Lecture #1: Bits, Bytes, and Binary

CS106E Spring 2018, Young

The binary number system underlies all modern computers. In this lecture we'll take a look at the binary number system and some of the implications of using binary numbers. Having a solid grounding in binary will set us up to explore digital images and digital music in the next two lectures.

Along the way, we'll see why we buy iPhones with 256 Gigabytes of memory instead of 250 billion bytes of memory. We'll discover that sometimes adding to a large positive number leads to a large negative number such as 32767 + 15 = -32754. We'll find that binary representation results in 0.1 + 0.2 = 0.30000000000000004.

We'll end with a quick look at how using the right encoding scheme can result in your foreign language website looking great, and the wrong encoding scheme can result in in Russian characters or in some cases just complete gibberish showing up in the middle of your French website.

What is a Bit?

- Internally all information is stored using electronic switches. These switches can be either on or off.
    - When a switch is on, we sometimes think of this as representing the number 0, or representing the condition "false".
    - When a switch is off, this corresponds to the number 1 or the condition "true".
    - This means we can think of all the switches in computer memory as representing a whole bunch of 0s and 1s.

- We are used to working with the decimal number system which has 10 digits (0, 1, 2, 3, …, 8, and 9). A number system having only 2 digits is called a ***binary number system***.

- We also sometimes say that the decimal number system is **base 10** and the binary number system is **base 2**.

- We say that each electronic switch represents a single *binary digit*. The term binary digit is shortened to ***bit***.

What is a Byte?

- Trying to look over an undifferentiated mass of 0s and 1s is difficult, so we organize them into groups of eight.
- A set of 8 bits is referred to as **byte**.

Bits and Bytes in other Mediums

- While bits and bytes are represented by the state of electronic switches in computer memory, they can be represented differently in other storage media.

- On magnetic disk, such as a hard drive, or on a magnetic tape we magnetically polarize a section of the disk in one direction or in the opposite direction to indicate values of 0 or 1.
- On an optical disk such as a CD, DVD, or Bluray, we place tiny pits on the disk, the depth of the pit determines the value of the bits which are being stored.

- Ultimately regardless of the type of media the data is still represented as binary values.

Counting in Binary

- Let's take a quick look at how to count in binary. This will lead us to a better understanding of how binary works and how it compares to decimal.

- We sometimes indicate whether a number is representing a binary number or a decimal number by using a subscript
  - 436_{10} is 436 in base 10 (i.e., in the decimal number system)
  - 1001_2 is 1001 in base 2 (i.e., in the binary number system)

- Counting in Decimal

  0 + 1 = 1
  1 + 1 = 2
  2 + 1 = 3
  3 + 1 = 4
  …
  8 + 1 = 9
  9 + 1 = ???

  When we get to 9 and add 1 to it in base 10, we don't have another digit so instead we reset the 1's column to 0 and instead put a 1 in the next column, which represents 10's so

  9 + 1 = 10

  Something similar happens to make 99 + 1 = 100 and 999 + 1 = 1000

- Counting in Binary

  0 + 1 = 1
  1 + 1 = ???

  Okay, we've now reached the point where we were when we tried adding $9_{10} + 1_{10}$ In binary there is no 2 digit, so instead we reset the 1's column and put a 1 in the next column which represents 2's so in binary

  1 + 1 = 10
  10 + 1 = 11
  11 + 1 = ???

  Now we're at the same position we were in decimal when we added $99_{10} + 1_{10}$ With binary we'll have to reset both the 1's column and the 2's column and carry the result into the 3rd column which represents 4's:

  11 + 1 = 100

- Here are the first sixteen binary digits and their decimal equivalents

0_{10}	0_2	8_{10}	1000_2
1_{10}	1_2	9_{10}	1001_2
2_{10}	10_2	10_{10}	1010_2
3_{10}	11_2	11_{10}	1011_2
4_{10}	100_2	12_{10}	1100_2
5_{10}	101_2	13_{10}	1101_2
6_{10}	110_2	14_{10}	1110_2
7_{10}	111_2	15_{10}	1111_2

Combinations of Binary Numbers

- Generally on the computer we'll need to determine in advance how many bits or bytes to set aside to represent a given quantity.
- The number of bits will determine the range of values that may be stored.

