

Recap and announcements

Section B: Digital Information

Representing numbers (cont.)

Adding 1-bit numbers

p	q	$p + q$
1	1	1 0
1	0	0 1
0	1	0 1
0	0	0 0

Adding 1-bit numbers

p	q	$p + q$	
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

p	q	$p + q$	
		LB	RB
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

Adding 1-bit numbers

p	q	$p + q$	
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

p	q	$p + q$	
		LB	RB
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

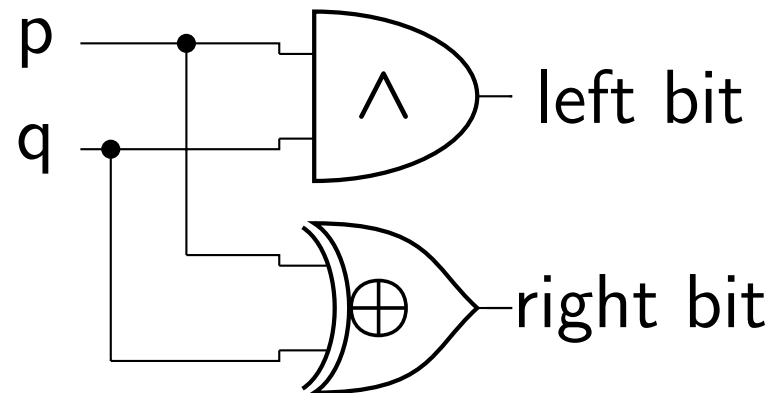
p	q	$p + q$	
		$p \wedge q$	$p \oplus q$
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

Adding 1-bit numbers

p	q	$p + q$	
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

p	q	$p + q$	
		LB	RB
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

p	q	$p + q$	
		$p \wedge q$	$p \oplus q$
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0



This circuit is called a **half adder** (2 bits in, 2 bits out)

Adding 1-bit numbers

p	q	r	$p + q + r$	
1	1	1	1	1
1	1	0	1	0
1	0	1	1	0
1	0	0	0	1
0	1	1	1	0
0	1	0	0	1
0	0	1	0	1
0	0	0	0	0

Adding 1-bit numbers

p	q	r	$p + q + r$	
1	1	1	1	1
1	1	0	1	0
1	0	1	1	0
1	0	0	0	1
0	1	1	1	0
0	1	0	0	1
0	0	1	0	1
0	0	0	0	0

p	q	r	$p + q + r$	
			LB	RB
1	1	1	1	1
1	1	0	1	0
1	0	1	1	0
1	0	0	0	1
0	1	1	1	0
0	1	0	0	1
0	0	1	0	1
0	0	0	0	0

CHALLENGE: Construct a circuit diagram for a circuit which implements the above (3 bits in, 2 bits out). We shall call such a circuit a **full adder**.

Meaningful renaming of bits in a full adder

p	q	carry	carry	sum
		in	out	out
1	1	1	1	1
1	1	0	1	0
1	0	1	1	0
1	0	0	0	1
0	1	1	1	0
0	1	0	0	1
0	0	1	0	1
0	0	0	0	0

4-bit addition via a cascade of full adders

A = 0110
+ B = 0011

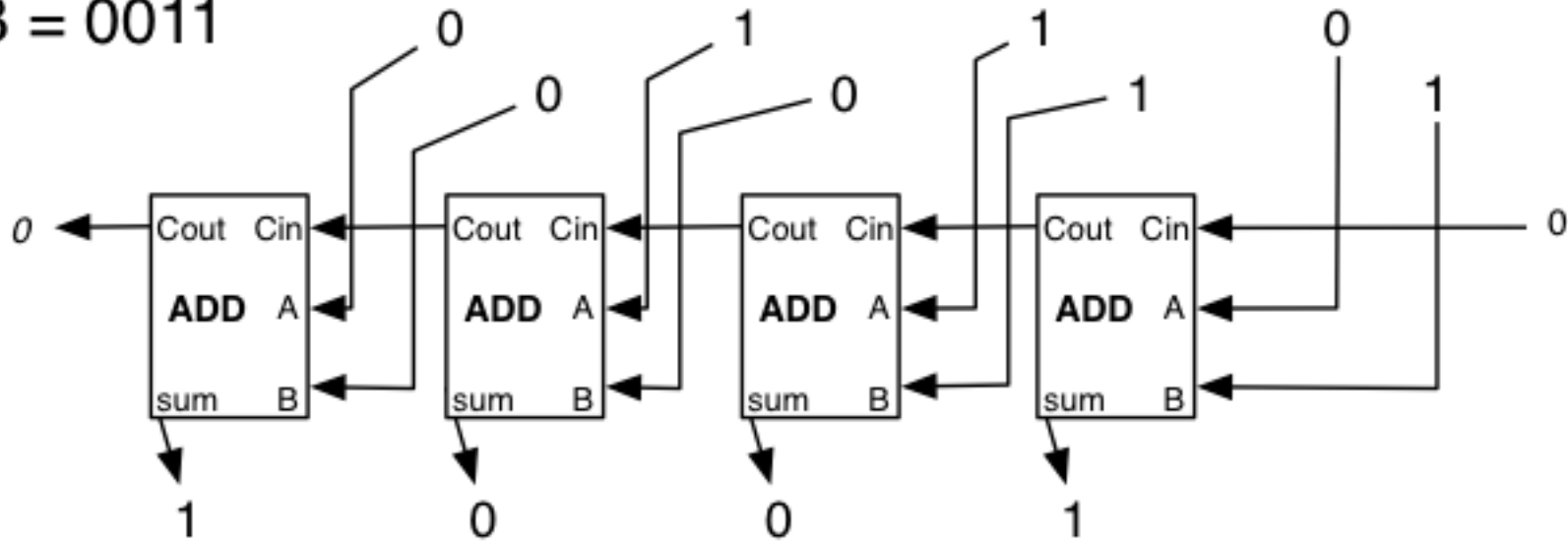


Diagram extracted from “Concepts in Computing” course notes at Dartmouth College, Hanover, New Hampshire, USA, 2009.

Circuits for integer subtraction

Since $x - y$ is the same as $x + (-y)$, we can accomplish integer subtraction provided we can convert y to $-y$ (because we already know how to add).

Our clever way of representing negative numbers means

- the integer representation of an integer y can be converted to the integer representation of $-y$ by toggling all bits and adding one.
- A NOT gate will toggle a single bit, and we already know how to build a circuit that adds, so we can build a circuit that adds one.

Something amazingly neat

Our clever way of representing negative numbers means

- the usual algorithm (the step-by-step process) that works for adding two non-negative integers, works for adding two negative integers and for adding a non-negative integer and a negative integer
- the carry bit from the most significant place in an addition can be ignored.
- this will work provided that the sum is in the range of integers that can be represented by the number of bits we have chosen.

Examples

4-bit examples:

$$\begin{array}{r} 4 \quad \quad 0 \ 1 \ 0 \ 0 \\ -6 \quad + \ 1 \ 0 \ 1 \ 0 \\ \hline -2 \quad \quad 1 \ 1 \ 1 \ 0 \end{array}$$

$$\begin{array}{r} 7 \quad \quad 0 \ 1 \ 1 \ 1 \\ -3 \quad + \ 1 \ 1 \ 0 \ 1 \\ \hline 4 \quad \quad (1) \ 0 \ 1 \ 0 \ 0 \end{array}$$

The carry bit in () is ignored.

8-bit example:

$$\begin{array}{r} 81 \quad \quad 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \\ -98 \quad + \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \\ \hline -17 \quad \quad 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 1 \end{array}$$

$$\left[\begin{array}{l} 81 = 64 + 16 + 1 \rightarrow 0101 \ 0001 \\ 98 = 64 + 32 + 2 \rightarrow 0110 \ 0010 \\ -98 \rightarrow 1001 \ 1101 + 1 \rightarrow 1001 \ 1110 \\ \hline 1110 \ 1111 \rightarrow -(0001 \ 0000 + 1) \\ \quad \quad \rightarrow -0001 \ 0001 \rightarrow -17 \end{array} \right]$$

Multiplication

Example:

$$\begin{array}{r} 1 \ 0 \ 1 \ 1 \\ \times 1 \ 1 \ 0 \ 0 \\ \hline 0 \ 0 \ 0 \ 0 \\ + 0 \ 0 \ 0 \ 0 \ 0 \\ + 1 \ 0 \ 1 \ 1 \ 0 \ 0 \\ + 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \\ \hline 1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \\ \hline 1 \quad 1 \quad 1 \quad 1 \end{array}$$

Multiplication

Example:

$$\begin{array}{r}
 1 \ 0 \ 1 \ 1 \\
 \times 1 \ 1 \ 0 \ 0 \\
 \hline
 0 \ 0 \ 0 \ 0 \\
 + 0 \ 0 \ 0 \ 0 \ 0 \\
 + 1 \ 0 \ 1 \ 1 \ 0 \ 0 \\
 + 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \\
 \hline
 1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \\
 \hline
 1 \quad 1 \quad 1 \quad 1
 \end{array}$$

Notice that no ‘multiplication tables’ are required since the only multiplications used are ‘times 0’ which results in all zeroes, and ‘times 1’ which has no effect.

Multiplication

Example:

$$\begin{array}{r} 1\ 0\ 1\ 1 \\ \times 1\ 1\ 0\ 0 \\ \hline 0\ 0\ 0\ 0 \\ + 0\ 0\ 0\ 0\ 0 \\ + 1\ 0\ 1\ 1\ 0\ 0 \\ + 1\ 0\ 1\ 1\ 0\ 0\ 0 \\ \hline 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0 \\ \hline 1\ 1\ 1\ 1 \end{array}$$

Notice that no ‘multiplication tables’ are required since the only multiplications used are ‘times 0’ which results in all zeroes, and ‘times 1’ which has no effect.

This is one of the bonuses of using binary in computers.

Multiplication by shifts and additions

For computer implementation of multiplication (within \mathbb{N}) we only need a 'shift' circuit (to move bits left and append a zero on the right) and addition circuits.

But we also need an *algorithm* to control the process.

An **algorithm** is a sequence of operations on an input. The result is called an output.

Examples: Addition, subtraction, multiplication, division.

Example

Here is very simple list processing algorithm, set out using a standard format for displaying algorithms:

Input: L : a list of students.

Output: O : list of students who have submitted their assignment.

Method:

Set $j = 1$, $O =$ empty list. (Initialization phase)

Loop: If $j = \text{length}(L) + 1$, stop.

 If $L[j]$ (the j -th element in L) has
 submitted the assignment, append $L[j]$ to O .

 Replace j by $j + 1$.

Repeat loop.

Algorithm for binary addition

This algorithm formalizes the method described earlier, which is implemented with a cascade of full adders.

Contrary to notation used in some earlier examples, in this context it is usual to *number the bits starting from the rightmost bit, which is called the 0-th bit*.

So the leftmost bit of an n -bit number is its $(n - 1)$ -th bit.

Example: For the number 01101001 we have

0	1	1	0	1	0	0	1
↑	↑	↑	↑	↑	↑	↑	↑
7-th	6-th	5-th	4-th	3-th	2-th	1-th	0-th
bit	bit	bit	bit	bit	bit	bit	bit

Algorithm for binary addition

Input: Two numbers p, q in base 2 with n bits.

Output: The sum $s = p + q$ in base 2 with $n + 1$ bits.

Method:

Set $j = 0$, $c = 0$, and the n -th bits of p and q to 0.

Loop: If $j = n + 1$ stop.

Add the j -th bits of p and q plus c .

Store the result part as the j -th bit of s .

Store the carry part as c (replacing the old value of c).

Replace j by $j + 1$.

Repeat loop

Example: Add $p = 0110$ and $q = 1111$ with $n = 4$.

Input: Two numbers p, q in base 2 with n bits.

Output: The sum $s = p + q$ in base 2 with $n + 1$ bits.

Method:

Set $j = 0$, $c = 0$, and the n -th bits of p and q to 0.

Loop: If $j = n + 1$ stop.

Add the j -th bits of p and q plus c .

Store the result part as the j -th bit of s .

Store the carry part as c (replacing the old value of c).

Replace j by $j + 1$.

Repeat loop

Algorithm for binary multiplication

This algorithm formalises the method described earlier, except that addition is done progressively rather than at the end.

The additions may be done by the algorithm of the previous slide.

This algorithm also assumes the existence of a shift algorithm, which shifts all the bits of a number one place to the left and then fills the vacant rightmost place with a 0.

Algorithm for binary multiplication

Input: Two numbers x, y in base 2 with n bits.

Output: The product $p = x \times y$ in base 2 with $2n$ bits.

Method:

Set $j = 0$ and $p = 0$.

Loop: If $j = n$ stop.

 If the j -th bit of y is 1 then replace p by $p + x$.

 Shift x .

 Replace j by $j + 1$.

Repeat loop.

Example: Multiply $x = 1111$ by $y = 0110$ with $n = 4$.

Input: Two numbers x, y in base 2 with n bits.

Output: The product $p = x \times y$ in base 2 with $2n$ bits.

Method:

Set $j = 0$ and $p = 0$.

Loop: If $j = n$ stop.

 If the j -th bit of y is 1 then replace p by $p + x$.

 Shift x .

 Replace j by $j + 1$.

Repeat loop.

Rational numbers

What is a rational number?

Recall that \mathbb{Q} is the set of **rational numbers**. A rational number is a number that can be represented as the ratio of two integers.

EXAMPLE $\frac{2}{3}$ is a rational number.

Please note that every integer is a rational number as, for example $6 = \frac{6}{1}$.

What is a rational number?

Recall that \mathbb{Q} is the set of **rational numbers**. A rational number is a number that can be represented as the ratio of two integers.

EXAMPLE $\frac{2}{3}$ is a rational number.

Please note that every integer is a rational number as, for example $6 = \frac{6}{1}$.

Are you happy with this definition?

What is a rational number?

Let $Q = \mathbb{Z} \times (\mathbb{Z} \setminus \{0\})$.

What does an element of Q look like?

What is a rational number?

Let $Q = \mathbb{Z} \times (\mathbb{Z} \setminus \{0\})$.

What does an element of Q look like?

The set Q may be partitioned so that any elements (n_1, d_1) and (n_2, d_2) of Q are in the same partition set if and only if $n_1 d_2 = n_2 d_1$.

So, for example, $\{(2, 3), (-2, -3), (4, 6), (-4, -6), (6, 9), (-6, -9), \dots\}$ is one of the sets in the partition

The sets in the partition may themselves be considered rational numbers. We usually write $\frac{2}{3}$ instead of

$\{(2, 3), (-2, -3), (4, 6), (-4, -6), (6, 9), (-6, -9), \dots\}$.

Representing rationals in a computer

For computer storage of any **non-zero** rational number q we need to express it using **scientific notation**. For any base b this is

$$q = (-1)^s \times m \times b^n$$

where

- $q \in \mathbb{Q}, q \neq 0$;
- $b \in \mathbb{Z}, b \geq 2$;
- $s \in \{0, 1\}$, (s is the **sign bit**)
- $m \in \mathbb{Q}, 1 \leq m < b$, (m is the '**mantissa**') and
- $n \in \mathbb{Z}$ (n is the **exponent**).

Given q and b , the values of s , m and n are uniquely determined by these conditions.

Example in base 10: $13.5 = (-1)^0 \times 1.35 \times 10^1$.

Examples

Example in base 10: $23.5 = (-1)^0 \times 2.35 \times 10^1$.

To save space, we store 235 instead of 2.35 because 2.35 is the only number between 1 and 10 with digits 1, 3, 5.

Examples

Example in base 10: $23.5 = (-1)^0 \times 2.35 \times 10^1$.

To save space, we store 235 instead of 2.35 because 2.35 is the only number between 1 and 10 with digits 1, 3, 5.

Examples in base 10:

- $-154 = (-1)^1 \times 1.54 \times 10^2$. Store m as 154.
- $0.031 = (-1)^0 \times 3.1 \times 10^{-2}$. Store m as 31.

The number of digits used to store m is called the **precision**.

IEEE half-precision

The shortest IEEE standard for representing rational numbers is called *half-precision floating point*. It uses a 16-bit word partitioned as in the diagram at right.

To store a rational number x it is first represented as $(-1)^s \times m \times 2^n$ with $1 \leq m < 2$. The sign bit s is stored as the left-most bit (bit 15), the mantissa m (called “significand” in IEEE parlance) is stored in the right-most 10 bits (bits 9 to 0), and the exponent n is stored in the 5 bits in between (bits 14 to 10). However...

IEEE half-precision

- *Only the fractional part of m is stored.* Because $1 \leq m < 2$, the binary representation of m **always** has the form $1 \cdot \star \star \star \dots$ where the stars stand for binary digits representing the fractional part of m . Hence there is no need to store the $1 \cdot$ part.
- *the exponent n is stored with an “offset”.* In order to allow for both positive and negative exponents, but to avoid another sign bit, the value stored is $n + 15$. In principle this means that the five exponent bits can store exponents in the range $-15 \leq n \leq 16$, but 00000 and 11111 are reserved for special purposes so in fact n is restricted to the range $-14 \leq n \leq 15$.

An example

A rational number x is stored in half-precision floating point as the word 5555_{16} . Which number is being represented?

An example

A rational number x is stored in half-precision floating point as the word 5555_{16} . Which number is being represented?

$$(5555)_{16} = \underbrace{(0101)}_{(5)} \underbrace{(0101)}_{(5)} \underbrace{(0101)}_{(5)} \underbrace{(0101)}_{(5)} = \begin{array}{c|c|c} (0) & (10101) & (0101010101) \\ s & n+15 & \text{frac.part of } m \\ & = 21 & \end{array}$$

$$\begin{aligned} \longrightarrow & (-1)^0 \times 1.0101010101 \times 2^{21-15} \\ &= 1.0101010101 \times 2^6 \\ &= 1010101.0101 \quad (\text{moving the binary point 6 places right}) \\ &= 64 + 16 + 4 + 1 + \frac{1}{4} + \frac{1}{16} = \boxed{85\frac{5}{16} = 85.3125_{10}}. \end{aligned}$$

A warning

WARNING: With limits on precision and exponent size, some rational numbers can only be stored inaccurately, if at all.

Of course, the same sort of thing is true for integers. But with integers we can represent ALL of the integers close enough to 0, so it is easier to understand which integers we can and cannot represent.

If you have a reason to need to represent rational numbers with an accuracy beyond the accuracy provided by some sort of standard set up, you can write dedicated software to represent numbers with greater precision.

Part B. Digital Information

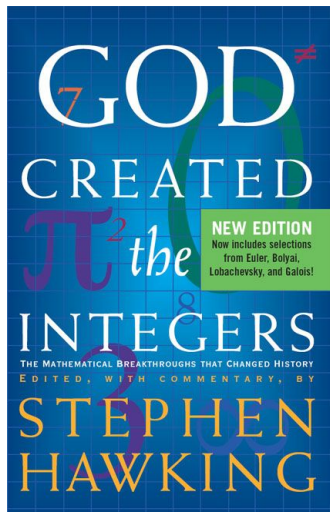
B1. Numbers.

