- In both cases $r < q$

  This characteristic is important!!
  (*modulus-compatibile*)

Rewriting g(x) to avoid overflow

$$
\begin{aligned}
g(x) &= ax \bmod m \\
&= ax - m\lfloor ax/m \rfloor \\
&= ax + [- m\lfloor x/q \rfloor + m\lfloor x/q \rfloor] - m\lfloor ax/m \rfloor \\
&= [ax - (aq+r)\lfloor x/q \rfloor] + [\, m\lfloor x/q \rfloor - m\lfloor ax/m \rfloor \,] \\
&= [a(x - q\lfloor x/q \rfloor) - r\lfloor x/q \rfloor\,] + [m\lfloor x/q \rfloor - m\lfloor ax/m \rfloor\,] \\
&= [a(x \bmod q) - r\lfloor x/q \rfloor\,] + m\,[\lfloor x/q \rfloor - \lfloor ax/m \rfloor\,] \\
&= \gamma(x) + m\,\delta(x)
\end{aligned}
$$

where

$\gamma(x) = a(x \bmod q) - r\lfloor x/q \rfloor$ and
$\delta(x) = \lfloor x/q \rfloor - \lfloor ax/m \rfloor$

Note: mods are done before multiplications!!!

Characterization of $\delta(x)$

$g(x) = \gamma(x) + m\,\delta(x)$

Theorem 2.2.1

If $m = aq+r$ is prime and $r < q$, for $x \in \chi_m$

$$\delta(x) = 0 \quad \text{or} \quad \delta(x) = 1$$

where

$$\delta(x) = \lfloor x/q \rfloor - \lfloor ax/m \rfloor$$

moreover

$$\delta(x) = 0 \ \text{ iff } \ \gamma(x) \in \chi_m$$
$$\delta(x) = 1 \ \text{ iff } \ -\gamma(x) \in \chi_m$$

where

$$\gamma(x) = a(x \bmod q) - r\lfloor x/q \rfloor$$

Computing g(x)

- evaluates $g(x) = ax \bmod m$ with no values > m-1

Algorithm 1

```
t = a * (x % q) - r * (x / q);        /* t = γ(x) */
if (t > 0)
        return (t);                   /* δ(x) = 0 */
else
        return (t + m);               /*  δ(x) = 1 */
```

- returns $g(x) = \gamma(x) + m\,\delta(x)$
- the *ax* product is "trapped" in $\delta(x)$
- no overflow !!

Modulus compatibility

- we must have r < q in $m = aq+r$
- multiplier *a* is *modulus-compatibile* (MC) with *m* iff r < q
- choose *a* MC with $m = 2^{31}-1$, then algorithm 1 can port to any 32-bit machine
- e.g.: a=48271 is MC with $m = 2^{31}-1$

  r = 3399 q = 44 488

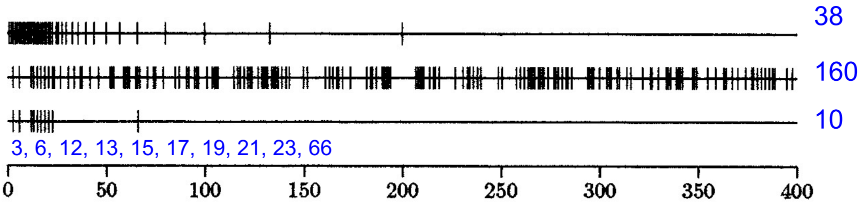
Modulus-Compatible MC and Full-Period FP

- no MC multipliers $> (m-1)/2$

- more densely distributed on low end [0, m-1]

- consider a tiny modulus $m = 401$:

(row 1: MC; row 2: FP; row 3: MC & FP)