    - For example, if I set aside just a single bit to represent a quantity, that may be used to represent the decimal numbers 0 or 1 or states with only two conditions such as true or false.
    - If I set aside 4 bits, we can represent the decimal numbers 0 to 15 (as shown above), we could instead use those 4 bits to represent positive and negative decimal numbers from -8 to +7, or we could represent conditions which have 16 possible states.
        - As an example, if we were storing information about Canadian residents, we could use 4 bits to represent which Province in Canada they lived in, since there are only 12 Provinces.

0000 = British Columbia	0011 = Quebec
0001 = Yukon	0100 = Ontario
0010 = Prince Edward Island	…

        - We could not represent states in the United States, since 4 bits cannot represent 50 different combinations

- In general we can represent 2^n states in n-bits

1 bit	2 states	$2^1 = 2$
2 bits	4 states	$2 \times 2 = 2^2 = 4$
3 bits	8 states	$2 \times 2 \times 2 = 2^3 - 8$
4 bits	16 states	$2 \times 2 \times 2 \times 2 = 2^4 = 16$
5 bits	32 states	
6 bits	64 states	
7 bits	128 states	
8 bits	256 states	

- If you've shopped for consumer electronics lately these numbers should look familiar to you.

    This is why we can purchase iPhones with memory amounts like 64, 128, and 256 not amounts like 50, 100, or 250.

k's. Megs, and Gigs.

- You may have noticed that in addition to consumer electronics coming in memory amounts such as 64, 128, or 256 we also say that we are purchasing them in megabytes, gigabytes, or terabytes. We don't buy 64 million bytes of memory or 32 billion bytes of memory.
    - Thousands, millions, and billions are powers of 10, not powers of 2
    - When working with the computer we need measuring amounts which reflect our binary foundation not the decimal system we are used to.

- In computing we use kilobytes, megabytes, and gigabtyes

    - A ***Kilobyte*** is 2^{10} = 1024 bytes.
        - This is the number of combinations we can store in 10 bits
        - It is almost, but not quite the same as 1000 bytes
    - A ***Megabyte*** is 2^{20} = 1,048,576 bytes.
        - This is the number of combinations we can store in 20 bits
        - It is slightly larger than a million bytes
    - A ***Gigabyte*** is 2^{30} = 1,073,741,824 bytes.
        - This is the number of combinations we can store in 30 bits
        - It is slightly larger than a billion bytes
    - A ***Terabyte*** is 2^{40} and roughly corresponds to a trillion

- You may also run into the following measurements:

    Tera = 2^{50}, Peta = 2^{60}, Exa = 2^{70}, Zetta = 2^{80}, Yotta = 2^{90}

    For example CISCO estimates that global Internet traffic currently exceeds 1 Zettabyte per year.

- While the majority of memory measurements inside the computer use these power-of-2 denominated amounts there are some exceptions
    - Electrical Engineers refuse to participate in binary, so measurements by them are generally based on the power of 10. For example data transfer rates over digital communications are generally based on powers of 10 not powers of 2.
    - Apple has decided powers of 2 are bad and has switched to powers of 10 in some measurements they display to users.

- In an attempt to distinguish between powers of 10 and powers of 2, some computer scientists use the terms kibi, mebi, and gibi (example: 64 mebibtyes). I haven't found these to be in particularly widespread usage, but you may run into them.
- In this system:
    - kilo = 10^3 whereas kibi = 2^{10}
    - mega = 10^6 whereas mebi = 2^{20}
    - giga = 10^9 whereas gibi = 2^{30}

Problems with Binary Numbers in Computers

- Problems with binary integers
    - As we saw previously we need to determine in advance how many bits to set aside to represent a given quantity

- If we don't set aside enough bits, we can't represent a given value.
    - For example if I set aside 7 bits to represent a person's age, that allows me to represent ages from 0 to 127
    - If longevity treatments allow humans to live beyond 127, my program is now broken.