Notes originally prepared by Pierre Portal
Editing and expansion by Malcolm Brooks.

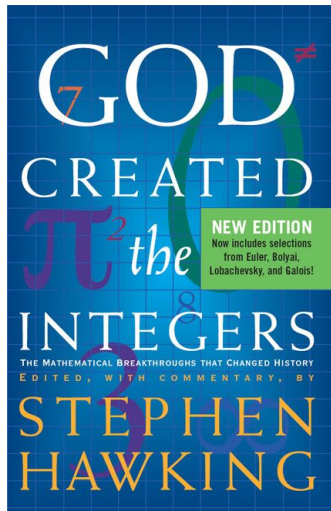
Text Reference (Epp)	3ed: Sections	1.5 (Binary arithmetic), 3.8 (Algorithms), 10.4 (Modular arithmetic)
	4ed: Sections	2.5 (Binary arithmetic), 4.8 (Algorithms), 8.4 (Modular arithmetic)
	5ed: Sections	2.5 (Binary arithmetic), 4.10 (Algorithms), 8.4 (Modular arithmetic)

In the beginning

In the beginning



In the beginning

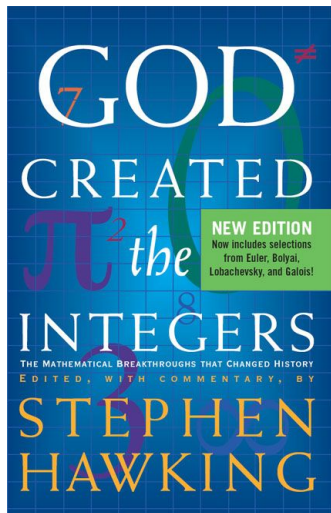


The title of Stephen Hawking's book is a reference to a remark attributed to the mathematician Leopold Kronecker (1823 — 1891).



God made the integers; all else is the work of man.

In the beginning



The title of Stephen Hawking's book is a reference to a remark attributed to the mathematician Leopold Kronecker (1823 — 1891).



God made the integers; all else is the work of man.

This controversial opinion, which was a reaction against set theory, is still held by many people today.

Natural numbers

We are all familiar with the **natural numbers** 1, 2, 3, 4,

Natural numbers

We are all familiar with the **natural numbers** 1, 2, 3, 4,

We have already been using the notation \mathbb{N} to denote the complete set of them.

Natural numbers

We are all familiar with the **natural numbers** 1, 2, 3, 4,

We have already been using the notation \mathbb{N} to denote the complete set of them.

Using set theory it is possible to *define* them like this:

Natural numbers

We are all familiar with the **natural numbers** 1, 2, 3, 4,

We have already been using the notation \mathbb{N} to denote the complete set of them.

Using set theory it is possible to *define* them like this:

- Define $0 := \emptyset$.

Natural numbers

We are all familiar with the **natural numbers** 1, 2, 3, 4,

We have already been using the notation \mathbb{N} to denote the complete set of them.

Using set theory it is possible to *define* them like this:

- Define $0 := \emptyset$.
- Given a set S , define $\text{next}(S) := S \cup \{S\}$.

Natural numbers

We are all familiar with the **natural numbers** 1, 2, 3, 4,

We have already been using the notation \mathbb{N} to denote the complete set of them.

Using set theory it is possible to *define* them like this:

- Define $0 := \emptyset$.
- Given a set S , define $\text{next}(S) := S \cup \{S\}$.
- Define $1 := \text{next}(0)$,

Natural numbers

We are all familiar with the **natural numbers** 1, 2, 3, 4,

We have already been using the notation \mathbb{N} to denote the complete set of them.

Using set theory it is possible to *define* them like this:

- Define $0 := \emptyset$.
- Given a set S , define $\text{next}(S) := S \cup \{S\}$.
- Define $1 := \text{next}(0)$, $2 := \text{next}(1) = \text{next}(\text{next}(0))$,

Natural numbers

We are all familiar with the **natural numbers** 1, 2, 3, 4,

We have already been using the notation \mathbb{N} to denote the complete set of them.

Using set theory it is possible to *define* them like this:

- Define $0 := \emptyset$.
- Given a set S , define $\text{next}(S) := S \cup \{S\}$.
- Define $1 := \text{next}(0)$, $2 := \text{next}(1) = \text{next}(\text{next}(0))$,
and more generally $n := \text{next}(n - 1)$.

Natural numbers

We are all familiar with the **natural numbers** 1, 2, 3, 4,

We have already been using the notation \mathbb{N} to denote the complete set of them.

Using set theory it is possible to *define* them like this:

- Define $0 := \emptyset$.
- Given a set S , define $\text{next}(S) := S \cup \{S\}$.
- Define $1 := \text{next}(0)$, $2 := \text{next}(1) = \text{next}(\text{next}(0))$,
and more generally $n := \text{next}(n - 1)$.
- Define \mathbb{N} to be the smallest set containing 1 and and
everything that can be generated by 'next' starting from 1.
(Such a set construction method is allowed in ZFC.)

Natural numbers

We are all familiar with the **natural numbers** 1, 2, 3, 4,

We have already been using the notation \mathbb{N} to denote the complete set of them.

Using set theory it is possible to *define* them like this:

- Define $0 := \emptyset$.
- Given a set S , define $\text{next}(S) := S \cup \{S\}$.
- Define $1 := \text{next}(0)$, $2 := \text{next}(1) = \text{next}(\text{next}(0))$,
and more generally $n := \text{next}(n - 1)$.
- Define \mathbb{N} to be the smallest set containing 1 and and
everything that can be generated by 'next' starting from 1.
(Such a set construction method is allowed in ZFC.)

Under this scheme, when we say *there are three apples* we mean

Natural numbers

We are all familiar with the **natural numbers** 1, 2, 3, 4,

We have already been using the notation \mathbb{N} to denote the complete set of them.

Using set theory it is possible to *define* them like this:

- Define $0 := \emptyset$.
- Given a set S , define $\text{next}(S) := S \cup \{S\}$.
- Define $1 := \text{next}(0)$, $2 := \text{next}(1) = \text{next}(\text{next}(0))$,
and more generally $n := \text{next}(n - 1)$.
- Define \mathbb{N} to be the smallest set containing 1 and and everything that can be generated by 'next' starting from 1.
(Such a set construction method is allowed in ZFC.)

Under this scheme, when we say *there are three apples* we mean

there is a bijection (one-to-one correspondence) between the set of apples and the set $\{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}$.

Introducing zero

The first extension to \mathbb{N} is to include zero.

Introducing zero

The first extension to \mathbb{N} is to include zero.

Though the natural numbers have been used and written since antiquity, the number zero was invented more recently:

Introducing zero

The first extension to \mathbb{N} is to include zero.

Though the natural numbers have been used and written since antiquity, the number zero was invented more recently:

- first by the Babylonians (as a place-holder only) in the 3rd century BC;

Introducing zero

The first extension to \mathbb{N} is to include zero.

Though the natural numbers have been used and written since antiquity, the number zero was invented more recently:

- first by the Babylonians (as a place-holder only) in the 3rd century BC;
- then independently by the Mayans (again place-holder only) in the fourth century CE;

Introducing zero

The first extension to \mathbb{N} is to include zero.

Though the natural numbers have been used and written since antiquity, the number zero was invented more recently:

- first by the Babylonians (as a place-holder only) in the 3rd century BC;
- then independently by the Mayans (again place-holder only) in the fourth century CE;
- and finally, independently and fully, by Hindu Indians probably in the fifth century CE.

Introducing zero

The first extension to \mathbb{N} is to include zero.

Though the natural numbers have been used and written since antiquity, the number zero was invented more recently:

- first by the Babylonians (as a place-holder only) in the 3rd century BC;
- then independently by the Mayans (again place-holder only) in the fourth century CE;
- and finally, independently and fully, by Hindu Indians probably in the fifth century CE.
- Zero eventually reached Europe, via Muslim Arab scholars and traders, in the twelfth century CE.

[See *A brief history of zero* <http://www.mediatinker.com/blog/archives/008821.html>]

Introducing zero

The first extension to \mathbb{N} is to include zero.

Though the natural numbers have been used and written since antiquity, the number zero was invented more recently:

- first by the Babylonians (as a place-holder only) in the 3rd century BC;
- then independently by the Mayans (again place-holder only) in the fourth century CE;
- and finally, independently and fully, by Hindu Indians probably in the fifth century CE.
- Zero eventually reached Europe, via Muslim Arab scholars and traders, in the twelfth century CE.

[See *A brief history of zero* <http://www.mediatinker.com/blog/archives/008821.html>]

I shall sometimes use the notation \mathbb{N}^\star to denote $\mathbb{N} \cup \{0\}$.

Integers

Logically the next extension of the number system is to the **negatives** of the natural numbers,

Integers

Logically the next extension of the number system is to the **negatives** of the natural numbers, then **fractions** (rational numbers)

Integers

Logically the next extension of the number system is to the **negatives** of the natural numbers, then **fractions** (rational numbers) and **decimals** (real numbers),

Integers

Logically the next extension of the number system is to the **negatives** of the natural numbers, then **fractions** (rational numbers) and **decimals** (real numbers), though historically (positive) rational and irrational numbers came before any negative numbers.

Integers

Logically the next extension of the number system is to the **negatives** of the natural numbers, then **fractions** (rational numbers) and **decimals** (real numbers), though historically (positive) rational and irrational numbers came before any negative numbers.

The history of all this is a fascinating chapter in the development of mathematics itself, but we will concentrate on just the numbers themselves, their properties and representations, leaving the history and mathematical foundations for another course.

(e.g. MATH3343 <https://studyat.anu.edu.au/2014/courses/MATH3343/details.html>)

Integers

Logically the next extension of the number system is to the **negatives** of the natural numbers, then **fractions** (rational numbers) and **decimals** (real numbers), though historically (positive) rational and irrational numbers came before any negative numbers.

The history of all this is a fascinating chapter in the development of mathematics itself, but we will concentrate on just the numbers themselves, their properties and representations, leaving the history and mathematical foundations for another course.

(e.g. MATH3343 <https://studyat.anu.edu.au/2014/courses/MATH3343/details.html>)

An **integer** is either a positive or negative natural number, or zero. The set of all integers is usually denoted by \mathbb{Z} . (The 'Z' stands for *zählen*, the German for 'numbers'.)

Integers

Logically the next extension of the number system is to the **negatives** of the natural numbers, then **fractions** (rational numbers) and **decimals** (real numbers), though historically (positive) rational and irrational numbers came before any negative numbers.

The history of all this is a fascinating chapter in the development of mathematics itself, but we will concentrate on just the numbers themselves, their properties and representations, leaving the history and mathematical foundations for another course.

(e.g. MATH3343 <https://studyat.anu.edu.au/2014/courses/MATH3343/details.html>)

An **integer** is either a positive or negative natural number, or zero.

The set of all integers is usually denoted by \mathbb{Z} .

(The 'Z' stands for *zählen*, the German for 'numbers'.)

$$\mathbb{Z} = \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$$

Real, rational and irrational numbers

A **real number** is any number that can, in principle, be expressed using decimal notation.

Real, rational and irrational numbers

A **real number** is any number that can, in principle, be expressed using decimal notation. Examples are 5, 5.5, -0.12345, etc.

Real, rational and irrational numbers

A **real number** is any number that can, in principle, be expressed using decimal notation. Examples are 5, 5.5, -0.12345, etc.

Saying '*in principal*' allows for infinitely long expressions such as

(a) $0.1\overline{37} = 0.13737 \dots$ ('37' repeats for ever; 'recurring decimal')

Real, rational and irrational numbers

A **real number** is any number that can, in principle, be expressed using decimal notation. Examples are 5, 5.5, -0.12345, etc.

Saying '*in principal*' allows for infinitely long expressions such as

(a) $0.1\overline{37} = 0.13737 \dots$ ('37' repeats for ever; 'recurring decimal')

(b) $\sqrt{5} = 2.2360679 \dots$ (never ending but no repeating patterns)

Real, rational and irrational numbers

A **real number** is any number that can, in principle, be expressed using decimal notation. Examples are 5, 5.5, -0.12345, etc.

Saying '*in principal*' allows for infinitely long expressions such as

(a) $0.1\overline{37} = 0.13737 \dots$ ('37' repeats for ever; 'recurring decimal')

(b) $\sqrt{5} = 2.2360679 \dots$ (never ending but no repeating patterns)

The set of all real numbers is denoted by \mathbb{R} .

Real, rational and irrational numbers

A **real number** is any number that can, in principle, be expressed using decimal notation. Examples are 5, 5.5, -0.12345, etc.

Saying '*in principal*' allows for infinitely long expressions such as

(a) $0.1\overline{37} = 0.13737 \dots$ ('37' repeats for ever; 'recurring decimal')

(b) $\sqrt{5} = 2.2360679 \dots$ (never ending but no repeating patterns)

The set of all real numbers is denoted by \mathbb{R} .

A **rational number** is a real number that can be expressed as the quotient (fraction) of an integer and a non-zero integer.

The set of all rational numbers is denoted by \mathbb{Q} . ('Q' for 'quotient'.)

$$\mathbb{Q} = \{q \in \mathbb{R} : \exists n, d \in \mathbb{Z} (q = \frac{n}{d}) \wedge (d \neq 0)\}.$$

Real, rational and irrational numbers

A **real number** is any number that can, in principle, be expressed using decimal notation. Examples are 5, 5.5, -0.12345, etc.

Saying '*in principal*' allows for infinitely long expressions such as

(a) $0.1\overline{37} = 0.13737 \dots$ ('37' repeats for ever; 'recurring decimal')

(b) $\sqrt{5} = 2.2360679 \dots$ (never ending but no repeating patterns)

The set of all real numbers is denoted by \mathbb{R} .

A **rational number** is a real number that can be expressed as the quotient (fraction) of an integer and a non-zero integer.

The set of all rational numbers is denoted by \mathbb{Q} . ('Q' for 'quotient'.)

$$\mathbb{Q} = \{q \in \mathbb{R} : \exists n, d \in \mathbb{Z} (q = \frac{n}{d}) \wedge (d \neq 0)\}.$$

Examples: $\frac{-3}{2}, \frac{68}{495} = \frac{136}{990} = 0.1\overline{37}$ (fraction expressions not unique).

Real, rational and irrational numbers

A **real number** is any number that can, in principle, be expressed using decimal notation. Examples are 5, 5.5, -0.12345, etc.

Saying '*in principal*' allows for infinitely long expressions such as

(a) $0.1\overline{37} = 0.13737 \dots$ ('37' repeats for ever; 'recurring decimal')

(b) $\sqrt{5} = 2.2360679 \dots$ (never ending but no repeating patterns)

The set of all real numbers is denoted by \mathbb{R} .

A **rational number** is a real number that can be expressed as the quotient (fraction) of an integer and a non-zero integer.

The set of all rational numbers is denoted by \mathbb{Q} . ('Q' for 'quotient'.)

$$\mathbb{Q} = \{q \in \mathbb{R} : \exists n, d \in \mathbb{Z} (q = \frac{n}{d}) \wedge (d \neq 0)\}.$$

Examples: $\frac{-3}{2}, \frac{68}{495} = \frac{136}{990} = 0.1\overline{37}$ (fraction expressions not unique).

An **irrational number** is a real number that is not rational.

Real, rational and irrational numbers

A **real number** is any number that can, in principle, be expressed using decimal notation. Examples are 5, 5.5, -0.12345, etc.

Saying '*in principal*' allows for infinitely long expressions such as

(a) $0.1\overline{37} = 0.13737 \dots$ ('37' repeats for ever; 'recurring decimal')

(b) $\sqrt{5} = 2.2360679 \dots$ (never ending but no repeating patterns)

The set of all real numbers is denoted by \mathbb{R} .

A **rational number** is a real number that can be expressed as the quotient (fraction) of an integer and a non-zero integer.

The set of all rational numbers is denoted by \mathbb{Q} . ('Q' for 'quotient').

$$\mathbb{Q} = \{q \in \mathbb{R} : \exists n, d \in \mathbb{Z} (q = \frac{n}{d}) \wedge (d \neq 0)\}.$$

Examples: $\frac{-3}{2}$, $\frac{68}{495} = \frac{136}{990} = 0.1\overline{37}$ (fraction expressions not unique).

An **irrational number** is a real number that is not rational.

Examples: $\sqrt{2}$, π .

Real, rational and irrational numbers

A **real number** is any number that can, in principle, be expressed using decimal notation. Examples are 5, 5.5, -0.12345, etc.

Saying '*in principal*' allows for infinitely long expressions such as

(a) $0.1\overline{37} = 0.13737 \dots$ ('37' repeats for ever; 'recurring decimal')

(b) $\sqrt{5} = 2.2360679 \dots$ (never ending but no repeating patterns)

The set of all real numbers is denoted by \mathbb{R} .

A **rational number** is a real number that can be expressed as the quotient (fraction) of an integer and a non-zero integer.

The set of all rational numbers is denoted by \mathbb{Q} . ('Q' for 'quotient'.)

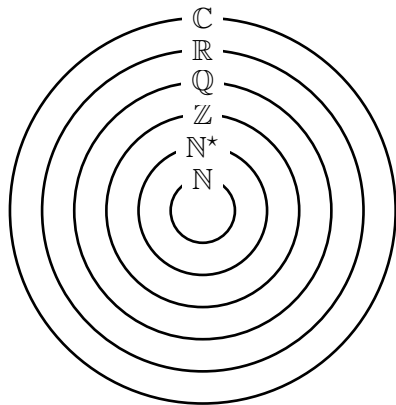
$$\mathbb{Q} = \{q \in \mathbb{R} : \exists n, d \in \mathbb{Z} (q = \frac{n}{d}) \wedge (d \neq 0)\}.$$

Examples: $\frac{-3}{2}, \frac{68}{495} = \frac{136}{990} = 0.1\overline{37}$ (fraction expressions not unique).

An **irrational number** is a real number that is not rational.

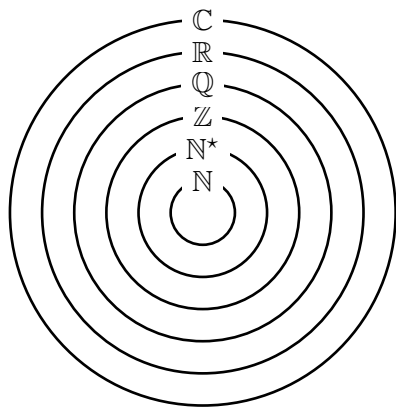
Examples: $\sqrt{2}, \pi$. All irrationals have decimal expressions like (b).

A hierarchy of number systems



[\mathbb{C} is the set of complex numbers,
not used at all in this course.]

