25

Random Numbers – How Can We Create Randomness in Computers?

Bruno Müller-Clostermann and Tim Jonischkat

Universität Duisburg-Essen, Essen, Germany

Algorithms are clever procedures to efficiently solve a variety of problems. In the preceding chapters we learned numerous examples for "normal" algorithms, like binary search, insertion sort, depth-first search in graphs and finding shortest paths. As a consequence one might assume that algorithms – despite all their cleverness and efficiency – are stubborn and purely replicative procedures yielding in any case perfect and unique solutions. Seemingly algorithms have nothing to do with randomness (or chance). But wait! When applying QUICKSORT the pivot-element is proposed to be selected randomly. The One-Time-Pad procedure uses keys that have been randomly chosen. In Fingerprinting numbers are randomly selected.

Tactical and strategic PC games likewise apply algorithms where randomness is highly desirable or even indispensable. Often the computer operates as an opponent, steering its actions by algorithms that imitate meaningful and intelligent behaviors. This is well known from interactive games like *Sims*, *SimCity*, the *Settlement Game*, and *World of Warcraft*. Under identical situations the computer is not expected to show identical behaviors, to the contrary a range of various effects and actions is pleasing. As a consequence, diversity and stimulation increase.

Generating random numbers or random events by throwing a die (getting numbers $1, 2, \ldots, 6$) or a coin (getting heads or tails) is obviously not possible in a programmed algorithm. On the other hand, can random behavior be programmed? Is it possible to create randomness by algorithms? The answer is as follows: Randomness is imitated by algorithms, which generate numbers that are apparently random. Hence such numbers are often called pseudorandom numbers (although the term random numbers is quite common). Here we consider well-known and approved procedures to construct random number generators. There are many fields of application for random numbers. Here we introduce two examples: A computer game and the so called Monte Carlo simulation for the determination of surface areas.

B. Vöcking et al. (eds.), *Algorithms Unplugged*,
DOI 10.1007/978-3-642-15328-0_25,

Fig. 25.1. Rock, paper, scissors (Picture credits: Tim Jonischkat)

Fig. 25.2. Coin: heads or tails; die: $1, 2, \ldots, 6$; roulette wheel: $0, 1, \ldots, 36$ (Picture credits: Lukasz Wolejko-Wolejszo, Toni Lozano)

A Tactical Game: "Rock, Paper, Scissors"

As a simple example for a programmed game we can consider an algorithm for the popular game "Rock, Paper, Scissors". The game works like this: As a player you have to choose one of three options: rock, paper or scissors (Fig. 25.1). Afterwards the algorithm is executed and also yields one of the three options. The evaluation of this round follows these rules: Rock beats scissors, scissors beats paper, paper beats rock. The winner gets one point and the next round follows.

How shall the algorithm proceed? A permanent alternation between "scissors" and "rock" would be a possible (and boring) tactic that is soon deciphered by a human opponent! Obviously the result of the algorithm should be *unforeseeable*, like throwing dice, drawing lottery numbers or spinning a roulette wheel.

A mechanical coupling of an algorithm with dice or a roulette wheel (Fig. 25.2) would be rather troublesome, in any case such a construction would be neither efficient nor clever. Hence an algorithmic procedure is needed that chooses seemingly at random among the three possibilities rock, paper and scissors. In other words, we need a *random number generator*.

Means for the Generation of Random Numbers: Modular Arithmetic

Before we have a closer look at the computation of random numbers, we need the concept of modular arithmetic. The modulo-function (also called mod-function) determines the remainder that is left over under division of

two natural numbers. For example consider the division of 27 by 12; since $27 = 2 \cdot 12 + 3$, we obtain the remainder 3. In the case of two arbitrary natural numbers x and m, we may divide x through m and find as result $x = a \cdot m + r$, where a is called quotient and r is called the remainder. The remainder r is a natural number from the interval $\{0, 1, \ldots, m-1\}$.

Examples for Modular Arithmetic

- $9 \bmod 8 = 1$
- $16 \bmod 8 = 0$
- $(9 + 6) \bmod 12 = 15 \bmod 12 = 3$
- $(6 \cdot 2 + 15) \bmod 12 = 27 \bmod 12 = 3$
- $1143 \bmod 1000 = 143$
  In the case of division by 1000 the remainder r is given by the last 3 digits.

