Binary number representation

The reason we humans represent numbers using decimal (the ten digits from 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, ..., 9\}, in particular, as sums of powers of ten, for example,

\[(238)_{10} = 2 \times 10^2 + 3 \times 10^1 + 8 \times 10^0\]

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

\[(11010)_2 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0.\]

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.

You know how to count in decimal, so 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 in the leftmost column.

<table>
<thead>
<tr>
<th>decimal</th>
<th>binary</th>
<th>binary (8 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>00000001</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>00000010</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>00000011</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>00000100</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>00000101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>00000110</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>00000111</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>00001000</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>00001001</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>00001010</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>00001011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>etc</td>
</tr>
</tbody>
</table>

In the left two columns, I only used as many digits or bits as I needed to represent the number. In the right column, I used a fixed number of bits, namely 8. 8 bits is called a byte.

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 '1' bits as a power of 2 (using the decimal representation you are familiar with) and then you add up these decimal numbers, e.g.

\[ 1101_2 = 16 + 8 + 2 = 26. \]

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

Here is an algorithm for converting a number to binary 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. fractions. If you want the remainder of the division, use “%” which is sometimes called the “mod” (or modulus) operator. After I present the algorithm, I will explain why it works.

**Algorithm 1** Convert decimal to binary

**INPUT:** a number \( m \)

**OUTPUT:** the number \( m \) expressed in base 2 using a bit array \( b[ ] \)

\[
i \leftarrow 0
\]

\[
\text{while } m > 0 \text{ do }
\]

\[
b[i] \leftarrow m \% 2
\]

\[
m \leftarrow m / 2
\]

\[
i \leftarrow i + 1
\]

\[
\text{end while}
\]

Note this algorithm doesn’t say anything about how \( m \) is represented. But in practice, since you are human, you will represent \( m \) in decimal.

**Example: Convert 241 to binary**

<table>
<thead>
<tr>
<th>( i )</th>
<th>( m )</th>
<th>( b[i] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>241</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>120</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

Thus, \((241)_{10} = (11110001)_{2}\). 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.
Let $m$ be a positive integer which is written in decimal. To divide $m$ by 10, we shift the digits to the right

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

We have dropped the rightmost digit 8 (which becomes the remainder) since integer division ignores terms with negative powers of 10 i.e $8 \times 10^{-1}$.

To multiply by 10, we shift the digits left by one place and put a 0 in the rightmost position. So, $23 \times 10 = 230$ and the reason is

$$23 = (2 \times 10^1 + 3 \times 10^0) \times 10 = 2 \times 10^2 + 3 \times 10^1 + 0 \times 10^0 = 230.$$ 

So dividing and the multiplying by 10 doesn’t get us back to the original number in this example. What is missing is the remainder part, which we dropped in the division. This missing part is $238 \% 10 = 8$.

Writing this in general for any positive integer $m$:

$$m = 10 \times (m/10) + (m \% 10).$$

In binary, the same idea holds. If we represent a number $m$ in binary and we divide by 2, then we shift right each bit by one position and drop the rightmost bit (which becomes the remainder). To multiply by 2, we shift the bits to the left by one position and put a 0 in the rightmost position. So, for example, if

$$m = (11011)_2 = 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^1 + 1 \times 2^0$$

then dividing by 2 gives

$$(11011)_2/2 = (1101)_2$$

then multiplying by 2 gives

$$(11010)_2 = 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^1.$$ 

More generally, for any $m$,

$$m = 2 \times (m/2) + (m \% 2).$$

Verify for yourself that, for any base that is a positive integer, and for any positive integer $m$, we can write

$$m = (m/base) \times base + (m \% base).$$

Notice that this is of the form,

$$dividend = quotient \times divisor + remainder.$$ 

or

$$a = q \times b + r$$

that we saw last lecture.
Now let’s apply these ideas to the algorithm for converting to binary. Representing a positive integer \( m \) in binary means that we write it as a sum of powers of 2:

\[
m = \sum_{i=0}^{n-1} b_i \cdot 2^i
\]

where \( b_i \) is a bit, which has a value either 0 or 1. So we write \( m \) in binary as a bit sequence \((b_{n-1} \ldots b_2 b_1 b_0)_2\). In particular,

\[
m \mod 2 = b_0 \\
m / 2 = (b_{n-1} \ldots b_1)_2
\]

Thus, we can see that the algorithm for converting to binary, which just repeats the mod and division operations, essentially just 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. The algorithm just reads off the rightmost bit of \( m \) each time that we divide it by 2.