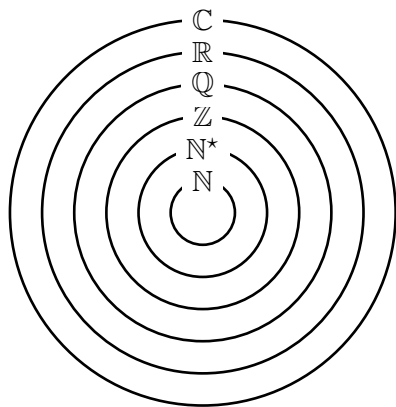
A hierarchy of number systems



The number systems \mathbb{Z} , \mathbb{N}^* and \mathbb{N} are 'discrete', as opposed to \mathbb{C} , \mathbb{R} and \mathbb{Q} which are 'continuous'.

[\mathbb{C} is the set of complex numbers, not used at all in this course.]

A hierarchy of number systems

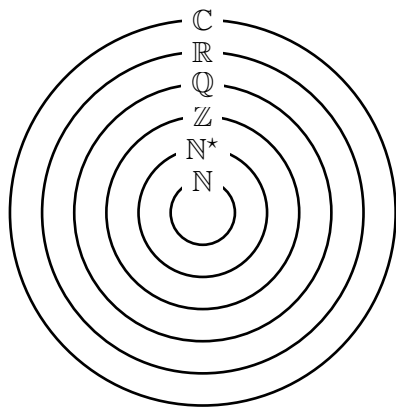


[\mathbb{C} is the set of complex numbers,
not used at all in this course.]

The number systems \mathbb{Z} , \mathbb{N}^* and \mathbb{N} are 'discrete', as opposed to \mathbb{C} , \mathbb{R} and \mathbb{Q} which are 'continuous'.

A system is **continuous** if and only if between any two distinct numbers in the system another number in the system can be found.

A hierarchy of number systems



[\mathbb{C} is the set of complex numbers, not used at all in this course.]

The number systems \mathbb{Z} , \mathbb{N}^* and \mathbb{N} are 'discrete', as opposed to \mathbb{C} , \mathbb{R} and \mathbb{Q} which are 'continuous'.

A system is **continuous** if and only if between any two distinct numbers in the system another number in the system can be found.

For example in \mathbb{R} , between 1 and 2 we have 1.5 (and many other numbers), but in \mathbb{N} there is no number between 1 and 2.

Positional notation

The **decimal digits** are the members of the set $\mathbb{Z}_{10} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

Positional notation

The **decimal digits** are the members of the set

$$\mathbb{Z}_{10} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

As we have all known since primary school, any natural number can be written using decimal digits and *positional notation*.

Positional notation

The **decimal digits** are the members of the set

$$\mathbb{Z}_{10} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

As we have all known since primary school, any natural number can be written using decimal digits and *positional notation*.

E.g. 1356 means $1 \times 10^3 + 3 \times 10^2 + 5 \times 10^1 + 6 \times 10^0$.

Positional notation

The **decimal digits** are the members of the set

$$\mathbb{Z}_{10} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

As we have all known since primary school, any natural number can be written using decimal digits and *positional notation*.

E.g. 1356 means $1 \times 10^3 + 3 \times 10^2 + 5 \times 10^1 + 6 \times 10^0$.

The same can be done with **binary digits** in $\mathbb{Z}_2 = \{0, 1\}$.

Positional notation

The **decimal digits** are the members of the set

$$\mathbb{Z}_{10} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

As we have all known since primary school, any natural number can be written using decimal digits and *positional notation*.

E.g. 1356 means $1 \times 10^3 + 3 \times 10^2 + 5 \times 10^1 + 6 \times 10^0$.

The same can be done with **binary digits** in $\mathbb{Z}_2 = \{0, 1\}$.

E.g. Using binary notation 10111 means

$$1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 16 + 4 + 2 + 1 = 23.$$

Positional notation

The **decimal digits** are the members of the set

$$\mathbb{Z}_{10} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

As we have all known since primary school, any natural number can be written using decimal digits and *positional notation*.

E.g. 1356 means $1 \times 10^3 + 3 \times 10^2 + 5 \times 10^1 + 6 \times 10^0$.

The same can be done with **binary digits** in $\mathbb{Z}_2 = \{0, 1\}$.

E.g. Using binary notation 10111 means

$$1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 16 + 4 + 2 + 1 = 23.$$

We write $10111_2 = 23_{10} = 23$, where the subscripts denote 'base':

Positional notation

The **decimal digits** are the members of the set

$$\mathbb{Z}_{10} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

As we have all known since primary school, any natural number can be written using decimal digits and *positional notation*.

E.g. 1356 means $1 \times 10^3 + 3 \times 10^2 + 5 \times 10^1 + 6 \times 10^0$.

The same can be done with **binary digits** in $\mathbb{Z}_2 = \{0, 1\}$.

E.g. Using binary notation 10111 means

$$1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 16 + 4 + 2 + 1 = 23.$$

We write $10111_2 = 23_{10} = 23$, where the subscripts denote 'base':

For $b \in \mathbb{N}$, $b > 1$ the **base b representation** of $n \in \mathbb{N}$ is

$$n = (d_1 d_2 \dots d_N)_b, \text{ meaning } d_1 \times b^{N-1} + d_2 \times b^{N-2} + \dots + d_N \times b^0.$$

*There are only 10 types of people:
those who can count in base 2, and those who can't.*

Common bases

Decimal ($b = 10$).

Common bases

Decimal ($b = 10$).

Binary ($b = 2$).

Example: $10101_2 = 2^4 + 2^2 + 1 = 21$.

Common bases

Decimal ($b = 10$).

Binary ($b = 2$).

Example: $10101_2 = 2^4 + 2^2 + 1 = 21$.

Octal ($b = 8$).

Example: $357_8 = (3(8^2) + 5(8) + 7)_{10} = 192 + 40 + 7 = 239$.

Common bases

Decimal ($b = 10$).

Binary ($b = 2$).

Example: $10101_2 = 2^4 + 2^2 + 1 = 21$.

Octal ($b = 8$).

Example: $357_8 = (3(8^2) + 5(8) + 7)_{10} = 192 + 40 + 7 = 239$.

Hexadecimal ('hex')

($b = 16$, $\mathbb{Z}_{16} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$).

Example: $13F = (16^2 + 3(16) + 15)_{10} = 256 + 48 + 15 = 319$.

Common bases

Decimal ($b = 10$).

Binary ($b = 2$).

Example: $10101_2 = 2^4 + 2^2 + 1 = 21$.

Octal ($b = 8$).

Example: $357_8 = (3(8^2) + 5(8) + 7)_{10} = 192 + 40 + 7 = 239$.

Hexadecimal ('hex')

($b = 16$, $\mathbb{Z}_{16} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$).

Example: $13F = (16^2 + 3(16) + 15)_{10} = 256 + 48 + 15 = 319$.

Binary is widely used internally by computers.

Common bases

Decimal ($b = 10$).

Binary ($b = 2$).

Example: $10101_2 = 2^4 + 2^2 + 1 = 21$.

Octal ($b = 8$).

Example: $357_8 = (3(8^2) + 5(8) + 7)_{10} = 192 + 40 + 7 = 239$.

Hexadecimal ('hex')

($b = 16$, $\mathbb{Z}_{16} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$).

Example: $13F = (16^2 + 3(16) + 15)_{10} = 256 + 48 + 15 = 319$.

Binary is widely used internally by computers.

Octal and hex are mostly used as easily translatable shorthands for binary, as we shall see.

Addition and subtraction in base b

In base 10:

Addition and subtraction in base b

In base 10:

$$\begin{array}{r} 123 + 678 : \\ \hline 801 \\ \hline 11 \end{array}$$

Addition and subtraction in base b

In base 10:

$$123 + 678 : \quad \begin{array}{r} 123 \\ + 678 \\ \hline 801 \\ \hline 11 \end{array}$$

$$123 - 78 : \quad \begin{array}{r} 123 \\ - 78 \\ \hline 45 \end{array}$$

Addition and subtraction in base b

In base 10:

$$123 + 678 : \quad \begin{array}{r} 123 \\ + 678 \\ \hline 801 \\ \hline 11 \end{array}$$

$$123 - 78 : \quad \begin{array}{r} 123 \\ - 78 \\ \hline 45 \end{array}$$

In base 2:

Addition and subtraction in base b

In base 10:

$$123 + 678 : \quad \begin{array}{r} 123 \\ + 678 \\ \hline 801 \\ 11 \end{array}$$

$$123 - 78 : \quad \begin{array}{r} 123 \\ - 78 \\ \hline 45 \end{array}$$

In base 2:

$$111_2 + 10_2 : \quad \begin{array}{r} 111 \\ + 10 \\ \hline 1001 \\ 1 \end{array}$$

Addition and subtraction in base b

In base 10:

$$123 + 678 : \quad \begin{array}{r} 123 \\ + 678 \\ \hline 801 \\ 11 \end{array}$$

$$123 - 78 : \quad \begin{array}{r} 123 \\ - 78 \\ \hline 45 \end{array}$$

In base 2:

$$111_2 + 10_2 : \quad \begin{array}{r} 111 \\ + 10 \\ \hline 1001 \\ 1 \end{array}$$

$$1010_2 - 111_2 : \quad \begin{array}{r} 1010 \\ - 111 \\ \hline 0011 \end{array}$$

Addition and subtraction in base b

In base 10:

$$123 + 678 : \quad \begin{array}{r} 123 \\ + 678 \\ \hline 801 \\ \hline 11 \end{array}$$

$$123 - 78 : \quad \begin{array}{r} 123 \\ - 78 \\ \hline 45 \end{array}$$

In base 2:

$$111_2 + 10_2 : \quad \begin{array}{r} 111 \\ + 10 \\ \hline 1001 \\ \hline 1 \end{array}$$

$$1010_2 - 111_2 : \quad \begin{array}{r} 1010 \\ - 111 \\ \hline 0011 \end{array}$$

In base 16:

Addition and subtraction in base b

In base 10:

$$123 + 678 : \quad \begin{array}{r} 1 \ 2 \ 3 \\ + \ 6 \ 7 \ 8 \\ \hline 8 \ 0 \ 1 \\ \hline 1 \ 1 \end{array}$$

$$123 - 78 : \quad \begin{array}{r} 1 \ 12 \ 13 \\ - \ 1 \ 71 \ 8 \\ \hline 4 \ 5 \\ \hline \end{array}$$

In base 2:

$$111_2 + 10_2 : \quad \begin{array}{r} 1 \ 1 \ 1 \\ + \ 1 \ 0 \\ \hline 1 \ 0 \ 0 \ 1 \\ \hline 1 \end{array}$$

$$1010_2 - 111_2 : \quad \begin{array}{r} 1 \ 10 \ 11 \ 10 \\ - \ 1 \ 11 \ 11 \ 1 \\ \hline 0 \ 0 \ 1 \ 1 \\ \hline \end{array}$$

In base 16:

$$456_{16} + ABC_{16} : \quad \begin{array}{r} 4 \ 5 \ 6 \\ + \ A \ B \ C \\ \hline F \ 1 \ 2 \\ \hline 1 \ 1 \end{array}$$

Addition and subtraction in base b

In base 10:

$$123 + 678 : \quad \begin{array}{r} 1 \ 2 \ 3 \\ + \ 6 \ 7 \ 8 \\ \hline 8 \ 0 \ 1 \\ \hline 1 \ 1 \end{array}$$

$$123 - 78 : \quad \begin{array}{r} 1 \ 1^2 \ 1^3 \\ - \ 1 \ 7_1 \ 8 \\ \hline 4 \ 5 \\ \hline \end{array}$$

In base 2:

$$111_2 + 10_2 : \quad \begin{array}{r} 1 \ 1 \ 1 \\ + \ 1 \ 0 \\ \hline 1 \ 0 \ 0 \ 1 \\ \hline 1 \end{array}$$

$$1010_2 - 111_2 : \quad \begin{array}{r} 1 \ 1^0 \ 1^1 \ 1^0 \\ - \ 1 \ 1_1 \ 1_1 \ 1 \\ \hline 0 \ 0 \ 1 \ 1 \\ \hline \end{array}$$

In base 16:

$$456_{16} + ABC_{16} : \quad \begin{array}{r} 4 \ 5 \ 6 \\ + \ A \ B \ C \\ \hline F \ 1 \ 2 \\ \hline 1 \ 1 \end{array}$$

$$F12_{16} - ABC_{16} : \quad \begin{array}{r} F \ 1^1 \ 1^2 \\ - \ A_1 \ B_1 \ C \\ \hline 4 \ 5 \ 6 \\ \hline \end{array}$$

Addition and subtraction in base b

In base 10:

$$123 + 678 : \begin{array}{r} 1\ 2\ 3 \\ +\ 6\ 7\ 8 \\ \hline 8\ 0\ 1 \\ \hline 1\ 1 \end{array}$$

$$123 - 78 : \begin{array}{r} 1\ 12\ 13 \\ -\ 1\ 71\ 8 \\ \hline 4\ 5 \end{array}$$

In base 2:

$$111_2 + 10_2 : \begin{array}{r} 1\ 1\ 1 \\ +\ 1\ 0 \\ \hline 1\ 0\ 0\ 1 \\ \hline 1 \end{array}$$

$$1010_2 - 111_2 : \begin{array}{r} 1\ 10\ 11\ 10 \\ -\ 1\ 11\ 11\ 1 \\ \hline 0\ 0\ 1\ 1 \end{array}$$

In base 16:

$$456_{16} + ABC_{16} : \begin{array}{r} 4\ 5\ 6 \\ +\ A\ B\ C \\ \hline F\ 1\ 2 \\ \hline 1\ 1 \end{array}$$

$$F12_{16} - ABC_{16} : \begin{array}{r} F\ 11\ 12 \\ -\ A_1\ B_1\ C \\ \hline 4\ 5\ 6 \end{array}$$

e.g. in right hand columns: $6 + C$ means $6 + 12_{10} = 18_{10} = 12_{16}$;

Addition and subtraction in base b

In base 10:

$$123 + 678 : \begin{array}{r} 1\ 2\ 3 \\ +\ 6\ 7\ 8 \\ \hline 8\ 0\ 1 \\ \hline 1\ 1 \end{array}$$

$$123 - 78 : \begin{array}{r} 1\ 12\ 13 \\ -\ 1\ 71\ 8 \\ \hline 4\ 5 \end{array}$$

In base 2:

$$111_2 + 10_2 : \begin{array}{r} 1\ 1\ 1 \\ +\ 1\ 0 \\ \hline 1\ 0\ 0\ 1 \\ \hline 1 \end{array}$$

$$1010_2 - 111_2 : \begin{array}{r} 1\ 10\ 11\ 10 \\ -\ 1\ 11\ 11\ 1 \\ \hline 0\ 0\ 1\ 1 \end{array}$$

In base 16:

$$456_{16} + ABC_{16} : \begin{array}{r} 4\ 5\ 6 \\ +\ A\ B\ C \\ \hline F\ 1\ 2 \\ \hline 1\ 1 \end{array}$$

$$F12_{16} - ABC_{16} : \begin{array}{r} F\ 11\ 12 \\ -\ A_1\ B_1\ C \\ \hline 4\ 5\ 6 \end{array}$$

e.g. in right hand columns: $6 + C$ means $6 + 12_{10} = 18_{10} = 12_{16}$;
 $12 - C$ means $18_{10} - 12_{10} = 6$.

Converting between bases: bit grouping

Base 2 to base 8: grouping blocks of 3 bits. Example:

$$\begin{array}{ccc} \underbrace{110} & \underbrace{001} & \underbrace{100} \\ 6 & 1 & 4 \end{array}$$

$$110001100_2 = 614_8.$$

Converting between bases: bit grouping

Base 2 to base 8: grouping blocks of 3 bits. Example:

$$\underbrace{110}_6 \quad \underbrace{001}_1 \quad \underbrace{100}_4 \quad 110001100_2 = 614_8.$$

Base 8 to base 2: reverse grouping blocks of 3 bits. Example:

$$\underbrace{7}_{111} \quad \underbrace{0}_{000} \quad \underbrace{3}_{011} \quad 703_8 = 111000011_2.$$

Converting between bases: bit grouping

Base 2 to base 8: grouping blocks of 3 bits. Example:

$$\begin{array}{ccc} \underbrace{110} & \underbrace{001} & \underbrace{100} \\ 6 & 1 & 4 \end{array} \quad 110001100_2 = 614_8.$$

Base 8 to base 2: reverse grouping blocks of 3 bits. Example:

$$\begin{array}{ccc} \underbrace{7} & \underbrace{0} & \underbrace{3} \\ 111 & 000 & 011 \end{array} \quad 703_8 = 111000011_2.$$

Base 2 to base 16: grouping blocks of 4 bits. Example:

$$\begin{array}{ccc} \underbrace{1010} & \underbrace{0101} & \underbrace{1001} \\ A & 5 & 9 \end{array} \quad 101001011001_2 = A59_{16}.$$

Converting between bases: bit grouping

Base 2 to base 8: grouping blocks of 3 bits. Example:

$$\underbrace{110}_6 \quad \underbrace{001}_1 \quad \underbrace{100}_4 \quad 110001100_2 = 614_8.$$

Base 8 to base 2: reverse grouping blocks of 3 bits. Example:

$$\underbrace{7}_{111} \quad \underbrace{0}_{000} \quad \underbrace{3}_{011} \quad 703_8 = 111000011_2.$$

Base 2 to base 16: grouping blocks of 4 bits. Example:

$$\underbrace{1010}_A \quad \underbrace{0101}_5 \quad \underbrace{1001}_9 \quad 101001011001_2 = A59_{16}.$$

Base 16 to base 2: reverse grouping blocks of 4 bits. Example:

$$\underbrace{7}_{0111} \quad \underbrace{F}_{1111} \quad \underbrace{8}_{1000} \quad 7F8_{16} = 011111111000_2.$$

Binary terminology in computing

A digit in base 2 is called a **bit** (contraction of **b**inary **dig**it).

Binary terminology in computing

A digit in base 2 is called a **bit** (contraction of **b**inary **dig**it).
The term was first used in information theory by Tukey in 1948.

Binary terminology in computing

A digit in base 2 is called a **bit** (contraction of **b**inary **dig**it).
The term was first used in information theory by Tukey in 1948.

A block of 8 bits is called a **byte** (play on 'bit' ; 'y' avoids confusion).

Binary terminology in computing

A digit in base 2 is called a **bit** (contraction of **b**inary **dig**it).
The term was first used in information theory by Tukey in 1948.

A block of 8 bits is called a **byte** (play on 'bit' ; 'y' avoids confusion).
The term was introduced by IBM around the early 60's when they started using an 8-bit character code known as EBCDIC.

Binary terminology in computing

A digit in base 2 is called a **bit** (contraction of **b**inary **dig**it).
The term was first used in information theory by Tukey in 1948.

A block of 8 bits is called a **byte** (play on 'bit' ; 'y' avoids confusion).
The term was introduced by IBM around the early 60's when they started using an 8-bit character code known as EBCDIC.

