CSCI 120 Introduction to Computation
It's only a matter of representation (draft)

Saad Mneimneh
Visiting Professor
Hunter College of CUNY

1 From bits to binary

Consider the number thirteen (nothing special about this number, but I picked it because I was born on that day). Now imagine I ask you to write the number thirteen on the board. You would most definitely write the following expression:

$$13$$

But, if you want, you could also write the expression:

$$\textit{thirteen}$$

What guides you to make the decision? It is a system of *representation*. Clearly, when someone says "the number thirteen", you know that we are most likely talking about numbers. So you invoke the system of representation for numbers. It happens to be the decimal system (a 10 digit alphabet).

But you certainly understand what "thirteen" is. It is a word in English that represents the number thirteen. Therefore, it is the same thing but using another alphabet (a 26 digit, or letter as commonly called, alphabet).

You even have more representations for number thirteen! You can imagine thirteen apples or thirteen oranges as a mental representation of that number. Therefore, thirteen is just a concept. You can express this concept in many ways. The fact that we use the decimal system more frequently is merely a biological coincidence (we have 10 fingers to count!).

Unfortunately, computers are less lucky. The way they evolved did not give them the chance to stick to the decimal system. Recall that the computer memory is just a reservoir of bits. Therefore, any concept must be represented using 0's and 1's only. Most importantly, and in particular, numbers.

That should not be a problem. We simply have to enrich our system of representations by adding one more representation or alphabet for numbers. The bits will serve as our digits (or letters) in this new alphabet. We call it the binary alphabet, or the binary number system (bi for two).

Before you get the chance to be surprised by the idea, note that it is possible to represent numbers using two digits only, namely 0 and 1. In fact, why be surprised at all? We can do it with only one digit after all. Here's how:

zero	0
one	00
two	000
three	0000
four	00000
five	000000
six	0000000
seven	00000000
⋮	⋮

Of course this is very inefficient. Imagine what the representation of a million would be. So we're lucky to have 0's and 1's.

2 Binary numbers: numbers in base 2

In 1679, Leibniz introduced the binary system for numbers and published it around the year 1701. The first computer to use binary representation for numbers was the Atanasoff-Berry machine developed at Iowa State University in 1941. When we say that we want to represent numbers in binary, instead of decimal, we are merely saying that we want to use **base 2**, instead of base 10.

In base 10, we have the units, the tens, the hundreds, the thousands, etc... For instance, the number 137 is $7 + 3 \times 10 + 1 \times 100$. Expressed as powers of 10, 137 is seven times ten to the power zero, plus three times ten to the power one, plus one times ten to the power two. This corresponds to the mathematical notation $137 = 7 \times 10^0 + 3 \times 10^1 + 1 \times 10^2 = 7 + 30 + 100$.

We can say almost the same thing in binary (i.e. base 2). We have the units, the twos, the fours, the eights, etc... Basically all the powers of 2. Let's compare:

2^0	1	10^0	1
2^1	2	10^1	10
2^2	4	10^2	100
2^3	8	10^3	1000
2^4	16	10^4	10000
2^5	32	10^5	100000
⋮	⋮	⋮	⋮

Since we are using base 2, we only need two digits 0 and 1, because a 2 can be expressed as a power of 2, much like the way we don't need a digit called **"10"** in the decimal system.

Let's figure out what number the binary expression 1101 represents. Well this is one thousand and one hundred and one in the decimal system. But since we are given that this is the binary representation of it, we need to treat it in base 2. So as we did for the case of decimal, this is 1 unit plus 0 twos plus 1 four plus 1 eight. Therefore, $1101 = 1 + 0 \times 2 + 1 \times 4 + 1 \times 8 = 1 + 0 + 4 + 8 = 13$. Of course, if we want to express 13 as a byte it will be 00001101. Now it is time for you to give me a random byte so we can see what number it represents (recall a byte is 8 bits). This will be a nice exercise to better understand binary numbers. Just in case, I am going to through some random bytes here:

10100101 11001010 00010001 11100111

3 Going from decimal to binary

It is probably easier to interpret a binary representation of a number than to come up with it. For instance, if you are given the binary representation 1101, you would tell that this is 13 by simply adding the corresponding powers of 2. But what if you are asked to come up with the binary representation of number 13, how would you do that?

