Ceng 111 – Fall 2021
Week 7b

Credit: Some slides are from the “Invitation to Computer Science” book by G. M. Schneider, J. L. Gersting and some from the “Digital Design” book by M. M. Mano and M. D. Ciletti.

Previously on CENG111!

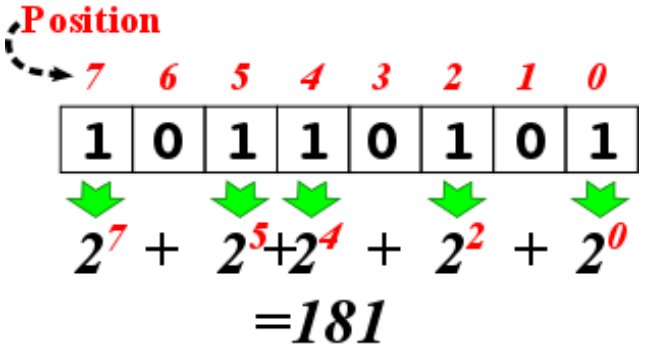
Binary Representation of Numeric Information

- Decimal numbering system
  - Base-10
  - Each position is a power of 10

    $3052 = 3 \times 10^3 + 0 \times 10^2 + 5 \times 10^1 + 2 \times 10^0$

- Binary numbering system
  - Base-2
  - Uses ones and zeros
  - Each position is a power of 2

    $1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$



Binary Representation of Numeric Information (continued)

- Sign/magnitude notation

  1 101 = - 5

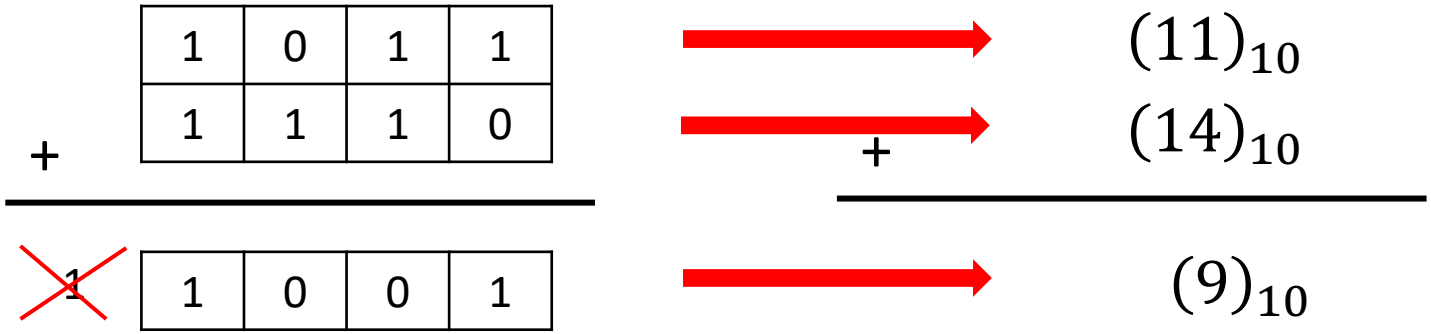
  0 101 = + 5

- Problems:
  - Two different representations for 0:
    - 1 000 = -0
    - 0 000 = +0
  - Addition & subtraction require a watch for the sign! Otherwise, you get wrong results:
    - 0 010 (+2) + 1 010 (-2) = 1 100 (-4)

Previously on CENG 111!

Arithmetic in Computers is Modular

Let's add two numbers in binary
(Assume that there is no sign bit)



In other words:

- Numbers larger than or equal to 16 (2^4) are discarded in a 4-bit representation.
- Therefore, $11 + 14$ yields 9 in this 4-bit representation.
- This is actually modular arithmetic:

$$11 + 14 \; mod \; 16 \equiv 9 \; mod \; 16$$

Previously on CENG111!

