Number Bases

Binary Numerals

We are so used to writing numbers down in the denary notation (e.g., we write $9 + 3 = 12$) that it may be hard to imagine other ways of representing them. However, there are many possible systems of numerals which can be used to display numbers. For example, the number twelve may be written as '12' (denary, or decimal), 'XII' (Roman), '1100' (binary), 'C' (hexadecimal), etc. Apart from Roman Numerals, all these systems use a **positional notation**, with a **base**, which is ten in the case of denary numerals.

When we write down the natural numbers in base ten, we use single digits for the first ten: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Then we use a combination of digits, 10, where the 1 is in a place which shows that it must be multiplied by the base, while the 0 is in the units place. We can carry on counting by increasing the units digit by 1 to give 11, 12, 13, 14, 15, 16, 17, 18, 19. When we get to nineteen (19), the units digit is the largest available, so we put it back to 0 and increase the tens digit by 1 to get 20. We can carry on counting in this way until we get to 00, when neither the unit digit nor the tens digit can increase any more, so we introduce a 1 in the hundreds place to get 100; the 1 now has a place value which is the square of the base ($10^2 = 10 \times 10 = 100$). A further example of the meaning of place notation is

$$\begin{aligned} 2506.37 &= 2 \times 1000 + 5 \times 100 + 0 \times 10 + 6 \times 1 + 3 \times 0.1 + 7 \times 0.01 \\ &= 2 \times 10^3 + 5 \times 10^2 + 0 \times 10^1 + 6 \times 10^0 + 3 \times 10^{-1} + 7 \times 10^{-2}. \end{aligned}$$

Internally, computers usually store data using large arrays of electronic, magnetic, or optical devices each of which can be in one of just two states. They therefore represent numbers in **binary**, with base two and just two possible digits, 0 and 1. In this system we can start counting at 0 in the same way as we do in denary, but run out of digits after 0 and 1. So we represent two as 10, then three as 11, but now neither the unit digit nor the twos digit can increase any more, so the next number, four, is written 100. The numbers continue as 101, 110, 111, 1000, 1001, 1010, 1011, 1100, etc. The place-values are powers of two instead of powers of ten. Thus, for example, binary 1101101 represents the denary number

$$1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 64 + 32 + 8 + 4 + 1 = 109.$$

The individual zeros and ones are called **bits**. We can also have a 'binary point' instead of a decimal point, so that, for example,

$$101.1101_2 = 1 \times 2^2 + \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-4}$$

$$= 1 \times 4 + 1 \times 1 + 1 \times \frac{1}{2} + 1 \times \frac{1}{4} + \times \frac{1}{16}$$
$$= 5.8125_{10}.$$

When there is any possibility of confusion about which base we are using in a numeral, as in the equation above involving different bases, a subscript can be used to write the base (in denary), e.g., $101_2 = 5_{10}$.

Just as in denary, we can add zeros on the left of numerals without changing the corresponding number, so, for example, the binary numeral 0010 represents just two just as 10 does.

Strings of bits are, by their nature, hard to read and use up a lot of space on paper. If we want to write numbers on paper as numerals which are close to the binary representation which is used by a computer internally, the **hexadecimal** system is commonly used. As the name suggests, hexadecimal numerals use a base which is the number sixteen. The word 'hexadecimal' is often abbreviated to **hex**.

In hexadecimal, we need sixteen digits. The usual symbols 0, 1, 2, ..., 9 are used for zero to nine, and then we use the letter A for ten, B for eleven, and so on. After F (fifteen), the next hex numeral is 10 (sixteen), and we can continue in the same way as for denary and binary.

The table below lists numerals in bases ten, two, and sixteen.

denary	**binary**	**hex**
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	10000	10
17	10001	11
18	10010	12
19	10011	13
20	10100	14

Since $16 = 2^4$, a binary numeral can be converted to hexadecimal by simply grouping the bits in fours and replacing each group of four by the corresponding

hex digit. So we can regard a hexadecimal numeral simply as a convenient way of writing down, with fewer symbols, a binary numeral.