Multipliers *a* between 1 and 400

MC and smallness

- multiplier *a* is "small" iff $a^2 < m$

- if *a* is small, then *a* is MC

  all multipliers from 1 to $\lfloor \sqrt{m} \rfloor = 46340$ *are MC*

- if *a* is MC, *a* is not necessarily small

  a=48271 is MC with $2^{31}-1$ *but is not small*

- start with a small (therefore MC) multiplier

  search until the first FP multiplier is found

Example: FPMC multipliers for m= 2^{31}-1

- For $m=2^{31}-1$ and FPMC a=7, there are 23093 FPMC multipliers

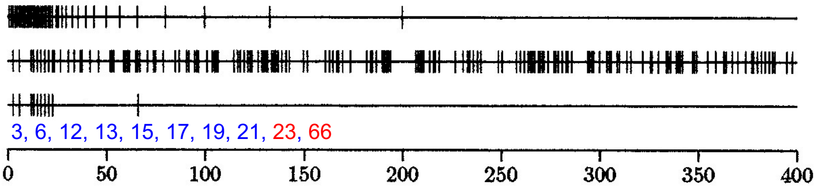
7^1 mod 2147483647 = 7
7^5 mod 2147483647 = 16807
7^{113039} mod 2147483647 = 41214
7^{188509} mod 2147483647 = 25697
7^{536035} mod 2147483647 = 63295
.
.

- a= 16807 is a “minimal” standard
- a= 48271 provides (slightly) more random sequences

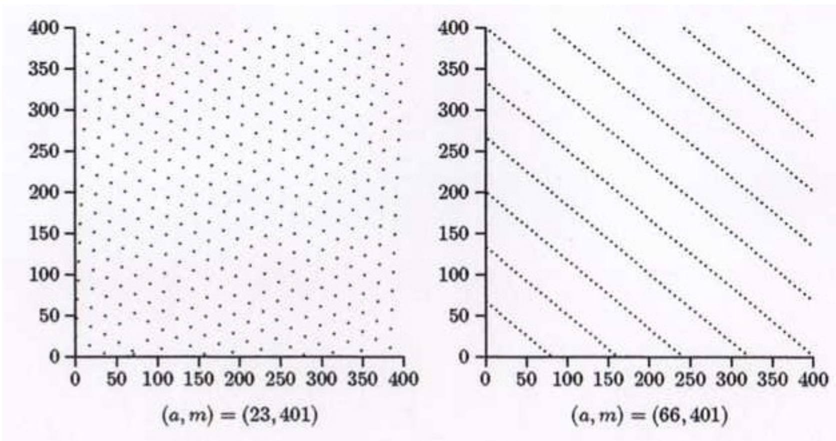
Randomness

- choose the FPMC multiplier that gives “most random” sequences
- no universal definition of randomness
- in 2-space $(x_0, x_1), (x_1, x_2), (x_2, x_3),\ldots$ form a lattice structure

- the first row shows 38 multipliers MC
- the second row shows 160 multipliers FP
- the third row shows 10 multipliers MC and FP



multipliers *a* between 1 and 400



Lehmer generator implementation with $(a,m) = (48271, 2^{31} - 1)$

```
Random(void) {
    static long state = 1;
    const long A = 48271;          /* multiplier*/
    const long M = 2147483647;     /* modulus */
    const long Q = M / A;          /* quotient */
    const long R = M % A;          /* remainder */
    long t = A * (state % Q) - R * (state / Q);
    if (t > 0)
        state = t;
    else
        state = t + M;
    return ((double) state / M);
}
```


A Not-As-Good RNG Library

- ANSI C library <stdlib.h> provides the function rand()
- simulates drawing from 1, 2, … m-1 with $m \geq 2^{15} - 1$
- value returned is not normalized; typical to use
  u = (double) rand() / RAND_MAX;
- ANSI C standard does not specify algorithm details
- for scientific work, avoid using rand() !!!

http://www.cplusplus.com/reference/cstdlib/rand/

rand

<cstdlib>

```
int rand (void);
```

Generate random number

Returns a pseudo-random integral number in the range between 0 and RAND_MAX.

This number is generated by an algorithm that returns a sequence of apparently non-related numbers each time it is called. This algorithm uses a seed to generate the series, which should be initialized to some distinctive value using function srand.

RAND_MAX is a constant defined in <cstdlib>.

A typical way to generate trivial pseudo-random numbers in a determined range using rand is to use the modulo of the returned value by the range span and add the initial value of the range:

```
v1 = rand() % 100;         // v1 in the range 0 to 99
v2 = rand() % 100 + 1;     // v2 in the range 1 to 100
v3 = rand() % 30 + 1985;   // v3 in the range 1985-2014
```

Notice though that this modulo operation does not generate uniformly distributed random numbers in the span (since in most cases this operation makes lower numbers slightly more likely).