Illustration of Modular Arithmetic

Modular arithmetic can be considered like a walk along a circle that carries the numbers $0, 1, 2, \ldots, m-1$. To find out x mod m (the perimeter the remainder r when dividing x through m) we start at position 0 and make x steps in clockwise direction. The number of completed revolutions along the circle equals the quotient a; the position we finally arrive at is the remainder r.

As an example consider the hour hand of a clock in the interval between January 1st, 0 a.m. and January 2nd, 7 p.m. Obviously a period of 43 hours has passed, yielding 3 complete revolutions of the hour hand (each revolution accounting for 12 hours) and another 7 steps that move the hour hand to 7 hours. Since a clock displays hours modulo 12, this number 7 is exactly the result of $r = 43 \bmod 12$. In modular arithmetic addition corresponds to movement in clockwise direction. We consider two examples (cf. Fig. 25.3).

If we start at position $x = 9$ and add the number 6, we have to take 6 steps and arrive at position 3; e.g., it holds that $(9 + 6) \bmod 12 = 3$ (cf. Fig. 25.3, left).

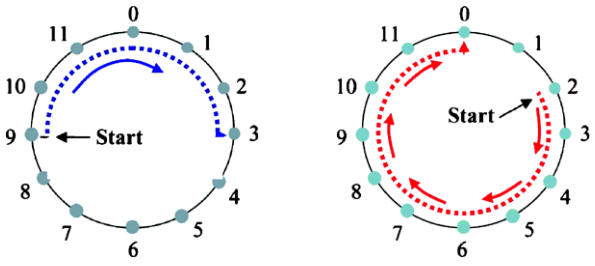

Fig. 25.3. Two examples of modular arithmetic: $9 \bmod 6 = 3$ and $(6 \cdot 2) \bmod 12 = 0$

Multiplying a number x with a factor a can be resolved into a series of additions; if we start at position $x = 2$ and multiply by $a = 6$, we have to add 5 times the value 2, i.e., we have to take 5 two-steps in clockwise direction, leading us to position 0; e.g., it holds that $(6 \cdot 2) \bmod 12 = 0$ (cf. Fig. 25.3, right).

In computer science modular arithmetic may be seen as the "natural" arithmetic because a computer's storage is always limited (finite); moreover, storage cells are also finite and can only store numbers up to a certain size.

An Algorithm for the Generation of Pseudorandom Numbers

The algorithm as given below employs modular arithmetic and provides random numbers from the interval $\{0, 1, \ldots, m-1\}$. The basic principle is quite simple: Take a starting value x_0, the factor a, the constant c, and the modulus m; compute the remainder r as $r = (a \cdot x_0 + c) \bmod m$. We set this value to be the first random number: $x_1 = r$. Now we use x_1 to compute the second random number $x_2 = (a \cdot x_1 + c) \bmod m$; proceeding with x_2 in the same way, we obtain a sequence of random numbers $x_1, x_2, x_3, \ldots$.

This procedure can be written down more precisely as follows:

$$\begin{aligned} x_1 &:= (a \cdot x_0 + c) \bmod m, \\ x_2 &:= (a \cdot x_1 + c) \bmod m, \\ x_3 &:= (a \cdot x_2 + c) \bmod m, \\ x_4 &:= (a \cdot x_3 + c) \bmod m, \\ &\text{etc.} \end{aligned}$$

In general notation we achieve an iterative procedure as follows:

$$x_{i+1} := (a \cdot x_i + c) \bmod m, \quad i = 0, 1, 2, \ldots.$$

To achieve a concrete random number generator, numerical values for factor a, constant c, and modulus m must be supplied; furthermore we need a starting value x_0, which is sometimes called the seed of the generator.