A block of 4 bits is called a **nibble**. (playful — half a bi(y)te).

Binary terminology in computing

A digit in base 2 is called a **bit** (contraction of **b**inary **dig**it).
The term was first used in information theory by Tukey in 1948.

A block of 8 bits is called a **byte** (play on 'bit' ; 'y' avoids confusion).
The term was introduced by IBM around the early 60's when they started using an 8-bit character code known as EBCDIC.

A block of 4 bits is called a **nibble**. (playful — half a bi(y)te).

A sequence of several adjacent bytes is called a **word**.

Binary terminology in computing

A digit in base 2 is called a **bit** (contraction of **b**inary **dig**it).
The term was first used in information theory by Tukey in 1948.

A block of 8 bits is called a **byte** (play on 'bit' ; 'y' avoids confusion).
The term was introduced by IBM around the early 60's when they started using an 8-bit character code known as EBCDIC.

A block of 4 bits is called a **nibble**. (playful — half a bi(y)te).

A sequence of several adjacent bytes is called a **word**.
The number of bytes varies, depending on the purpose of the word.

Binary terminology in computing

A digit in base 2 is called a **bit** (contraction of **b**inary **dig**it).
The term was first used in information theory by Tukey in 1948.

A block of 8 bits is called a **byte** (play on 'bit' ; 'y' avoids confusion).
The term was introduced by IBM around the early 60's when they started using an 8-bit character code known as EBCDIC.

A block of 4 bits is called a **nibble**. (playful — half a bi(y)te).

A sequence of several adjacent bytes is called a **word**.
The number of bytes varies, depending on the purpose of the word.
For example a 2-byte word can store integers in the range from 0 to $2^{16} - 1 = 65\,535$. (0000 0000 0000 0000₂ to 1111 1111 1111 1111₂.)

More than \mathbb{N} : \mathbb{Z}

To store integers in a range that includes negative integers , an extra **sign bit** is used.

More than \mathbb{N} : \mathbb{Z}

To store integers in a range that includes negative integers , an extra **sign bit** is used. The implementation is designed to facilitate addition and subtraction.

More than \mathbb{N} : \mathbb{Z}

To store integers in a range that includes negative integers, an extra **sign bit** is used. The implementation is designed to facilitate addition and subtraction. Example using nibble length:

nibble	decimal value	nibble	decimal value
0000	0	1000	-8
0001	1	1001	-7
0010	2	1010	-6
0011	3	1011	-5
0100	4	1100	-4
0101	5	1101	-3
0110	6	1110	-2
0111	7	1111	-1

More than \mathbb{N} : \mathbb{Z}

To store integers in a range that includes negative integers, an extra **sign bit** is used. The implementation is designed to facilitate addition and subtraction. Example using nibble length:

nibble	decimal value	nibble	decimal value
0000	0	1000	-8
0001	1	1001	-7
0010	2	1010	-6
0011	3	1011	-5
0100	4	1100	-4
0101	5	1101	-3
0110	6	1110	-2
0111	7	1111	-1

When the leading digit (sign bit) of the nibble is 1:

$$\text{value}(1d_2d_3d_4) = -[(\tilde{d}_2\tilde{d}_3\tilde{d}_4)_2 + 001_2], \text{ where } \tilde{d} \text{ is given by } \begin{array}{c|c|c} d : & 0 & 1 \\ \hline \tilde{d} : & 1 & 0 \end{array}$$

More than \mathbb{N} : \mathbb{Z}

To store integers in a range that includes negative integers, an extra **sign bit** is used. The implementation is designed to facilitate addition and subtraction. Example using nibble length:

nibble	decimal value	nibble	decimal value
0000	0	1000	-8
0001	1	1001	-7
0010	2	1010	-6
0011	3	1011	-5
0100	4	1100	-4
0101	5	1101	-3
0110	6	1110	-2
0111	7	1111	-1

When the leading digit (sign bit) of the nibble is 1:

value($1d_2d_3d_4$) = $-[(\tilde{d}_2\tilde{d}_3\tilde{d}_4)_2 + 001_2]$, where \tilde{d} is given by

$d :$	0	1
$\tilde{d} :$	1	0

i.e. toggle all bits; add one; negate.

More than \mathbb{N} : \mathbb{Z}

To store integers in a range that includes negative integers, an extra **sign bit** is used. The implementation is designed to facilitate addition and subtraction. Example using nibble length:

nibble	decimal value	nibble	decimal value
0000	0	1000	-8
0001	1	1001	-7
0010	2	1010	-6
0011	3	1011	-5
0100	4	1100	-4
0101	5	1101	-3
0110	6	1110	-2
0111	7	1111	-1

When the leading digit (sign bit) of the nibble is 1:

value($1d_2d_3d_4$) = $-[(\tilde{d}_2\tilde{d}_3\tilde{d}_4)_2 + 001_2]$, where \tilde{d} is given by

$d :$	0	1
$\tilde{d} :$	1	0

i.e. **toggle all bits; add one; negate.**

Example: value(1101) = $-(0010 + 1)_2 = -11_2 = -3_{10}$.

More than \mathbb{Z} : \mathbb{Q}

For computer storage of any **non-zero** rational number q we need to express it using **scientific notation** with base 2.

More than \mathbb{Z} : \mathbb{Q}

For computer storage of any **non-zero** rational number q we need to express it using **scientific notation** with base 2.

For any base b this is

$$q = (-1)^s \times m \times b^n$$

More than \mathbb{Z} : \mathbb{Q}

For computer storage of any **non-zero** rational number q we need to express it using **scientific notation** with base 2.

For any base b this is

where

$$q = (-1)^s \times m \times b^n$$

- $q \in \mathbb{Q}, q \neq 0;$

More than \mathbb{Z} : \mathbb{Q}

For computer storage of any **non-zero** rational number q we need to express it using **scientific notation** with base 2.

For any base b this is

where

$$q = (-1)^s \times m \times b^n$$

- $q \in \mathbb{Q}, q \neq 0$;
- $b \in \mathbb{N}, b \geq 2$;

More than \mathbb{Z} : \mathbb{Q}

For computer storage of any **non-zero** rational number q we need to express it using **scientific notation** with base 2.

For any base b this is

where

$$q = (-1)^s \times m \times b^n$$

- $q \in \mathbb{Q}, q \neq 0$;
- $b \in \mathbb{N}, b \geq 2$;
- $s \in \{0, 1\}$, (s is the **sign bit**)

More than \mathbb{Z} : \mathbb{Q}

For computer storage of any **non-zero** rational number q we need to express it using **scientific notation** with base 2.

For any base b this is

where

$$q = (-1)^s \times m \times b^n$$

- $q \in \mathbb{Q}$, $q \neq 0$;
- $b \in \mathbb{N}$, $b \geq 2$;
- $s \in \{0, 1\}$, (s is the **sign bit**)
- $m \in \mathbb{Q}$, $1 \leq m < b$, (m is the '**mantissa**'))

More than \mathbb{Z} : \mathbb{Q}

For computer storage of any **non-zero** rational number q we need to express it using **scientific notation** with base 2.

For any base b this is

where

$$q = (-1)^s \times m \times b^n$$

- $q \in \mathbb{Q}$, $q \neq 0$;
- $b \in \mathbb{N}$, $b \geq 2$;
- $s \in \{0, 1\}$, (s is the **sign bit**)
- $m \in \mathbb{Q}$, $1 \leq m < b$, (m is the '**mantissa**') and
- $n \in \mathbb{Z}$ (n is the **exponent**).

More than \mathbb{Z} : \mathbb{Q}

For computer storage of any **non-zero** rational number q we need to express it using **scientific notation** with base 2.

For any base b this is

where

$$q = (-1)^s \times m \times b^n$$

- $q \in \mathbb{Q}$, $q \neq 0$;
- $b \in \mathbb{N}$, $b \geq 2$;
- $s \in \{0, 1\}$, (s is the **sign bit**)
- $m \in \mathbb{Q}$, $1 \leq m < b$, (m is the '**mantissa**') and
- $n \in \mathbb{Z}$ (n is the **exponent**).

Given q and b , the values of s , m and n are uniquely determined by these conditions.

More than \mathbb{Z} : \mathbb{Q}

For computer storage of any **non-zero** rational number q we need to express it using **scientific notation** with base 2.

For any base b this is

where

$$q = (-1)^s \times m \times b^n$$

- $q \in \mathbb{Q}$, $q \neq 0$;
- $b \in \mathbb{N}$, $b \geq 2$;
- $s \in \{0, 1\}$, (s is the **sign bit**)
- $m \in \mathbb{Q}$, $1 \leq m < b$, (m is the '**mantissa**') and
- $n \in \mathbb{Z}$ (n is the **exponent**).

Given q and b , the values of s , m and n are uniquely determined by these conditions.

Example in base 10: $13.5 = (-1)^0 \times 1.35 \times 10^1$.

More than \mathbb{Z} : \mathbb{Q}

For computer storage of any **non-zero** rational number q we need to express it using **scientific notation** with base 2.

For any base b this is

where

$$q = (-1)^s \times m \times b^n$$

- $q \in \mathbb{Q}$, $q \neq 0$;
- $b \in \mathbb{N}$, $b \geq 2$;
- $s \in \{0, 1\}$, (s is the **sign bit**)
- $m \in \mathbb{Q}$, $1 \leq m < b$, (m is the '**mantissa**') and
- $n \in \mathbb{Z}$ (n is the **exponent**).

Given q and b , the values of s , m and n are uniquely determined by these conditions.

Example in base 10: $13.5 = (-1)^0 \times 1.35 \times 10^1$.

To save space, we store 135 instead of 1.35 because 1.35 is the only number between 1 and 10 with digits 1, 3, 5.

Storing rationals — more examples

Examples in base 10:

- $-154 = (-1)^1 \times 1.54 \times 10^2$. Store m as 154.

Storing rationals — more examples

Examples in base 10:

- $-154 = (-1)^1 \times 1.54 \times 10^2$. Store m as 154.
- $0.031 = (-1)^0 \times 3.1 \times 10^{-2}$. Store m as 31.

Storing rationals — more examples

Examples in base 10:

- $-154 = (-1)^1 \times 1.54 \times 10^2$. Store m as 154.
- $0.031 = (-1)^0 \times 3.1 \times 10^{-2}$. Store m as 31.

The number of digits used to store m is called the **precision**.

Storing rationals — more examples

Examples in base 10:

- $-154 = (-1)^1 \times 1.54 \times 10^2$. Store m as 154.
- $0.031 = (-1)^0 \times 3.1 \times 10^{-2}$. Store m as 31.

The number of digits used to store m is called the **precision**.

Example in base 2, precision 4, 4-bit exponent n ($-8 \leq n \leq 7$):

Storing rationals — more examples

Examples in base 10:

- $-154 = (-1)^1 \times 1.54 \times 10^2$. Store m as 154.
- $0.031 = (-1)^0 \times 3.1 \times 10^{-2}$. Store m as 31.

The number of digits used to store m is called the **precision**.

Example in base 2, precision 4, 4-bit exponent n ($-8 \leq n \leq 7$):

$\underbrace{1}_s \quad \underbrace{0101}_m \quad \underbrace{0011}_n$ or, alternatively, $\underbrace{1}_s \quad \underbrace{1010}_m \quad \underbrace{0011}_n$

Storing rationals — more examples

Examples in base 10:

- $-154 = (-1)^1 \times 1.54 \times 10^2$. Store m as 154.
- $0.031 = (-1)^0 \times 3.1 \times 10^{-2}$. Store m as 31.

The number of digits used to store m is called the **precision**.

Example in base 2, precision 4, 4-bit exponent n ($-8 \leq n \leq 7$):

$$\underbrace{1}_s \quad \underbrace{0101}_m \quad \underbrace{0011}_n \quad \text{or, alternatively,} \quad \underbrace{1}_s \quad \underbrace{1010}_m \quad \underbrace{0011}_n$$

$$m = 1.01_2 = 1(2^0) + 0(2^{-1}) + 1(2^{-2}) = 1 + \frac{1}{4} = \frac{5}{4}. \quad n = 3_{10}.$$

Storing rationals — more examples

Examples in base 10:

- $-154 = (-1)^1 \times 1.54 \times 10^2$. Store m as 154.
- $0.031 = (-1)^0 \times 3.1 \times 10^{-2}$. Store m as 31.

The number of digits used to store m is called the **precision**.

Example in base 2, precision 4, 4-bit exponent n ($-8 \leq n \leq 7$):

$$\underbrace{1}_s \quad \underbrace{0101}_m \quad \underbrace{0011}_n \quad \text{or, alternatively,} \quad \underbrace{1}_s \quad \underbrace{1010}_m \quad \underbrace{0011}_n$$

$$m = 1.01_2 = 1(2^0) + 0(2^{-1}) + 1(2^{-2}) = 1 + \frac{1}{4} = \frac{5}{4}. \quad n = 3_{10}.$$

$$\text{So } q = -\frac{5}{4} \times 2^3 = -10_{10}.$$

Storing rationals — more examples

Examples in base 10:

- $-154 = (-1)^1 \times 1.54 \times 10^2$. Store m as 154.
- $0.031 = (-1)^0 \times 3.1 \times 10^{-2}$. Store m as 31.

The number of digits used to store m is called the **precision**.

Example in base 2, precision 4, 4-bit exponent n ($-8 \leq n \leq 7$):

$$\underbrace{1}_s \quad \underbrace{0101}_m \quad \underbrace{0011}_n \quad \text{or, alternatively,} \quad \underbrace{1}_s \quad \underbrace{1010}_m \quad \underbrace{0011}_n$$

$$m = 1.01_2 = 1(2^0) + 0(2^{-1}) + 1(2^{-2}) = 1 + \frac{1}{4} = \frac{5}{4}. \quad n = 3_{10}.$$

$$\text{So } q = -\frac{5}{4} \times 2^3 = -10_{10}.$$

NOTE: In practice, when using binary, the “1.” part of the mantissa m is not stored, since it is implied.

Storing rationals — more examples

Examples in base 10:

- $-154 = (-1)^1 \times 1.54 \times 10^2$. Store m as 154.
- $0.031 = (-1)^0 \times 3.1 \times 10^{-2}$. Store m as 31.

The number of digits used to store m is called the **precision**.

Example in base 2, precision 4, 4-bit exponent n ($-8 \leq n \leq 7$):

$$\underbrace{1}_s \quad \underbrace{0101}_m \quad \underbrace{0011}_n \quad \text{or, alternatively,} \quad \underbrace{1}_s \quad \underbrace{1010}_m \quad \underbrace{0011}_n$$

$$m = 1.01_2 = 1(2^0) + 0(2^{-1}) + 1(2^{-2}) = 1 + \frac{1}{4} = \frac{5}{4}. \quad n = 3_{10}.$$

$$\text{So } q = -\frac{5}{4} \times 2^3 = -10_{10}.$$

NOTE: In practice, when using binary, the “1.” part of the mantissa m is not stored, since it is implied. So in 4-bit precision 1.01_2 would be stored as 0100 (with *no* alternative).

Storing rationals — more examples

Examples in base 10:

- $-154 = (-1)^1 \times 1.54 \times 10^2$. Store m as 154.
- $0.031 = (-1)^0 \times 3.1 \times 10^{-2}$. Store m as 31.

The number of digits used to store m is called the **precision**.

Example in base 2, precision 4, 4-bit exponent n ($-8 \leq n \leq 7$):

$$\underbrace{1}_s \quad \underbrace{0101}_m \quad \underbrace{0011}_n \quad \text{or, alternatively,} \quad \underbrace{1}_s \quad \underbrace{1010}_m \quad \underbrace{0011}_n$$

$$m = 1.01_2 = 1(2^0) + 0(2^{-1}) + 1(2^{-2}) = 1 + \frac{1}{4} = \frac{5}{4}. \quad n = 3_{10}.$$

$$\text{So } q = -\frac{5}{4} \times 2^3 = -10_{10}.$$

NOTE: In practice, when using binary, the “1.” part of the mantissa m is not stored, since it is implied. So in 4-bit precision 1.01_2 would be stored as 0100 (with *no* alternative).

WARNING: With limits on precision and exponent size, some rational numbers can only be stored inaccurately, if at all.

Circuit for 1-bit addition

Adding 1-bit numbers p and q :

p	q	$p + q$	
		left bit	right bit
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Circuit for 1-bit addition

Adding 1-bit numbers p and q :

p	q	$p + q$	
		left bit	right bit
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Circuit for left bit: AND gate.

Circuit for 1-bit addition

Adding 1-bit numbers p and q :

p	q	$p + q$	
		left bit	right bit
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Circuit for left bit: AND gate.

Circuit for right bit: XOR gate.

Circuit for 1-bit addition

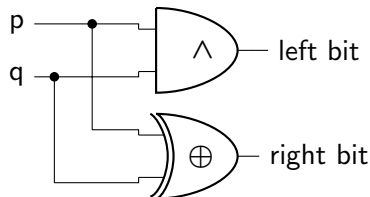
Adding 1-bit numbers p and q :

p	q	$p + q$	
		left bit	right bit
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Circuit for left bit: AND gate.

Circuit for right bit: XOR gate.

The two gates in parallel make a **half adder** (2 bits in, 2 bits out):



Circuit for multi-bit addition

Circuit for multi-bit addition

Consider binary sum at right, particularly the middle column:

$$\begin{array}{r} 111 \\ + \quad 11 \\ \hline 1010 \\ \hline 11 \end{array}$$

Circuit for multi-bit addition

Consider binary sum at right, particularly the middle column:

$$\begin{array}{r} 111 \\ + \quad 11 \\ \hline 1010 \\ \hline 11 \end{array}$$

For all but the right-most position of such a sum we need to be able to add, potentially, **three** bits

Circuit for multi-bit addition

$$\begin{array}{r}
 111 \\
 + \quad 11 \\
 \hline
 1010 \\
 \hline
 11
 \end{array}$$

Consider binary sum at right, particularly the middle column:

For all but the right-most position of such a sum we need to be able to add, potentially, **three** bits — the bits from p and q and a the carry-over bit from the position to the right (the 'carry(in)').

Circuit for multi-bit addition

$$\begin{array}{r}
 111 \\
 + \quad 11 \\
 \hline
 1010 \\
 \hline
 11
 \end{array}$$

Consider binary sum at right, particularly the middle column:

For all but the right-most position of such a sum we need to be able to add, potentially, **three** bits — the bits from p and q and a the carry-over bit from the position to the right (the 'carry(in)').

This will produce, potentially, a 2-bit number, with left output a 'carry(out)' and right output the 'result'.

Circuit for multi-bit addition

$$\begin{array}{r}
 1\ 1\ 1 \\
 + \quad 1\ 1 \\
 \hline
 1\ 0\ 1\ 0 \\
 \hline
 1\ 1
 \end{array}$$

Consider binary sum at right, particularly the middle column:

For all but the right-most position of such a sum we need to be able to add, potentially, **three** bits — the bits from p and q and a the carry-over bit from the position to the right (the 'carry(in)').