Binary Representation of Numeric Information (continued)

- Two's complement instead of sign-magnitude representation
  - Positive numbers have a leading 0.
    - 5 => 0101
  - The representation for negative numbers is found by subtracting the absolute value from 2^N for an N-bit system:
    - -5 => $2^4 - 5 = 16 - 5 = (11)_{10} => (1011)_2$
- Advantages:
  - 0 has a single representation: +0 = 0000, -0 = 0000
  - Arithmetic works fine without checking the sign bit:
    - 1011 (-5) + 0110 (6) = 0001 (1)
    - 1011 (-5) + 0011 (3) = 1110 (-2)

Binary Representation of Numeric Information (continued)

Previously on CENG111!

- Shortcut to convert from "two's complement" :
  - If the leading bit is zero, no need to convert.
  - If the leading bit is one, invert the number and add 1.
- What is our range?
  - With 2's complement we can represent numbers from -2^{N-1} to $2^{N-1} - 1$ using N bits.
  - 8 bits: -128 to +127.
  - 16 bits: -32,768 to +32,767.
  - 32 bits: -2,147,483,648 to +2,147,483,647.

Binary Number	Decimal Value	Value in Two's Complement
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

Previously on CENG111!

Binary Representation of Numeric Information (continued)

- Example:
  - We want to compute: 12 – 6
  - 12 => 01100
  - -6 => -(00110) => (11001)+1 => (11010)
- 12 – 6 =

```
  01100
+ 11010
-----------
  00110 => 6
```

So, addition and subtraction operations are simpler in the Two's Complement representation

Previously on CENG111!

Why does Two's Complement work?

- One perspective:
  - Inversion and addition of a 1-bit correspond effectively to subtraction from 0 – i.e., negative of a number.
  - Negative of a binary number X: $(00\ldots00)_2$ - $(X)_2$
  - Note that $(00\ldots00)_2 = (11\ldots11)_2 + (1)_2$
  - In other words:
    - $(00\ldots00)_2$ - $(X)_2 = (11\ldots11)_2 - (X)_2 + (1)_2$.

(i.e., how we find two's complement)

Inversion

Why does Two's Complement work?

- Or, equivalently:
  - i - j mod 2^N = i + (2^N-j) mod 2^N
  - Example:
    - Consider X and Y are positive numbers.
    - $X + (-Y) = X + (2^N - Y)$
      $= 2^N - (Y - X) = -(Y - X) = X - Y$

Previously on CENG111!

Why does Two's Complement work?

- A smart trick used in mechanical calculators
  - To subtract b from a, invert b and add that to a. Then discard the most significant digit.

http://en.wikipedia.org/wiki/Method_of_complements

Today

- Data Representation

Administrative Notes

- Scratch assignment
- Midterm date:
  - 22 December, Wednesday, 18:00

Binary Representation of Real Numbers

Conversion of the digits after the dot into binary:

- 1st Way:
  - 0.375 → 0x½ + 1x¼ + 1x1/8 → 011
- 2nd Way:

	Fraction		Multiplier		Whole		Fraction
Step 1	0.375	×	2	=	0	.	75
Step 2	0.75	×	2	=	1	.	5
Step 3	0.5	×	2	=	1	.	0

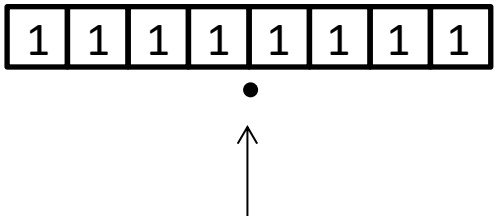
The result:

.	0	1	1

Continue until fraction is zero

Binary Representation of Real Numbers

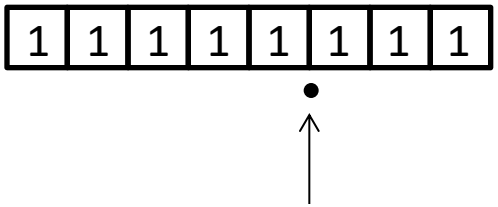
- Approach 1: Use fixed-point
  - Similar to integers, except that there is a decimal point.
  - E.g.: using 8 bits:



