Bits, Bytes and Bases

# Introduction

You'll be familiar from primary school with the idea of writing numbers with hundreds, tens and units. So if we see a number such as 739, what we really have is 7 lots of 100, 3 lots of 10 and 9 lots of 1. We can write this a bit more mathematically as:

739 = (7 \* 100) + (3 \* 10) + (9 \* 1)

Notice, too, that we can re-write the number 100 as 102, 10 as 101 and 1 as 100. So this gives us:

739 = (7 \* 102,) + (3 \* 101) + (9 \* 100)

We're not limited, either, by just having three columns, and the power notation saves a lot of space. So we can produce huge numbers like:

620571 = (6 \* 105) + (2 \* 104) + (0 \* 103) + (5 \* 102) + (7 \* 101) + (1 \* 100)

# The binary system

In the above system, we call 10 the **base** of the system, since we are writing our numbers as a sum of elements of the form *x \* 10y*. (Aside: this is closely related to the base of a logarithm.) Notice that we can, in principle, make y go as high as we like, but x must be a number between 0 and 9. (If x is 10, then we would add 1 to the column with y + 1 instead). This is also called the **decimal** system.

There's nothing special though about 10 (humans just like it because we have 10 fingers) - so we can use another number instead, as long as we are careful to restrict x to the appropriate range. Particularly useful for us is the **binary**, or base 2, system.

So, to write a number in binary, we need to write it as a sum of elements of the form *x \* 2y*. As before, we are restricted in x, but to what range?

We can definitely have x = 0 or x = 1. If we have x = 2, then we would get (with y = 0):

2 \* 20 = 2 = 1 \* 21

So we add 1 to the "twos" column. Thus in binary, we write 2 as 10, since 2 = (1 \* 21) + (0 \* 20).

In other words, we are only allowed to have x = 0 or x = 1.

Another example: take the decimal number 81. How can we write this in binary?

Note that 81 is bigger than 64, which is equal to 26. So we will have a 1 in the "sixty-fours" column. To find out what's left, we take 64 from 81, leaving 17. This is bigger than 16 but smaller than 32. So we will have a 0 in the "thirty-twos" column but a 1 in the "sixteens" column. Taking 16 from 17 leaves 1, giving a 1 in the units column and zeroes elsewhere. Therefore:

81 = (1 \* 64) + (1 \* 16) + (1 \* 1)

81 = (1 \* 26) + (0 \* 25) + (1 \* 24) + (0 \* 23) + (0 \* 22) + (0 \* 21) + (1 \* 20)

81 = 1010001

Notice how much longer numbers are in binary!

# Exercise 1

In this exercise, we will practice converting numbers between binary and decimal by hand.

#### Section A

(\*) Convert the following numbers to binary.

1. 7
2. 12
3. 16
4. 23
5. 54
6. 98
7. 243

#### Section B

(\*) Convert the following numbers to decimal.

1. 101
2. 1110
3. 10001
4. 101010
5. 110110
6. 1010100
7. 11110101

# Why do we care?

At the moment, binary seems like a very inefficient way to write numbers, so why bother? The answer is that, without binary, we could not have had the computer revolution!

The power of binary numbers comes from the fact that there are only two possible values for each binary digit: 0 or 1. So we could look at binary digits in a different way. Imagine you have a series of electrical switches in a row. Some of them are on; others are off. We can encapsulate the precise configuration of the switches in a binary number, by setting each digit to be 1 if the corresponding switch is on and 0 otherwise. Another, very common, interpretation is to say that 1 represents something True while 0 represents something False.

This sort of interpretation is exactly how computers work. In the end, in order to perform any task, the computer must switch various switches on and off according to its program. This program is - you guessed! - a stream of binary digits telling the computer exactly which switches need to be turned on at any given moment.

A single binary digit is called a **bit** (which is just a shortening of **bi**nary digi**t**). In fact, in order to process the program more efficiently, the computer will read information in blocks of eight bits, known as a **byte**. (lesser known is that 4 bits is called a nibble – which I think is very cute)

# Exercise 2

(\*) What is the maximum value that can be stored in a byte?