This will produce, potentially, a 2-bit number, with left output a 'carry(out)' and right output the 'result'.

p	q	carry (in)	carry (out)	result
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0

p	q	carry (in)	carry (out)	result
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Circuit for multi-bit addition

$$\begin{array}{r}
 1\ 1\ 1 \\
 + \quad 1\ 1 \\
 \hline
 1\ 0\ 1\ 0 \\
 \hline
 1\ 1
 \end{array}$$

Consider binary sum at right, particularly the middle column:

For all but the right-most position of such a sum we need to be able to add, potentially, **three** bits — the bits from p and q and a the carry-over bit from the position to the right (the 'carry(in)').

This will produce, potentially, a 2-bit number, with left output a 'carry(out)' and right output the 'result'.

p	q	carry (in)	carry (out)	result
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0

p	q	carry (in)	carry (out)	result
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

A logic circuit (3 in, 2 out) which achieves this is called a **full adder**.

Circuit for multi-bit addition

$$\begin{array}{r}
 1\ 1\ 1 \\
 + \quad 1\ 1 \\
 \hline
 1\ 0\ 1\ 0 \\
 \hline
 1\ 1
 \end{array}$$

Consider binary sum at right, particularly the middle column:

For all but the right-most position of such a sum we need to be able to add, potentially, **three** bits — the bits from p and q and a the carry-over bit from the position to the right (the 'carry(in)').

This will produce, potentially, a 2-bit number, with left output a 'carry(out)' and right output the 'result'.

p	q	carry (in)	carry (out)	result
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0

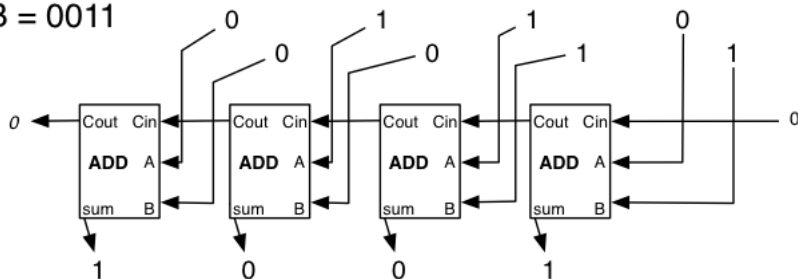
p	q	carry (in)	carry (out)	result
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

A logic circuit (3 in, 2 out) which achieves this is called a **full adder**.

For adding two n -bit numbers we use a 'cascade' of n full adders.

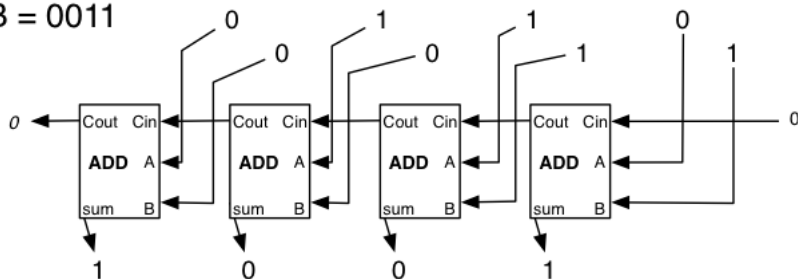
Example: 4-bit addition via a cascade of full adders

A = 0110
+ B = 0011



Example: 4-bit addition via a cascade of full adders

A = 0110
+ B = 0011



(Diagram extracted from “[Concepts in Computing](#)” course notes at Dartmouth College, Hanover, New Hampshire, USA, 2009.

<http://www.cs.dartmouth.edu/~cbk/classes/4/notes/19.php>

Circuits for Integer subtraction

In computer arithmetic, $x - y$ is treated as $x + (-y)$.

Circuits for Integer subtraction

In computer arithmetic, $x - y$ is treated as $x + (-y)$.

For integers this involves first converting y to $-y$, then adding.

Circuits for Integer subtraction

In computer arithmetic, $x - y$ is treated as $x + (-y)$.

For integers this involves first converting y to $-y$, then adding.

This works because of the special way negative integers are stored.

Circuits for Integer subtraction

In computer arithmetic, $x - y$ is treated as $x + (-y)$.

For integers this involves first converting y to $-y$, then adding.

This works because of the special way negative integers are stored.

4-bit examples:

$$\begin{array}{r}
 4 \qquad 0100 \\
 -6 \quad +1010 \\
 \hline
 -2 \quad 1110
 \end{array}$$

Circuits for Integer subtraction

In computer arithmetic, $x - y$ is treated as $x + (-y)$.

For integers this involves first converting y to $-y$, then adding.

This works because of the special way negative integers are stored.

4-bit examples:

$$\begin{array}{r} 4 \quad \quad 0100 \\ -6 \quad + 1010 \\ \hline -2 \quad \quad 1110 \end{array}$$

$$\begin{array}{r} 7 \quad \quad 0111 \\ -3 \quad + 1101 \\ \hline 4 \quad (1)0100 \end{array}$$

Circuits for Integer subtraction

In computer arithmetic, $x - y$ is treated as $x + (-y)$.

For integers this involves first converting y to $-y$, then adding.

This works because of the special way negative integers are stored.

4-bit examples:

$$\begin{array}{r} 4 \quad \quad 0100 \\ -6 \quad + 1010 \\ \hline -2 \quad \quad 1110 \end{array}$$

$$\begin{array}{r} 7 \quad \quad 0111 \\ -3 \quad + 1101 \\ \hline 4 \quad (1)0100 \end{array}$$

↑ This carry bit is ignored.

Circuits for Integer subtraction

In computer arithmetic, $x - y$ is treated as $x + (-y)$.

For integers this involves first converting y to $-y$, then adding. This works because of the special way negative integers are stored.

4-bit examples:

$$\begin{array}{r} 4 \quad 0100 \\ -6 \quad +1010 \\ \hline -2 \quad 1110 \end{array}$$

$$\begin{array}{r} 7 \quad 0111 \\ -3 \quad +1101 \\ \hline 4 \quad (1)0100 \end{array}$$

↑ This carry bit is ignored.

8-bit example:

$$\begin{array}{r} 81 \quad 01010001 \\ -98 \quad +10011110 \\ \hline -17 \quad 11101111 \end{array}$$

$$\left[\begin{array}{l} 81 = 64 + 16 + 1 \rightarrow 0101\ 0001 \\ 98 = 64 + 32 + 2 \rightarrow 0110\ 0010 \\ -98 \rightarrow 1001\ 1101 + 1 \rightarrow 1001\ 1110 \\ \hline 1110\ 1111 \rightarrow -(0001\ 0000 + 1) \\ \rightarrow -0001\ 0001 \rightarrow -17 \end{array} \right]$$

Circuits for Integer subtraction

In computer arithmetic, $x - y$ is treated as $x + (-y)$.

For integers this involves first converting y to $-y$, then adding. This works because of the special way negative integers are stored.

4-bit examples:

$$\begin{array}{r} 4 \quad 0100 \\ -6 \quad + 1010 \\ \hline -2 \quad 1110 \end{array}$$

$$\begin{array}{r} 7 \quad 0111 \\ -3 \quad + 1101 \\ \hline 4 \quad (1)0100 \end{array}$$

↑ This carry bit is ignored.

8-bit example:

$$\begin{array}{r} 81 \quad 01010001 \\ -98 \quad + 10011110 \\ \hline -17 \quad 11101111 \end{array}$$

$$\left[\begin{array}{l} 81 = 64 + 16 + 1 \rightarrow 0101\ 0001 \\ 98 = 64 + 32 + 2 \rightarrow 0110\ 0010 \\ -98 \rightarrow 1001\ 1101 + 1 \rightarrow 1001\ 1110 \\ \hline 1110\ 1111 \rightarrow -(0001\ 0000 + 1) \\ \rightarrow -0001\ 0001 \rightarrow -17 \end{array} \right]$$

So the only extra circuit required for subtraction of integers is a negation circuit (which would be needed anyway).

Circuits for Integer subtraction

In computer arithmetic, $x - y$ is treated as $x + (-y)$.

For integers this involves first converting y to $-y$, then adding. This works because of the special way negative integers are stored.

4-bit examples:

$$\begin{array}{r} 4 \quad 0100 \\ -6 \quad +1010 \\ \hline -2 \quad 1110 \end{array}$$

$$\begin{array}{r} 7 \quad 0111 \\ -3 \quad +1101 \\ \hline 4 \quad (1)0100 \end{array}$$

↑ This carry bit is ignored.

8-bit example:

$$\begin{array}{r} 81 \quad 01010001 \\ -98 \quad +10011110 \\ \hline -17 \quad 11101111 \end{array}$$

$$\left[\begin{array}{l} 81 = 64 + 16 + 1 \rightarrow 0101\ 0001 \\ 98 = 64 + 32 + 2 \rightarrow 0110\ 0010 \\ -98 \rightarrow 1001\ 1101 + 1 \rightarrow 1001\ 1110 \\ \hline 1110\ 1111 \rightarrow -(0001\ 0000 + 1) \\ \rightarrow -0001\ 0001 \rightarrow -17 \end{array} \right]$$

So the only extra circuit required for subtraction of integers is a negation circuit (which would be needed anyway).

Since negation uses toggle-plus-one, all we need is a toggle circuit for each bit, and this is just a NOT gate.

Multiplication

Example:

$$\begin{array}{r}
 1\ 0\ 1\ 1 \\
 \times 1\ 1\ 0\ 0 \\
 \hline
 0\ 0\ 0\ 0 \\
 + 0\ 0\ 0\ 0\ 0 \\
 + 1\ 0\ 1\ 1\ 0\ 0 \\
 + 1\ 0\ 1\ 1\ 0\ 0\ 0 \\
 \hline
 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0 \\
 \hline
 1\ 1\ 1\ 1
 \end{array}$$

Multiplication

Example:

$$\begin{array}{r}
 1\ 0\ 1\ 1 \\
 \times 1\ 1\ 0\ 0 \\
 \hline
 0\ 0\ 0\ 0 \\
 +\ 0\ 0\ 0\ 0\ 0 \\
 +\ 1\ 0\ 1\ 1\ 0\ 0 \\
 +\ 1\ 0\ 1\ 1\ 0\ 0\ 0 \\
 \hline
 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0 \\
 \hline
 1\ 1\ 1\ 1
 \end{array}$$

Notice that no 'multiplication tables' are required since the only multiplications used are 'times 0' which results in all zeroes, and 'times 1' which has no effect.

Multiplication

Example:

$$\begin{array}{r}
 1 \ 0 \ 1 \ 1 \\
 \times 1 \ 1 \ 0 \ 0 \\
 \hline
 0 \ 0 \ 0 \ 0 \\
 + 0 \ 0 \ 0 \ 0 \ 0 \\
 + 1 \ 0 \ 1 \ 1 \ 0 \ 0 \\
 + 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \\
 \hline
 1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \\
 \hline
 1 \ 1 \ 1 \ 1
 \end{array}$$

Notice that no 'multiplication tables' are required since the only multiplications used are 'times 0' which results in all zeroes, and 'times 1' which has no effect.

This is one of the bonuses of using binary in computers.

Multiplication

Example:

$$\begin{array}{r}
 1 \ 0 \ 1 \ 1 \\
 \times 1 \ 1 \ 0 \ 0 \\
 \hline
 0 \ 0 \ 0 \ 0 \\
 + 0 \ 0 \ 0 \ 0 \ 0 \\
 + 1 \ 0 \ 1 \ 1 \ 0 \ 0 \\
 + 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \\
 \hline
 1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \\
 \hline
 1 \ 1 \ 1 \ 1
 \end{array}$$

Notice that no ‘multiplication tables’ are required since the only multiplications used are ‘times 0’ which results in all zeroes, and ‘times 1’ which has no effect.

This is one of the bonuses of using binary in computers.

For computer implementation of multiplication (within \mathbb{N}) we only need a ‘shift’ circuit (to move bits left and append a zero on the right) and addition circuits.

Multiplication

Example:

$$\begin{array}{r}
 1\ 0\ 1\ 1 \\
 \times 1\ 1\ 0\ 0 \\
 \hline
 0\ 0\ 0\ 0 \\
 + 0\ 0\ 0\ 0\ 0 \\
 + 1\ 0\ 1\ 1\ 0\ 0 \\
 + 1\ 0\ 1\ 1\ 0\ 0\ 0 \\
 \hline
 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0 \\
 \hline
 1\ 1\ 1\ 1
 \end{array}$$

Notice that no ‘multiplication tables’ are required since the only multiplications used are ‘times 0’ which results in all zeroes, and ‘times 1’ which has no effect.

This is one of the bonuses of using binary in computers.

For computer implementation of multiplication (within \mathbb{N}) we only need a ‘shift’ circuit (to move bits left and append a zero on the right) and addition circuits.

But we also need an *algorithm* to control the process.

Algorithms

An **algorithm** is a sequence of operations on an input. The result is called an output.

Algorithms

An **algorithm** is a sequence of operations on an input. The result is called an output.

Examples:

- Addition, subtraction, multiplication, division.

Algorithms

An **algorithm** is a sequence of operations on an input. The result is called an output.

Examples:

- Addition, subtraction, multiplication, division.
- Here is very simple list processing algorithm, set out using a standard format for displaying algorithms:

Algorithms

An **algorithm** is a sequence of operations on an input. The result is called an output.

Examples:

- Addition, subtraction, multiplication, division.
- Here is very simple list processing algorithm, set out using a standard format for displaying algorithms:

Input: L: a list of students.

Algorithms

An **algorithm** is a sequence of operations on an input. The result is called an output.

Examples:

- Addition, subtraction, multiplication, division.
- Here is very simple list processing algorithm, set out using a standard format for displaying algorithms:

Input: L : a list of students.

Output: O : list of students who have submitted their assignment.

Algorithms

An **algorithm** is a sequence of operations on an input. The result is called an output.

Examples:

- Addition, subtraction, multiplication, division.
- Here is very simple list processing algorithm, set out using a standard format for displaying algorithms:

Input: L : a list of students.

Output: O : list of students who have submitted their assignment.

Method:

Set $j = 1$, $O =$ empty list. (Initialization phase)

Algorithms

An **algorithm** is a sequence of operations on an input. The result is called an output.

Examples:

- Addition, subtraction, multiplication, division.
- Here is very simple list processing algorithm, set out using a standard format for displaying algorithms:

Input: L : a list of students.

Output: O : list of students who have submitted their assignment.

Method:

Set $j = 1$, $O =$ empty list. (Initialization phase)

Loop: If $j = \text{length}(L) + 1$, stop.

Algorithms

An **algorithm** is a sequence of operations on an input. The result is called an output.

Examples:

- Addition, subtraction, multiplication, division.
- Here is very simple list processing algorithm, set out using a standard format for displaying algorithms:

Input: L : a list of students.

Output: O : list of students who have submitted their assignment.

Method:

Set $j = 1$, $O =$ empty list. (Initialization phase)

Loop: If $j = \text{length}(L) + 1$, stop.

If $L[j]$ (the j -th element in L) has submitted the assignment, append $L[j]$ to O .

Algorithms

An **algorithm** is a sequence of operations on an input. The result is called an output.

Examples:

- Addition, subtraction, multiplication, division.
- Here is very simple list processing algorithm, set out using a standard format for displaying algorithms:

Input: L : a list of students.

Output: O : list of students who have submitted their assignment.

Method:

Set $j = 1$, $O =$ empty list. (Initialization phase)

Loop: If $j = \text{length}(L) + 1$, stop.

If $L[j]$ (the j -th element in L) has submitted the assignment, append $L[j]$ to O .

Replace j by $j + 1$.

Algorithms

An **algorithm** is a sequence of operations on an input. The result is called an output.

Examples:

- Addition, subtraction, multiplication, division.
- Here is very simple list processing algorithm, set out using a standard format for displaying algorithms:

Input: L : a list of students.

Output: O : list of students who have submitted their assignment.

Method:

Set $j = 1$, $O =$ empty list. (Initialization phase)

Loop: If $j = \text{length}(L) + 1$, stop.

If $L[j]$ (the j -th element in L) has submitted the assignment, append $L[j]$ to O .

Replace j by $j + 1$.

Repeat loop.

Algorithm for binary addition

Algorithm for binary addition

This algorithm formalizes the method described earlier, which is implemented with a cascade of full adders.

Algorithm for binary addition

This algorithm formalizes the method described earlier, which is implemented with a cascade of full adders.

Contrary to notation used in some earlier examples, in this context it is usual to *number the bits starting from the rightmost bit, which is called the 0-th bit.*

Algorithm for binary addition

This algorithm formalizes the method described earlier, which is implemented with a cascade of full adders.

Contrary to notation used in some earlier examples, in this context it is usual to *number the bits starting from the rightmost bit, which is called the 0-th bit*.

So the leftmost bit of an n -bit number is its $(n - 1)$ -th bit.

Algorithm for binary addition

This algorithm formalizes the method described earlier, which is implemented with a cascade of full adders.

Contrary to notation used in some earlier examples, in this context it is usual to *number the bits starting from the rightmost bit, which is called the 0-th bit*.

So the leftmost bit of an n -bit number is its $(n - 1)$ -th bit.

Input: Two numbers p, q in base 2 with n bits.

Algorithm for binary addition

This algorithm formalizes the method described earlier, which is implemented with a cascade of full adders.

Contrary to notation used in some earlier examples, in this context it is usual to *number the bits starting from the rightmost bit, which is called the 0-th bit*.

So the leftmost bit of an n -bit number is its $(n - 1)$ -th bit.

Input: Two numbers p, q in base 2 with n bits.

Output: The sum $s = p + q$ in base 2 with $n + 1$ bits.

Algorithm for binary addition

This algorithm formalizes the method described earlier, which is implemented with a cascade of full adders.

Contrary to notation used in some earlier examples, in this context it is usual to *number the bits starting from the rightmost bit, which is called the 0-th bit*.

So the leftmost bit of an n -bit number is its $(n - 1)$ -th bit.

Input: Two numbers p, q in base 2 with n bits.

Output: The sum $s = p + q$ in base 2 with $n + 1$ bits.

Method:

Set $j = 0$, $c = 0$, and the n -th bits of p and q to 0.

Algorithm for binary addition

This algorithm formalizes the method described earlier, which is implemented with a cascade of full adders.

Contrary to notation used in some earlier examples, in this context it is usual to *number the bits starting from the rightmost bit, which is called the 0-th bit*.

So the leftmost bit of an n -bit number is its $(n - 1)$ -th bit.

Input: Two numbers p, q in base 2 with n bits.

Output: The sum $s = p + q$ in base 2 with $n + 1$ bits.

Method:

Set $j = 0$, $c = 0$, and the n -th bits of p and q to 0.

Loop: If $j = n + 1$ stop.

Algorithm for binary addition

This algorithm formalizes the method described earlier, which is implemented with a cascade of full adders.

Contrary to notation used in some earlier examples, in this context it is usual to *number the bits starting from the rightmost bit, which is called the 0-th bit*.