Setting $a = 5$, $c = 1$, $m = 16$ and $x_0 = 1$ results in these calculations

$$\begin{aligned} x_1 &:= (5 \cdot 1 + 1) \bmod 16 = 6, \\ x_2 &:= (5 \cdot 6 + 1) \bmod 16 = 15, \\ x_3 &:= (5 \cdot 15 + 1) \bmod 16 = 12, \\ x_4 &:= (5 \cdot 12 + 1) \bmod 16 = 13, \\ x_5 &:= (5 \cdot 13 + 1) \bmod 16 = 2, \\ x_6 &:= (5 \cdot 2 + 1) \bmod 16 = 11, \end{aligned}$$

$$x_7 := (5 \cdot 11 + 1) \bmod 16 = 8,$$
$$x_8 := (5 \cdot 8 + 1) \bmod 16 = 9,$$
$$\text{etc.}$$

Obviously this sequence is completely determined by the prescribed calculations. That is, for a given starting value x_0 we will always get the same, strictly defined and reproducible elements; such a behavior is called deterministic. By choosing another starting value x_0, the entry point into the sequence can be newly selected for another run of the algorithm.

Periodic Behavior

If we continue the above calculation we see that after 16 steps the sequence returns to its start value 1, and that each of the 16 possible numbers from the interval $\{0, 1, \ldots, 15\}$ has occurred exactly once. Computing the next 16 values $x_{16}, x_{17}, \ldots, x_{31}$, the sequence will repeat itself, and we notice a reproducing behavior, here with a period of length 16. When the modulus m is set to a very large number and furthermore we choose the factor a, and the constant c precisely, larger periods are achieved. In the ideal case we manage to reach a full period of length m. Sometimes programming languages provide a built-in random number generator or provide it through a function library; the programming language Java offers a full-period generator with parameter values $a = 252149003917$, $c = 11$ and $m = 2^{48}$.

Simulation of True Random Number Generators

Pseudorandom numbers from the interval $\{0, 1, \ldots, m-1\}$ are basic for many applications. Examples are the simulation of throwing a coin yielding heads or tails, throwing a die resulting in one of the values $1, 2, 3, 4, 5$ and 6, or spinning a roulette wheel providing the 37 possibilities $0, 1, \ldots, 36$.

Let us assume that for each example the random generating devices coin, die and roulette wheel are fair and do produce the corresponding outcomes with probabilities $\frac{1}{2}$, $\frac{1}{6}$ and $\frac{1}{37}$. To simulate dice throwing we need a procedure to transform a random number $x \in \{0, 1, \ldots, m-1\}$ in a number $z \in \{0, 1\}$, where 0 stands for "heads" and 1 for "tails". An easy procedure could be a mapping of "small" numbers to the value 0, and of "large" numbers to 1, or mathematically expressed: $z := 0$ if $x < \frac{m}{2}$, $z := 1$ if $x \geq \frac{m}{2}$. In the case of the roulette wheel we could use as transformation procedure $z := x \bmod 37$, and for the die $z := x \bmod 6 + 1$.

The Algorithm for Rock, Paper, Scissors

Now, after all, we return to our tactical game. We need an algorithm that decides randomly (or at least apparently random) between the three prospects:

rock, paper and scissors. To this end we use a random number generator that generates for each round a random number x; by the calculation $z := x \bmod 3$ we obtain a new number that can just provide one of the values 0, 1 and 2. Dependent on the actual value of z the algorithm chooses rock (0), paper (1) or scissors (2). This algorithm has been implemented as a small program, the rock-paper-scissors-applet.

A selection of four different (pseudo) random number generators is available; the first one is given just as the extreme case of a fixed sequence of numbers 0, 1 and 2.

1. Deterministic: The fixed sequence of numbers $2, 0, 1, 1, 0, 0, 0, 2, 1, 0, 2$
2. RNG-016: $a = 5$, $c = 1$, $m = 16$ and start value $x_0 = 1$ (period is of length 16)
3. RNG-100: $a = 81$, $c = 1$, $m = 100$ and start value $x_0 = 10$ (period is of length 100)
4. Java Generator: The generator of programming language Java: "java.util.Random"

The algorithm NEXTRANDOMNUMBER uses the input value x from the interval $\{0, 1, \ldots, m-1\}$ and the constant parameters a, c and m. With **return** the computed value from the interval $\{0, 1, \ldots, m-1\}$ is passed back. This value is the next random number following the input number x.

```
1  procedure NEXTRANDOMNUMBER (x)
2  begin
3      return (a · x + c) mod m
4  end
```

