Highest Common Factor

- It is often useful to be able to calculate the highest common factor of two numbers a and b (otherwise known as the greatest common divisor).
- We shall write $h = \text{hcf}(a, b)$ for this number. It is the largest integer that divides exactly into both a and b. For example, $\text{hcf}(24, 30) = 6$.
- What is the hcf of 18 and 25?

Algorithm for Computing H.C.F.

- We shall use the prime factorisations of a and b. Let $p_1, \ldots, p_n$ be the set of primes that divide either a or b, and write:

$$a = p_1^{l_1} \cdots p_n^{l_n} \qquad \text{and} \qquad b = p_1^{m_1} \cdots p_n^{m_n}.$$

  Here some of the powers l_i, m_i may be zero.

- Then it is not hard to show that

$$\text{hcf}(a, b) = p_1^{k_1} \cdots p_n^{k_n}, \tag{1}$$

  where $k_i = \min(l_i, m_i)$. For example,

$$24 = 2^3 \times 3^1 \times 5^0 \qquad \text{and} \qquad 30 = 2^1 \times 3^1 \times 5^1. \tag{2}$$

  So $k_1 = 1$, $k_2 = 1$, and $k_3 = 0$. Hence
  $\text{hcf}(24, 30) = 2^1 \times 3^1 \times 5^0 = 6$, as before.

- Write down the prime factorisations of 18 and 25. Use them to calculate $\text{hcf}(18, 25)$.

Euclid's Algorithm

So we have a workable algorithm, but it is very inefficient because it depends on the prime factorisation. One of the most famous (and earliest) algorithms for any task is Euclid's algorithm for computing the hcf(a, b). It is based on the following argument.

- If $a \leq b$, then we can divide b by a to obtain a quotient q and a remainder r, where $0 \leq r < a$. This means that $b = qa + r$.
- Now, if $r = 0$, then a divides b, and hence hcf$(a, b) = a$ and we are done.
- Otherwise, it is easy to show that hcf$(a, b) =$ hcf(r, a). We can then continue by dividing r into a.
- Because $r < a$, we can guarantee that the algorithm will eventually terminate, and it is much more efficient than using the prime factorisation.

Euclid's Algorithm: Worked Example

Let us find the hcf of 568 and 208:

$$568 = 2 \times 208 + 152$$

$$208 = 1 \times 152 + 56$$

$$152 = 2 \times 56 + 40$$

$$56 = 1 \times 40 + 16$$

$$40 = 2 \times 16 + 8$$

$$16 = 2 \times 8 + 0$$

Thus $\text{hcf}(208, 568) = 8$.

Euclid's Algorithm II

- Euclid's algorithm provides extra information: it enables us to find integers m and n such that $ma + nb = \text{hcf}(a, b)$. (We shall see why this is useful when we look at cryptography).
- This is done by working backwards through the computation from the penultimate line.

$$
\begin{aligned}
8 &= 40 - 2 \times 16 \\
&= 40 - 2 \times (56 - 1 \times 40) = 3 \times 40 - 2 \times 56 \\
&= 3 \times (152 - 2 \times 56) - 2 \times 56 = 3 \times 152 - 8 \times 56 \\
&= 3 \times 152 - 8 \times (208 - 1 \times 152) = 11 \times 152 - 8 \times 208 \\
&= 11 \times (568 - 2 \times 208) - 8 \times 208 = 11 \times 568 - 30 \times 208
\end{aligned}
$$

- Use Euclid's algorithm to find $\text{hcf}(24, 30)$. Also find integers m and n such that $24m + 30n = \text{hcf}(24, 30)$.

Solution

Use Euclid's algorithm to find hcf$(24, 30)$. Also find integers m and n such that $24m + 30n = \text{hcf}(24, 30)$.

$$30 = 1 \times 24 + 6 \qquad\qquad 6 = 1 \times 30 - 1 \times 24$$

$$24 = 4 \times 6 + 0$$

Hence $6 = \text{hcf}(24, 30)$ and $6 = 1 \times 30 - 1 \times 24$.

Rational Numbers

- The next number system extends the integers so that it is closed under all four arithmetic operators: addition, subtraction, multiplication and division.
- This is the rational numbers $\mathbb{Q}$, which consists of integer fractions.
- Thus $\mathbb{Q}$ consists of all numbers that can be written in the form a/b with a and b in $\mathbb{Z}$ and $b \neq 0$.
- Why are the rational numbers important? Every decimal or binary number which recurs must be rational. Because a number that is represented on a computer only has a finite number of non-zero values after the decimal point, eventually it recurs (with the repeated value 0). Hence every number represented on a computer must be rational.

