Binary numbers

The reason humans represent numbers using decimal (the ten digits 0,1, ... ,9) is that we have ten fingers. There is no other reason than that. There is nothing special otherwise about the number ten. Computers don't represent numbers using decimal. Instead, they represent numbers using binary, or "base 2". Let's make sure we understand what binary representations of numbers are. We'll start with positive integers.

In decimal, we write numbers using *digits* $\{0, 1, \ldots, 9\}$, in particular, as sums of powers of ten. For example,

$$(238)_{10} \;=\; 2 * 10^2 + 3 * 10^1 + 8 * 10^0$$

In binary, we represent numbers using *bits* $\{0, 1\}$, in particular, as a sum of powers of two:

$$(11010)_2 \;=\; 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0$$

I have put little subscripts (10 and 2) to indicate that we are using a particular representation (decimal or binary). We don't need to always put this subscript in, but sometimes it helps[2].

Let's consider how to count in binary. You should verify that the binary representation is a sum of powers of 2 that indeed corresponds to the decimal representation on the left.

```
decimal        binary
-------        ------
   0              0
   1              1
   2             10
   3             11
   4            100
   5            101
   6            110
   7            111
   8           1000
   9           1001
  10           1010
  11           1011
      etc
```

Just as we did last lecture with base 8, let's add two numbers which are written in binary:

```
  11010          26
+ 1111           15
  -----          --
 101001          41
```

Make sure you see how this is done, namely how the "carries" work. For example, in column 0, we add $0 + 1$ to get 1 and no carry. In column 1, we add $1 + 1$ (in fact, $1 * 2^1 + 1 * 2^1$) and we get $2 * 2^1 = 2^2$ and so we carry a 1 over column 2. Do not proceed further until you understand this.

[2]Note however that the subscript itself is represented in some fixed base. For convinience let's use base ten because it is a lot easier for us to read and write. However, computers might disagree! To be completely base-independent we could write the base using a different notation like roman numerals... $(238)_{\mathrm{X}}$ and $(11010)_{\mathrm{II}}$. But let's not go crazy...

Converting from decimal to binary

It is trivial to convert a number from a binary representation to a decimal representation. You just need to know the decimal representation of the various powers of 2.

$2^0 = 1,\ 2^1 = 2,\ 2^2 = 4,\ 2^3 = 8,\ 2^4 = 16,\ 2^5 = 32,\ 2^6 = 64,\ 2^7 = 128,\ 2^8 = 256,\ 2^9 = 512,\ 2^{10} = 1024,\ \ldots$

Then, for any binary number, you write each of its bits as a power of 2 (in decimal) and then you add up these decimal numbers, e.g.

$$11010_2 \ = \ 16 + 8 + 2 = 26.$$

The other direction is more challenging. How do you convert from a decimal number to a binary number?

Here is an algorithm for doing so which is so simple you could have learned it in grade school. The algorithm repeatedly divides by 2 and the "remainder" bits give us the binary representation. Recall that "/" is the integer division operation which ignores the remainder i.e. the fractional part. If you want the remainder of the division, use "%".

Algorithm 3 Convert integer to binary

INPUT: a number m
OUTPUT: the number m **expressed in base 2 using a bit array** $b[\,]$

```
i ← 0
while m > 0 do
    b[i] ← m%2
    m ← m/2
    i ← i + 1
end while
```

Example: Convert 241 to binary

i	b[i]	m
		241
0	1	120
1	0	60
2	0	30
3	0	15
4	1	7
5	1	3
6	1	1
7	1	0

and so $(241)_{10} = (11110001)_2$. Note that there are an infinite number of 0's on the left which are higher powers of 2 which we ignore.

Why does this algorithm work? To answer this question, it helps to recall a few properties of multiplication and division. Let's go back to base 10 where we have a better intuition.

Suppose we have a positive integer m which is written in decimal (as a sum of powers of 10) and we then multiply m by 10. There is a simple way to get the result, 10 m, namely shift the digits left by one place and put a 0 in the rightmost position. So, $238 * 10 = 2380$ and the reason is

$$238 * 10 \;=\; (2 * 10^2 + 2 * 10^1 + 8 * 10^0) * 10 = 2 * 10^3 + 3 * 10^2 + 8 * 10^1 + 0 * 10^0.$$

Similarly, to divide a number m by 10, we shift the digits to the right

$$238/10 \;=\; (2 * 10^2 + 3 * 10^1 + 8 * 10^0)/10 = 2 * 10^1 + 3 * 10^0$$

We have dropped the rightmost digit 8 (which becomes the remainder) since we are doing integer division, and thus ignoring terms with negative powers of 10 i.e $8 * 10^{-1}$.

In binary, the same idea holds. If we represent a number m in binary and multiply by 2, we shift the bits to the left by one position and put a 0 in the rightmost position. So, e.g. if

$$m = (11010)_2 \;=\; 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0$$

then multiplying by 2 gives

$$(110100)_2 \;=\; 1 * 2^5 + 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0.$$

If we divide by 2, then we shift right by one position and drop the rightmost bit (which becomes the remainder).