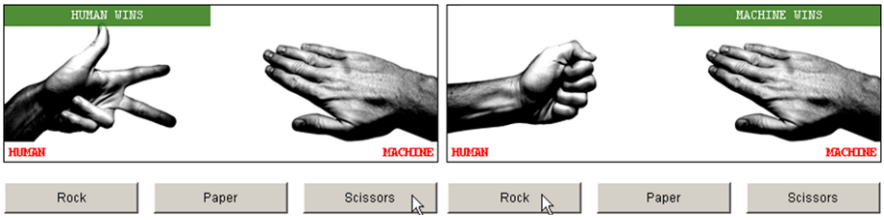

Fig. 25.4. Scissors cut paper (*left*), rock is wrapped by paper (*right*)

Table 25.1. State of game after three rounds: human 2, machine 1

Round	Human			Machine
1	Rock	0	1	Paper
2	Scissors	1	1	Paper
3	Rock	2	1	Scissors

This procedure RANDOMNUMBEREXAMPLE(n) is to illustrate the usage of the procedure NEXTRANDOMNUMBER(x). Firstly parameters a, c and m are initialized. The start value is set to value 1. Each invocation of procedure NEXTRANDOMNUMBER(x) changes the value of x to a new value that is returned and printed out. Additionally the value x mod 3 is also printed out.

```
1   procedure randomNumberExample (n)
2   begin
3       a := 5; c := 1; m := 16;
4       x := 1;
5       for i := 1 to n do
6           x := nextRandomNumber(x);
7           print(x);
8           print(x mod 3)
9       endfor
10  end
```

The generated random numbers are as follows:
x_i: 6, 15, 12, 13, 2, 11, 8, 9, 14, 7, ...
x_i modulo 3: 0, 0, 0, 1, 2, 2, 2, 1, 2, 1, ...

The pictures show two possible outcomes of a round "human" versus "machine" (Fig. 25.4) and the state of the game after several rounds (Table 25.1).

Figure 25.5 illustrates the working principle of the selected random number generator (here RNG-016). The current calculation is highlighted, and furthermore the future values are already visible! Hence, one can forecast the future decision of the machine (a small cheat).

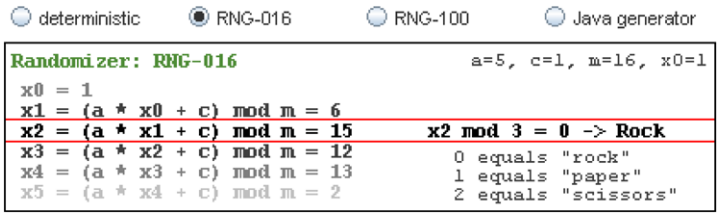

Fig. 25.5. Algorithm of the machine

Monte Carlo Simulation: Determination of Areas Using "Random Rain"

An important field of application for random numbers is the so called Monte Carlo simulation, named after the gambling casino in Monte Carlo. Monte Carlo simulation can be used – for example – to determine the areas of irregular shaped geometric figures by means of "random rain". The term "random rain" describes the situation where many two-dimensional random points (x, y) are hitting a flat surface like rain drops. A random point on a flat surface is given as an (x, y)-pair of random coordinates, where x-value and y-value are random values from the real interval $[0, 1]$. To this end random values $x \in \{0, 1, \ldots, m-1\}$ are transformed to real values between 0.0 and 1.0; the transformation rule is plainly given as $x := x/(m-1)$.

If an arbitrary area is placed into a unit square, i.e., all edges have length 1.0, and random points are thrown into the square, a fraction will hit the area whereas the others will miss it. Consider Fig. 25.6 for an example.

An estimate for the area is obtained by counting the hits and calculation of the term $A = (\text{hit count})/(\text{drop count})$. To achieve a good estimate the number of drops should be rather large, say many millions or even some billions of random points must be thrown. Of course, carrying out the algorithm by hand is out of discussion; as a substitute a programmed algorithm running on a computer has no problem to drop some million random points into a square.

As an example for the practical application of the Monte Carlo technique we consider a technique for the determination of the few first digits of the famous mathematical constant $\pi = 3.14159\ldots$. This can be done by throwing random points (x, y) into the unit circle. We consider a unit square (a square with side length 1.0 and area 1.0 as well), with an inscribed quadrant of the unit circle. Since the circle has radius $r = 1$ its area is $A = r^2 \cdot \pi = \pi$ and the area of the quadrant is $\frac{\pi}{4}$.