Well, let's ask the following question. If you are given the number thirteen, how do you know that 13 is the decimal representation? You know it because you are so used to it by now. But how do you really obtain it? You simply ask what is the largest power of ten that is less than or equal to thirteen. The answer is ten ($10^1 = 10$). How many tens do we have in thirteen? The answer is one. Therefore, we should have a 1 in the tens position. Now we are left with three. What is the largest power of ten that is less than or equal to three? The answer is one ($10^0 = 1$). How many ones do we have in three? The answer is three. Therefore, we should have a 3 in the units position. We are left with zero. We stop. We obtain 13.

What have we done here? Well, note that this is similar to the problem of returning the exact change using the smallest number of coins. For instance, in the American currency, we have 1 cent, 5 cents, 10 cents, and 25 cents. If we want to return 46 cents, we can do it in may ways; for instance, 46 coins of 1 cent each (of course no one would like that!). But if we do it in such a way that we try to minimize the number of coins, then this may give us a unique representation of 46 cents. In our case, we would use $25 + 2 \times 10 + 1$. In particular, if we **use the larger values first**, then we obtain a unique way of returning 46 cents (and minimize the number of coins). Let's do the analogy with the previous algorithm. Aha! Algorithm? Yes, that's what we did before: a method for performing a certain task. The input is a number, the output is a representation of that number. The method: use the larger values first.

Here it goes. What is the largest coin that is less than or equal to 46? The answer is 25. How many 25's do we have in 46? The answer is 1. Therefore, we should have **1** 25 cent coin. We are left with 21. What is the largest coin that is less than or equal to 21? The answer is 10. How many 10's do we have in 21? The answer is 2. Therefore, we should have **2** 10 cent coins. We are left with 1. What is the largest coin that is less than or equal to 1? Then answer is 1. How many 1's do we have in 1? The answer is 1. Therefore, we should have **1** 1 cent coin. We are left with 0. We obtain 1 25 cent coin, 2 10 cent coins, and 1 1 cent coin.

Now let's apply the same algorithm for binary representations. We can consider the binary system as a currency system with the following coins (powers of 2): 1 cent, 2 cents, 4 cents, 8 cents, 16 cents, 32 cents, 64 cents, etc...

We are after the binary representation of the number 13. What is the largest power of 2 that is less than or equal to 13 (now we are using the decimal representation in writing because we are after the binary one, so there is no confusion)? The answer is 8. How many 8's do we have in 13? The answer is 1. Therefore, we should have a **1** in the eights position. We are left with 5. What is the largest power of 2 that is less than or equal to 5? The answer is 4. How many 4's do we have in 5? The answer is 1. Therefore, we should have a **1** in the fours position. We are left with 1. What is the largest power of 2 that is less than or equal to 1. The answer is 1. How many 1's do we have in 1? The answer is 1. Therefore, we should have a **1** in the ones position. We are left with 0. We obtain 00001101.

4 A better algorithm

In the previous algorithm, we had to repeatedly guess the largest value (among the ones we have, e.g. 1, 2, 4, 8, 16, 32, ...) that fits. Generally, we do not like guessing. Luckily, we don't have to. We can actually work backward. Here's how. Let's stick to the same problem of finding the binary representation of number 13. We start by dividing 13 by 2 and record the remainder. We get 1, since $13 = 2 \times 6 + 1$. The quotient is 6. Now we repeat the same steps starting from 6. We divide 6 by 2 and record the remainder. We get 0, since $6 = 2 \times 3 + 0$. The quotient is 3. Now we repeat the same steps starting from 3. We divide 3 by 2 and record the remainder. We get 1, since $3 = 2 \times 1 + 1$. The quotient is 1. Now we repeat the same steps starting from 1. We divide 1 by 2 and record the remainder. We get 1, since $1 = 2 \times 0 + 1$. The quotient is 0. We stop. Let's write down the remainders we got in order from right to left. We get: 1101, which is the binary representation of 13.

The algorithm:

- divide the value by 2 and record the remainder
- as long as the quotient obtained is not 0, continue to divide the newest quotient by 2 and record the remainder
- now that a quotient of 0 has been obtained, the binary representation of the original value consists of the remainders listed from right to left in the order they were recorded

5 Addition

Of course numbers would be of not much use if we can't perform arithmetic operations on them. The basic arithmetic operation is addition. Good news: addition (also true for subtraction, multiplication, and division) works the same way (no reason for it not to).

$$\begin{array}{r} 0 \\ +\ 0 \\ \hline 0 \end{array} \qquad \begin{array}{r} 0 \\ +\ 1 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ +\ 0 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ +\ 1 \\ \hline 10 \end{array}$$

Try to add 13 and 13 in binary:

$$\begin{array}{r} 1101 \\ +\ 1101 \\ \hline \end{array}$$