<table>
<thead>
<tr>
<th>( i )</th>
<th>( m )</th>
<th>( b[i] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>((11110001)_2)</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>((1111000)_2)</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>((11110)_2)</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>((1111)_2)</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>((111)_2)</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>((11)_2)</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>((1)_2)</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Arithmetic in binary**

Let’s add two numbers which are written in binary. I’ve written the binary representation on the left and the decimal representation on the right.

\[
\begin{align*}
11010 \quad &\leftarrow \text{carries} \\
11010 + 1011 &\quad 26 + 11 \\
\hline
100101 &\quad 37
\end{align*}
\]

Make sure you see how this is done, namely how the “carries” work. For example, in column 0, we have \(0 + 1\) and get 1 and there is no carry. In column 1, we have \(1 + 1\) (in fact, \(1 \cdot 2^1 + 1 \cdot 2^1\)) and we get \(2 \cdot 2^1 = 2^2\) and so we carry a 1 over column 2 which represents the \(2^2\) terms. Make sure you understand how the rest of the carries work.
How many bits $N$ do we need to represent $n$?

[ASIDE: During the lecture, I switched variables at this point. From here on today, $n$ (not $m$) is the integer I am representing and $N$ (not $n$) is the number of bits. Hopefully this didn’t lead to too much confusion.]

Let $N$ be the number of bits needed to represent an integer $n$, that is,

$$n = b_{N-1}2^{N-1} + b_{N-2}2^{N-2} + \cdots + b_12 + b_0$$

where $b_{N-1} = 1$, that is, we only use as many bits as we need. This is similar to the idea that in decimal we don’t usually write, for example, 0000364 but rather we write 364. We don’t write 0000364 because the 0’s on the left don’t contribute anything.

Let’s derive an expression for how many bits $N$ we need to represent $n$. I will do so by deriving an upper bound and a lower bound for $N$. First, the upper bound: Since $b_{N-1} = 1$ and since each of the other $b_i$’s is either 0 or 1 for $0 \leq i < N-1$, we have

$$n \leq 2^{N-1} + 2^{N-2} + \cdots + 2 + 1. \quad (*)$$

To go further, I will next use of the following fact which some of you have not seen before: For any real number number $x$,

$$\sum_{i=0}^{N-1} x^i = \frac{x^N - 1}{x - 1}.$$ 

The proof of this fact is as follows. Take the sum on the left and multiply by $x - 1$. What happens?

$$\sum_{i=0}^{N-1} x^i(x - 1) = \sum_{i=1}^{N} x^i - \sum_{i=0}^{N-1} x^i = x^N - 1$$

where we have paid close attention to the indexes of the sum.

If we consider the case $x = 2$ we get

$$\sum_{i=0}^{N-1} 2^i = 2^N - 1.$$ 

For example, think of $N = 4$. We are saying that $16 - 1 = 8 + 4 + 2 + 1$.

Anyhow, getting back to our problem of deriving an upper bound on $N$, from equation (*) above, we have

$$n \leq 2^{N-1} + 2^{N-2} + \cdots + 2 + 1$$

$$= 2^N - 1 \quad \text{which I just proved}$$

$$< 2^N \quad \text{which makes the following step cleaner}$$

Taking the log (base 2) of both sides gives:

$$\log n < N.$$
Now let’s derive a lower bound. Since we are only considering the number of bits that we need, we have \( b_{N-1} = 1 \), and since the \( b_i \) are either 0 or 1 for any \( i < N - 1 \), we can conclude

\[
n \geq 1 \times 2^{N-1} + 0 \times 2^{N-1} + \ldots + 0 \times 2^1 + 0 \times 2^0 = 2^{N-1}
\]

and so

\[
\log n \geq N - 1.
\]

Putting the two inequalities together gives

\[
N - 1 \leq \log n < N
\]

Noting that \( N \) is an integer, we conclude that \( N - 1 \) is the largest integer that is less than or equal to \( \log_2 n \), that is \( N - 1 \) is \( \log_2 n \) rounded down. We write: \( N - 1 = \text{floor}(\log_2 n) \) where ”floor” just means ”round the number down, i.e. it is a definition. Thus, the number of bits in the binary representation of \( n \) is always:

\[
N = \text{floor}(\log_2 n) + 1.
\]

This is a rather complicated expression, and I don’t expect you to remember it exactly. What I do expect you to remember is that \( N \) grows roughly as \( \log_2 n \), that is, \( N \) is \( O(\log_2 n) \). Also, note it follows that the algorithm for converting an integer \( n \) into binary takes time \( O(\log_2 n) \) since the number of passes through the loop is just the number of bits \( N \).

**Other number representations**

Today we have considered only positive integers. Of course, sometimes we want to represent negative integers, and sometimes we want to represent numbers that are not integers, e.g. fractions. These other number representations are taught in COMP 273. If you wish to learn about them now, please have a look at the lecture notes on my COMP 273 public web page.