Rational Arithmetic

We can define the four arithmetic operators on $\mathbb{Q}$ as follows:

$$\frac{a_1}{b_1} +_{\mathbb{Q}} \frac{a_2}{b_2} = \frac{a_1 b_2 + a_2 b_1}{b_1 b_2}$$

$$\frac{a_1}{b_1} -_{\mathbb{Q}} \frac{a_2}{b_2} = \frac{a_1 b_2 - a_2 b_1}{b_1 b_2}$$

$$\frac{a_1}{b_1} \times_{\mathbb{Q}} \frac{a_2}{b_2} = \frac{a_1 a_2}{b_1 b_2}$$

$$\frac{a_1}{b_1} \div_{\mathbb{Q}} \frac{a_2}{b_2} = \frac{a_1 b_2}{b_1 a_2} \quad \text{if } a_2 \neq 0$$

We can easily show that $\mathbb{Q}$ extends $\mathbb{Z}$ by noting that the set of rationals of the form $a/1$ is equivalent to $\mathbb{Z}$.

Real Numbers

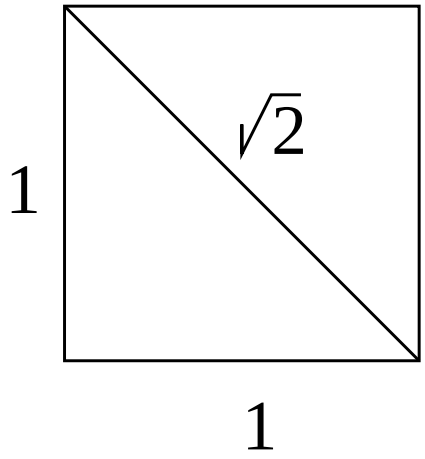
- The final number system we need is the real numbers, denoted by $\mathbb{R}$.
- We can take this system to be all the infinite decimal numbers (both recurring and non-recurring). To be precise, we must note that certain decimals are exactly the same. For example:

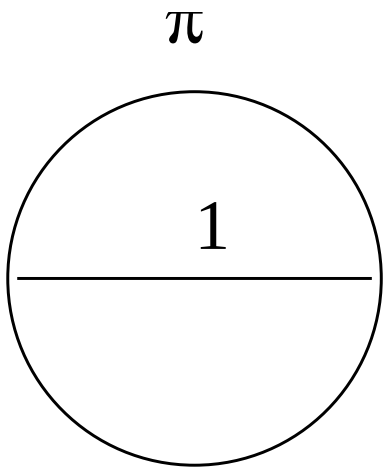
$$42.7899999\cdots = 42.79000000.$$

- Arithmetic on the real line is defined in the familiar way.

Real Numbers and Geometry

- Why do we need real numbers? The simple answer is that the rationals leave a lot of gaps.
- The real numbers represent all the points on a line: there are lengths that are not rational numbers, and these are called irrational numbers.
- Two famous examples of irrational numbers are $\sqrt{2}$ and π.





Summary

1. There are four main 'standard' number systems: natural numbers $\mathbb{N}$; integers $\mathbb{Z}$; rationals $\mathbb{Q}$; reals $\mathbb{R}$.
2. We must distinguish between mathematical definitions of number systems and their implementation in software.
3. Natural numbers are for counting; they are defined by a zero and a successor function $Sn = n + 1$.
4. The Σ notation is a compact way of expressing sums.
5. The integers add negative numbers to the natural numbers. There is no first element.
6. Prime numbers are the basic building blocks for the multiplicative structure of the integers. The prime factorisation of an integer is essentially unique.
7. Euclid's algorithm is an efficient method for computing the highest common factor of two integers.
8. Rational numbers are the ratio of two integers.
9. Real numbers are all infinite decimals.
10.
11.

Modular Arithmetic

Not another number system! What is it good for?

Random number generators. There are many applications where an element of randomness is needed, and modular arithmetic is a good way of generating random numbers.

