Today:
- Pseudo-random Numbers (Chapter 11)
- What use are random sequences?
- What are “random sequences“?
- Pseudo-random sequences.
- How to get one.
- Relevant Java library classes and methods.
- Random permutations.
Why Random Sequences?

• Choose statistical samples
• Simulations
• Random algorithms
• Cryptography:
  - Choosing random keys
  - Generating streams of random bits (e.g., SSL xor's your data with a regeneratable, pseudo-random bit stream that only you and the recipient can generate).
• And, of course, games
What Is a “Random Sequence”?

• How about: “a sequence where all numbers occur with equal frequency”?
  - Like 1, 2, 3, 4, ...?

• Well then, how about: “an unpredictable sequence where all numbers occur with equal frequency?”
  - Like 0, 0, 0, 1, 1, 2, 2, 2, 2, 3, 4, 4, 0, 1, 1, 1, ...?

• Besides, what is wrong with 0, 0, 0, 0, ... anyway? Can’t that occur by random selection?
Pseudo-Random Sequences

- Even if definable, a “truly” random sequence is difficult for a computer (or human) to produce.

- For most purposes, need only a sequence that satisfies certain statistical properties, even if deterministic.

- Sometimes (e.g., cryptography) need sequence that is hard or impractical to predict.

- Pseudo-random sequence: deterministic sequence that passes some given set of statistical tests.

- For example, look at lengths of runs: increasing or decreasing contiguous subsequences.

- Unfortunately, statistical criteria to be used are quite involved. For details, see Knuth.
Generating Pseudo-Random Sequences

• Not as easy as you might think.

• Seemingly complex jumbling methods can give rise to bad sequences.

• **Linear congruential method** is a simple method that has withstood test of time:

\[
X_0 = \text{arbitrary seed}
\]

\[
X_i = (aX_{i-1} + c) \mod m, \quad i > 0
\]

• Usually, \( m \) is large power of 2.

• For best results, want \( a \equiv 5 \mod 8 \), and \( a, c, m \) with no common factors.

• This gives generator with a **period of** \( m \) (length of sequence before repetition), and reasonable **potency** (measures certain dependencies among adjacent \( X_i \).)

• Also want bits of \( a \) to “have no obvious pattern” and pass certain other tests (see Knuth).

• **Java uses** \( a = 25214903917 \), \( c = 11 \), \( m = 2^{48} \), to compute 48-bit pseudo-random numbers but I haven’t checked to see how good this is.
What Can Go Wrong?

• Short periods, many impossible values: E.g., \( a, c, m \) even.

• Obvious patterns. E.g., just using lower 3 bits of \( X_i \) in Java’s 48-bit generator, to get integers in range 0 to 7. By properties of modular arithmetic,

\[
X_i \mod 8 = (25214903917X_{i-1} + 11 \mod 2^{48}) \mod 8 \\
= (5(X_{i-1} \mod 8) + 3) \mod 8
\]

so we have a period of 8 on this generator; sequences like

\[0, 1, 3, 7, 1, 2, 7, 1, 4, \ldots\]

are impossible. This is why Java doesn't give you the raw 48 bits.

• Bad potency leads to bad correlations.

- E.g. Take \( c = 0, a = 65539, m = 2^{31} \), and make 3D points:

\[(X_i/S, X_{i+1}/S, X_{i+2}/S)\]

where \( S \) scales to a unit cube.

- Points will be arranged in parallel planes with voids between.

- So, “random points” won’t ever get near many points in the cube.
Other Generators

• Additive generator:

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

• Other choices than 24 and 55 possible.

• This one has period of \( 2^f(2^{55} - 1) \), for some \( f < e \).

• Simple implementation with circular buffer:

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

• where \( X[0 \ldots 54] \) is initialized to some “random” initial seed values.
Adjusting Range and Distribution

• Given raw sequence of numbers, $X_i$, from above methods in range (e.g.) 0 to $2^{48}$, how to get uniform random integers in range 0 to $n - 1$?

• If $n = 2^k$, is easy: use top $k$ bits of next $X_i$ (bottom $k$ bits not as “random”)

• For other $n$, be careful of slight biases at the ends. For example, if we compute $X_i/(2^{48}/n)$ using all integer division, and if $(2^{48}/n)$ doesn’t come out even, then you can get $n$ as a result (which you don’t want).

• Easy enough to fix with floating point, but can also do with integers; one method (used by Java for type int):

```c
/** Random integer in the range 0 .. n-1, n>0. */
int nextInt (int n) {
    long X = next random long (0 \leq X < 2^{48});
    if (n is 2^k for some k) return top k bits of X;
    int MAX = largest multiple of n that is < 2^{48};
    while (X_i \geq MAX) X = next random long (0 \leq X < 2^{48});
    return X_i / (MAX/n);
}
```
Arbitrary Bounds

• How to get arbitrary range of integers ($L$ to $U$)?
• To get random float, $x$ in range $0 \leq x < d$, compute

  return $d \times \text{nextInt}(1<<24) / (1<<24)$;

• Random double a bit more complicated: need two integers to get enough bits.

  long bigRand = ((long) nextInt(1<<26) << 27) + (long) nextInt(1<<27);
  return $d \times \text{bigRand} / (1L << 53)$;
Other Distributions

• Can also turn uniform random integers into arbitrary other distributions, like the Gaussian.