Converting between bases

Binary to denary

To convert a number from its binary representation to the denary form in which we are more used to seeing it, we just need to add the place values for the positions of the 1s. For example, to convert 10011_2 to denary:

	1		0		0		1		1		binary
	2^4		2^3		2^2		2^1		2^0		place values (in denary)
	16		8		4		2		1		place values
$=$	16	$+$	0	$+$	0	$+$	2	$+$	1	$=$	19_{10}

Fractions pose no additional problems; we just need to get the place values right. For example, to convert 1101.011_2 to denary:

Test Yourself Visit the URL below to try a numbas exam:
`https://numbas.mathcentre.ac.uk/question/44478/binary-to-decimal-worksheet/`

Hexadecimal to denary

The same method can be used to convert from other bases to denary, except that we have to multiply the place value by the digit in each place. For example, to convert $30C.A3_{16}$ to denary:

Denary to binary

First, a reminder about remainders. Given any integer a and a positive integer b, the integer a can always be written in the form

$$a = qb + r,$$

where b is an integer and r is a positive integer. In this case we call q the **quotient** and r the **remainder**. For example, when 19 is divided by 5, the quotient is 3 and the remainder is 4, since

$$19 = 3 \times 5 + 4.$$

To convert a whole number from denary to binary, we can use repeated division.

For example, to convert 113_{10} to its binary representation, we divide by 2, giving a quotient of 56 and a remainder of 1. So the last bit is 1. The rest are found by starting the process again with 56 instead of 113, then with the new

	1		1		0		1	.	0		1		1		binary
	2^3		2^2		2^1		2^0		2^{-1}		2^{-2}		2^{-3}		place values (in denary)
	8		4		2		1		0.5		0.25		0.125		place values
$=$	8	$+$	4	$+$	0	$+$	1	$+$	0	$+$	0.25	$+$	0.125	$=$	13.375_{10}

$$
\begin{array}{cccccccccccc}
 & 3 & & 0 & & C & . & A & & 3 & & \text{hexadecimal} \\
 & 16^2 & & 16^1 & & 16^0 & & 16^{-1} & & 16^{-2} & & \text{place values (in denary)} \\
 & 256 & & 16 & & 1 & & \frac{1}{16} & & \frac{1}{256} & & \text{place values} \\
= & 3\times 256 & + & 0 & + & 12\times 1 & + & 10\times\frac{1}{16} & + & 3\times\frac{1}{256} & & \\
= & 768 & + & 0 & + & 12 & + & 0.625 & + & 0.0117\ldots & = & 780.6367\ldots_{10}
\end{array}
$$

$$
\begin{array}{rcrcrcl}
113 & \div & 2 & = & 56 & \text{rem.} & 1 \\
56 & \div & 2 & = & 28 & \text{rem.} & 0 \\
28 & \div & 2 & = & 14 & \text{rem.} & 0 \\
14 & \div & 2 & = & 7 & \text{rem.} & 0 \\
7 & \div & 2 & = & 3 & \text{rem.} & 1 \\
3 & \div & 2 & = & 1 & \text{rem.} & 1 \\
1 & \div & 2 & = & 0 & \text{rem.} & 1
\end{array}
$$

quotient 28, and so on until the quotient is 0 and we can stop. The remainders, which are in the last column above, give the binary representation of the number. However, we must remember to read the upwards. So $113_{10} = 1110001_2$.

Test Yourself Visit the URL below to try a numbas exam:
`https://numbas.mathcentre.ac.uk/question/10755/decimal-to-binary`

Hexadecimal to binary

We have already seen how to convert from hexadecimal to binary. Each hex digits is replaced by four bits. (Leading zeroes must be included to make sure that each digit does give exactly four bits.) For example,

$$6\,0\,C\,3_{16} = 0110\,0000\,1100\,0011_2 = 110000011000011_2.$$

Binary to hexadecimal

To convert from binary to hexadecimal, we reverse the above process: group the bits in fours and convert each group of four to a hex digit. The grouping starts on the right-hand side, and the easiest way to do this is to add leading zeros if necessary to make the number of bits a multiple of four. For example,

$$1001101101110_2 = 0001\,0011\,0110\,1110_2 = 1\,3\,6\,E_{16}.$$

Denary to hexadecimal

One way is to convert from denary to binary and then from binary to hexadecimal. Alternatively, use the iterative division method as from denary to binary above, except that the division is by 16 rather than 2.

