L13: Binary Representation of Data

Zeros and ones

Lucas A. J. Bastien

E7 Spring 2017, University of California at Berkeley

February 15, 2017

Version: release

Announcements

Lab 04 is due on February 17 at 12 pm (noon)

Wednesday:

- Binary representation of data

Friday:

- Discussion, Practice questions
- Written feedback

Submit your own work for E7 assignments

We will control for plagiarism in your E7 submissions

- **Submit your own code!** It is okay to talk about the general approach used to solve a problem with other students, it is **not** okay to share your code
- Penalty for plagiarism: undropable -100 for the lab (for both the original author and the copier(s))

It is better to have a dropable 0, 50, or 70 for a given lab than an undroppable -100

The only code that is not your own code and that you can copy and re-use (with or without modifications) in your E7 assignments is:

- The code found in **this semester's E7 lectures/discussions**:
  - Lecture slides, diaries, m-files that I upload to bCourses
- The code in the **solutions to this semester's E7 assignments**
  - These solutions will be posted on bCourses

What is binary representation of data

Binary representation of data: representation of data using only two different "symbols" (typically: zeros and ones)

Binary representations that you may already have heard of:

- Morse Code
- Braille

A **bit** is a digit that can take only one of two values: 0 or 1

Decimal system

"**deci**" means "**10**"

The numbers that we usually use rely on the decimal system (also known as the base 10 system). For example:

$$\begin{array}{ccccccccc} & 4 & & 5 & & 0 & & 3 \\ = & 4 \times 10^3 & + & 5 \times 10^2 & + & 0 \times 10^1 & + & 3 \times 10^0 \end{array}$$

In the decimal system:

- There are **ten** different digits (0 to 9)
- Each digit is "multiplied by a power of **ten**"

Binary system: introduction

"**binary**" means "**2**"

Example of binary representation:

$$
\begin{array}{ccccccccc}
 & 1 & & 1 & & 0 & & 1 \\
\rightarrow & 1 \times 2^3 & + & 1 \times 2^2 & + & 0 \times 2^1 & + & 1 \times 2^0 \\
= & 13 & & & & & &
\end{array}
$$

In binary systems:

- There are **two** different digits (0 and 1)
- Each digit is "multiplied by a power of **two**"

Three binary representations of integers

Example with 8 bits:

	1^{st} bit	2^{nd} bit	3^{rd} bit	4^{sth} bit	5^{th} bit	6^{th} bit	7^{th} bit	8^{th} bit
Unsigned	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Sign-magnitude	sign	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Two's complement	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

$$11101000 \to 2^7 + 2^6 + 2^5 + 2^3 = 232 \qquad (\textit{unsigned})$$

$$\to -(2^6 + 2^5 + 2^3) = -104 \qquad (\textit{sign magnitude})$$

$$\to -2^7 + 2^6 + 2^5 + 2^3 = -24 \qquad (\textit{two's complement})$$

Binary representation: practice question

How many different numbers can be represented with n bits?

(A) n

(B) $n!$

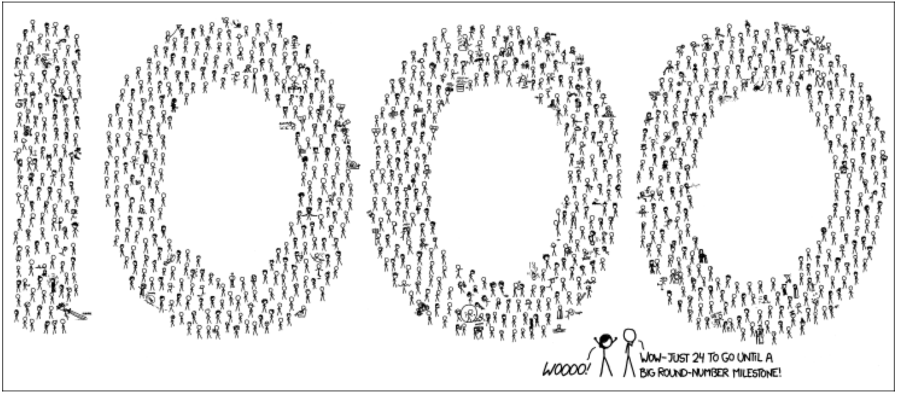
(C) n^2

(D) 2^n

- The first bit can have one of two values
- The second bit can have one of two values
- The third bit can have one of two values
- And so on...

Round numbers

"Just 24 to go until a big round-number milestone!"



- The unsigned binary representation of 1000 is 1111101000
- The unsigned binary representation of 1024 is 10000000000

Image from https://xkcd.com/1000/, retrieved on February 14$^{\text{th}}$ 2017

Floating point numbers

In the decimal system, digits after the decimal point represent negative powers of ten. For example:

$$475.865 = 4 \times 10^2 + 7 \times 10^1 + 5 \times 10^0 + 8 \times 10^{-1} + 6 \times 10^{-2} + 5 \times 10^{-3}$$

We can use a similar approach with the binary system:

$$1011.11 \to 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 11.75$$

With a fixed number of bits (*e.g.*, 32 bits or 64 bits), **where to put the decimal point?**

- Too few bits for the decimal part: low accuracy
- Too many bits for the decimal part: cannot represent large numbers

