## Random Numbers

### Step 1

The phenomenon of *randomness* has had numerous applications throughout history, including being used in dice games (as early as the 3,000s BCE in Mesopotamia), coin-flips (which originally were interpreted as the expression of divine will and were later used in games and decision-making), the shuffling of playing cards (which were used in divination as well as in games), and countless other applications.

To Computer Scientists, *randomness* is essential to a special class of algorithms known as **Randomized Algorithms**: algorithms that employ a degree of randomness as part of their logic. **Randomized Algorithms** can be broken down into two classes of algorithms: **Las Vegas Algorithms**, which use the random input to reduce the expected running time or memory usage but are guaranteed to terminate with a correct result, and **Monte Carlo Algorithms**, which have a chance of producing an incorrect result (but hopefully perform "pretty well" on average).

In the context of data structures specifically, *randomness* is essential to a special class of data structures known as **Randomized Data Structures**, which are data structures that incorporate some form of random input to determine their structure and data organization, which in turn effects their performance in finding, inserting, and removing elements.

For all of the applications listed above, we used the vague term *randomness*, but we can be more concrete and specify that these are all applications of **random number generation**. For example, rolling a *k*-sided dice can be considered equivalent to generating a random number from 1 to *k*, flipping a coin can be considered equivalent to generating a random number that can only be 0 or 1, etc. Even with **Randomized Algorithms** and **Randomized Data Structures**, any applied randomness can be broken down into randomly generating numbers.

In this section, we will discuss computational methods of **random number generation**.

### Step 2

It might seem strange, but it turns out that it can be extremely difficult to achieve **true random number generation**. The method by which we generate **true random numbers** is by measuring some physical phenomenon that is expected to be random and then compensating for possible biases in the measurement process. Example sources of physical randomness include measuring atmospheric noise, thermal noise, and other external electromagnetic and quantum phenomena: things that demonstrate natural *entropy* (or disorder).

The speed at which entropy can be harvested from natural sources is dependent on the underlying physical phenomena being measured, which is typically significantly slower than the speed of a computer processor. As a result, the computer processes that measure these sources of entropy are said to be "blocking": they have to slow down (or halt entirely) until enough entropy is harvested to meet the demand. As a result, **true random number generation** is typically very *slow*.

It turns out, however, that there exist computational algorithms that can produce long sequences of seemingly random results, which are in fact completely determined by a shorter initial value, known as a **seed**. Basically, you create an instance of a random number, and you *seed* it with some number, which is then used to generate the sequence of "random" numbers. This type of a random number generator is said to be **pseudo-random** because it *seems* to be random for all intensive purposes, but given the seed, it's fully deterministic.

For example, let's say that yesterday, we seeded some random number generator using the integer 42 as our seed. The sequence of numbers we received as we queried the random number generator *seemed* to be randomly distributed. However, let's say that today, we seed the random number generator, again with the integer 42 as our seed. The sequence of numbers we receive today will

be *completely identical* to the sequence of numbers we received yesterday! Hence, the random number generator is **pseudo-random**. Because these algorithms for generating **pseudo-random** numbers are not bound by any physical measurements, they are *much faster* to generate than **true random number generation**.

**STOP and Think:** We mentioned that generating **true random numbers** is very slow, but generating **pseudo-random numbers** based off of some seed number is very fast. Is there some way we can merge the two approaches to get reasonably good randomness that is fast to generate?

## Step 3

It turns out that we can incorporate both approaches of **random number generation** to generate a sequence of random numbers that are "random enough" for practical uses, but in a fashion that is much faster than generating a sequence of **true random numbers**. We can take a **pseudo-random number generator** and *seed* it with a **true random number**! The result is a sequence of random numbers that can be generated quickly, but that changes upon each execution of our program.

For example, let's say that we have a program that outputs a sequence of random numbers using a **pseudo-random number generator**, but seeded with some natural **true random number** (e.g. thermal noise in the computer circuitry). If we ran the program yesterday, it would have measured some amount of thermal noise (let's call it *x*), then it would have seeded the **pseudo-random number generator** with *x*, and then it would have outputted a sequence of random numbers. If we then run the same program today, it would measure some amount of thermal noise that would be *different* from yesterday's thermal noise (let's call it *y*), then it would seed the **pseudo-random number generator** with *y* (instead of *x*), and it would generate an entirely different sequence of random numbers! Also, even though we still have to generate a **true random number**, which is slow, because we only have to generate a *single* **true random number**—and not an entire sequence of them—it is actually a fast enough operation.