Let's put the left and right shifts together. For any positive integer m, we can write

$$m = 10 * (m/10) + (m\%10),$$

for example,

$$549 = 540 + 9 = 10 * (549/10) + (549 \;\%\; 10).$$

More generally, for any positive integer – call it *base* – we can write

$$m = base * (m/base) + (m \;\%\; base),$$

and in particular, in binary we use $base = 2$, so

$$m = 2 * (m/2) + (m \;\%\; 2).$$

Now let's apply these ideas to the algorithm. Representing a positive integer m in binary means that we write it

$$m = \sum_{i=0}^{n-1} b_i \; 2^i$$

where b_i is a bit, *i.e* either 0 or 1. So we write m in binary as a bit sequence $(b_{n-1}\; b_{n-2}\; \ldots\; b_2\; b_1\; b_0)$. In particular,

$$\begin{aligned} m \;\%\; 2 &= b_0 \\ m \;/\; 2 &= (b_{n-1} \ldots b_2 b_1) \end{aligned}$$

Thus, the algorithm essentially just uses repeated division and mod to read off the bits of the binary representation of the number.

If you are still not convinced, let's run another example where we "know" the answer from the start and we'll see that the algorithm does the correct thing. Suppose our number is $m = 241$, which is $(11110001)_2$ in binary.

i	b[i]	m
		11110001
0	1	1111000
1	0	111100
2	0	11110
3	0	1111
4	1	111
5	1	11
6	1	1
7	1	0

and so the remainders are just the bits used in the binary representation of the number.

Interestingly, the same algorithm works for any base. For example, let's convert 238 from decimal to base 5, that is, let's write 238 as a sum of powers of 5. We apply the same algorithm as earlier but now we divide by 5 at each step and take the remainder.

i	b[i]	m
		238
0	3	47
1	2	9
2	4	1
3	1	0

and so $(238)_{10} = (1423)_5$. Verify this by converting the latter back into decimal by summing powers of 5.

Binary fractions

Up to now we have only talked about integers. We next talk about binary representations of fractional numbers, that is, numbers that lie between the integers. Take a decimal number such as 26.375. We write this as:

$$(26.375)_{10} \;=\; 2 * 10^1 + 6 * 10^0 + 3 * 10^{-1} + 7 * 10^{-2} + 5 * 10^{-2}.$$

The "." is called the *decimal point*.

One uses an analogous representation using binary numbers, *e.g.*

$$(11010.011)_2 \;=\; 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 + 0 * 2^{-1} + 1 * 2^{-2} + 1 * 2^{-3}$$

where "." is called the *binary point*. Check for yourself that this is the same number as above, namely

$$16 + 8 + 2 + 0.25 + 0.125 \;=\; 26.375.$$

How do we convert from decimal to binary for such fractional numbers in general ? Suppose we have a number x that is written in decimal and has a finite number of digits to the right of the binary point. Let's look at a particular example.

Let $x = 4.67$ and let's convert it to binary. Since $x = 4 + .67$, we know the answer will have the form $(100.\_\_\_)_2$ since 100 is the binary representation of 4 and .67 is a sum of negative powers of 2. So we just need to find the bits to the right of the binary point. To get the first say five bits, we multiply and divide by 2 five times (or alternatively, as was suggested in class, we just directly multiply 0.67 by $2^5 = 32$ and also divide by 2^5).

$$\begin{aligned} 0.67 &= 1.34 * 2^{-1} \\ &= 2.68 * 2^{-2} \\ &= 5.36 * 2^{-3} \\ &= 10.72 * 2^{-4} \\ &= 21.44 * 2^{-5} \end{aligned}$$

We convert $(21)_{10}$ from decimal to binary which gives $(10101.\_\_)_2$ and then we shift the binary point left by five places. Thus $(0.67)_{10} = (0.10101\_\_\_)_2$, and so $(4.67)_{10} = (100.10101\_\_\_)_2$. If we drop the unspecified part to have a finite number of bits only, then we have an approximation $(4.67)_{10} \approx (100.10101)_2$.

As was observed in class, we can sometimes obtain a better approximation than truncating (chopping) the unknown bits. In this example, the approximation error is $0.44 * 2^{-5}$ and since 0.44 is closer to 0 than to 1, we are better off truncating. For other examples, however, it might be better to "round up". This is the case of the one example in the slides, where we had

$$\begin{aligned} (0.247)_{10} &= 3.952 * 2^{-4} \\ &= (0.0011\_\_)_2 \end{aligned}$$

Since 0.952 is closer to 1 than to 0, we would obtain a better approximation if we replaced $0.952 * 2^{-4}$ by $1 * 2^{-4}$, and so the approximation would be $(.247)_{10} \approx (0.0100)_2$.