Definitions:

Algorithms -
Step-by-step description of how to accomplish a task.

High-level Language -
Languages that are readable by humans, these languages are compiled into lower-level languages that can be interpreted by the computer. Anyone can view these languages, and any computer with the appropriate software installed can compile the associated high-level language.

Intermediate Language -
A partially compiled language, which can be compiled further into machine language. Intermediate languages are typically able to be interpreted by many different computers.

Machine Language -
Binary (0's and 1's) are interpreted and executed by the computer. Machine language is typically unique to specific computers, and therefore cannot be interpreted by many different computers.

Concepts:

Numeric Systems -
Each column in a specific system contains 'x' number of values, where 'x' correlates to the base of the system. IE: A binary (bi implies 2) system can hold 2 values in each digit (0, 1), whereas a decimal (deci implies 10) system can hold 10 values in each column (0 through 9).

We can calculate the value of any number in any base using this information.

First we have to understand how the values of a column are affected by the position of the column. The magnitude of each column can be determined by raising the base of the system to the position of the column, and multiplying the product by the value represented in the column. IE: In a decimal system, if we wanted to determine the magnitude of the second column (_ 2 _ _) in the number '7495' we would begin by identifying the base, column position, and value. The base of the decimal system is 10, the column position is 2, and the value is 4. We raise the base of the system to the column position: 10^2 = 100. We then multiply this product by the value, 4: 4*100 = 400. The magnitude of the second column is 400.

Here are a couple full examples of calculating the entire magnitude of a number (all columns) in other bases:

Binary: 1101  base = 2  column span = {3, 2, 1, 0}
(2^3)*1 = (8)*1 = 8
(2^2)*1 = (4)*1 = 4
(2^1)*0 = (2)*0 = 0
(2^0)*1 = (1)*1 = 1
8 + 4 + 0 + 1 = 13
1101 (base 2) converted into decimal, is 13.
Octal: 273   base = 8   column span = {2, 1, 0}

\[(8^2)\times2 = (64)\times2 = 128 \]
\[(8^1)\times7 = (8)\times7 = 56 \]
\[(8^0)\times3 = (1)\times3 = 3 \]
\[128 + 56 + 3 = 187 \]

273 (base 8) converted into decimal, is 187.

Hexadecimal: 1AF   base = 16   column span = {2, 1, 0}

*NOTE* since you cannot have two digits in one column, then letters are designated to represent values greater than 9. We need to represent another 6 possible values; so, we use the letters 'A' through 'F'. Where \{A, B, C, D, E, F\} represent \{10, 11, 12, 13, 14, 15\}, respectively.

\[(16^2)\times1 = (256)\times1 = 256 \]
\[(16^1)\timesA = (16)\times10 = 160 \]
\[(16^0)\timesF = (1)\times15 = 15 \]
\[256 + 160 + 15 = 431 \]

1AF (base 16) converted into decimal, is 431.

Converting binary to hexadecimal can be done very easily.

Break the binary number into 'bytes' (which represent 4 bits each, where each bit is a column). IE: for the binary number 1010111001110011, we can separate the columns into bytes to read as 1010 1110 0111 0011. What is the maximum value of each byte? 1111 (base 2) converted into decimal, is 15. Since 0 through 15 are possible value, a hexadecimal number can be used to represent each byte. By calculating the decimal value of each byte, and then converting to hexadecimal:

\[1010 = 10 = A \]
\[1110 = 14 = E \]
\[0111 = 7 = 7 \]
\[0011 = 3 = 3 \]

This means we know that 1010 1110 0111 0011 (base 2) is equal to A E 7 3 (base 16), or 1010111001110011 (base 2) = AE73 (base 16).

Converting binary to hexadecimal by each byte is extremely useful.

