**Understanding how numbers are stored in the computer using only 0 and 1:**

Hi, up for a light read on CS fundamentals? I am very excited for this post. Integers form the basis of everything we do on the computer. But, do you really know how computers store numbers?  
Will someone ever ask you this in an interview? Probably not.  
Will it help you as a programmer? Yes :)  
This will be a good tutorial for the beginners and a quick revision for the experienced.  
Let's begin. I hope there are no errors but if they have creeped in, let me know in the comments.

The day-to-day system of writing numbers is called decimal. It uses 10 digits (0 to 9) to represent any number. When we write a number in decimal, each digit is multiplied by a power of 10, which is it's distance from the rightmost bit.  
For example, 89 = (8 \* 10^1) + (9 \* 10^0).

In a similar way, we can write all numbers using only 0 and 1, and multiplying each digit by  
a power of 2, which is it's distance from the rightmost bit.  
A computer only knows 0's and 1's. That's why we need to represent numbers in binary.

Let's do some simple calculations:-

i) What does (101) represent in binary?  
A. (101) = (1 \* 2^2) + (0 \* 2^1) + (1 \* 2^0) = (4) + (0) + (1) = 5.

ii) What does (111) represent in binary?  
A. (111) = (1 \* 2^2) + (1 \* 2^1) + (1 \* 2^0) = (4) + (2) + (1) = 7.

Hope, this was clear. Also, when we say that a bit is set, it means that the bit is 1. To set a bit means to make it 1.

Now, let's put our newly gained knowledge to use.

Notice that, since 2 is an even number, all powers of 2 will contribute an even number, except the rightmost bit, which contributes 1. Hence, in every odd number, the last bit will be set.  
Let's take some examples to make that sink.

Let's represent some odd decimal numbers in Binary:-

Decimal | Binary

3 | 11

5 | 101

9 | 1001

15 | 1111

Do you see that the rightmost bit is 1 for all the odd numbers? Similarly, the rightmost bit is 0 for all even numbers. It makes sense right? Since the last bit contributes 1 which is an odd number.

Hopefully everything was clear upto this point.

Now, if we have 3 binary digits, what is the maximum and minimum integer that can be represented?  
The minimum is (000) which is 0. Maximum is (111) which is (7).

IMP: An n-bit binary number can represent numbers in the range 0 to (2^n)-1.

Now, you might ask, how do we represent negative numbers then?  
Well, let's see how we can represent -5.  
Since, 5 is represented as (101), we might represent -5 as -(101). But, we don't really want to include special symbols while storing numbers inside the computer. So, we discard this idea.

We can use the sign-magnitude representation for our problem.  
We will use the left-most bit as the sign bit. If the leftmost bit is 0, it means that it is positive and if it is 1, the number is negative.

Hence,  
5 is represented as (0101) whereas (-5) is represented as (1101). (Left most bit is the sign bit)  
7 is represented as (0111) whereas (-7) is represented as (1111).  
20 is represented as (0001 0100) whereas (-20) is represented as (1001 0100).

Notice that I used 8 bits to represent 20. Most operating systems will use 32 bits to represent integers. So, most bits in between will be filled with 0.

So, in summary, in sign-magnitude representation,  
i) The left most bit is the sign bit.  
ii) Rest all bits represent the magnitude of the number.  
Example. (1000 0010) represents (-2) because left most bit is 1 so it's negative.

Now, let's see if you have understood till now:-

Can you guess what the following binary numbers in sign-magnitude represent?  
(0010)  
(1010)  
(1111)  
(0000 1001)  
(1000 0110)

I hope you did them, you can find correct answers at the end of the post. (No, kidding. Just do it yourselves :) )

Now, we have solved the problem of representing negative numbers without using any special symbol.  
I have a task for you. Let's stick to 4 bits to represent numbers.  
Can you write sign-magnitude for 0?

We can write 0 as (0000) or (1000). One is +0 and other is -0.  
If you have been programming for a while, you might realize that the operation of comparing something to 0 is done quite frequently and it's unnecessary to check it's equality with both these numbers. It would be easier if we had just one representation for zero.

In short, the first issue is, 0 has two representations .i.e.(0000) and (1000).

Great! Let's do some operations. Do you remember the rules for binary addition?

Here they are:

0 + 0 = 0 with carry as 0  
1 + 0 = 1 with carry as 0  
0 + 1 = 1 with carry as 0  
1 + 1 = 0 with carry as 1

Addition of two numbers will take place bitwise .i.e. we add two numbers bit by bit.

Let's add 2 and 5.

(2) = (0010)  
(5) = (0101)

2 + 5 = 0010 + 0101 = 0010

+ 0101

= 0111

What does (0111) represent? 7 right? +7 to be precise.

Try adding -3 and 4.

(-3) = (1011)  
(+4) = (0100)

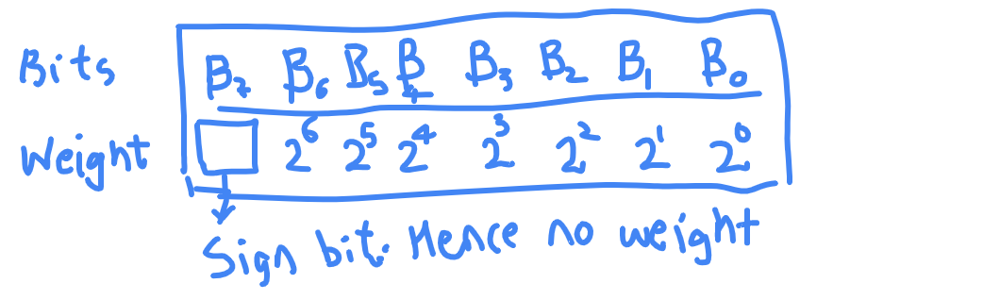
-3 + (+4) = 1011 + 0100 = 1111

Wew! What does 1111 represent? -7? Yep. But, is that the answer? Nope. This is another issue with sign-magnitude representation. We need to consider the sign of both numbers before doing any operation.