- When a number in a calculation exceeds the maximum number which can be represented, we have ***overflow***.
    - For example, using 16-bits we can represent numbers from -32,768 to +32,767.
    - If I have a variable storing 32,767 and I add one to it, I don't get 32,768, since I can't represent that in my 16-bits.
    - Instead I get -32,768.
    - If you're working with a computer and have a very large positive number and it suddenly unexpectedly changes to a very large negative number, there's a very good chance you've just experienced overflow.

- Working with binary floating point numbers can also cause problems.

    - 0.1 + 0.2 = 0.30000000000000004
        - We know 0.1 + 0.2 should equal 0.3.
        - However, if we run this in many computer languages we'll discover it actually equals 0.30000000000000004
    - 0.7 + 0.1 = 0.7999999999999999
        - This clearly should equal 0.8
        - But instead we get 0.7999999999999999

    - What's going on here?
        - Some numbers we cannot represent in a set number of decimal places.
            - 2 / 3 doesn't equal 0.67 or 0.66667 or even 0.6666667 instead we write that 2/3 = $0.\overline{6}$ where the overline above the 6 indicates a repeating decimal.
            - Irrational numbers such as Pi, also can't be represented in a finite number of decimal places.
        - Repeating numbers and irrational numbers are not the same in binary and decimal.
            - The number 0.1 in base 10 is equal to $0.0\overline{0011}$ or 0.0001100110011001100110011… in binary
        - Inside the computer, floating-point numbers are represented in a set number of bits, just as integer numbers are. While intuitively we would expect 2/3rds when represented by a set number of decimal places to round to 0.66667, we don't expect something like 0.1 + 0.7 to cause problems, but it does because our instincts about decimal numbers doesn't carry to binary numbers.

    - *To prevent unexpected surprises such as the ones we've seen here, programmers are cautioned not to store money using standard floating point techniques.*

Storing text and words using bits and bytes

- We can encode text using bits by using different bit combinations to represent different letters.
    - For example, we could say that the bit combination 1000001 corresponds to the letter 'A' and the bit combination 1000010 corresponds to the letter 'B'.
    - In fact this is exactly how these letters are represented in the computer.

- The standard encoding of text inside a computer is called ***ASCII***.[1]
    - ASCII stores upper-case letters, lower-case letters, punctuation characters, along with some special control characters using different combinations of 7-bits.
- A number of different schemes were developed to support International Text
    - ASCII only used 128 combinations of the 256 available in a byte, so the ISO-8859 standard used the remaining combinations to represent other types of characters.
        - ISO-8859-1 for example used the other 128 characters to represent Western European characters such as à, é, ô, or ñ.
        - ISO-8859-7 used the extra combinations for Greek letters like Δ, λ, or π
    - The SHIFT-JIS character set used 16-bits to represent Japanese characters
    - These different schemes were not compatible and loading a file encoded with one scheme in a program meant for another scheme will result in gibberish.

- The ***Unicode*** character encoding was developed to provide a common character format for all human languages.
    - Modern Unicode represents characters using anywhere from 1-byte to 4-bytes, depending on the character represented.
    - The 1-byte characters correspond directly to the original ASCII standard. So Unicode and ASCII can interoperate well.
    - However, as 4-bytes gives us many, many possible bit combinations, Unicode can be used to represent almost every language used by humans.
        - Unicode includes bit combination for 87,882 characters used for Chinese, Japanese, and Korean.
        - It includes encodings for Egyptian Hieroglyphs, Sumerian Cuneiform, and Mycenaean Linear B
        - Unicode includes bit combinations to represent musical notes
        - Unicode also includes bit combinations for **Emojis**.

    - There are several different methods for encoding Unicode. The variant you are most likely to run into is **UTF-8**. This encoding is widely used on the web.

[1] ASCII stands for American Standard Code for Information Interchange, but you don't need to know that, and in fact many Computer Scientists probably don't.