CS61B Lecture 35

Monday, April 20, 2020

Random Sequences

Why do we use random sequences? There are many uses for it:

- Choose statistical samples
- Simulations
- Random algorithms
- Cryptography
- And of course, games

What is a random sequence anyway? We have several possible definitions.

- A sequence where all numbers occur with equal frequency.
  1, 2, 3, 4, ...
- An unpredictable sequence where all numbers occur with equal frequency.
  0, 0, 0, 1, 1, 2, 2, 2, 2, 2, 3, 4, 4, 0, 1, 1, 1, ...
- Besides, what is wrong with 0, 0, 0, 0... anyway? Couldn't we get this by random selection?

If you statistics and probability spaces, you'll get a formal definition. As it turns out, even if it is definable, a "truly" random sequence is difficult for a computer or human to produce.

Pseudo-Random Sequences (PRS)

For most purposes, we only need a sequence that satisfies a certain statistical property, even if it is deterministic. We need a sequence that is hard, or impractical, to predict.

A pseudo-random sequence is a deterministic sequence that passes some given set of statistical tests. For example, look at lengths of runs increasing or decreasing contiguous subsequences. Unfortunately, the statistical criteria to be used are quite involved. For more details, see Knuth's book.

Generating PRSes

It is not as easy as you might think, because seemingly complex jumbling methods can give rise to bad sequences. The linear congruential method is a simple method used by Java:

$$X_0 = \text{arbitrary seed}$$
$$X_i = (aX_{i-1} + c)\%m, i > 0$$

Usually, `m` is a large power of 2. For the best results, we want

$$a \equiv 5\%8$$

and

$$a, c, m$$

without common factors.

This gives a generator with a period of `m` (the length of sequence before repetition), and reasonable potency (measures certain dependencies among adjacent x-values).

We want bits of `a` to have no obvious pattern and pass certain other tests (see Knuth).

Java uses

$$\begin{aligned} a &= 25214903917 \\ c &= 11 \\ m &= 2^{48} \end{aligned}$$

to compute 48-bit pseudo-random numbers, which is good enough for many purposes. But it is important to note it's not cryptographically secure.

What Can Go Wrong?

If we choose these numbers wrong, we can end up with a PRS that has short periods and having many impossible values (such as if `a`, `c`, `m` were all even).

Another potential shortfall is obvious patterns, such as just using the lower 3 bits of x-values in Java's 48-bit generator, to get integers in the range of 0 to 7. By the properties of modular arithmetic:

$$\begin{aligned} X_i \% 8 &= (25214903917 X_{i-1} + 11 \% 2^{48}) \% 8 \\ &= (5(X_{i-1} \% 8) + 3) \% 8 \end{aligned}$$

So we have a period 8 in this generator, sequences like:

$$0, 1, 3, 7, 1, 2, 7, 1, 4...$$

are impossible, which is why Java doesn't give you the raw 48 bits (and will usually take the top bits than last bits).

Bad potency leads to bad correlations: the infamous IBM generator RANDU used

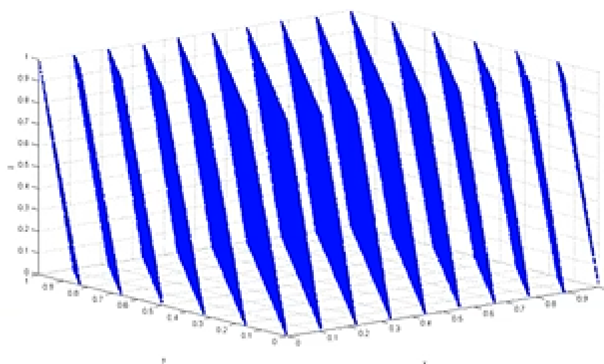
$$c = 0, a = 65539, m = 2^{31}$$

When RANDU is used to make 3D points:

$$(X_i / S, X_{i+1} / S, X_{i+2} / S)$$

where `S` scales to a unit cube.