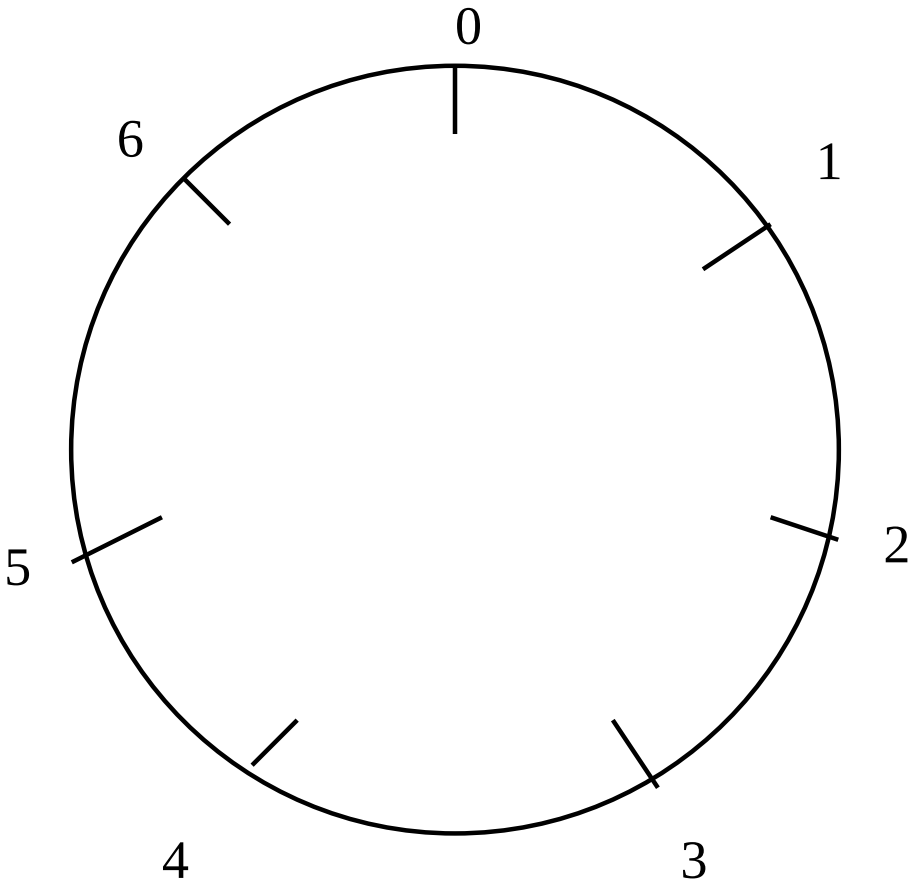
- For a card game you will certainly need to shuffle the pack (i.e. select all the cards in a random order). For a board game you will need to roll the dice (i.e. select at random two integers in the range 1 to 6).
- For an action game, you may want some randomness to represent external variations and prevent predictability.
- To simulate the randomness in the real world you may need to generate events with a certain probability. For example, in a traffic simulation vehicles enter the system at certain random times.

Cryptography. The transmission of information that you don't want to share requires a method of encrypting it so that only the receiver can decrypt it. Modular arithmetic is the basis of the most commonly used cryptographic system on the Internet: the RSA algorithm.

Error-correcting codes. There are many situations in which a large amount of data has to be transmitted quickly and reliably. Obvious examples are communication by telephone and the transmission of television signals to and from satellites. In practice, noise (random fluctuations or external disturbances) may corrupt signals. Error-correcting codes are used to encode values in a digital form and incorporating extra information so that even if the signal is slightly modified, the original values can be recovered. Modular matrices are the basis of most practical error-correcting codes.

Modular Numbers

- If n is a positive integer, then we denote by $\mathbb{Z}_n$ the set $\{0, 1, \ldots, n-1\}$, and call it the integers modulo n (or just mod n).
- We cannot just use the usual definitions of the integer operations because then $\mathbb{Z}_n$ would not be closed under these operations. For example, $2 + 2 = 4$, but 4 is not in $\mathbb{Z}_3$.
- We shall, however, use the same symbols $+$ and $\times$ to represent modular addition and modular multiplication. This can be a bit confusing until you are used to it.



Modular Addition

- To define modular addition $a + b \bmod n$, we first calculate $a + b$ using integer addition, and then compute the remainder r when divided by n; r is guaranteed to be in the range $0 \leq r < n$ so is a member of $\mathbb{Z}_n$.
- For example, to calculate $2 + 3 \bmod 4$, we first of all calculate $2 + 3 = 5$ using integer addition, and then find the remainder when 5 is divided by 4, which is 1. We conclude that

$$2 + 3 \bmod 4 = 1 \bmod 4.$$

Modular Multiplication

- We define modular multiplication $a \times b \bmod n$ by the following algorithm. First compute $a \times b$ in integer arithmetic, and then find the remainder when it is divided by n.
- For example, to calculate $2 \times 3 \bmod 4$, we first calculate $2 \times 3 = 6$ using integer multiplication, and then find the remainder when 6 is divided by 4, which is 2. We conclude that

$$2 \times 3 \bmod 4 = 2 \bmod 4.$$

Example: $\mathbb{Z}_5$

$+$	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

$\times$	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

Exercise Draw up modular arithmetic tables for $\mathbb{Z}_4$. What are the key differences between the multiplication tables for $\mathbb{Z}_4$ and $\mathbb{Z}_5$? (Hint: look for zeros).