Binary arithmetic

Doing arithmetic with numbers written in binary is really not very different from doing arithmetic with numbers written in denary; it is much simpler, because the multiplication and addition tables are very simple, but of course calculations take more steps because the representations are usually longer.

For addition, we need to know that 0 added to any number gives that same number, and, in denary, we need to know the sums of all pairs of numbers from one to nine, for example, $1 + 1 = 2$, $2 + 3 = 5$, $6 + 7 = 13$, etc. In binary all we need to know is that $1 + 1 = 10$. It also helps, when dealing with carries, to know that $1 + 1 + 1 = 11$.

Addition

For example,

Denary:

	9	4	
+	1	9	
1	1	3	
carries→ 1	1		

Binary:

	0	1	0	1	1	1	1	0
+	0	0	0	1	0	0	1	1
	0	1	1	1	0	0	0	1
carries→			1	1	1	1		

Subtraction

Subtraction also works in the same way as in denary, but we need to be careful to be accurate with the 'borrows'. For example,

Denary:

	2	3	0	1
−	1	9	9	4
	0	3	0	7
borrows→	1	1	1	

Binary:

	0	1	0	1	1	1	1	0
−	0	0	0	1	0	0	1	1
	0	1	0	0	1	0	1	1
borrows→						1	1	

Hence, in denary, $2301 - 1994 = 307$, and, in binary, $1011110 - 10011 = 1001011$. To be sure that you understand how the borrows work, it is useful to work out carefully an example with multiple borrows. For example,

Denary:

2	0	0	0	3
−	1	0	0	5
1	8	9	9	8

Binary:

	0	1	0	0	0	0	0	0
−	0	0	0	0	0	0	0	1
	0	0	1	1	1	1	1	1

Subtraction with a negative result

In the above examples, a smaller number was subtracted from a larger. Of course, when we do a subtraction we need to check which number is larger, and make the answer negative if a large number is subtracted from a smaller. Just as in denary, we can recognise which number is larger by adding leading zeros if necessary to make the number of bits the same in each, then look from the left until a bit is different in the two numbers, and then the number with the 1 bit in this position is larger.

For example, in denary, $1994 - 2301 = -(2301 - 1994) = -307$, and, in binary, $10011 - 1011110 = -(1011110 - 10011) = -1001011$.

The processing units of computers have circuitry built in to do addition of binary numerals. However, subtraction is often done by the two's complement method, which will be described later.

Integer storage and complements

Computers store integers in binary, using a fixed number of bytes. Typical lengths are 2 bytes (16 bits), 4 bytes (32 bits), and 8 bytes (64 bits). However,

for now, we'll assume just 8 bits.

There are obviously only a finite number of possible bit strings with a fixed amount of store, 2^n possibilities with n bits. Integers can be stored only up to some maximum value. For 8 bits, the maximum is 127 if negative numbers are included, 255 if not.

Bit strings representing positive (or zero) integers are padded out with zeros on the left so that they can be stored in the appropriate number of bytes. For example, if integers occupy 32 bits, then the number 6 would be stored as

$$0000\,0000\,0000\,0000\,0000\,0000\,0000\,0110.$$

Programming languages may allow the programmer some choice in the number of bytes to be used for each integer variable, for example by specifying its type as `int` or `short int` or `long int`. Some languages allow a program to read the amount of storage allocated to any variable or type and the largest integer value which can be stored. If the integer is too large, then, of course, it is not possible to store it. An example of the problems this may cause is given below, in the section on **overflow**.

It would seem that, using using 32 bits, the largest possible integer should be 2^32-1. However, this does not allow for negative integers. One bit is needed to store the sign, so that only (for example) 31 bits out of 4 bytes can be used for positive numbers. Thus the largest positive integer which can be stored is $2^31-1 = 2147483647$, represented by the hex number $7FFFFFFF_{16}$.

The sign bit of an integer is normally the left hand bit (0 for positive or zero, 1 for negative). However, for ease of doing arithmetic, the rest of the bits are not simply a representation of the absolute value of the number, but instead negative integers are usually stored in **two's complement** form; more details are given below.