The points would be arranged in parralel planes with voids between. So "random points" won't ever get many points in the cube:



Additive generators

$$X_n = \begin{cases} \text{arbitrary value} & , n < 55 \\ (X_{n-24} + X_{n-55}) \% 2^e & , n \geq 55 \end{cases}$$

Other choices than 24 and 55 are possible. This generator has a period of

$$2^f(2^{55}-1)$$

for some

$$f < e$$

Here's a simple implementation with a circular buffer:

```
i = (i + 1) % 55;
x[i] += x[(i + 31) % 55]; // Why +31 instead of -24?
return x[i]; // modulo 2^32
```

where

```
x[0 ... 54]
```

is initialized to some "random" initial seed values.

Cryptographic PRN Generators

The simple form of linear congruential generators mean that one can predict future values after seeing relatively few outputs, not good if we want unpredictable output. A cryptographic PRN generator (also called CPRNG, not PRS because S indicates there is a pattern) has the properties that:

- Given `k` bits of a sequence, no polynomial-time algorithm can guess the next-bit with better than 50% accuracy.
- Given the current state of the generator, it is also infeasible to reconstruct the bits it generated in getting to that state.

To do this, we start with a good block cipher -- an encryption algorithm that encrypts blocks of `N` bits (and not like Enigma, which works 1 byte at a time). AES is an example.

As a seed, provide a key, `K`, and an initialization value `I`. The `j`-th pseudo-random number is now

$$E(K, I + j)$$

where

$$E(x, y)$$

is now the encryption of message `y` using key `x`.

Adjusting Range and Distribution

Given a raw sequence of numbers from above methods in range, for example, 0 to 2^48, how can we get uniform random integers in the range 0 to `n`-1?

If `n` is a power of 2, it's easy, we simply use the top `k` bits of the next x-value (the bottom k-bits are not as random).

For other `n` values, we must be careful of slight biases at the ends. For example, if we compute:

$$\frac{X_i}{\frac{2^{48}}{n}}$$

using all integer division, and if

$$\frac{2^{48}}{n}$$

is rounded down, then you get `n` as a result.

If you try to fix that by computing

$$\frac{2^{48}}{n-1}$$

instead, the probability of getting `n`-1 will be wrong.

To fix the bias problems where `n` does not evenly divide 2^48, Java throws out values after the largest multiple of `n` less than 2^48:

```
int nextInt(int n) {
    long X = next random long; // (0 <= X <= 2^48)
    if ( n is 2^k for some k ) {
        return top k bits of X;
    }
    int MAX = largest multiple of N < 2^48
    while (Xi >= MAX) {
        X = next random long;
    }
    return Xi / (MAX / n);
}
```

Arbitrary Bounds

How to get an arbitrary range of integers (L to U)?

To get random `float`, x, in range between 0 inclusive and d exclusive, compute

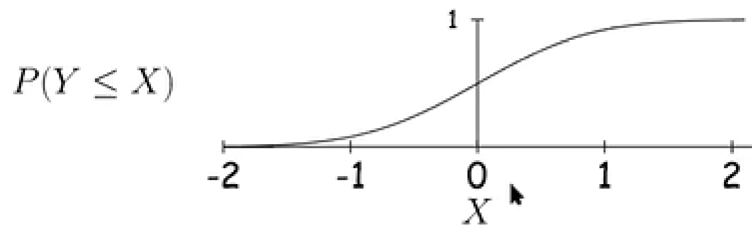
```
d*nextInt(1<<24) / (1<<24);
```

Random `double` is a bit more complicated, because we need two integers to get enough bits.

```
long bigRand = ((long) nextInt(1<<26) << 27) + (long) nextInt(1<<27);
return d * bigRand / (1L << 53);
```

General Distribution

Our distribution between two finite bounds is a distribution, but what if we wanted to achieve a different distribution? Say the normal distribution?