$= 1\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0 + 1\times2^{-1} + 1\times2^{-2} + 1\times2^{-3} + 1\times2^{-4}$
$= 15.9375$

Binary Representation of Real Numbers

- Location of the decimal point changes the value of the number.
  - E.g.: using 8 bits:



$= 1\times2^4 + 1\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0$
$1\times2^{-1} + 1\times2^{-2} + 1\times2^{-3} = 31.875$

Assumed decimal point

Binary Representation of Real Numbers

- Problems with fixed-point:
  - Limited in the maximum and minimum values that can be represented.
  - For instance, using 32-bits, reserving 1-bit for the sign and putting the decimal point after 16 bits from the right, the maximum positive value that can be stored is slightly less than 2^{15}.
  - Allowing larger values gives away from the precision (the decimal part).

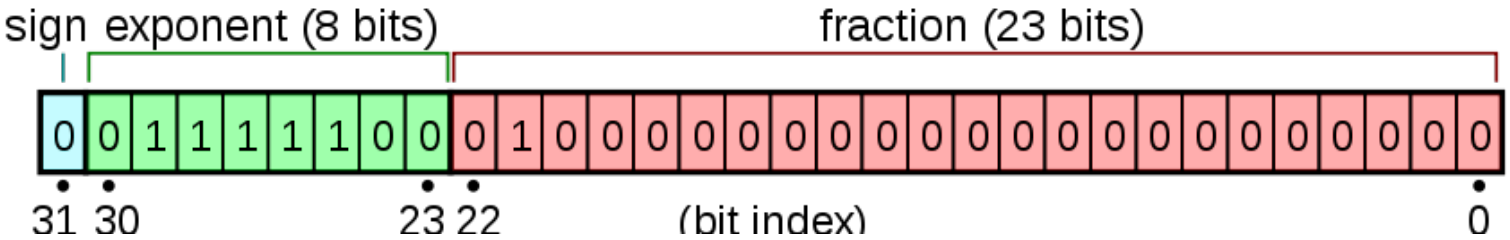
Binary Representation of Real Numbers

- Solution: Use scientific notation: a x 2^b *(or* $\pm M \times B^{\pm E}$*)*
  - Example: 5.75
    - 5 → 101
    - 0.75 → ½ + ¼ → $2^{-1}+2^{-2}$ → $(0.11)_2$
    - 5.75 → $(101.11)_2 \times 2^0$
- Number is then normalized so that the first significant digit is immediately to the left of the binary point
  - Example: 1.0111×2^2
- We take and store the mantissa and the exponent.

Binary Representation of Real Numbers

- This needs some standardization for:
  - where to put the decimal point
  - how to represent negative numbers
  - how to represent numbers less than 1

IEEE 32bit Floating-Point Number Representation



= 0.15625

$$= (-1)^{\text{sign}}(1.b_{-1}b_{-2}...b_{-23})_2 \times 2^{e-127}$$

- M x 2^E $(2-2^{-23})\times 2^{127}$
- Exponent (E): 8 bits
  - Add 127 to the exponent value before storing it
  - **E can be 0 to 255 with 127 representing the real zero.**
- Fraction (M - Mantissa): 23 bits
- $2^{128} = 1.70141183 \times 10^{38}$

IEEE 32bit Floating-Point Number Representation

- Example: 12.375
- The digits before the dot:
  - $(12)_{10} \rightarrow (1100)_2$
- The digits after the dot:
  - 1st Way: 0.375 → 0x½ + 1x¼ + 1x1/8 → 011
  - 2nd Way: Multiply by 2 and get the integer part until 0:
    - 0.375 x 2 = 0.750 = 0 + 0.750
    - 0.750 x 2 = 1.50 = 1 + 0.50
    - 0.50 x 2 = 1.0 = 1 + 0.0
- $(12.375)_{10} = (1100.011)_2$
- NORMALIZE (move the point): $(1100.011)_2 = (1.100011)_2 x 2^3$
- Exponent: 3, adding 127 to it, we get 1000 0010
- Fraction: 100011
- Then our number is: 0 10000010 10001100000000000000000

Why add bias to the exponent?