The IEEE standard for floating point numbers

Motivation for the standard:

- All data in computers are in binary format
- Computers need to be able to represent large numbers
- Computers need to be able to represent small numbers with accuracy
- Computer memory and hard-drive space is not infinite

The IEEE standard defines algorithms to represent floating point numbers with variable accuracy

- Using 32 bits ("single precision")
- Using 64 bits ("double precision")

The IEEE standard for floating point numbers

$$\boxed{\text{number} = (-1)^s 2^{e-b}(1+f)}^{\star}$$

- s is the first bit
- b is called the bias, it allows for negative powers of 2
- f is called the significand, it allows for accuracy

Example with 32 bits (single precision):

Bit 1	Bits 2 through 9	Bits 10 through 32
1	0 1 1 1 1 1 0 0	0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Indicates the **sign**:
Positive if 0
Negative if 1

Indicate the value of e
(Unsigned integer)
$\rightarrow$ **magnitude** of the number

Indicate the value of f as:
$2^{-1}, 2^{-2}, \ldots, 2^{-23}$
$\rightarrow$ gives **accuracy**
(*i.e.* significant digits)

*this is the general formula, there are special cases

The IEEE standard for floating point numbers

$$\boxed{\text{number} = (-1)^s 2^{e-b}(1+f)}$$

Example with 32 bits (single precision):

Bit 1	Bits 2 through 9	Bits 10 through 32
1	0 1 1 1 1 1 0 0	0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
s	e	f

- $s = 1$, the number is negative
- $e = 2^6 + 2^5 + 2^4 + 2^3 + 2^2 = 124$
- $b = 127$ (constant)
- $f = 2^{-2} + 2^{-3} = 0.375$

$$\text{number} = (-1)^1 \times 2^{124-127} \times (1 + 0.375) = -0.171875$$

Single and double precision

$$\text{number} = (-1)^s 2^{e-b}(1+f)$$

Single precision

- s: 1 bit
- e: 8 bits
- $b = 2^7 - 1$: 127
- f: 23 bits
- Total: 32 bits = 4 bytes

Double precision

- s: 1 bit
- e: 11 bits
- $b = 2^{10} - 1$: 1023
- f: 52 bits
- Total: 64 bits = 8 bytes

Matlab uses double precision by default for numerical values

```
>> a = 1;
>> whos
  Name      Size            Bytes  Class     Attributes
  a         1x1                 8  double
```

Single and double precision (continued)

Single precision

- **Range:** $\pm \approx 3.4 \times 10^{38}$
- Can represent $2^{32} \approx 10^{9}$ different numbers

Double precision

- **Range:** $\pm \approx 1.8 \times 10^{308}$
- Can represent $2^{64} \approx 10^{19}$ different numbers

Matlab cannot represent all real numbers

There is an **infinite number of real numbers**. Matlab uses a **finite number of bits** to represent each number. **Consequence:** there is a non-zero gap between any two consecutive numbers that one can represent in binary. For example with single precision (32-bits):

The binary representation of 15 is:

0	1 0 0 0 0 0 1 0	1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

The next smaller number that we can represent is 14.99999904632568359375000000:

0	1 0 0 0 0 0 1 0	1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

The next bigger number that we can represent is 15.00000095367431640625000000:

0	1 0 0 0 0 0 1 0	1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

Measure the gap between two "consecutive" numbers

The Matlab built-in function eps measures the gap around a number

- Small gap (high accuracy) if the number has a small magnitude
- Big gap (low accuracy) if the number has a large magnitude
  - the gap is still small compared to the number itself (*i.e.* the relative accuracy is still high)

```
>> eps(0)
ans =
  4.9407e-324

>> eps(1e-10)
ans =
   1.2925e-26

>> eps(1e100)
ans =
   1.9427e+84

>> eps(-1e100)
ans =
   1.9427e+84
```

Binary representations of characters

ASCII: American Standard Code for Information Interchange

The ASCII standard associates with each character a numerical (integer) code. Each of these codes can then be represented in binary format

Character	ASCII code
A	65
B	66
Z	90
[	90
]	91
a	97
b	98
z	122

The ASCII table contains 128 characters (a–z, A–Z, 0–9, punctuation)

ASCII (continued)

In Matlab, when characters are used in arithmetic expressions, they are converted to their corresponding numerical codes. Use functions `double` and `char` to convert between numerical codes and characters

- Codes 0 to 127 are converted to corresponding ASCII characters
- Conversion of higher codes depends on computer's configuration

```
>> double('Hello')
ans =
    72   101   108   108   111

>> 'Hello' * 2
ans =
   144   202   216   216   222

>> 'Hello' + 'lab04'
ans =
   180   198   206   156   163

>> char([72, 101, 108, 108, 111])
ans =
Hello
```

ASCII (continued)

Limitation of ASCII?

- Limited to English alphabet
- Few punctuation and other symbols

Widespread alternative: Unicode

- Characters present in ASCII have the same code in unicode (backward compatibility)
- Many alphabets
- Even Emojis!

```
>> char([9786, 32, 87, 101, 100, 110, 101, 115, 100, 97, ...
         121, 33, 32, 73, 32, 104, 111, 112, 101, ...
         32, 121, 111, 117, 32, 10084, 32, 69, 55, 33])
```