![Graph]

\[ P(x) \]

-2 -1 0 1 2

• Curve is the desired probability distribution (\( P(x) \) is the probability that a certain random variable is \( \leq x \).)

• Choose \( y \) uniformly between 0 and 1, and the corresponding \( x \) will be distributed according to \( P \).
Computing Arbitrary Discrete Distribution

• Example from book: want integer values $X_i$ with $\Pr(X_i = 0) = 1/12$, $\Pr(X_i = 1) = 1/2$, $\Pr(X_i = 2) = 1/3$, $\Pr(X_i = 3) = 1/12$:

$$
\begin{align*}
\text{Legend:} \\
0: & \quad \text{top}, \\
1: & \quad \text{top}, \\
2: & \quad \text{top}, \\
3: & \quad \text{bot}.
\end{align*}
$$

0 \quad 1 \quad 2 \quad 3

• To get desired probabilities, choose floating-point number, $0 \leq R_i < 4$, and see what color you land on.

• ≤ 2 colors in each beaker ≡ ≤ 2 colors between $i$ and $i + 1$.

```
return (R_i \% 1.0 > v[(int) R_i]) \\
\quad ? \text{top[(int) R_i]} \\
\quad : \text{bot[R_i];}
```

where

$$
\begin{align*}
\text{v} & = \{ 1.0/3.0, 2.0/3.0, 0, 1.0/3.0 \}; \\
\text{top} & = \{ 1, 2, 2, 1 \}, \\
\text{bot} & = \{ 0, 1, \text{/** ANY */0, 3} \};
\end{align*}
$$
Java Classes

- **Math.random()**: random double in \([0..1]\).

- **Class java.util.Random**: a random number generator with constructors:
  
  - `Random()` generator with “random” seed (based on time).
  - `Random(seed)` generator with given starting value (reproducible).

- **Methods**

  - `next(k)` \(k\)-bit random integer
  - `nextInt(n)` int in range \([0..n]\).
  - `nextLong()` random 64-bit integer.
  - `nextBoolean()`, `nextFloat()`, `nextDouble()` Next random values of other primitive types.
  - `nextGaussian()` normal distribution with mean 0 and standard deviation 1 (“bell curve”).

- **Collections.shuffle(L, R)** for list \(R\) and Random \(R\) permutes \(L\) randomly (using \(R\)).
# Shuffling

- A shuffle is a random permutation of some sequence.
- Obvious dumb technique for sorting \( N \)-element list:
  - Generate \( N \) random numbers
  - Attach each to one of the list elements
  - Sort the list using random numbers as keys.
- Can do quite a bit better:

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

- Example:

<table>
<thead>
<tr>
<th>Swap items</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>A♣</td>
<td>2♣</td>
<td>3♠</td>
<td>A♥</td>
<td>2♥</td>
<td>3♥</td>
</tr>
<tr>
<td>5 ⇨ 1</td>
<td>A♣</td>
<td>3♥</td>
<td>3♠</td>
<td>A♥</td>
<td>2♥</td>
<td>2♠</td>
</tr>
<tr>
<td>4 ⇨ 2</td>
<td>A♣</td>
<td>3♥</td>
<td>2♥</td>
<td>A♥</td>
<td>3♠</td>
<td>2♠</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Swap items</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 ⇨ 3</td>
<td>A♣</td>
<td>3♥</td>
<td>2♥</td>
<td>A♥</td>
<td>3♠</td>
<td>2♠</td>
</tr>
<tr>
<td>2 ⇨ 0</td>
<td>2♥</td>
<td>3♥</td>
<td>A♣</td>
<td>A♥</td>
<td>3♠</td>
<td>2♠</td>
</tr>
<tr>
<td>1 ⇨ 0</td>
<td>3♥</td>
<td>2♥</td>
<td>A♣</td>
<td>A♥</td>
<td>3♠</td>
<td>2♠</td>
</tr>
</tbody>
</table>
Random Selection

• Same technique would allow us to select $N$ items from list:

```java
/** Permute L and return sublist of $K \geq 0$ 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)
        swap element i-1 of L with element
        R.nextInt (i) of L;
    return L.sublist (L.size ()-k, L.size ());
}
```

• Not terribly efficient for selecting random sequence of $K$ distinct integers from $[0..N)$, with $K \ll N$. 
Alternative Selection Algorithm (Floyd)

/** 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

<table>
<thead>
<tr>
<th>i</th>
<th>s</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4</td>
<td>[4]</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>[2, 4]</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>[5, 2, 4]</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>[5, 8, 2, 4]</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>[5, 8, 2, 4, 9]</td>
</tr>
</tbody>
</table>

selectRandomIntegers (10, 5, R)