Signed/Unsigned Numbers -

So far, we have only expressed binary values as unsigned numbers, limiting ourselves to the positive realm. However, we need a means in which to express negative numbers as well. This is accomplished through the use of a two’s complement system. With signed numbers, the first bit (or sign-bit) of a binary value is used as an indicator of positive or negative direction. If the sign-bit is 0, the number is positive. If the sign-bit is 1, the number is negative. The two’s complement rule says to flip all the bits and then add one to get the number, which allows us to get to the -16 value. IE: In a signed number system, the binary value 00000 would equal 0 in decimal, and the binary value 10000 would equal -16 in decimal. Notice that 0 is included in the positive range, whereas 0 is not included in the negative range. The 16 possible values of a positive 4-bit binary number range from 0 to 15, while the 16 possible values of a negative 4-bit binary number range from -1 to -16.
Range of Numbers -

Now that we understand how numeric values are represented in different base systems, and how the number of bits correlate to the possible magnitude of a number, we can calculate the range of a number based on the number of bits.

Using a binary system as an example, let us look at how exactly the number of bits correlate with the range of signed and unsigned numbers. Let 'n' represent the number of bits used in the representation of a binary value. If n=1 there are only 2 possible values (0,1). If n=2 there are 4 possible values (00, 01, 10, 11).

The pattern we observe here is that the number of possible values represented by an n-bit unsigned binary number is $2^n$ because there are two possibilities, 0 or 1, in n places. For example, telephone numbers have 10 different possibilities in 10 different spots, which means we can have up to $10^{10}$ different telephone numbers. Since all unsigned numbers are positive, the number of possible values range from 0 to $(2^n)-1$. Subtracting 1 from $2^n$ accommodates for the fact that 0 is included as one of our possible values. For instance, as addressed earlier, a 4-bit number has a maximum positive magnitude of 15, accounting for 16 ($2^4$) possible values from 0 to 15.

For signed numbers, one less bit will be available to represent magnitude. Therefore, the maximum number of positive values for a signed number range from 0 to $(2^{(n-1)})-1$. For instance, a 4-bit binary number (0111) has a sign-bit of value 0, indicating a positive value. If we look at the remaining bits (n-1 bits) we can determine the maximum positive range for this number. Since there are 3 remaining bits, the maximum possible value is $(2^3)-1$, just like the unsigned numbers, but the minimum is $-(2^3)$, instead of zero as an automatic unsigned minimum.

When calculating the range of negative signed numbers (unsigned cannot be negative), we use the same method we did for calculating positive signed ranges, except we do not include 0 as one of the possible values. In other words, one bit will be used as a sign bit to represent negative polarity, and the range will span -1 to $-(2^{(n-1)})$. For a 4-bit negative number the range will be determined by 3 bits, with an initial value of -1. 3 bits can represent $2^3$ (8) possible values {-1, -2, -3, -4, -5, -6, -7, -8}.

In summary, the formulas for calculating a numbers range are as follows:

* Subtract 1 from n if signed.
* Subtract 1 from range if positive.

Unsigned minimum = 0  
Unsigned maximum = $(2^n)-1$

Signed minimum = $-(2^{(n-1)})$  
Signed maximum = $(2^{(n-1)})-1$

Negative numbers -

An important thing to notice is how we count negative values, and how it differs from counting positive values. For positive values, we have a sign-bit of 0, and we begin counting from 0000 to 0111 (for a 4 bit number). For negative values, we have a sign-bit of 1, and we begin counting from 1111 to 1000. This means the value of the signed binary
number 1111 in decimal is -1, and the value of the signed binary number 1000 in decimal is -8.

Positive counting: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111.
0, 1, 2, 3, 4, 5, 6, 7.

Negative counting: 1111, 1110, 1101, 1100, 1011, 1010, 1001, 1000.
-1, -2, -3, -4, -5, -6, -7, -8.

Overflow behavior -
Notice what happens when we add 1 to the maximum positive value:

In binary: (signed) 0111 + 0001 = 1000 (unsigned) 1111
+ 0001 = 0000
In decimal: (signed) 7 + 1 = -8 (unsigned) 15
+ 1 = 0

This demonstrates overflow behavior. Binary addition will continue to count as we would expect. But when the sign-bit is flipped, by means of carry-over from the addition of 1, the sign changes, and the remaining bits are interpreted differently as a result. This behavior (in decimal representation) appears to the user as if adding 1 to the positive value of greatest magnitude, causes the value to flip to the negative value of greatest magnitude. This works inversely as well when subtracting 1 from the negative value of greatest magnitude.