In a 32 bit two's complement representation, the hex number 80000000_{16} corresponds to the smallest storable negative number, $-2^{31} = -2147483648$, and as we increment the bit string the size of this negative number drops progressively until we reach $FFFFFFFF_{16}$, which corresponds to -1.

Some languages allow a variable to be declared as a natural number (essentially an `unsigned` number), so that it can take only positive or zero values; all the available bits can then be used for the value.

The two's complement representation is chosen to make addition and subtraction of signed numbers easy to implement. To see how this works we first look at the equivalent idea in denary, which is ten's complement.

Ten's complement

Suppose that a child has learnt how to add numbers (including 'carries'), but is having difficulties with the 'borrows' involved in subtraction. They want to calculate 4375 - 1659; what can they do?

Being a resourceful child, they realise that subtraction from a number where all the digits are 9s won't cause any problems. It is easy to work out $9999-1659$ instead. To do this, all one needs to do is to replace each 0 in the number by 9, each 9 by 0, each 1 by 8, each 8 by 1, and so on.

$$\begin{array}{rr} & 9999 \\ - & 1659 \\ \hline & 8340 \\ \hline \end{array}$$

Now it is easy to work out $4375+8340$, which is $4375+(9999-1659)$. To get the right answer to the original problem, 9999 must be subtracted. Since 9999 is $10000-1$, this can be achieved by adding 1 and subtracting 10000, which can also be done without any 'borrows'. To summarise,

$$\begin{aligned} 4375 - 1659 &= 4375 + (9999 - 1659) - 9999 \\ &= 4375 + (9999 - 1659 + 1) - 10000. \end{aligned}$$

The number in brackets $(9999 - 1659 + 1)$ is called the **ten's complement** of 1659. To form it, as mentioned above, we swap 0s with 9s, 1s with 8s, 2s with 7s, 3s with 6s, and 4s with 5s. Then add 1. So, to do the subtraction, the complement of 1659, which is 8341, is added to 4375:

$$\begin{array}{rr} & 4375 \\ + & 8341 \\ \hline & 12716 \end{array} \quad \text{(tens complement of 1659)}$$

To calculate $4375 - 1659$, all that is needed now is to subtract 10000, or, in other words, to discard the 1 which has been carried on the left. The result is 2716/

Note that the complement method works with fixed length numbers. Leading zeros must be added if necessary to make the two numbers the same lenght (and, if necessary, an extra zero to make each left hand digit less than 5.) To calculate $4375 - 67$, we have to convert 67 to 0067. Then both numbers have four digits. The complement of 0067 is 9933.

Two's complement

The binary equivalent of ten's complement is two's complement. To calculate the two's complement of a number written in binary, we subtract it from a number consisting of a string of 1s (giving the one's complement) and then add 1 to give the two's complement. Subtracting from a string of 1s is very easy; each 0 in the number becomes 1 and each 1 becomes 0.

Example 1

As a first example, to keep bit strings short, we'll work with 8-bit numbers. We work out $1111010 - 101101$ which is $01111010 - 00101101$ when leading zeros are added to give 8-bit strings. We form the complement of the number to be subtracted as follows:

$$\begin{array}{rrr} & 1111 & 1111 \\ - & 0010 & 1101 \\ \hline & 1101 & 0010 \end{array} \quad \text{(one's complement)}$$

$$\begin{array}{rrr} + & & 1 \\ \hline & 1101 & 0011 \end{array} \quad \text{(two's complement)}$$

Now we add the two's complement to the number from which we are subtracting:

$$\begin{array}{rrr} & 0111 & 1010 \\ + & 1101 & 0011 \\ \hline & 10100 & 1001 \end{array} \quad \text{(complement of number being subtracted)}$$

Discarding the 1 carried at the left, the result is 0100 1101.

Example 2

So far, we've been doing subtractions which give a positive answer. Now we use the complement method to subtract a large number from a small one. As an example, we swap the numbers above and work out 00101101 − 01111010.

Following the same method as above, we form the two's complement of 01111010.