We have discussed the two major drawbacks of sign-magnitude representation. We will see how we can improve our current representation using the two's complement representation. The drawbacks are overcome by two's complement representation. This is the reason sign-magnitude represenation is rarely used. But, we don't deny it's existence, it is still used in some areas and it's better to just know about it.

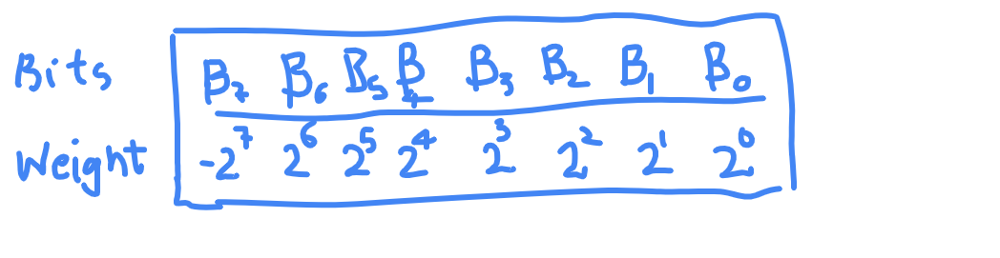
Let's see what two's complement representation of a number is.

Firstly let's define the weight of a bit. The weight of a bit is the amount it contributes to the actual number if the corresponding bit in the number is set. Look at the following image to understand better.



This is what we have seen already. This is what we did to get the value of (0101) as 5. By multiplying the weight by corresponding bit.

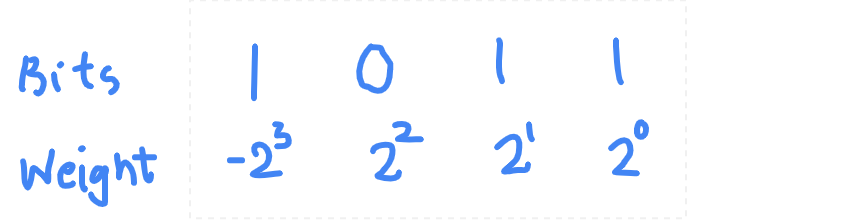
The left most bit ofcourse doesn't have any weight as it just aids us in knowing if number is positive or negative.

This is where it becomes interesting. The left-most bit is sign-bit in Twos complement too but now, it also has a weight just like every other bit albeit with negative sign.  


Notice the weight of Bit7(B7).

This tremendously changes how each integer is stored in computer.

Let's take 1011 for example. Let's see what it represents.



We use the same method of getting the value.  
Multiply the bit and weight.  
So (1011) in Twos complement is:  
1 \* (-(2^3)) + 0 \* (2^2) + 1 \* (2^1) + 1 \* (2^0)  
= -8 + 0 + 2 + 1  
= -8 + 3  
= -5

Hence, 1011 represents -5.

What is representation for -3? 1101. (Please verify it by adding weights)  
What is representation for +4? 0100. (Please verify this too)

The representation for 0 in Twos complement is 0000 .i.e +0. So, a simple comparison to know if a value is 0 or not.

Alright, a small question before we part.

If I give you a number in Twos complement, can you tell me if it is positive or negative?

In sign-magnitude, it is simple, just look at the sign bit and you know if it is positive or negative. But, what about 2's complement?  
Think before you read further.

The answer is yes, you can know if a number if +ve or -ve just by looking at leftmost bit.

If leftmost bit is 0, the number is always greater than or equal to zero because each weight except the leftmost is positive and leftmost bit is 0, it means that there is no negative contribution from any bit and so value can never be negative.

If leftmost bit is 1, the number is always negative. Even if all other bits are 1 then too the number is negative. Let's check it.

(1111) = (-8 + 4 + 2 + 1) = -1

(1101) = (-8 + 4 + 0 + 1) = -3

You can prove it with simple mathematics involving powers of 2 but that is left as an exercise for the reader.

From now, we will write all the representations in twos complement.  
Also, if you want to know, how an integer is represented in your programming language, you can use following code:

For example, C++ integer is of 32 bits. So, if I want to know all the bits of a number 'n', I can run the following code. I believe that the code will be similar in other programming languages with minor changes.

void printTwosComplementRepresentation(int n)

{

// We start from 31 and go to 0.

for (int i = 31; i >= 0; --i)

{

if ((1 << i) & n) // Check if the ith bit is set or not

std::cout << 1 ;

else std::cout << 0 ;

}

}

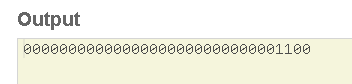
int main()

{

printTwosComplementRepresentation(12);

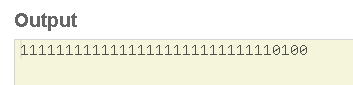
return 0;

}

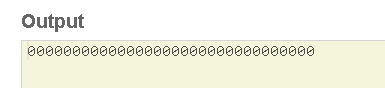


Above image is for +12.

For -12, it looks something like below.



Now, I am interested in knowing how 0 is represented. I ran the same program with n as 0 and it gave following output.



So, leftmost bit is 0 in 0.

Do you wonder what if leftmost bit is 1 and all other bits are 0?  
Since, we know that weight of leftmost bit is negative, the final number will be very big negative number and it's absolute value will be a power of 2.

So, that's it. This is how numbers are represented in memory. If you want to learn how various operations are done with Twos complement, you are free to check out other amazing sources on the web.