Tutorial 1

Gidon Rosalki

2025-03-26

1 Counting systems

We like base 10, which looks like 78396, computers use base 2, which is more 1101000010, and the Romans used letters MMCCCCXXI. These are all methods of representing amounts. A **base** is a method of representing numbers based of multiples of that number. The common method today is base 10, which contains the numbers 0 to 9, and each in a number represents that value multiplied by the relevant power of 10. For example

$$(5783.2)_{10} = 5 \times 10^3 + 7 \times 10^2 + 8 \times 10^1 + 3 \times 10^0 + 2 \times 10^{-1}$$

This can be generalised to many numbers, not just 10, with n digits.
Computer like base 2 since that is easy to represent using electricity, since 0 and 1 can be represented by simply if there is or is not electricity.
To represent a number in base r, let there be $r \geq 2 \in \mathbb{N}$, then a number in base r is of the style:

$$a_n a_{n-1} \dots a_1 a_0 . a_{-1} \dots a_{-m}$$

where $\forall j \ a_j \in \{0, 1, \dots, r-1\}$. Its value is defined as

$$\sum_{i=-m}^{n} a_i r^i$$

1.1 Arithmetic operations

Just because we are in a different base, does not mean that our arithmetic operations have changed any. You must simply be working in the new base for all the operations that you may wish to carry out.

2 Moving between numerical bases

To move to base 10, we have a simple algorithm:
Input: A number in some base $(a_n \dots a_{-m})_r$
Output: The value representation in base 10: $\sum_{i=-m}^{n} a_i r^i$
Example: $(253.1)_6 = 2 \cdot 6^2 + 5 \cdot 6^1 + 1 \cdot 6^0 + 1 \cdot 6^{-1} = (105.167)_{10}$

Change base 1

A number in some base *input*
Value representation in base 10 *output*
1: $i \leftarrow 0$
2: **while** $N > 0$ **do**
3: $\quad a_i \leftarrow N \mod r$
4: $\quad N \leftarrow \left\lfloor \frac{N}{r} \right\rfloor$
5: $\quad i \leftarrow i + 1$
6: **end while**

We can also use similar algorithms to transfer from base 10. We will now prove correctness for this algorithm. Let us

assume that the representation of the number N according to base r is

$$\begin{aligned} N &= a_n r^n + \dots a_0 r^0 \\ &= a_n r^n + \dots a_0 \\ &= r\left(a_n r^{n-1} + \dots + a_1\right) + a_0 \end{aligned}$$

So dividing by r gives us $a_n r^{n-1} + \dots + a_1$ with a remainder of a_0. After k steps we get the same concept with a remainder of a_k, so that may be our inductive step, so a proper inductive proof would generate the required statement that the algorithm is correct.

This algorithm works for moving from base a to base b, but you will probably get confused doing that, so it is instead recommended to go from a to 10, and then from 10 to b.
A thing to note is that bases that are the powers of 2 split apart into base 2 numbers incredibly nicely, for example in base 16, each number represents 4 binary numbers in the binary representation. This comes from the laws of logarithms:

$$\log_b(n) \cdot \log_c(b) = \log_c(n)$$

3 Representation of negative numbers in base 2

We need the following: To represent negative numbers, and to simplify mathematical operations as much as possible. We will look at today sign and magnitude representation, twos complement, and ones complement.

3.1 Size and magnitude

The $n-1$ number represents the sign: 0 is positive, 1 is negative. The rest of the numbers are the value, as we saw at the start

$$N = (-1)^{a_{n-1}} \sum_{i=0}^{n-2} a_i 2^i$$

For example:

- $01001 = (-1)^0 \cdot \left(2^3 + 2^0\right) = 9$
- $11101 = (-1)^1 \cdot \left(2^3 + 2^2 + 2^0\right) = -13$
- $00000 = 0$
- $10000 = 0$

Using n bits this can represent all the numbers from $-\left(2^{n-1} - 1\right)$ to $\left(2^{n-1} - 1\right)$. This is very readable, easy to use, but arithmetic operations are very complex, and there are two ways to represent 0.

3.2 One's complement

$$N = -a_{n-1}\left(2^{n-1} - 1\right) + \sum_{i=0}^{n-2} a_i 2^i$$

Examples:

- $01001 = -0 \cdot \left(2^4 - 1\right) + \left(2^3 + 2^0\right) = 1 + 8 = 9$
- $11011 = -1\left(2^4 - 1\right) + \left(2^3 + 2^1 + 2^0\right) = -15 + 11 = -4$

Here addition and subtraction is more simple, add the two numbers, and if there is a carry, remove it and add 1 to the result. This is simple to implement, easy to read, simple arithmetic operations, but has 2 ways to represent 0.

3.3 Two's complement

$$N = -a_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

Examples:

- $01001 = -0 \cdot 2^4 + \left(2^3 + 2^0\right) = 9$

- $11100 \to -(0011+1) \to -0100 = -2^2 = -4$

Here it is easy to implement, with simple arithmetic operations, and only 1 way to represent 0, but it is not particularly readable. Additionally, there can be overflow if the result of the addition/subtraction cannot be represented with the provided number of bits.

4 Floating points

The mass of the sum is about $1.9884335 \times 10^{30} kg$, which is about 31 numbers, or 103 bits. To deal with this we will use floating points, We will split the section to two areas, the mantissa (M) and the exponent (E) (The first bit is still the sign (S)). The result is $n = (-1)^S \cdot M \cdot 2^E$. Since this means that there are many ways to represent the same number, we want that E and M will give us a single representation, and will thus require $1 \leq |M| < 2$. This is called **normalised**. Which is to say, if there is a mentissa with the value M in binary, the whole number will be $N = 1.M \cdot 2^E$. We will say that the floating point representation is **biased** with B if the value is represented by $N = M \cdot 2^{E-B}$. This means that the exponent can be both negative and positive, and therefore we can represent both very large, and very small numbers. To represent the number in its final form, with normalisation and bias will be

$$N = 1.M \cdot 2^{E-B}$$

4.1 Examples

$E = 110,\ M = 00000 \implies 1.M \cdot 2^{E-B} = (1.00000)_2 \cdot 2^{6-3} = (1000.00)_2 = (8)_{10}$
$E = 110,\ M = 00001 \implies (1.00001)_2 \cdot 2^{6-3} = (1.00001)_2 \cdot 2^3 = (1000.01)_2 = (8.25)_{10}$
$E = 001,\ M = 00000 \implies (1.00000)_2 \cdot 2^{1-3} = (0.01000)_2 = (0.25)_{10}$
$E = 001,\ M = 00001 \implies (1.00001)_2 \cdot 2^{-2} = (0.0100001)_2 = (0.25785125)_{10}$

According to IEEE 754, for floats we have 8 bits for the exponent, 1 for the sign of the mantissa, and 23 for the mantissa itself. For doubles we have 1 for the sign, 11 for the exponent, and 52 for the mantissa.