## Step 4

In most programming languages, the built-in random number generator will generate a single random integer at a time, where the random integer will typically range between 0 and some language-specific maximum value. In C++, the random number generator is the **`rand()`** function, which generates a random integer between **0** and **`RAND_MAX`**, inclusive (the exact value of `RAND_MAX` depends on the exact C++ Standard Template Library, or STL, that is on your system). Before we call `rand()`, however, we must first seed it by calling the **`srand()`** function (which stands for **s**eed **`rand`**) and passing our seed integer as a parameter. It is common to use the current system time as a seed.

```
srand(time(NULL));   // seed random number generator using system time
int number = rand(); // generate random number from 0 through RAND_MAX
```

What if we wanted more control over the range of numbers from which we sample? For example, what if we want to generate a random integer in the range from 0 through 99? It turns out that we can do this fairly trivially using the **modulo** operator! Recall that the modulo operator (%) returns the remainder of a division operation. Specifically, if we call *a* % *b*, we are computing the remainder of *a* divided by *b*. If we wanted to generate a number from 0 through 99, we could do the following:

```
int number = rand() % 100; // generate random number and map to range 0 through 99
```

What about if we wanted to generate a random number from 1 to 100?

```
int number = (rand() % 100) + 1 // generate random number, map to 0-99, then add 1 --> 1-100
```

What about if we had C++ vector called myVec, and we wanted to choose a random element from it?

```
vector<string> myVec;
// add stuff to myVec
int index = rand() % myVec.size(); // randomly choose an index
string element = myVec[index];     // grab the element at the randomly-chosen index
```

The possibilities are endless! All of the examples above were sampling from the **Uniform Distribution** (i.e., every outcome was equally likely), but we can become even more clever and use statistics knowledge to sample from other distributions (e.g. Gaussian, Exponential, Poisson, etc.) by performing some combination of operations using techniques like the ones above. In short, *any* random sampling from *any* distribution can be broken down into sampling from a Uniform Distribution.

## Step 5

**CODE CHALLENGE:** Using the C++ random number generator `rand()`, implement a **fair coin** that outputs either **1 or 0** with equal probability. **Do *not* seed the random number generator within the function you will be implementing!** We will seed the random number generator for you.

**Hint:** If you need help using `rand()`, either refer to the C++ Reference, or look at the examples on the previous step.

**Note:** In the Sample Input, the 1 represents how many times we will run your code. It is for grading purposes, so please ignore it.

**Sample Input:**

```
1
```

**Sample Output:**

```
0
```

**To solve this problem please visit https://stepik.org/lesson/31402/step/5**

## Step 6

**CODE CHALLENGE:** Using the C++ random number generator `rand()`, implement a **fair *k*-sided die** that outputs one of the numbers from **1 through *k*, inclusive,** with equal probability. **Do *not* seed the random number generator within the function you will be implementing!** We will seed the random number generator for you.

**Hint:** If you need help using `rand()`, either refer to the C++ Reference, or look at the examples on the previous step.

**Note:** In the Sample Input, the 1 on the top line represents how many times we will run your code. It is for grading purposes, so please ignore it. The bottom line is *k*.

**Sample Input:**

```
1
6
```

**Sample Output:**

```
6
```

**To solve this problem please visit https://stepik.org/lesson/31402/step/6**

## Step 7

**CODE CHALLENGE:** Using the C++ random number generator `rand()`, implement a **biased coin** that outputs **1 with probability *p*** and **0 with probability 1–*p***. **Do *not* seed the random number generator within the function you will be implementing!** We will seed the random number generator for you.

**Hint:** If you need help using `rand()`, either refer to the C++ Reference, or look at the examples on the previous step.

**Note:** In the Sample Input, the 1 on the top line represents how many times we will run your code. It is for grading purposes, so please ignore it. The bottom line is *p*.

---

**Sample Input:**

```
1
0.7
```

**Sample Output:**

```
1
```

**To solve this problem please visit https://stepik.org/lesson/31402/step/7**

## Step 8

We hope that you are now comfortable with **random number generation** in general as well as in C++ specifically. Also, we hope you now understand the distinction between **true random numbers** (truly random, but can only be generated by harvesting natural entropy, which is *slow*) and **pseudo-random numbers** (seemingly random, but deterministic upon the seed, but can be generated *very quickly*). Also, remember that you can combine these two **random number generation** approaches by using a **true random number** to *seed* a **pseudo-random number generator**.

No matter what problem you are trying to solve, if you need to generate a random number in some certain range, or if you need to sample from some distribution, or if you need to do something possibly even more fancy, rest assured that any of these tasks can be broken down into sampling random integers.

Later in this text, we will come back to **random number generation** as it applies to the various **Randomized Data Structures** we will be discussing.