Random Numbers

- In many applications, want to be able to generate random numbers, permutations, etc.
  - Since computers are deterministic, true randomness does not exist
  - Settle for pseudo-randomness: sequence of numbers that looks random, but is deterministically generated
  - Most random number generators use a “linear congruential generator” (LCG):
    - Start with a seed number $X_0$
    - Next “random” number given by: $X_{n+1} = (aX_n + c) \mod m$
    - Effectiveness is very sensitive to choice of $a$, $c$, and $m$
    - Note: the sequence of random numbers must eventually cycle

The `rand()` Function

- In C/C++, there exists a function for generating pseudo-random numbers: `int rand(void)`
  - Part of `<stdlib.h>` library
  - Returns integer between 0 and `RAND_MAX`, inclusive
  - Values supposed to be “uniformly” distributed in range
    - `RAND_MAX` guaranteed to be at least 32767 (often larger)
  - In most implementations, uses LCG
  - Seeded using: `void srand(unsigned int seed)`
    - Often set to current system time: `srand(time(NULL));`
  - For our purposes, we accept approximation: 
    - $(\text{rand()}/((\text{double})\text{RAND_MAX} + 1)) \sim \text{Uni}[0, 1)$

The rand() Function

- In C/C++, there exists a function for generating pseudo-random numbers: `int rand(void)`
  - Part of `<stdlib.h>` library
  - Returns integer between 0 and `RAND_MAX`, inclusive
    - Values supposed to be “uniformly” distributed in range
      - `RAND_MAX` guaranteed to be at least 32767 (often larger)
    - In most implementations, uses LCG
  - Seeded using: `void srand(unsigned int seed)`
    - Often set to current system time: `srand(time(NULL));`
  - For our purposes, we accept approximation:
    - $(\text{rand()}/((\text{double})\text{RAND_MAX} + 1)) \sim \text{Uni}[0, 1)$

Shuffling Deck of Cards

- Want to generate a random permutation of set
  - E.g., completely shuffle a deck of cards
  - Want all permutations to be equally likely
  ```
  void shuffle(int arr[], int n) {
    for(int i = n - 1; i > 0; i--) {
      double u = uniformRand(0, 1);  // u in [0, 1)
      int pos = (int)((i + 1) * u);
      swap(arr[i], arr[pos]);
    }
  }
  ```
  - Has $n!$ execution paths, but only $n!$ permutations
  - Consider $n = 3$:
    - Can generate $3^3 = 27$ possible execution paths
    - But, only $3! = 6$ possible permutations
    - $27 / 6 = 4.5$ (not integer), so not all permutations equally likely!

Another Good Way to Shuffle

- Common (easy) way to shuffle
  ```
  void shuffle(int arr[], int n) {
    double *keys = new double[n];
    for(int i = 0; i < n; i++) {
      keys[i] = uniformRand(0, 1);  // u in [0, 1)
    }
    SortUsingKeys(arr, keys, n);
    delete[] keys;
  }
  ```
  - Pros: all permutations equally likely, easy to code
  - Cons: O(n log n) due to sort vs. O(n) for our first method

Bad, But Common, Shuffle

- Common mistake in creating random permutation
  ```
  void badShuffle(int arr[], int n) {
    for(int i = 0; i < n; i++) {
      double u = uniformRand(0, 1);  // u in [0, 1)
      int pos = (int)(n * u);
      swap(arr[i], arr[pos]);
    }
  }
  ```
  - Has $n^n$ execution paths, but only $n!$ permutations
  - Consider $n = 3$:
    - Can generate $3^3 = 27$ possible execution paths
    - But, only $3! = 6$ possible permutations
    - $27 / 6 = 4.5$ (not integer), so not all permutations equally likely!

Generating Distributions

- Given ability to generate numbers $\sim \text{Uni}(0, 1)$
  - How can we generate other distributions?
    - First method we consider is “Inverse Transform”
    - Want to simulate a continuous distribution function $F$
      - Let $U \sim \text{Uni}(0, 1)$
    - Define $X = F^{-1}(U)$ (inverse: $F^{-1}(a) = b \iff F(b) = a$)
      - Note: $P(X \leq x) = P(F^{-1}(U) \leq x) = P(U \leq F(x)) = F(x)$
      - Thus, $X$ will have distribution $F$
        - Can use method for discrete distributions with some modification
Continuous Inverse Transform

- Need to invert distribution function
  - Want to generate exponential distribution: \( X \sim \text{Exp}(\lambda) \)
  - CDF: \( F(X = x) = 1 - e^{-\lambda x} \) where \( x \geq 0 \)
  - To compute inverse, let \( F(X) = 1 - e^{-\lambda x} = u \), solve for \( x \):
    \[
    e^{-\lambda x} = 1 - u \Leftrightarrow -\lambda x = \log(1 - u) \Leftrightarrow x = -\log(1 - u)/\lambda.
    \]
  - So, \( F^{-1}(U = u) = x = -\log(1 - u)/\lambda \)
  - Since \( U \sim \text{Uni}(0, 1) \), also have \( (1 - U) \sim \text{Uni}(0, 1) \)
  - Simplify: \( X = F^{-1}(U) = -\log(U)/\lambda \)

  - Note: closed-form inverse may not always exist
    - Normal distribution doesn’t have closed-form inverse

Discrete Inverse Transform

- Recall form of CDF for discrete distribution:
  - Here is \( F(X = x) \) where \( X \sim \text{Poi}(3) \):
  - Generate \( U \sim \text{Uni}(0, 1) \) (e.g., \( u = 0.7 \))
  - As \( x \) increases, determine first \( F(x) \geq U \) (e.g., \( x = 4 \))
  - Return that value of \( x \)

