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

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.

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.

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?
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 \( X \), 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:
    \[
    \left( \frac{X_i}{S}, \frac{X_{i+1}}{S}, \frac{X_{i+2}}{S} \right), \text{where } S \text{ scales to a unit cube.}
    \]
  - 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 \( 2^f(2^{35} - 1) \), for some \( f < e \).

  • Simple implementation with circular buffer:
    \[
    i = (i+1) \mod 55; \quad X[i] += X[(i+31) \mod 55]; \\
    \text{// Why +31 (55-24) instead of -24?} \quad \text{return } X[i]; \quad \text{/* modulo } 2^{32} */
    \]
  
  • Where \( X[0 .. 54] \) 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):

  ```java
  /** 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 \leq 2^{48};
    while (X, >= MAX) X = next random long (0 \leq X < 2^{48});
    return X_i / (MAX/n);
  }
  ```
Arbitrary Bounds

- How to get arbitrary range of integers \((L, U)\)?
- To get random float, \(x\) in range \(0 \leq x < d\), compute
  \[
  \text{return } d \times \text{nextInt}(1 << 24) / (1 << 24);
  \]
- Random double a bit more complicated: need two integers to get enough bits.
  \[
  \text{long bigRand = ((long) nextInt(1<<26) << 27) + (long) nextInt(1<<27);}
  \]
  \[
  \text{return } d \times \text{bigRand} / (1L << 53);
  \]

Other Distributions

- Can also turn uniform random integers into arbitrary other distributions, like the Gaussian.
  \[
  P(x)
  \]
  \[
  \begin{array}{c}
    -2 \\
    -1 \\
    0 \\
    1 \\
    2
  \end{array}
  \]
- 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 \(Pr(X_i = 0) = 1/12\), \(Pr(X_i = 1) = 1/2\), \(Pr(X_i = 2) = 1/3\), \(Pr(X_i = 3) = 1/12\):

  \[
  \begin{array}{cccc}
    0 & 1 & 2 & 3 \\
    \hline
    \text{Legend:} & 0: & 1: & 2: & 3: \\
    & 0 & 1 & 2 & 3 & 4
  \end{array}
  \]
  \[
  \begin{array}{cccc}
    0 & 1 & 2 & 3 & 4 \\
    \hline
    \text{Legend:} & 0: & 1: & 2: & 3: \\
    & 0 & 1 & 2 & 3 & 4
  \end{array}
  \]

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

  \[
  \text{return } (R_i \% 1.0 > v[(\text{int}) R_i])
  \]
  \[
  \begin{array}{cccc}
    0 & 1 & 2 & 3 & 4 \\
    \hline
    \text{Legend:} & 0: & 1: & 2: & 3: \\
    & 0 & 1 & 2 & 3 & 4
  \end{array}
  \]

- \(\leq 2\) colors in each beaker \(\equiv\) \(\leq 2\) colors between \(i\) and \(i + 1\).

Java Classes

- Math.random(): random double in \([0..1)\).
- Class java.util.Random: a random number generator with constructors:
  \[
  \begin{array}{c}
    \text{Random()} \text{ generator with "random" seed (based on time).} \\
    \text{Random(}\text{seed}) \text{ generator with given starting value (reproducible).}
  \end{array}
  \]
- Methods
  \[
  \begin{array}{c}
    \text{nextInt(}k) k\text{-bit random integer} \\
    \text{nextInt(}n) \text{ int in range [0..n].} \\
    \text{nextLong()} \text{ random 64-bit integer.} \\
    \text{nextBoolean(), nextFloat(), nextDouble()} \text{ Next random values of other primitive types.} \\
    \text{nextGaussian()} \text{ normal distribution with mean 0 and standard deviation 1 ("bell curve").}
  \end{array}
  \]
- 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>A♥</td>
<td>3♣</td>
<td>2♥</td>
<td>5♣</td>
</tr>
<tr>
<td>5 &lt;=&gt; 1</td>
<td>A♥</td>
<td>3♣</td>
<td>A♥</td>
<td>2♥</td>
<td>2♥</td>
<td>5♣</td>
</tr>
<tr>
<td>4 &lt;=&gt; 2</td>
<td>A♣</td>
<td>3♣</td>
<td>2♥</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>=0 randomly */
  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)**

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