Exercise: Solution

$+$	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

$\times$	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	0	2
3	0	3	2	1

Note that the rows of the multiplication table for $\mathbb{Z}_5$ only have zeros in the first column, while the row for 2 in $\mathbb{Z}_4$ contains a zero elsewhere. Also the columns after the first in $\mathbb{Z}_5$ just permute the non-zero values, but that is not true for $\mathbb{Z}_4$.

Prime Modulus

the product of two non-zero values can give a result of zero: this is quite different from integer multiplication. For example,

$$2 \times 2 = 0 \bmod 4 \qquad \text{and} \qquad 3 \times 4 = 0 \bmod 6.$$

This can happen only if the modulus n is not prime. If the modulus p is prime, three important properties hold:

Cancellation

$$\text{If } a \neq 0 \text{ and } \quad a \times b = a \times c \bmod p \quad \text{then} \quad b = c \bmod p \tag{3}$$

Inverse

$$\text{If } a \neq 0 \text{ then } \quad \exists y \quad \text{such that} \quad x \times y = 1 \bmod p. \tag{4}$$

Fermat's Little Theorem

$$\text{If } a \neq 0 \quad a^{p-1} = 1 \bmod p. \tag{5}$$

Multiplicative Inverses

- Fermat's little theorem tells us that $a^{p-1} = 1 \bmod p$
- It follows that $a \times a^{p-2} = 1 \bmod p$, and hence that a^{p-2} is the inverse of a. For example,

$$2^{-1} = 2^3 = 8 = 3 \bmod 5.$$

Exercise Use Fermat's little theorem to find the inverses of 3 and 4 modulo 5. Check your answers against the multiplication table shown in the $\mathbb{Z}_5$ table.

Exercise: Solution

$$
\begin{aligned}
3^{-1} &= 3^{p-2} \\
&= 3^3 \\
&= 9 \times 3 \bmod 5 \\
&= 4 \times 3 \bmod 5 \\
&= 12 \bmod 5 \\
&= 2 \bmod 5
\end{aligned}
$$

$$
\begin{aligned}
4^{-1} &= 4^{p-2} \\
&= 4^3 \\
&= 16 \times 4 \bmod 5 \\
&= 1 \times 4 \bmod 5 \\
&= 4 \bmod 5.
\end{aligned}
$$

Ada Implementation

In Ada 95, the integer types are subdivided into signed integer types and modular types. The modular types are unsigned integer types for which the arithmetic cycles round. We can define a type for arithmetic modulo 5 as follows:

```
1   TYPE Mod_5 is MOD 5;
2   A, B, C, D: Mod_5;
```

We can now add, subtract, multiply and exponentiate values of this type with the standard arithmetic operators +, -, * and ** and the arithmetic is automatically performed mod 5.

```
3   A := 3; B := 4;
4   C := A + B;     -- Sets C to 2
5   D := A*C;       -- Sets D to 1
6   B := A**3;      -- Sets B to 2
```

- An alternative approach to using modular arithmetic in Ada is to use the `mod` operator on integers.
- For example, the following statement is true mathematically:

$$a + b \bmod n = (a \bmod n + b \bmod n) \bmod n. \tag{6}$$

- The corresponding statement about Ada expressions is also true:

```
(a + b) mod n = (a mod n + b mod n) mod n
```