# The hexadecimal system

So, we have seen the power of binary in making computers run. But we humans still have a problem. Writing down binary takes ages! And it's very easy for the numbers to all blur together and miss out a digit. For example, if you wanted to write down the number 42923 in binary, it would be:

1010011110101011

That's not very readable, or very easy to work out. However, as we've seen in Exercise 1, it's not entirely straight-forward to convert between decimal and binary. The decimal interpretation doesn't really have much relationship to the binary.

So, when we are programming, we will often use a system called **hexadecimal**, which uses base 16 instead. A number in hexadecimal is written as a sum of elements of the form *x \* 16y*. Now there is a problem. In decimal, the maximum value of x was 9; in binary, it was 1. Both of these are one less than the base. So naturally, in hexadecimal, the maximum value of x should be 15. But we can't use 15 off the bat, since:

15 = (1 \* 161) + (5 \* 160) = 21 in decimal

The work-around for this is to define some new "digits": A = 10 B = 11 C = 12 D = 13 E = 14 F = 15

(Aside: both upper and lower case can be used.) So a hexadecimal number might look like 2A, which in decimal is:

2A = (2 \* 161) + (A \* 160) = (2 \* 161) + (10 \* 160) = 42

Why is hexadecimal better? We can write any set of four binary digits as a single hexadecimal digit. In effect, the last four columns of the binary number correspond to the last column of the hexadecimal, the previous four for the "sixteens" column, and so forth. So our big number above is written in hex (a common abbreviation for hexadecimal) as:

A7AB

Notice how much shorter this is, and the chance of making a mistake is much lower! To produce this, we simply divide the binary number into chunks of 4, and individually convert each chunk into hex.

# Exercise 3

(\*) We have three lists of numbers: a decimal list, a binary list and a hexadecimal list. They're not in order, but they represent the same numbers. Match them up.

#### List A: decimal

* 72
* 54
* 165
* 93
* 37
* 203
* 177
* 44

#### List B: binary

* 10100101
* 10110001
* 1001000
* 100101
* 101100
* 110110
* 1011101
* 11001011

#### List C: hexadecimal

* 25
* B1
* CB
* 48
* 5D
* 36
* 2C
* A5

# Using bases in Python

Note, it’s not the number itself that is binary/hexadecimal, just how it’s written down.

In python we can use any of the three representations. To signify a number is written in hexadecimal you need to put a 0x on the front and to signify a number is written in binary you put 0b on the front. For example:

number\_of\_apples = 10

seconds\_left = 0x1F

memory\_free = 0b100000000

In a python program, text is represented as strings (with quotes around) eg. “hello”.

We can create text representing a number using the following

* str(memory\_free) uses base 10 so returns “256”
* hex(number\_of\_apples) uses base 16 so returns “0xA”
* bin(seconds\_left) uses base 2 so returns “0b11111”

We can turn a string containing a number back into a number using the int function. You just need to tell it what base you working with.

* int(string,2) takes a binary string (with or without 0b at the beginning) and converts to an integer
* int(string,16) takes a hexadecimal string (with or without 0x at the beginning) and converts to an integer
* int(string,10) takes a decimal string and converts to an integer. Base 10 is the default so you can also just use int(string)

# Exercise 4

(\*) Have a go at using these functions in python

# Bit manipulation

When writing programs you might want to know the value of a particular bit in a number, eg. bit 3 of 0b10110 would be 0 because we start from bit 0 on the right and work left.

To write a program that does this out you need to know about some new mathematical operations.

#### Bit shift

This operation does what it says. It moves all the bits to the left (for left bit shift, with symbol <<) or right (for right bit shift, with symbol >>). Example:

* 0b10010 >> 2 means shift right twice dropping the digits off the front to give 0b100
* 0b101 << 1 means shift left once adding 0s on the front to give 0b1010

Shifting left by one is the same as what other mathematical operation?

#### Logic functions

If you want to learn more check out this website from Stanford University <https://web.stanford.edu/class/cs101/bits-bytes.html>