	1111	1111	
−	0111	1010	
	1000	0101	(one's complement)
+		1	
	1000	0110	(two's complement)

Now we add the two's complement to the number from which we are subtracting:

	0010	1101	
+	1000	0110	(complement of number being subtracted)
	1011	0011	

This time, there is no carry bit on the left, and the sign bit of the answer is 1. This means that the answer is negative. In two's complement form, the result is 1011 0011. To see which negative number this represents, we can make it positive by forming its two's complement. This is 01001101. The answer represents −1001101.

Example 3

For an example with 16 bit numbers, we work out 0001 0001 0001 0111 − 0000 0110 0111 1011. We form the complement of the number to be subtracted as follows:

	0000	0110	0111	1011	
	1111	1001	1000	0100	(one's complement)
+				1	
	1111	1001	1000	0101	(two's complement)

Now we add the two's complement to the number from which we are subtracting:

	0001	0001	0001	0111	
+	1111	1001	1000	0101	(complement of number being subtracted)
1	0000	1010	1001	1100	

Discarding the 1 carried at the left, the result is 0000 1010 1001 1100.

Example 4

Now we use the complement method to subtract a large number from a small one. As an example, we swap the numbers above and work out 0000 0110 0111 1011− 0001 0001 0001 0111.

Following the same method as above, we form the two's complement of 0001 0001 0001 0111.

	0001	0001	0001	0111	
	1110	1110	1110	1000	(one's complement)
+				1	
	1110	1110	1110	1001	(two's complement)

Then we add the complement to 0000 0110 0111 1011.

$$\begin{array}{rcccc} & 0000 & 0110 & 0111 & 1011 \\ + & 1110 & 1110 & 1110 & 1001 \\ \hline & 1111 & 0101 & 0110 & 0100 \end{array}$$ (complement of number being subtracted)

The thing to notice is that there is now no carry from the left hand side. This indicates that the result is negative, and the bit string we've produced is the two's complement of the absolute value of the result. To check this, we form the complement:

$$\begin{array}{rcccc} & 1111 & 0101 & 0110 & 0100 \\ \hline & 0000 & 1010 & 1001 & 1011 \\ + & & & & 1 \\ \hline & 0000 & 1010 & 1001 & 1100 \end{array}$$

The answer is the binary number -101010011100, as expected.

Storage of negative numbers

If we wish to add a negative number $-y$ (where y is positive) to a positive number x, the result is the same as doing the subtraction $x - y$. It is therefore useful to store negative numbers in a computer's memory in two's complement form. Then a negative number can be added to a positive number using exactly the same method (and so the same electronic circuitry) as for adding two positive numbers. If the result is negative then it will be in twos complement form and so can be stored immediately without any extra work,. Addition also works correctly when two negative numbers in two's complement form are added in the same way.

For example, -5, or -101 in binary, is stored as its complement 11111011 if only 8 bits are available. With more bits, the complement has extra leading ones; e.g., 1111111111111011 with 16 bits.

Hence the usual method of storing negative integers is as two's complements, so that all numbers can be added without taking account of their sign. Subtraction can also be done in the same way for all numbers; the number to be subtracted is complemented (converting it from positive to negative or negative to positive, or zero to zero) and added.

An alternative way to understand numbers represented in two's complement form is to regard the sign bit (left hand bit) as having a negative place value. The other bits have their usual place values. With an 8 bit representation, the sign bit has place value -128. So, for example, 11111011 in binary represents $(-128) + 64 + 32 + 16 + 8 + 2 + 1 = -5$.

Overflow

One problem with fixed length integer storage is overflow. If we add two positive integers, the result will sometimes be larger than the maximum integer which can be stored in the number of bits allocated.

As an example using 8 bit numbers, suppose we work out $101101 + 1111010$.

$$\begin{array}{rcc} & 0010 & 1101 \\ + & 0111 & 1010 \\ \hline & 1010 & 0111 \end{array}$$

There is no carry bit on the left, but the sign bit of the answer is 1. This means that the answer is negative, which of course is impossible when two

positive numbers are added. The problem is the answer is too big for the number of bits used.

For a 16 bit example, we add 0100 0000 1010 0011 to 0100 0100 0101 0100. The result is 1000 0100 1111 0111. This bit string, with the left hand bit having the value 1, represents a negative integer in complement form. Overflow has occurred.

The same problem can occur when two negative integers are added and the result has a 0 as the left hand bit and so looks like a positive or zero number.