Simulation and Random Number Generators

- **Simulation**: modeling of a real-world system requiring at least part of the processes to be based on random numbers.
- They need **random number generators.** The numbers are called pseudo-random in fact since they are generated by a deterministic algorithm based on a recurrence relation.
- When such a generator returns numbers in a range, like [-MAXINT, MAXINT], the numbers don't repeat until the entire interval was covered.

Random Number Generator

- We'll suppose that we have the following function
  `long randint( long first, long last);`
  returning a random number between first and last, such that
- if first $\leq$ i $\leq$ last, the probability that randint returns i is 1/(last-first+1).
- Assume that we also have the function
  `double randouble();`
  returning a random value in [0,1).

Code

- Example of the function rand() : if x is the previous value, then the next one is computed as x1 = (a * x+b) mod c, where a, b, c are large prime numbers. rand() returns a random integer between [-MAXINT, MAXINT].
- ```
  long randint(long first, long second) {
    return first + abs(rand()) %
                   (second-first+1);
  }
  ```
- ```
  double randouble() {
    return double(abs(rand()))/MAXINT;
  }
  ```


Examples

- Choosing a random element of an array[0..n-1]:
  r = randint(0, n-1);
  return a[r];
- Simulating an experiment with a die that has a biased face, twice more likely to be drawn than the others.
- Drawing a card from a shuffled deck.
- For n experiments of throwing a die, what is the sum of all the numbers showing up?

Randomizing Arrays and Files

- Suppose that we want to simulate a card game. Suppose that the cards are of the following type:

```
struct card_type {
  int rank;
  int suit;
};
```

- We would like to write a function shuffle such that it gives each of the 52! permutations of cards a fair chance to be produced.


```
class deck_type {
  public:
    enum suit_type {CLUB=1, DIAMOND,
      HEART, SPADE};
    enum rank_type {TWO=2, THREE, FOUR,
      FIVE, SIX, SEVEN, EIGHT, NINE,
      TEN, JACK, QUEEN, KING, ACE};
    deck_type();  // Constructor
    void shuffle();
  private:
    card_type deck[53];
};
```

```
// Constructor
deck_type::deck_type()
{
  int i = 1;
  for (int rank = TWO; rank <=ACE;
       ++rank)
    for (int suit = CLUB;
         suit <= SPADE; ++suit) {
      deck[i].suit = suit;
      deck[i].rank = rank;
      ++i;
    }
}
```



```
Shuffling Algorithm 1:
  for (int k = 1; k <= 52; ++k)
  {
     r = randint(1,52);
     swap(deck[r],deck[k]);
  }

Shuffling Algorithm 2:
  for (int k = 1; k <= 51; ++k)
  {
     r = randint(k,52);
     swap(deck[r],deck[k]);
  }
```

First Algorithm

- Each possible transposition (swap) is a root of a subtree.
- Total number of leaves in the tree is 52^{52}. Each of them is a permutation but 52^{52} is not a multiple of 52! so the probability of each permutation can't be the same.


```
void select_sample (file F, int N, int K, file G)
{
  bool in_sample[N+1] = {false};
  int count = 0;
  while (count < K) {
      int r = randint(1,N);
      if (!in_sample[r]) {
          in_sample[r] = true;
          ++count;
      }
  }
  for (int i = 1; count > 0; ++i) {
      "read an object from file F into a
       temporary location Temp"
      if (in_sample[i]) {
          "write the object in Temp out to file G"
          --count;
      }
  }
}
```

```
bool select_sample (file F, int N, int K,
  file G)
{
  if ( N < 0  ||  K < 0  ||  K > N )
    return false;
  while (K > 0) {
      F >> Temp;
      int r = randint(1,N);
      if (r <= K) {
          G << Temp;
          --K;
        }
      --N;
    }
  return true;
}
```



```
bool select_sample (int K, file F, array a)
{
  int M = 0;
  while (M < K) {
    if ("at end of file F")
      return false;
    a[++M] = "the next object from file F";
  }
  while ("not at end of file F") {
    a[0] = "the next object from file F";
    r = randint(1,M+1);
    if (r <= K)
      a[r] = a[0];
    ++M;
  }
  return true;
}
```