Bring on the Code

- Discrete inverse transform
  - Assumes PMF, \( p(x) \), available for distribution modeled
    - E.g., Bernoulli, Binomial, Poisson, many others
  ```c
  int discreteInverseTransform() {
    double u = uniformRand(0, 1); // u in [0, 1)
    int x = 0;
    double F_so_far = p(x);
    while (F_so_far < u) {
      x++;
      F_so_far += p(x);
    }
    return x;
  }
  ```

Rejection Filtering

- Want to simulate random variable \( X \) with PDF \( f(x) \)
  - Assume we can simulate \( Y \) with PDF \( g(y) \) where \( Y \) has same range as \( X \)
  ```c
  double rejectionFilter() {
    while (true) {
      double u = uniformRand(0, 1); // u in [0, 1)
      double y = randomValueFromDistributionOfY();
      if (u <= f(y)/(c * g(y))) return y;
    }
  }
  ```
  - where constant \( c \geq f(y)/g(y) \) for all \( y \)
  - Number iterations of loop \( \sim \text{Geo}(1/c) \)
  - Proof of correctness in Ross, Chap. 10.2.2

Generating Normal Random Variable

- Want to simulate random variable \( Z \sim \text{N}(0, 1) \)
  - PDF for \( |Z| \):
    - \( f(z) = \frac{1}{\sqrt{2\pi}} e^{-z^2/2} \) where \(-\infty < z < \infty\)
  - Will simulate using \( Y \sim \text{Exp}(1) \) (from inverse transform)
    - PDF: \( g(y) = e^{-y} \) where \( 0 \leq y < \infty \)
    - \[
    \frac{f(x)}{g(x)} = \frac{\frac{1}{\sqrt{2\pi}} e^{-z^2/2}}{\frac{\lambda}{\sqrt{2\pi}} e^{-\lambda z^2/2}} = \frac{1}{\lambda} \frac{e^{-(x^2/2) - \lambda z^2/2}}{e^{-x^2/2}} \leq \frac{1}{\lambda} \leq c
    \]
    - So, we obtain: \( \frac{f(x)}{c \cdot g(x)} = e^{-(x^2/12)} \)

Applying Rejection Filtering

- Note: \( \frac{f(x)}{c \cdot g(x)} = e^{-(x^2/12)} \) \( c = \frac{\lambda}{\sqrt{2\pi}} = 1.32 \)
  ```c
  double rejectionFilterNormal() {
    while (true) {
      double u = uniformRand(0, 1); // u in [0, 1)
      double y = -ln(uniformRand(0, 1));
      if (u <= exp(-((y - 1)*(y - 1)) / 2)) return y;
    }
  }
  ```
  ```c
  double normal() {
    double x = rejectionFilterNormal();
    double u = uniformRand(0, 1);
    if (u < 0.5) return (x);
    else return (-x);
  }
  ```
Computing Integrals

- Given ability to generate numbers $\sim \text{Uni}(0, 1)$
  - Want to compute (approximate) value of an integral
  - Useful when integral has no closed form
    - Or may have closed form, but don’t know how to derive it
- Basic idea
  - Consider graph of function
  - Throw “darts” at graph
  - Determine percentage below function
  - Multiply by area into which you threw darts
    - Square: (domain of integral) x (maximum value of function)
- This is called “Monte Carlo Integration”
  - Named after area in Monaco known for its casinos

Monte Carlo Integration

- Consider integral: $\int_0^2 e^x dx$
  
```java
double integrate() {
    // number of “darts” to throw
    int NUM_POINTS = 1000000;
    int numBelowFn = 0;
    for(int i = 0; i < NUM_POINTS; i++) {
        double x = uniformRand(0, 1) * 2.0;
        double y = uniformRand(0, 1) * exp(2.0);
        if (y < exp(x)) numBelowFn++;
    }
    double fraction = (double)numBelowFn/NUM_POINTS;
    return (2.0 * exp(2.0) * fraction);
}
```

How Well Does This Do?

- Consider integral: $\int_0^2 e^x dx$
  - Analytically:
    $\int_0^2 e^x dx = e^2 - 1 = 6.389$
  - Monte Carlo Integration:
    | NUM_POINTS | Computed value (to 4 significant digits) |
    |------------|-----------------------------------------|
    | 10         | 5.911                                   |
    | 100        | 6.650                                   |
    | 1,000      | 6.872                                   |
    | 10,000     | 6.239                                   |
    | 100,000    | 6.387                                   |
    | 1,000,000  | 6.391                                   |
    | 10,000,000 | 6.388                                   |

Computing Statistics Via Simulation

- Recall example:
  ```java
  int Recurse() {
      int x = randomInt(1, 3); // Equally likely values
      if (x == 1) return 3;
      else if (x == 2) return (5 + Recurse());
      else return (7 + Recurse());
  }
  ```
  - Wanted to compute $E[Y]$
    - Analytically, derived $E[Y] = 15$
    - Can approximate by simulation: run algorithm many times, compute average
  
How Well Does This Do?

- Recall example:
  ```java
  int Recurse() {
      int x = randomInt(1, 3); // Equally likely values
      if (x == 1) return 3;
      else if (x == 2) return (5 + Recurse());
      else return (7 + Recurse());
  }
  ```
  - Simulation
    | # of runs | Average value (to 5 significant digits) |
    |-----------|-----------------------------------------|
    | 10        | 16.800                                  |
    | 100       | 15.080                                  |
    | 1,000     | 15.920                                  |
    | 10,000    | 14.844                                  |
    | 100,000   | 14.977                                  |
    | 1,000,000 | 14.999                                  |