So the leftmost bit of an n -bit number is its $(n - 1)$ -th bit.

Input: Two numbers p, q in base 2 with n bits.

Output: The sum $s = p + q$ in base 2 with $n + 1$ bits.

Method:

Set $j = 0$, $c = 0$, and the n -th bits of p and q to 0.

Loop: If $j = n + 1$ stop.

Add the j -th bits of p and q plus c .

Algorithm for binary addition

This algorithm formalizes the method described earlier, which is implemented with a cascade of full adders.

Contrary to notation used in some earlier examples, in this context it is usual to *number the bits starting from the rightmost bit, which is called the 0-th bit*.

So the leftmost bit of an n -bit number is its $(n - 1)$ -th bit.

Input: Two numbers p, q in base 2 with n bits.

Output: The sum $s = p + q$ in base 2 with $n + 1$ bits.

Method:

Set $j = 0$, $c = 0$, and the n -th bits of p and q to 0.

Loop: If $j = n + 1$ stop.

Add the j -th bits of p and q plus c .

Store the result part as the j -th bit of s .

Algorithm for binary addition

This algorithm formalizes the method described earlier, which is implemented with a cascade of full adders.

Contrary to notation used in some earlier examples, in this context it is usual to *number the bits starting from the rightmost bit, which is called the 0-th bit*.

So the leftmost bit of an n -bit number is its $(n - 1)$ -th bit.

Input: Two numbers p, q in base 2 with n bits.

Output: The sum $s = p + q$ in base 2 with $n + 1$ bits.

Method:

Set $j = 0$, $c = 0$, and the n -th bits of p and q to 0.

Loop: If $j = n + 1$ stop.

Add the j -th bits of p and q plus c .

Store the result part as the j -th bit of s .

Store the carry part as c (replacing the old value of c).

Algorithm for binary addition

This algorithm formalizes the method described earlier, which is implemented with a cascade of full adders.

Contrary to notation used in some earlier examples, in this context it is usual to *number the bits starting from the rightmost bit, which is called the 0-th bit*.

So the leftmost bit of an n -bit number is its $(n - 1)$ -th bit.

Input: Two numbers p, q in base 2 with n bits.

Output: The sum $s = p + q$ in base 2 with $n + 1$ bits.

Method:

Set $j = 0$, $c = 0$, and the n -th bits of p and q to 0.

Loop: If $j = n + 1$ stop.

Add the j -th bits of p and q plus c .

Store the result part as the j -th bit of s .

Store the carry part as c (replacing the old value of c).

Replace j by $j + 1$.

Algorithm for binary addition

This algorithm formalizes the method described earlier, which is implemented with a cascade of full adders.

Contrary to notation used in some earlier examples, in this context it is usual to *number the bits starting from the rightmost bit, which is called the 0-th bit*.

So the leftmost bit of an n -bit number is its $(n - 1)$ -th bit.

Input: Two numbers p, q in base 2 with n bits.

Output: The sum $s = p + q$ in base 2 with $n + 1$ bits.

Method:

Set $j = 0$, $c = 0$, and the n -th bits of p and q to 0.

Loop: If $j = n + 1$ stop.

Add the j -th bits of p and q plus c .

Store the result part as the j -th bit of s .

Store the carry part as c (replacing the old value of c).

Replace j by $j + 1$.

Repeat loop

Algorithm for binary multiplication

This algorithm formalises the method demonstrated a couple of slides back (slide 22), except that addition is done progressively rather than at the end.

Algorithm for binary multiplication

This algorithm formalises the method demonstrated a couple of slides back (slide 22), except that addition is done progressively rather than at the end.

The additions may be done by the algorithm of the previous slide.

Algorithm for binary multiplication

This algorithm formalises the method demonstrated a couple of slides back (slide 22), except that addition is done progressively rather than at the end.

The additions may be done by the algorithm of the previous slide.

This algorithm also assumes the existence of a shift algorithm, which shifts all the bits of a number one place to the left and then fills the vacant rightmost place with a 0.

Algorithm for binary multiplication

This algorithm formalises the method demonstrated a couple of slides back (slide 22), except that addition is done progressively rather than at the end.

The additions may be done by the algorithm of the previous slide.

This algorithm also assumes the existence of a shift algorithm, which shifts all the bits of a number one place to the left and then fills the vacant rightmost place with a 0.

Input: Two numbers x, y in base 2 with n bits.

Algorithm for binary multiplication

This algorithm formalises the method demonstrated a couple of slides back (slide 22), except that addition is done progressively rather than at the end.

The additions may be done by the algorithm of the previous slide.

This algorithm also assumes the existence of a shift algorithm, which shifts all the bits of a number one place to the left and then fills the vacant rightmost place with a 0.

Input: Two numbers x, y in base 2 with n bits.

Output: The product $p = xy$ in base 2 with $2n$ bits.

Algorithm for binary multiplication

This algorithm formalises the method demonstrated a couple of slides back (slide 22), except that addition is done progressively rather than at the end.

The additions may be done by the algorithm of the previous slide.

This algorithm also assumes the existence of a shift algorithm, which shifts all the bits of a number one place to the left and then fills the vacant rightmost place with a 0.

Input: Two numbers x, y in base 2 with n bits.

Output: The product $p = xy$ in base 2 with $2n$ bits.

Method:

Set $j = 0$ and $p = 0$.

Algorithm for binary multiplication

This algorithm formalises the method demonstrated a couple of slides back (slide 22), except that addition is done progressively rather than at the end.

The additions may be done by the algorithm of the previous slide.

This algorithm also assumes the existence of a shift algorithm, which shifts all the bits of a number one place to the left and then fills the vacant rightmost place with a 0.

Input: Two numbers x, y in base 2 with n bits.

Output: The product $p = xy$ in base 2 with $2n$ bits.

Method:

Set $j = 0$ and $p = 0$.

Loop: If $j = n$ stop.

Algorithm for binary multiplication

This algorithm formalises the method demonstrated a couple of slides back (slide 22), except that addition is done progressively rather than at the end.

The additions may be done by the algorithm of the previous slide.

This algorithm also assumes the existence of a shift algorithm, which shifts all the bits of a number one place to the left and then fills the vacant rightmost place with a 0.

Input: Two numbers x, y in base 2 with n bits.

Output: The product $p = xy$ in base 2 with $2n$ bits.

Method:

Set $j = 0$ and $p = 0$.

Loop: If $j = n$ stop.

If the j -th bit of y is 1 then replace p by $p + x$.

Algorithm for binary multiplication

This algorithm formalises the method demonstrated a couple of slides back (slide 22), except that addition is done progressively rather than at the end.

The additions may be done by the algorithm of the previous slide.

This algorithm also assumes the existence of a shift algorithm, which shifts all the bits of a number one place to the left and then fills the vacant rightmost place with a 0.

Input: Two numbers x, y in base 2 with n bits.

Output: The product $p = xy$ in base 2 with $2n$ bits.

Method:

Set $j = 0$ and $p = 0$.

Loop: If $j = n$ stop.

If the j -th bit of y is 1 then replace p by $p + x$.

Shift x .

Algorithm for binary multiplication

This algorithm formalises the method demonstrated a couple of slides back (slide 22), except that addition is done progressively rather than at the end.

The additions may be done by the algorithm of the previous slide.

This algorithm also assumes the existence of a shift algorithm, which shifts all the bits of a number one place to the left and then fills the vacant rightmost place with a 0.

Input: Two numbers x, y in base 2 with n bits.

Output: The product $p = xy$ in base 2 with $2n$ bits.

Method:

Set $j = 0$ and $p = 0$.

Loop: If $j = n$ stop.

If the j -th bit of y is 1 then replace p by $p + x$.

Shift x .

Replace j by $j + 1$.

Algorithm for binary multiplication

This algorithm formalises the method demonstrated a couple of slides back (slide 22), except that addition is done progressively rather than at the end.

The additions may be done by the algorithm of the previous slide.

This algorithm also assumes the existence of a shift algorithm, which shifts all the bits of a number one place to the left and then fills the vacant rightmost place with a 0.

Input: Two numbers x, y in base 2 with n bits.

Output: The product $p = xy$ in base 2 with $2n$ bits.

Method:

Set $j = 0$ and $p = 0$.

Loop: If $j = n$ stop.

If the j -th bit of y is 1 then replace p by $p + x$.

Shift x .

Replace j by $j + 1$.

Repeat loop.

The 'mod' and 'div' operations

Given any integer n and given any natural number d , there is exactly one way to express n as an integer multiple qd of d plus a non-negative 'remainder' r less than the 'divisor' d .

The 'mod' and 'div' operations

Given any integer n and given any natural number d , there is exactly one way to express n as an integer multiple qd of d plus a non-negative 'remainder' r less than the 'divisor' d .

$$\forall n \in \mathbb{Z} \forall d \in \mathbb{N} \exists! q \in \mathbb{Z} \exists! r \in \mathbb{N}^* (n = qd + r) \wedge (0 \leq r < d)$$

The 'mod' and 'div' operations

Given any integer n and given any natural number d , there is exactly one way to express n as an integer multiple qd of d plus a non-negative 'remainder' r less than the 'divisor' d .

$$\forall n \in \mathbb{Z} \forall d \in \mathbb{N} \exists! q \in \mathbb{Z} \exists! r \in \mathbb{N}^* (n = qd + r) \wedge (0 \leq r < d)$$

We define: $q = n \text{ div } d$; $r = n \text{ mod } d$.

The 'mod' and 'div' operations

Given any integer n and given any natural number d , there is exactly one way to express n as an integer multiple qd of d plus a non-negative 'remainder' r less than the 'divisor' d .

$$\forall n \in \mathbb{Z} \forall d \in \mathbb{N} \exists! q \in \mathbb{Z} \exists! r \in \mathbb{N}^* (n = qd + r) \wedge (0 \leq r < d)$$

We define: $q = n \text{ div } d$; $r = n \text{ mod } d$.

Examples:

n	d	$n = qd + r$	q	r
60	7			

The 'mod' and 'div' operations

Given any integer n and given any natural number d , there is exactly one way to express n as an integer multiple qd of d plus a non-negative 'remainder' r less than the 'divisor' d .

$$\forall n \in \mathbb{Z} \forall d \in \mathbb{N} \exists! q \in \mathbb{Z} \exists! r \in \mathbb{N}^* (n = qd + r) \wedge (0 \leq r < d)$$

We define: $q = n \text{ div } d$; $r = n \text{ mod } d$.

Examples:

n	d	$n = qd + r$	q	r
60	7	$60 = 8(7) + 4$		

The 'mod' and 'div' operations

Given any integer n and given any natural number d , there is exactly one way to express n as an integer multiple qd of d plus a non-negative 'remainder' r less than the 'divisor' d .

$$\forall n \in \mathbb{Z} \forall d \in \mathbb{N} \exists! q \in \mathbb{Z} \exists! r \in \mathbb{N}^* (n = qd + r) \wedge (0 \leq r < d)$$

We define: $q = n \text{ div } d$; $r = n \text{ mod } d$.

Examples:

n	d	$n = qd + r$	q	r
60	7	$60 = 8(7) + 4$	$8 = 60 \text{ div } 7$	

The 'mod' and 'div' operations

Given any integer n and given any natural number d , there is exactly one way to express n as an integer multiple qd of d plus a non-negative 'remainder' r less than the 'divisor' d .

$$\forall n \in \mathbb{Z} \forall d \in \mathbb{N} \exists! q \in \mathbb{Z} \exists! r \in \mathbb{N}^* (n = qd + r) \wedge (0 \leq r < d)$$

We define: $q = n \text{ div } d$; $r = n \text{ mod } d$.

Examples:

n	d	$n = qd + r$	q	r
60	7	$60 = 8(7) + 4$	$8 = 60 \text{ div } 7$	$4 = 60 \text{ mod } 7$

The 'mod' and 'div' operations

Given any integer n and given any natural number d , there is exactly one way to express n as an integer multiple qd of d plus a non-negative 'remainder' r less than the 'divisor' d .

$$\forall n \in \mathbb{Z} \forall d \in \mathbb{N} \exists! q \in \mathbb{Z} \exists! r \in \mathbb{N}^* (n = qd + r) \wedge (0 \leq r < d)$$

We define: $q = n \text{ div } d$; $r = n \text{ mod } d$.

Examples:

n	d	$n = qd + r$	q	r
60	7	$60 = 8(7) + 4$	$8 = 60 \text{ div } 7$	$4 = 60 \text{ mod } 7$
-60	7			

The 'mod' and 'div' operations

Given any integer n and given any natural number d , there is exactly one way to express n as an integer multiple qd of d plus a non-negative 'remainder' r less than the 'divisor' d .

$$\forall n \in \mathbb{Z} \forall d \in \mathbb{N} \exists! q \in \mathbb{Z} \exists! r \in \mathbb{N}^* (n = qd + r) \wedge (0 \leq r < d)$$

We define: $q = n \text{ div } d$; $r = n \text{ mod } d$.

Examples:

n	d	$n = qd + r$	q	r
60	7	$60 = 8(7) + 4$	$8 = 60 \text{ div } 7$	$4 = 60 \text{ mod } 7$
-60	7	$-60 = -9(7) + 3$		

The 'mod' and 'div' operations

Given any integer n and given any natural number d , there is exactly one way to express n as an integer multiple qd of d plus a non-negative 'remainder' r less than the 'divisor' d .

$$\forall n \in \mathbb{Z} \forall d \in \mathbb{N} \exists! q \in \mathbb{Z} \exists! r \in \mathbb{N}^* (n = qd + r) \wedge (0 \leq r < d)$$

We define: $q = n \text{ div } d$; $r = n \text{ mod } d$.

Examples:

n	d	$n = qd + r$	q	r
60	7	$60 = 8(7) + 4$	$8 = 60 \text{ div } 7$	$4 = 60 \text{ mod } 7$
-60	7	$-60 = -9(7) + 3$	$-9 = -60 \text{ div } 7$	

The 'mod' and 'div' operations

Given any integer n and given any natural number d , there is exactly one way to express n as an integer multiple qd of d plus a non-negative 'remainder' r less than the 'divisor' d .

$$\boxed{\forall n \in \mathbb{Z} \forall d \in \mathbb{N} \exists! q \in \mathbb{Z} \exists! r \in \mathbb{N}^* (n = qd + r) \wedge (0 \leq r < d)}$$

We define: $q = n \text{ div } d$; $r = n \text{ mod } d$.

Examples:

n	d	$n = qd + r$	q	r
60	7	$60 = 8(7) + 4$	$8 = 60 \text{ div } 7$	$4 = 60 \text{ mod } 7$
-60	7	$-60 = -9(7) + 3$	$-9 = -60 \text{ div } 7$	$3 = -60 \text{ mod } 7$

The 'mod' and 'div' operations

Given any integer n and given any natural number d , there is exactly one way to express n as an integer multiple qd of d plus a non-negative 'remainder' r less than the 'divisor' d .

$$\forall n \in \mathbb{Z} \forall d \in \mathbb{N} \exists! q \in \mathbb{Z} \exists! r \in \mathbb{N}^* (n = qd + r) \wedge (0 \leq r < d)$$

We define: $q = n \text{ div } d$; $r = n \text{ mod } d$.

Examples:

n	d	$n = qd + r$	q	r
60	7	$60 = 8(7) + 4$	$8 = 60 \text{ div } 7$	$4 = 60 \text{ mod } 7$
-60	7	$-60 = -9(7) + 3$	$-9 = -60 \text{ div } 7$	$3 = -60 \text{ mod } 7$

Class examples:

- Find $87 \text{ mod } 13$.

The 'mod' and 'div' operations

Given any integer n and given any natural number d , there is exactly one way to express n as an integer multiple qd of d plus a non-negative 'remainder' r less than the 'divisor' d .

$$\forall n \in \mathbb{Z} \forall d \in \mathbb{N} \exists! q \in \mathbb{Z} \exists! r \in \mathbb{N}^* (n = qd + r) \wedge (0 \leq r < d)$$

We define: $q = n \text{ div } d$; $r = n \text{ mod } d$.

Examples:

n	d	$n = qd + r$	q	r
60	7	$60 = 8(7) + 4$	$8 = 60 \text{ div } 7$	$4 = 60 \text{ mod } 7$
-60	7	$-60 = -9(7) + 3$	$-9 = -60 \text{ div } 7$	$3 = -60 \text{ mod } 7$

Class examples:

- Find $87 \text{ mod } 13$. Answer: 9 [since $87 = 6(13) + 9$]

The 'mod' and 'div' operations

Given any integer n and given any natural number d , there is exactly one way to express n as an integer multiple qd of d plus a non-negative 'remainder' r less than the 'divisor' d .

$$\forall n \in \mathbb{Z} \forall d \in \mathbb{N} \exists! q \in \mathbb{Z} \exists! r \in \mathbb{N}^* (n = qd + r) \wedge (0 \leq r < d)$$

We define: $q = n \text{ div } d$; $r = n \text{ mod } d$.

Examples:

n	d	$n = qd + r$	q	r
60	7	$60 = 8(7) + 4$	$8 = 60 \text{ div } 7$	$4 = 60 \text{ mod } 7$
-60	7	$-60 = -9(7) + 3$	$-9 = -60 \text{ div } 7$	$3 = -60 \text{ mod } 7$

Class examples:

- Find $87 \text{ mod } 13$. Answer: 9 [since $87 = 6(13) + 9$]
- Find $-100 \text{ div } 13$:

The 'mod' and 'div' operations

Given any integer n and given any natural number d , there is exactly one way to express n as an integer multiple qd of d plus a non-negative 'remainder' r less than the 'divisor' d .

$$\forall n \in \mathbb{Z} \forall d \in \mathbb{N} \exists! q \in \mathbb{Z} \exists! r \in \mathbb{N}^* (n = qd + r) \wedge (0 \leq r < d)$$

We define: $q = n \text{ div } d$; $r = n \text{ mod } d$.

Examples:

n	d	$n = qd + r$	q	r
60	7	$60 = 8(7) + 4$	$8 = 60 \text{ div } 7$	$4 = 60 \text{ mod } 7$
-60	7	$-60 = -9(7) + 3$	$-9 = -60 \text{ div } 7$	$3 = -60 \text{ mod } 7$

Class examples:

- Find $87 \text{ mod } 13$. Answer: 9 [since $87 = 6(13) + 9$]
- Find $-100 \text{ div } 13$: Answer: -8 [since $-100 = (-8)(13) + 4$]

The division algorithm

The ‘primary school’ method of finding quotient and remainder is to use *repeated subtraction*. This only works for non-negative n .

The division algorithm

The ‘primary school’ method of finding quotient and remainder is to use *repeated subtraction*. This only works for non-negative n .

Input: $n \in \mathbb{N}^*$ and $d \in \mathbb{N}$.

Output: $q = n \operatorname{div} d$ and $r = n \bmod d$.

The division algorithm

The ‘primary school’ method of finding quotient and remainder is to use *repeated subtraction*. This only works for non-negative n .