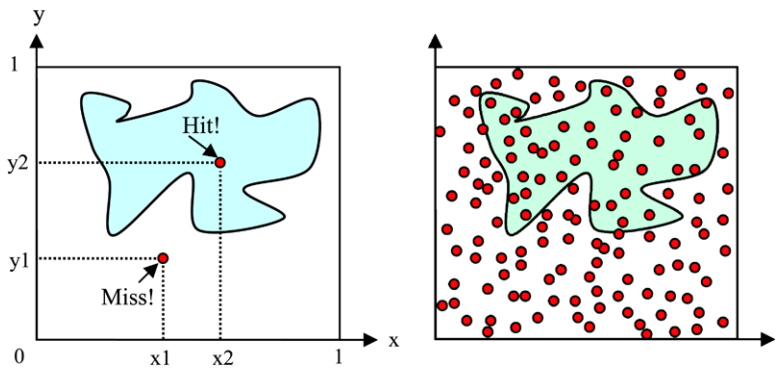

Fig. 25.6. Two random points (x_1, y_1) and (x_2, y_2), a hit and a miss (*left*); random rain: Count the number of drops hitting the area (*right*)

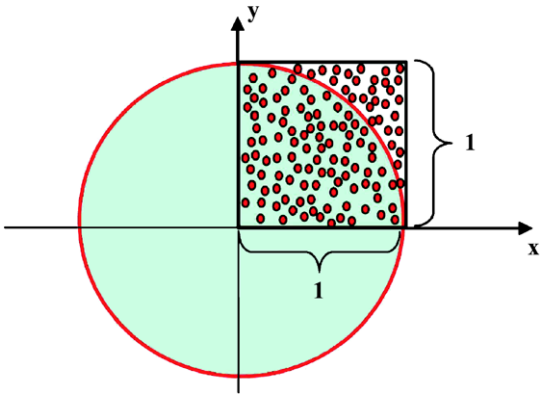

Fig. 25.7. How many points hit the quadrant?

If T is the number of hits in the quadrant and N is the number of hits in the unit square, we can approximately calculate $\pi \approx 4 \cdot \frac{T}{N}$. The figure shows 130 random points where 102 points fall into the quadrant, i.e., we calculate $\pi \approx 4 \cdot \frac{102}{130} \approx 3.1384$. This result is still not very precise, although with 100,000 or some millions or even one billion points the result should become significantly better. This procedure is summarized more precisely in the algorithm RandomRain.

The algorithm RandomRain delivers an estimate for the mathematical constant π. We assume a unit square (side length = 1) with area 1.0.

```
1   procedure RandomRain (n)
2   begin
3       a := 1103515245; c := 12345; m := 4294967296;      // parameters
4       z := 1; hits := 0;      // start values
5       for i := 1 to n do
6           z := (a · z + c) mod m;
7           x := z/(m − 1);
8           z := (a · z + c) mod m;
9           y := z/(m − 1);
10          if inCircle(x,y) then
11              hits := hits + 1
12          endif
13      endfor
14      return 4 · hits/n
15  end
```

Comment: The function inCircle() tests whether the point (x, y) is located inside the circle area. Therefore it uses the equation $x^2 + y^2 = r^2$; the point (x, y) is counted as a hit in the unit circle if $x^2 + y^2 \leq 1$.

There are an abundant number of applications of Monte Carlo simulations in engineering and the natural sciences. In computer science the field of randomized (or probabilistic) algorithms developed out of Monte Carlo simulation.

Further Reading

1. A starting point for further reading is Wikipedia:
   http://en.wikipedia.org/wiki/Pseudorandom_number_generator
2. Modular arithmetics as used for pseudorandom number generation is essential for other fields within computer science, e.g., for techniques like Public-Key Cryptography (Chap. 16), Fingerprinting (Chap. 19), the One-Time Pad (Chap. 15) and Simulated Annealing (Chap. 41).
3. Other fields of application for random numbers are stochastic simulation programs that are used to investigate the performance of complex systems like computer networks, mobile systems and Web services.