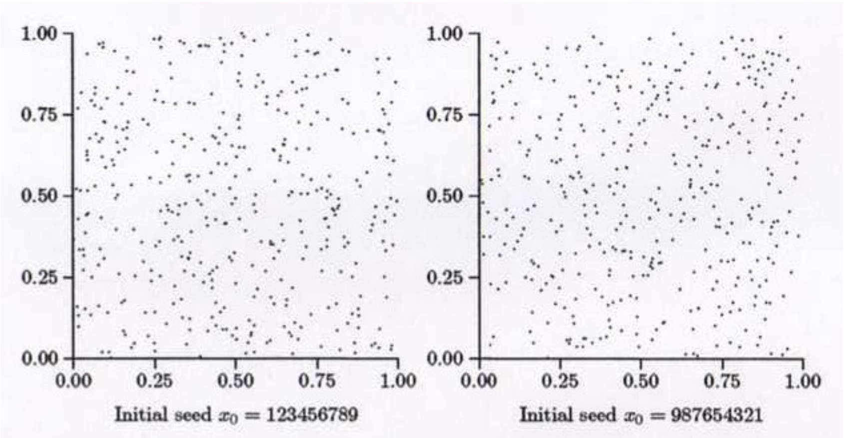
C++ supports a wide range of powerful tools to generate random and pseudo-random numbers (see <random> for more info).

- defined in the source files rng.h and rng.c
- based on the implementation considered here
  - double Random(void)
  - void PutSeed(long seed)
  - void GetSeed(long *seed)
  - void TestRandom(void)
- initial seed can be set directly, via prompt or by system clock
- PutSeed() and GetSeed() often used together
- *a*=48271 is the default multiplier

Example using the RNG

- generates 400 2-space points at random

```
seed = 123456789;            /* or 987654321 */
PutSeed(seed);
x0 = Random();
for (i = 0; i < 400; i++) {
        xi+1 = Random();
        Plot(xi, xi+1);        /* grafics function */
}
```

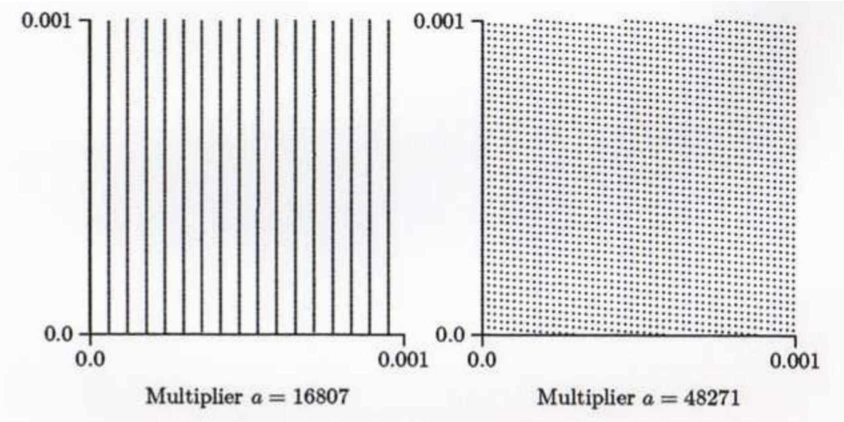
Observations on Randomness

- no lattice structure is evident
- appearance of randomness is an illusion
- if all $m - 1 = 2^{31} - 2$ points were generated, lattice would be evident
- herein lies distinction between *ideal* and *good* generator !!

Example

- plotting all pairs (x_i, x_{i+1}) for $m = 2^{31} - 1$ would give a black square
- any tiny square should appear approximately the same
- zoom in the square with opposite corners (0, 0) and (0.001, 0.001)

```
seed = 123456789;
PutSeed(seed);
x0 = Random();
for (i = 0; i < 2147483646; i++) {
        xi+1 = Random();
        if ((xi < 0.001) and (xi+1 < 0.001))
                Plot(xi, xi+1);
}
```



• further justification for using *a*=48271 over *a*=16807

considerations

• only 20 random numbers were needed
• seed x_0= 109.869.724
• resulting 20 random numbers

0.64 0.72 0.77 0.93 0.82 0.88 0.67 0.76 0.84 0.84
0.74 0.76 0.80 0.75 0.63 0.94 0.86 0.63 0.78 0.67

not discard outliers

→ Replicating simulation many times!!!!
So averaging the unusual cases