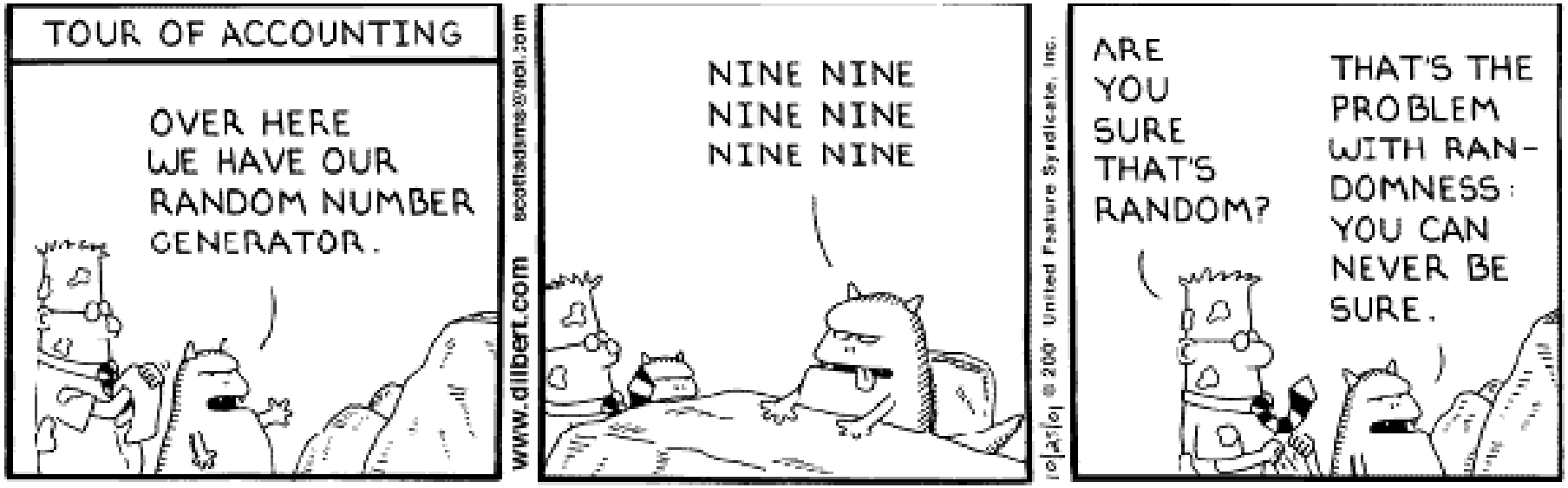
Random Number Generators

God does not play dice. *Albert Einstein*

- Whether God does or does not play dice, it is a fact that many computer programs need a source of random numbers in order to inject some unpredictability into their operation.
- If you really want truly random numbers, then the only way to provide is to build some form of complex physical system and use its output.
- What is needed is an algorithm that generates random numbers. Of course, this is impossible, since an algorithm is deterministic (i.e., predictable). Instead, we use an algorithm that generates pseudo-random numbers. Such an algorithm generates a sequence of numbers that appear to be random under a wide range of statistical tests (even though they aren't truly random).

Pseudo-Random Number Generators

- If we generate numbers which are pseudo-randomly distributed in the range $[0, 1]$, then in a sequence of 1000 such numbers, we would expect about 100 to be in the range $[0, 0.1]$, and we can test whether the difference between the actual number in a given sequence and 100 is statistically significant.
- In any sequence of truly random numbers, all patterns of numbers will occur eventually.



Congruential Random Number Generators

- Most practical random number generators are based on simple congruential algorithms of the form

$$I_{j+1} \equiv aI_j \bmod m.$$

- This algorithm generates a sequence of numbers in $\mathbb{Z}_m$ given by I_1, I_2, I_3, etc. If a and m are chosen carefully, you can generate $m-1$ different integers.
- These are usually converted to random numbers in the range $[0, 1]$ by dividing by m.

Seeds

- The first number in the sequence is called the seed.
- Suppose that we set the seed to a particular value (42, say) and then generate 10 numbers. If we reset the seed to the same value again and generate 10 numbers, we will get the same 10 numbers.
- This might seem to be a nuisance, given that we want unpredictable behaviour, but is actually very helpful. Suppose that you have a large system using a random number generator and that midway through a run the program goes wrong (e.g. it crashes). To debug the problem, you will need to run the system again under identical conditions to find out what has gone wrong and prove that you have removed the error. However, if you don't know what the seed was when you first ran the system, you cannot run it again in an identical way.
- The solution is to run the system from a known seed, but to use different seeds for each run.

Example

- We will choose $m = 5$, $a = 2$ and the seed $I_1 = 1$.
- Then the sequence of numbers is

$$I_1 = 1, I_2 = 2, I_3 = 4, I_5 = 3, I_5 = 1 \ldots \tag{7}$$

- Note how we get 4 distinct integers: we say that the period of the generator is 4.
- If we want real numbers, we divide by 5 and get

$$R_1 = 0.2, R_2 = 0.4, R_3 = 0.8, R_4 = 0.6, R_5 = 0.2 \ldots \tag{8}$$

- What happens if we choose a seed of 0?

Parameter Choice

- It is important to realise that the choice of a and m can have a big effect on the quality of your random number generator, and is best left to experts in number theory.
- (That is, don't just make up some values, but look them up in a reliable text, and make sure that you transcribe them into your program accurately.)
- Consider the values $a = 2$ and $m = 2147483647$. If we start with a seed set to 1, then the first ten values will all be less than $2^{10} \approx 1000$; as fractions they will all be less than $10^{-6} = 0.000001$.
- Another way of saying this is that small numbers are followed by several small numbers; this is a highly undesirable correlation between numbers in the sequence.