Input: $n \in \mathbb{N}^*$ and $d \in \mathbb{N}$.

Output: $q = n \operatorname{div} d$ and $r = n \bmod d$.

Method:

Set $r = n$, $q = 0$.

The division algorithm

The ‘primary school’ method of finding quotient and remainder is to use *repeated subtraction*. This only works for non-negative n .

Input: $n \in \mathbb{N}^*$ and $d \in \mathbb{N}$.

Output: $q = n \operatorname{div} d$ and $r = n \operatorname{mod} d$.

Method:

Set $r = n$, $q = 0$.

Loop: If $r < d$ stop.

The division algorithm

The ‘primary school’ method of finding quotient and remainder is to use *repeated subtraction*. This only works for non-negative n .

Input: $n \in \mathbb{N}^*$ and $d \in \mathbb{N}$.

Output: $q = n \operatorname{div} d$ and $r = n \bmod d$.

Method:

Set $r = n$, $q = 0$.

Loop: If $r < d$ stop.

Replace r by $r - d$.

The division algorithm

The ‘primary school’ method of finding quotient and remainder is to use *repeated subtraction*. This only works for non-negative n .

Input: $n \in \mathbb{N}^*$ and $d \in \mathbb{N}$.

Output: $q = n \operatorname{div} d$ and $r = n \operatorname{mod} d$.

Method:

Set $r = n$, $q = 0$.

Loop: If $r < d$ stop.

Replace r by $r - d$.

Replace q by $q + 1$.

The division algorithm

The ‘primary school’ method of finding quotient and remainder is to use *repeated subtraction*. This only works for non-negative n .

Input: $n \in \mathbb{N}^*$ and $d \in \mathbb{N}$.

Output: $q = n \operatorname{div} d$ and $r = n \bmod d$.

Method:

Set $r = n$, $q = 0$.

Loop: If $r < d$ stop.

Replace r by $r - d$.

Replace q by $q + 1$.

Repeat loop

The division algorithm

The ‘primary school’ method of finding quotient and remainder is to use *repeated subtraction*. This only works for non-negative n .

Input: $n \in \mathbb{N}^*$ and $d \in \mathbb{N}$.

Output: $q = n \operatorname{div} d$ and $r = n \bmod d$.

Method:

Set $r = n$, $q = 0$.

Loop: If $r < d$ stop.

Replace r by $r - d$.

Replace q by $q + 1$.

Repeat loop

Some small modifications to the algorithm allow it cope also with negative n .

The division algorithm

The ‘primary school’ method of finding quotient and remainder is to use *repeated subtraction*. This only works for non-negative n .

Input: $n \in \mathbb{N}^*$ and $d \in \mathbb{N}$.

Output: $q = n \operatorname{div} d$ and $r = n \bmod d$.

Method:

Set $r = n$, $q = 0$.

Loop: If $r < d$ stop.

Replace r by $r - d$.

Replace q by $q + 1$.

Repeat loop

Some small modifications to the algorithm allow it cope also with negative n .

Could you make them?

Computability



A problem is called **computable** if there is an algorithm (encodable in a 'Turing machine') to solve it.

Computability



A problem is called **computable** if there is an algorithm (encodable in a 'Turing machine') to solve it.

A Turing machine (Alan Turing 1912 - 1954) is a machine that reads a (possibly infinite) sequence of binary digits one digit at a time, can overwrite a digit, and can move to the next or the preceding digit, and change 'state'. It has a finite but unlimited memory of states, and its action at each step is determined by its current state and current digit of the sequence.

Computability



A problem is called **computable** if there is an algorithm (encodable in a 'Turing machine') to solve it.

A Turing machine (Alan Turing 1912 - 1954) is a machine that reads a (possibly infinite) sequence of binary digits one digit at a time, can overwrite a digit, and can move to the next or the preceding digit, and change 'state'. It has a finite but unlimited memory of states, and its action at each step is determined by its current state and current digit of the sequence.

Turing machines are thought experiments representing any computer that can possibly exist. Actual computers are faster but can't do more!

Computability



A problem is called **computable** if there is an algorithm (encodable in a 'Turing machine') to solve it.

A Turing machine (Alan Turing 1912 - 1954) is a machine that reads a (possibly infinite) sequence of binary digits one digit at a time, can overwrite a digit, and can move to the next or the preceding digit, and change 'state'. It has a finite but unlimited memory of states, and its action at each step is determined by its current state and current digit of the sequence.

Turing machines are thought experiments representing any computer than can possibly exist. Actual computers are faster but can't do more!

Some problems (many in fact) are not computable, e.g. find whether or not a diophantine equation has a solution (Hilbert's 10th problem).

Computability



A problem is called **computable** if there is an algorithm (encodable in a 'Turing machine') to solve it.

A Turing machine (Alan Turing 1912 - 1954) is a machine that reads a (possibly infinite) sequence of binary digits one digit at a time, can overwrite a digit, and can move to the next or the preceding digit, and change 'state'. It has a finite but unlimited memory of states, and its action at each step is determined by its current state and current digit of the sequence.

Turing machines are thought experiments representing any computer that can possibly exist. Actual computers are faster but can't do more!

Some problems (many in fact) are not computable, e.g. find whether or not a diophantine equation has a solution (Hilbert's 10th problem). Martin Davis has a very readable article:

http://www.maa.org/sites/default/files/pdf/upload_library/22/Ford/MartinDavis.pdf

Congruence modulo n

Let $n \in \mathbb{N}$. The **congruence modulo n** relation $R_n \subseteq \mathbb{Z} \times \mathbb{Z}$ is defined by

$$aR_nb \iff \exists k \in \mathbb{Z} ; a = b + kn.$$

We write **$a \equiv b \pmod{n}$** .

Congruence modulo n

Let $n \in \mathbb{N}$. The **congruence modulo n** relation $R_n \subseteq \mathbb{Z} \times \mathbb{Z}$ is defined by

$$aR_nb \iff \exists k \in \mathbb{Z} ; a = b + kn.$$

We write **$a \equiv b \pmod{n}$** .

Example: $-17 \equiv 15 \pmod{8}$ since $-17 = 15 + (-4)8$.

Congruence modulo n

Let $n \in \mathbb{N}$. The **congruence modulo n** relation $R_n \subseteq \mathbb{Z} \times \mathbb{Z}$ is defined by

$$aR_nb \iff \exists k \in \mathbb{Z} ; a = b + kn.$$

We write **$a \equiv b \pmod{n}$** .

Example: $-17 \equiv 15 \pmod{8}$ since $-17 = 15 + (-4)8$.

Lemma:

$$\forall a, b \in \mathbb{Z} \forall n \in \mathbb{N} [a \equiv b \pmod{n}] \iff [a \bmod n = b \bmod n]$$

Congruence modulo n

Let $n \in \mathbb{N}$. The **congruence modulo n** relation $R_n \subseteq \mathbb{Z} \times \mathbb{Z}$ is defined by

$$aR_nb \iff \exists k \in \mathbb{Z} ; a = b + kn.$$

We write **$a \equiv b \pmod{n}$** .

Example: $-17 \equiv 15 \pmod{8}$ since $-17 = 15 + (-4)8$.

Lemma:

$$\forall a, b \in \mathbb{Z} \forall n \in \mathbb{N} [a \equiv b \pmod{n}] \iff [a \bmod n = b \bmod n]$$

Examples: $[-17 \equiv 15 \pmod{8}] \implies [-17 \bmod 8 = 15 \bmod 8]$.

Congruence modulo n

Let $n \in \mathbb{N}$. The **congruence modulo n** relation $R_n \subseteq \mathbb{Z} \times \mathbb{Z}$ is defined by

$$aR_nb \iff \exists k \in \mathbb{Z} ; a = b + kn.$$

We write **$a \equiv b \pmod{n}$** .

Example: $-17 \equiv 15 \pmod{8}$ since $-17 = 15 + (-4)8$.

Lemma:

$$\forall a, b \in \mathbb{Z} \forall n \in \mathbb{N} [a \equiv b \pmod{n}] \iff [a \bmod n = b \bmod n]$$

Examples: $[-17 \equiv 15 \pmod{8}] \implies [-17 \bmod 8 = 15 \bmod 8]$.

$[493 \bmod 7 = 3 = 780 \bmod 7] \implies [493 \equiv 780 \pmod{7}]$

Congruence modulo n

Let $n \in \mathbb{N}$. The **congruence modulo n** relation $R_n \subseteq \mathbb{Z} \times \mathbb{Z}$ is defined by

$$aR_nb \iff \exists k \in \mathbb{Z} ; a = b + kn.$$

We write **$a \equiv b \pmod{n}$** .

Example: $-17 \equiv 15 \pmod{8}$ since $-17 = 15 + (-4)8$.

Lemma:

$$\forall a, b \in \mathbb{Z} \forall n \in \mathbb{N} \quad [a \equiv b \pmod{n}] \iff [a \bmod n = b \bmod n]$$

Examples: $[-17 \equiv 15 \pmod{8}] \implies [-17 \bmod 8 = 15 \bmod 8]$.

$[493 \bmod 7 = 3 = 780 \bmod 7] \implies [493 \equiv 780 \pmod{7}]$

For any $n \in \mathbb{N}$ and any $a \in \mathbb{Z}$ the **congruence class $[a]_n$** (or 'equivalence class') of a modulo n is defined by

$$[a]_n = \{m \in \mathbb{Z} : m \equiv a \pmod{n}\}.$$

Congruence modulo n

Let $n \in \mathbb{N}$. The **congruence modulo n** relation $R_n \subseteq \mathbb{Z} \times \mathbb{Z}$ is defined by

$$aR_nb \iff \exists k \in \mathbb{Z} ; a = b + kn.$$

We write **$a \equiv b \pmod{n}$** .

Example: $-17 \equiv 15 \pmod{8}$ since $-17 = 15 + (-4)8$.

Lemma:

$$\forall a, b \in \mathbb{Z} \forall n \in \mathbb{N} \quad [a \equiv b \pmod{n}] \iff [a \bmod n = b \bmod n]$$

Examples: $[-17 \equiv 15 \pmod{8}] \implies [-17 \bmod 8 = 15 \bmod 8]$.

$[493 \bmod 7 = 3 = 780 \bmod 7] \implies [493 \equiv 780 \pmod{7}]$

For any $n \in \mathbb{N}$ and any $a \in \mathbb{Z}$ the **congruence class $[a]_n$** (or 'equivalence class') of a modulo n is defined by

$$[a]_n = \{m \in \mathbb{Z} : m \equiv a \pmod{n}\}.$$

Example: $[13]_5 = \{\dots, -7, -2, 3, 8, 13, \dots\}$.

Congruence modulo n

Let $n \in \mathbb{N}$. The **congruence modulo n** relation $R_n \subseteq \mathbb{Z} \times \mathbb{Z}$ is defined by

$$aR_nb \iff \exists k \in \mathbb{Z} ; a = b + \underline{kn}.$$

We write **$a \equiv b \pmod{n}$** .

Example: $-17 \equiv 15 \pmod{8}$ since $-17 = 15 + (-4)8$.

Lemma:

$$\forall a, b \in \mathbb{Z} \forall n \in \mathbb{N} [a \equiv b \pmod{n}] \iff [a \bmod n = b \bmod n]$$

Examples: $[-17 \equiv 15 \pmod{8}] \implies [-17 \bmod 8 = 15 \bmod 8]$.

$[493 \bmod 7 = 3 = 780 \bmod 7] \implies [493 \equiv 780 \pmod{7}]$

For any $n \in \mathbb{N}$ and any $a \in \mathbb{Z}$ the **congruence class $[a]_n$** (or 'equivalence class') of a modulo n is defined by

$$[a]_n = \{m \in \mathbb{Z} : m \equiv a \pmod{n}\}.$$

Example: $[13]_5 = \{\dots, -7, -2, 3, 8, 13, \dots\}$.

Lemma: R_n induces the partition $\{[0]_n, [1]_n, \dots, [n-1]_n\}$ on \mathbb{Z}

Modular arithmetic

Theorem: For any $a_1, a_2, b_1, b_2 \in \mathbb{Z}$ and $n \in \mathbb{N}$:

$$\boxed{\begin{array}{l} \underline{a_1} \equiv \underline{a_2} \pmod{n} \\ \underline{b_1} \equiv \underline{b_2} \pmod{n} \end{array}} \implies \boxed{\begin{array}{l} a_1 \pm b_1 \equiv a_2 \pm b_2 \pmod{n} \\ a_1 \times b_1 \equiv a_2 \times b_2 \pmod{n} \end{array}}$$

Modular arithmetic

Theorem: For any $a_1, a_2, b_1, b_2 \in \mathbb{Z}$ and $n \in \mathbb{N}$:

$$\begin{array}{l} a_1 \equiv a_2 \pmod{n} \\ b_1 \equiv b_2 \pmod{n} \end{array} \implies \begin{array}{l} a_1 \pm b_1 \equiv a_2 \pm b_2 \pmod{n} \\ a_1 \times b_1 \equiv a_2 \times b_2 \pmod{n} \end{array}$$

Example: Modulo 7 throughout:

$$\begin{array}{l} \underline{27} \equiv -1 \\ \underline{36} \equiv 1 \end{array} \implies \begin{array}{l} 27 + 36 \equiv -1 + 1 \equiv 0 \\ 27 \times 36 \equiv -1 \times 1 \equiv -1 \end{array}$$

Modular arithmetic

Theorem: For any $a_1, a_2, b_1, b_2 \in \mathbb{Z}$ and $n \in \mathbb{N}$:

$$\begin{array}{l} a_1 \equiv a_2 \pmod{n} \\ b_1 \equiv b_2 \pmod{n} \end{array} \implies \begin{array}{l} a_1 \pm b_1 \equiv a_2 \pm b_2 \pmod{n} \\ a_1 \times b_1 \equiv a_2 \times b_2 \pmod{n} \end{array}$$

Example: Modulo 7 throughout:

$$\begin{array}{l} 27 \equiv -1 \\ 36 \equiv 1 \end{array} \implies \begin{array}{l} 27+36 \equiv -1+1 = 0 \\ 27 \times 36 \equiv -1 \times 1 = -1 \end{array}$$

Checks:

$$\begin{array}{l} 27+36 = 63 = 0+9(7) \\ 27 \times 36 = 972 = -1+39(7) \end{array}$$

Modular arithmetic

Theorem: For any $a_1, a_2, b_1, b_2 \in \mathbb{Z}$ and $n \in \mathbb{N}$:

$$\begin{array}{l} a_1 \equiv a_2 \pmod{n} \\ b_1 \equiv b_2 \pmod{n} \end{array} \implies \begin{array}{l} a_1 \pm b_1 \equiv a_2 \pm b_2 \pmod{n} \\ a_1 \times b_1 \equiv a_2 \times b_2 \pmod{n} \end{array}$$

Example: Modulo 7 throughout:

$$\begin{array}{l} 27 \equiv -1 \\ 36 \equiv 1 \end{array} \implies \begin{array}{l} 27+36 \equiv -1+1 = 0 \\ 27 \times 36 \equiv -1 \times 1 = -1 \end{array}$$

Checks:

$$\begin{array}{l} 27+36 = 63 = 0+9(7) \\ 27 \times 36 = 972 = -1+39(7) \end{array}$$

Corollary: $\forall a, b \in \mathbb{Z}$
 $\forall m, n \in \mathbb{N}$

$$\begin{array}{l} a \times b \equiv (a \bmod n) \times (b \bmod n) \pmod{n} \\ a^m \equiv (a \bmod n)^m \pmod{n} \end{array}$$

Modular arithmetic

Theorem: For any $a_1, a_2, b_1, b_2 \in \mathbb{Z}$ and $n \in \mathbb{N}$:

$$\begin{array}{l} a_1 \equiv a_2 \pmod{n} \\ b_1 \equiv b_2 \pmod{n} \end{array} \implies \begin{array}{l} a_1 \pm b_1 \equiv a_2 \pm b_2 \pmod{n} \\ a_1 \times b_1 \equiv a_2 \times b_2 \pmod{n} \end{array}$$

Example: Modulo 7 throughout:

$$\begin{array}{l} 27 \equiv -1 \\ 36 \equiv 1 \end{array} \implies \begin{array}{l} 27+36 \equiv -1+1 = 0 \\ 27 \times 36 \equiv -1 \times 1 = -1 \end{array}$$

Checks:

$$\begin{array}{l} 27+36 = 63 = 0+9(7) \\ 27 \times 36 = 972 = -1+39(7) \end{array}$$

Corollary: $\forall a, b \in \mathbb{Z}$
 $\forall m, n \in \mathbb{N}$

$$\begin{array}{l} a \pm b \equiv (a \bmod n) \pm (b \bmod n) \pmod{n} \\ a^m \equiv (a \bmod n)^m \pmod{n} \end{array}$$

Examples: Modulo 7 throughout:

$$379 - 803 \equiv 1 - 5 = -4 \equiv 3$$

Modular arithmetic

Theorem: For any $a_1, a_2, b_1, b_2 \in \mathbb{Z}$ and $n \in \mathbb{N}$:

$$\begin{array}{l} a_1 \equiv a_2 \pmod{n} \\ b_1 \equiv b_2 \pmod{n} \end{array} \implies \begin{array}{l} a_1 \pm b_1 \equiv a_2 \pm b_2 \pmod{n} \\ a_1 \times b_1 \equiv a_2 \times b_2 \pmod{n} \end{array}$$

Example: Modulo 7 throughout:

$$\begin{array}{l} 27 \equiv -1 \\ 36 \equiv 1 \end{array} \implies \begin{array}{l} 27+36 \equiv -1+1 = 0 \\ 27 \times 36 \equiv -1 \times 1 = -1 \end{array}$$

Checks:

$$\begin{array}{l} 27+36 = 63 = 0+9(7) \\ 27 \times 36 = 972 = -1+39(7) \end{array}$$

Corollary: $\forall a, b \in \mathbb{Z}$
 $\forall m, n \in \mathbb{N}$

$$\begin{array}{l} a \pm b \equiv (a \bmod n) \pm (b \bmod n) \pmod{n} \\ a^m \equiv (a \bmod n)^m \pmod{n} \end{array}$$

Examples: Modulo 7 throughout:

$$379 - 803 \equiv 1 - 5 = -4 \equiv 3$$

$$379 \times 803 \equiv 1 \times 5 = 5$$

Modular arithmetic

Theorem: For any $a_1, a_2, b_1, b_2 \in \mathbb{Z}$ and $n \in \mathbb{N}$:

$$\begin{array}{l} a_1 \equiv a_2 \pmod{n} \\ b_1 \equiv b_2 \pmod{n} \end{array} \implies \begin{array}{l} a_1 \pm b_1 \equiv a_2 \pm b_2 \pmod{n} \\ a_1 \times b_1 \equiv a_2 \times b_2 \pmod{n} \end{array}$$

Example: Modulo 7 throughout:

$$\begin{array}{l} 27 \equiv -1 \\ 36 \equiv 1 \end{array} \implies \begin{array}{l} 27+36 \equiv -1+1 = 0 \\ 27 \times 36 \equiv -1 \times 1 = -1 \end{array}$$

Checks:

$$\begin{array}{l} 27+36 = 63 = 0+9(7) \\ 27 \times 36 = 972 = -1+39(7) \end{array}$$

Corollary: $\forall a, b \in \mathbb{Z}$
 $\forall m, n \in \mathbb{N}$

$$\begin{array}{l} a^{\pm} \equiv (a \bmod n)^{\pm} \pmod{n} \\ a^m \equiv (a \bmod n)^m \pmod{n} \end{array}$$

Examples: Modulo 7 throughout:

$$379 - 803 \equiv 1 - 5 = -4 \equiv 3$$

$$803^5 \equiv 5^5 = 25 \times 25 \times 5$$

$$379 \times 803 \equiv 1 \times 5 = 5$$

$$\equiv 4 \times 4 \times 5 = 80 \equiv 3$$

Prime numbers, arithmetic

As you know, a **prime** is a natural number $p > 1$ with no divisors other than 1 and itself.