Curve is the desired probability distribution.

$$P(Y \leq X)$$

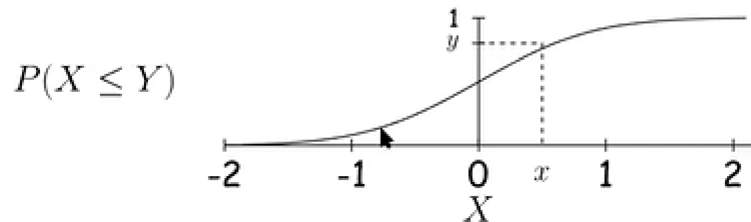
is the probability that random variable

$$Y$$

is

$$\leq X$$

The solution is choose `y` uniformly between 0 and 1, and the corresponding `x` will be distributed according to `P`.



Java Classes

Here are a few useful tools in Java to do probability distribution:

- `Math.random()` gives a random double in `[0...1)`.
- Class `java.util.Random` is a random number generator with constructors:
    - `Random()` generator with a "random" seed based on time.
    - `Random(seed)` generator with a given starting value (reproducible).
- Methods:
    - `next(k)` gives a `k`-bit random integer.
    - `nextInt(n)` gives an `int` in range `[0...n)`.
    - `nextLong()` gives a random 64-bit integer.
    - `nextBoolean()`, `nextFloat()`, `nextDouble()` give the next random values of other primitive types.
    - `nextGaussian()` gives a normal distribution with mean 0 and STD 1 to produce a bell curve.
- `Collections.shuffle(L,R)` for list `L` and `Random R` permutes `L` randomly, using `R`.

Shuffling

A shuffle is a random permutation of some sequence, and is an obvious, dumb technique for sorting an `N`-element list:

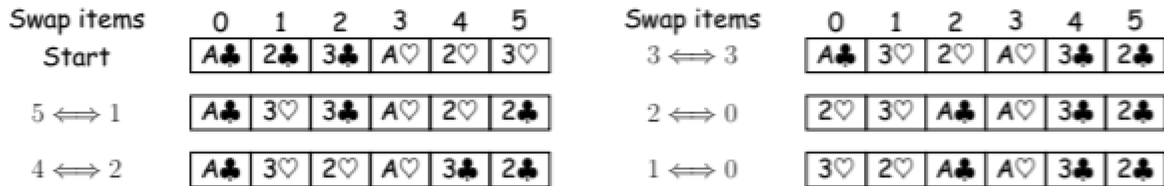
- Generate `N` random numbers.
- Attach each to one of the list elements.

- Sort the list using random numbers as keys.

We can do quite a bit better:

```
void shuffle(List L, Random R) {
    for (int i = L.size(); i > 0; i--) {
        L.get(i-1).swap(L.get(R.nextInt(i)));
    }
}
```

- Example:



Random Selection

Same technique allows us to select `N` items from a list:

```
/** Permute L and return sublist of K>=0 randomly
 * chosen elements of L, using R as random source. */
List select(List L, int k, Random R) {
    for (int i = L.size(); i+k > L.size(); i -= 1)
        L.get(i-1).swap(L.get(R.nextInt(i)));
    return L.sublist(L.size()-k, L.size());
}
```

It's not terribly efficient for selecting random sequence of `K` distinct integers from `[0...N)` with

$$K << N$$

Alternative Selection Algorithm (Floyd)

Here is another approach:

```
/** Random sequence of K distinct integers
* from 0...N-1, 0<=K<=N. */
IntList selectInts(int N, int K, Random R) {
    IntList S = new IntList();
    for (int i = N - K; i < N; i += 1) {
        // All values in S are < i
        int s = R.randInt(i+1); // 0 <= s <= i < N
        if (s == S.get(j) for some j) {
            // Insert value i (which can't be there
            // yet) after the s (i.e., at a random
            // place other than the front)
            S.add(j+1, i);
        } else {
            // Insert random value s at front
            S.add(0, s);
        }
    return S;
```

```
}
```

Example

i	s	S
5	4	[4]
6	2	[2, 4]
7	5	[5, 2, 4]
8	5	[5, 8, 2, 4]
9	4	[5, 8, 2, 4, 9]

`selectRandomIntegers(10, 5, R)`