11, 13
31, ~~6~~

Prime numbers, arithmetic

As you know, a **prime** is a natural number $p > 1$ with no divisors other than 1 and itself. Using 'mod' this can be expressed as

$$p \text{ is prime} \iff (p \in \mathbb{N}) \wedge (p > 1) \wedge (\forall n \in \{2, \dots, p-1\} \ p \bmod n \neq 0).$$

Prime numbers, arithmetic

As you know, a **prime** is a natural number $p > 1$ with no divisors other than 1 and itself. Using 'mod' this can be expressed as

$$p \text{ is prime} \iff (p \in \mathbb{N}) \wedge (p > 1) \wedge (\forall n \in \{2, \dots, p-1\} \ p \bmod n \neq 0).$$

The first few primes are 2,3,5,7,11,13,17,19,...

Prime numbers, arithmetic

As you know, a **prime** is a natural number $p > 1$ with no divisors other than 1 and itself. Using 'mod' this can be expressed as

$$p \text{ is prime} \iff (p \in \mathbb{N}) \wedge (p > 1) \wedge (\forall n \in \{2, \dots, p-1\} \ p \bmod n \neq 0).$$

The first few primes are 2,3,5,7,11,13,17,19,...

The **greatest common divisor** of $a, b \in \mathbb{N}$, written $\gcd(a, b)$, is the largest $n \in \mathbb{N}$ such that $a \bmod n = 0$ and $b \bmod n = 0$.

Prime numbers, arithmetic

As you know, a **prime** is a natural number $p > 1$ with no divisors other than 1 and itself. Using 'mod' this can be expressed as

$$p \text{ is prime} \iff (p \in \mathbb{N}) \wedge (p > 1) \wedge (\forall n \in \{2, \dots, p-1\} \ p \bmod n \neq 0).$$

The first few primes are 2,3,5,7,11,13,17,19,...

The **greatest common divisor** of $a, b \in \mathbb{N}$, written **gcd**(a, b), is the largest $n \in \mathbb{N}$ such that $a \bmod n = 0$ and $b \bmod n = 0$.

If $\text{gcd}(a, b) = 1$, a, b are called **relatively prime**.

Prime numbers, arithmetic

As you know, a **prime** is a natural number $p > 1$ with no divisors other than 1 and itself. Using 'mod' this can be expressed as


$$p \text{ is prime} \iff (p \in \mathbb{N}) \wedge (p > 1) \wedge (\forall n \in \{2, \dots, p-1\} \ p \bmod n \neq 0).$$

The first few primes are 2,3,5,7,11,13,17,19,...

The **greatest common divisor** of $a, b \in \mathbb{N}$, written **gcd**(a, b), is the largest $n \in \mathbb{N}$ such that $a \bmod n = 0$ and $b \bmod n = 0$.

If $\text{gcd}(a, b) = 1$, a, b are called **relatively prime**.

Examples: $\text{gcd}(30, 75) = 15$;

 42
1, 2, 3, 6, 5, 15, 30

Prime numbers, arithmetic

As you know, a **prime** is a natural number $p > 1$ with no divisors other than 1 and itself. Using 'mod' this can be expressed as

$$p \text{ is prime} \iff (p \in \mathbb{N}) \wedge (p > 1) \wedge (\forall n \in \{2, \dots, p-1\} \ p \bmod n \neq 0).$$

The first few primes are 2,3,5,7,11,13,17,19,...

The **greatest common divisor** of $a, b \in \mathbb{N}$, written **gcd**(a, b), is the largest $n \in \mathbb{N}$ such that $a \bmod n = 0$ and $b \bmod n = 0$.

If $\text{gcd}(a, b) = 1$, a, b are called **relatively prime**.

Examples: $\text{gcd}(30, 75) = 15$;

$\text{gcd}(30, \underline{77}) = 1$, so 30,77 are relatively prime.

1, 7, 11, 77

Prime numbers, arithmetic

As you know, a **prime** is a natural number $p > 1$ with no divisors other than 1 and itself. Using 'mod' this can be expressed as

$$p \text{ is prime} \iff (p \in \mathbb{N}) \wedge (p > 1) \wedge (\forall n \in \{2, \dots, p-1\} \ p \bmod n \neq 0).$$

The first few primes are 2,3,5,7,11,13,17,19,...

The **greatest common divisor** of $a, b \in \mathbb{N}$, written **gcd**(a, b), is the largest $n \in \mathbb{N}$ such that $a \bmod n = 0$ and $b \bmod n = 0$.

If $\text{gcd}(a, b) = 1$, a, b are called **relatively prime**.

Examples: $\text{gcd}(30, 75) = 15$;

$\text{gcd}(30, 77) = 1$, so 30,77 are relatively prime.

Fermat's little theorem, (proof omitted):

If p is prime and $a \in \mathbb{N}$ satisfies $\text{gcd}(a, p) = 1$ then $a^{p-1} \bmod p = 1$.

Prime numbers, arithmetic

As you know, a **prime** is a natural number $p > 1$ with no divisors other than 1 and itself. Using 'mod' this can be expressed as

$$p \text{ is prime} \iff (p \in \mathbb{N}) \wedge (p > 1) \wedge (\forall n \in \{2, \dots, p-1\} \ p \bmod n \neq 0).$$

The first few primes are 2,3,5,7,11,13,17,19,...

The **greatest common divisor** of $a, b \in \mathbb{N}$, written **gcd**(a, b), is the largest $n \in \mathbb{N}$ such that $a \bmod n = 0$ and $b \bmod n = 0$.

If $\text{gcd}(a, b) = 1$, a, b are called **relatively prime**.

Examples: $\text{gcd}(30, 75) = 15$;

$\text{gcd}(30, 77) = 1$, so 30,77 are relatively prime.

Fermat's little theorem, (proof omitted):

If p is prime and $a \in \mathbb{N}$ satisfies $\text{gcd}(a, p) = 1$ then $a^{p-1} \bmod p = 1$.

Example: $\text{gcd}(6, \underline{11}) = 1$, so $6^{10} \bmod 11 \equiv 1$.

Prime numbers, arithmetic

As you know, a **prime** is a natural number $p > 1$ with no divisors other than 1 and itself. Using 'mod' this can be expressed as

$$p \text{ is prime} \iff (p \in \mathbb{N}) \wedge (p > 1) \wedge (\forall n \in \{2, \dots, p-1\} \ p \bmod n \neq 0).$$

The first few primes are 2,3,5,7,11,13,17,19,...

The **greatest common divisor** of $a, b \in \mathbb{N}$, written **gcd**(a, b), is the largest $n \in \mathbb{N}$ such that $a \bmod n = 0$ and $b \bmod n = 0$.

If $\text{gcd}(a, b) = 1$, a, b are called **relatively prime**.

Examples: $\text{gcd}(30, 75) = 15$;

$\text{gcd}(30, 77) = 1$, so 30,77 are relatively prime.

Fermat's little theorem, (proof omitted):

If p is prime and $a \in \mathbb{N}$ satisfies $\text{gcd}(a, p) = 1$ then $a^{p-1} \bmod p = 1$.

Example: $\text{gcd}(6, 11) = 1$, so $6^{10} \bmod 11 = 1$.

Check: $6^{10} = \underline{60\,466\,176} = \underline{5\,496\,925} \times \underline{11} + \underline{1}$.

RSA Cryptography

only assessed in graduate assignment B for MATH6005



Rivest, Shamir, Adleman 1977

RSA Cryptography

only assessed in graduate assignment B for MATH6005

The bare bones of RSA are:

- A message is encoded as sequence of bytes e.g. using UTF-8.



Rivest, Shamir, Adleman 1977

RSA Cryptography

only assessed in graduate assignment B for MATH6005

The bare bones of RSA are:

- A message is encoded as sequence of bytes e.g. using UTF-8.
- The bytes are grouped into words of fixed length, e.g. 64 bytes.



Rivest, Shamir, Adleman 1977

8 bits

RSA Cryptography

only assessed in graduate assignment B for MATH6005

The bare bones of RSA are:

- A message is encoded as sequence of bytes e.g. using UTF-8.
- The bytes are grouped into words of fixed length, e.g. 64 bytes.
- Each word is treated as a (large) integer m expressed in binary, e.g. $\underline{0} \leq m < \underline{2^{512}} \approx 10^{154}$.



Rivest, Shamir, Adleman 1977

RSA Cryptography

only assessed in graduate assignment B for MATH6005

The bare bones of RSA are:

- A message is encoded as sequence of bytes e.g. using UTF-8.
- The bytes are grouped into words of fixed length, e.g. 64 bytes.
- Each word is treated as a (large) integer m expressed in binary, e.g. $0 \leq m < 2^{512} \approx 10^{154}$.
- RSA is used to encrypt each m as another integer c .



Rivest, Shamir, Adleman 1977

RSA Cryptography

only assessed in graduate assignment B for MATH6005



Rivest, Shamir, Adleman 1977

The bare bones of RSA are:

- A message is encoded as sequence of bytes e.g. using UTF-8.
- The bytes are grouped into words of fixed length, e.g. 64 bytes.
- Each word is treated as a (large) integer m expressed in binary, e.g. $0 \leq m < 2^{512} \approx 10^{154}$.
- RSA is used to encrypt each m as another integer c .
- c is represented as a binary word, with similar format to m .

RSA Cryptography

only assessed in graduate assignment B for MATH6005



Rivest, Shamir, Adleman 1977

The bare bones of RSA are:

- A message is encoded as sequence of bytes e.g. using UTF-8.
- The bytes are grouped into words of fixed length, e.g. 64 bytes.
- Each word is treated as a (large) integer m expressed in binary, e.g. $0 \leq m < 2^{512} \approx 10^{154}$.
- RSA is used to encrypt each m as another integer c .
- c is represented as a binary word, with similar format to m .
- An RSA code has a 'public' key for encrypting but a secret 'private' key for decrypting.

RSA Cryptography

only assessed in graduate assignment B for MATH6005



Rivest, Shamir, Adleman 1977

The bare bones of RSA are:

- A message is encoded as sequence of bytes e.g. using UTF-8.
- The bytes are grouped into words of fixed length, e.g. 64 bytes.
- Each word is treated as a (large) integer m expressed in binary, e.g. $0 \leq m < 2^{512} \approx 10^{154}$.
- RSA is used to encrypt each m as another integer c .
- c is represented as a binary word, with similar format to m .
- An RSA code has a 'public' key for encrypting but a secret 'private' key for decrypting.
- The **public key** is a pair of (large) integers (e, n) , where $n = pq$ for distinct primes p, q , and e is relatively prime to $\underbrace{(p-1)(q-1)}_{2, q}$.

RSA Cryptography

only assessed in graduate assignment B for MATH6005



Rivest, Shamir, Adleman 1977

The bare bones of RSA are:

- A message is encoded as sequence of bytes e.g. using UTF-8.
- The bytes are grouped into words of fixed length, e.g. 64 bytes.
- Each word is treated as a (large) integer m expressed in binary, e.g. $0 \leq m < 2^{512} \approx 10^{154}$.
- RSA is used to encrypt each m as another integer c .
- c is represented as a binary word, with similar format to m .
- An RSA code has a 'public' key for encrypting but a secret 'private' key for decrypting.
- The **public key** is a pair of (large) integers (e, n) , where $n = pq$ for distinct primes p, q , and e is relatively prime to $(p-1)(q-1)$.
- The encryption rule is $c = m^e \bmod n$.

RSA Cryptography

only assessed in graduate assignment B for MATH6005



Rivest, Shamir, Adleman 1977

The bare bones of RSA are:

- A message is encoded as sequence of bytes e.g. using UTF-8.
- The bytes are grouped into words of fixed length, e.g. 64 bytes.
- Each word is treated as a (large) integer m expressed in binary, e.g. $0 \leq m < \underline{2^{512}} \approx 10^{154}$.
- RSA is used to encrypt each m as another integer c .
- c is represented as a binary word, with similar format to m .
- An RSA code has a 'public' key for encrypting but a secret 'private' key for decrypting.
- The **public key** is a pair of (large) integers (e, n) , where $n = pq$ for distinct primes p, q , and e is relatively prime to $(p-1)(q-1)$.
- The encryption rule is $\boxed{c = m^e \bmod n.}$
- The **private key** is an integer \underline{d} , with $\underline{ed} \bmod \underline{(p-1)(q-1)} = 1$.

RSA Cryptography

only assessed in graduate assignment B for MATH6005



Rivest, Shamir, Adleman 1977

The bare bones of RSA are:

- A message is encoded as sequence of bytes e.g. using UTF-8.
- The bytes are grouped into words of fixed length, e.g. 64 bytes.
- Each word is treated as a (large) integer m expressed in binary, e.g. $0 \leq m < 2^{512} \approx 10^{154}$.
- RSA is used to encrypt each m as another integer c .
- c is represented as a binary word, with similar format to m .
- An RSA code has a 'public' key for encrypting but a secret 'private' key for decrypting.
- The **public key** is a pair of (large) integers (e, n) , where $n = \underline{pq}$ for distinct primes p, q , and e is relatively prime to $(p-1)(q-1)$.
- The encryption rule is $\boxed{c = m^e \text{ mod } n.}$
- The **private key** is an integer d , with $ed \text{ mod } (p-1)(q-1) = 1$.
- The decryption rule is $\boxed{m = c^d \text{ mod } n.}$

RSA Cryptography — comments

only assessed in graduate assignment B for MATH6005

- To send an RSA encrypted message to B, A first asks for B's public key (n, e) . This can be transmitted without security.

RSA Cryptography — comments

only assessed in graduate assignment B for MATH6005

- To send an RSA encrypted message to B, A first asks for B's public key (n, e) . This can be transmitted without security.
- The security of RSA is entirely dependent on the difficulty of finding d , given n and e .

RSA Cryptography — comments

only assessed in graduate assignment B for MATH6005

- To send an RSA encrypted message to B, A first asks for B's public key (n, e) . This can be transmitted without security.
- The security of RSA is entirely dependent on the difficulty of finding d , given n and e .
- There is a straight-forward way to compute d from e if you know $(p-1)(q-1)$. For example, the method is taught in MATH2301, along with more discussion about RSA.

RSA Cryptography — comments

only assessed in graduate assignment B for MATH6005

- To send an RSA encrypted message to B, A first asks for B's public key (n, e) . This can be transmitted without security.
- The security of RSA is entirely dependent on the difficulty of finding d , given n and e .
- There is a straight-forward way to compute d from e if you know $(p-1)(q-1)$. For example, the method is taught in MATH2301, along with more discussion about RSA.
- So the security of RSA actually depends entirely on the difficulty of factorizing very large numbers like n . *i.e. finding p and q*

RSA Cryptography — comments

only assessed in graduate assignment B for MATH6005

- To send an RSA encrypted message to B, A first asks for B's public key (n, e) . This can be transmitted without security.
- The security of RSA is entirely dependent on the difficulty of finding d , given n and e .
- There is a straight-forward way to compute d from e if you know $(p-1)(q-1)$. For example, the method is taught in MATH2301, along with more discussion about RSA.
- So the security of RSA actually depends entirely on the difficulty of factorizing very large numbers like n .
- Factorization algorithms have advanced enormously in the last 30 years, but they are still not fast enough to deal with numbers like n , typically with around a thousand bits.

RSA Cryptography — comments

only assessed in graduate assignment B for MATH6005

- To send an RSA encrypted message to B, A first asks for B's public key (n, e) . This can be transmitted without security.
- The security of RSA is entirely dependent on the difficulty of finding d , given n and e .
- There is a straight-forward way to compute d from e if you know $(p-1)(q-1)$. For example, the method is taught in MATH2301, along with more discussion about RSA. *games, graphs, and machines*
- So the security of RSA actually depends entirely on the difficulty of factorizing very large numbers like n .
- Factorization algorithms have advanced enormously in the last 30 years, but they are still not fast enough to deal with numbers like n , typically with around a thousand bits.
- Generating large primes, and factorizing large integers are topics in **number theory**, taught, for example, in MATH3301.

Number theory / cryptography

RSA Cryptography — comments

only assessed in graduate assignment B for MATH6005

- To send an RSA encrypted message to B, A first asks for B's public key (n, e) . This can be transmitted without security.
- The security of RSA is entirely dependent on the difficulty of finding d , given n and e .
- There is a straight-forward way to compute d from e if you know $(p-1)(q-1)$. For example, the method is taught in MATH2301, along with more discussion about RSA.
- So the security of RSA actually depends entirely on the difficulty of factorizing very large numbers like n .
- Factorization algorithms have advanced enormously in the last 30 years, but they are still not fast enough to deal with numbers like n , typically with around a thousand bits.
- Generating large primes, and factorizing large integers are topics in **number theory**, taught, for example, in MATH3301.
- For more on RSA, you could start with the Wikipedia article
[http://en.wikipedia.org/wiki/RSA_\(algorithm\)](http://en.wikipedia.org/wiki/RSA_(algorithm))

Adleman clip

RSA Cryptography — comments

only assessed in graduate assignment B for MATH6005

- To send an RSA encrypted message to B, A first asks for B's public key (n, e) . This can be transmitted without security.
- The security of RSA is entirely dependent on the difficulty of finding d , given n and e .
- There is a straight-forward way to compute d from e if you know $(p-1)(q-1)$. For example, the method is taught in MATH2301, along with more discussion about RSA.
- So the security of RSA actually depends entirely on the difficulty of factorizing very large numbers like n .
- Factorization algorithms have advanced enormously in the last 30 years, but they are still not fast enough to deal with numbers like n , typically with around a thousand bits.
- Generating large primes, and factorizing large integers are topics in **number theory**, taught, for example, in MATH3301.
- For more on RSA, you could start with the Wikipedia article
[http://en.wikipedia.org/wiki/RSA_\(algorithm\)](http://en.wikipedia.org/wiki/RSA_(algorithm))

Adleman clip