- It helps in comparing the exponents of the same-sign real-numbers without looking out for the sign of the exponent.

Binary Number	Decimal Value	Value in Two's Complement	Value with bias 7
0000	0	0	-7
0001	1	1	-6
0010	2	2	-5
0011	3	3	-4
0100	4	4	-3
0101	5	5	-2
0110	6	6	-1
0111	7	7	0
1000	8	-8	1
1001	9	-7	2
1010	10	-6	3
1011	11	-5	4
1100	12	-4	5
1101	13	-3	6
1110	14	-2	7
1111	15	-1	8

To read more on this:
https://blog.angularindepth.com/the-mechanics-behind-exponent-bias-in-floating-point-9b3185083528

IEEE 32bit Floating-Point Number Representation

- Zero:
  - Exponent: All zeros
  - Fraction: All zeros
  - +0 and -0 are different numbers but they are equal!
- Infinity:
  - Exponent: All ones
  - Fraction: All zeros
- Not a number (NaN):
  - Exponent: All ones
  - Fraction: non-zero fraction.

http://steve.hollasch.net/cgindex/coding/ieeefloat.html

IEEE 32bit Floating-Point Number Representation

- What is the maximum positive IEEE floating point value that can be stored?
  - Just less than 2^{128} [$(2 - 2^{-23}) \times 2^{127}$ to be specific]
  - Why? 2^{128} is reserved for NaN.

- Check out these useful links:
  - http://steve.hollasch.net/cgindex/coding/ieeefloat.html
  - http://babbage.cs.qc.cuny.edu/IEEE-754.old/Decimal.html

IEEE 32bit Floating-Point Number Representation

- Now consider 4.1:
  - 4 => $(100)_2$
  - 0.1 =>
    - x 2 = 0.2 = 0 + 0.2
    - x 2 = 0.4 = 0 + 0.4
    - x 2 = 0.8 = 0 + 0.8
    - x 2 = 1.6 = 1 + 0.6
    - x 2 = 1.2 = 1 + 0.2
    - x 2 = 0.4 = 0 + 0.4
    - x 2 = 0.8 = 0 + 0.8
    - …….

- So,
  - Representing a fraction which is a multiple of $1/2^n$ is lossless.
  - Representing a fraction which is not a multiple of $1/2^n$ leads to accuracy loss.

BINARY REPRESENTATION OF TEXT ETC.

Binary Representation of Textual Information

- Characters are mapped onto binary numbers
  - ASCII (American Standard Code for Information Interchange) code set
    - Originally: 7 bits per character; 128 character codes
  - Unicode code set
    - 16 bits per character
  - UTF-8 (Universal Character Set Transformation Format) code set.
    - Variable number of 8-bits.

Binary Representation of Textual Information (cont'd)

ASCII
7 bits long

Decimal	Binary	Val.
48	00110000	0
49	00110001	1
50	00110010	2
51	00110011	3
52	00110100	4
53	00110101	5
54	00110110	6
55	00110111	7
56	00111000	8
57	00111001	9
58	00111010	:
59	00111011	;
60	00111100	<
61	00111101	=
62	00111110	>
63	00111111	?
64	01000000	@
65	01000001	A
66	01000010	B

Hex.	Unicode	Charac.
0x30	0x0030	0
0x31	0x0031	1
0x32	0x0032	2
0x33	0x0033	3
0x34	0x0034	4
0x35	0x0035	5
0x36	0x0036	6
0x37	0x0037	7
0x38	0x0038	8
0x39	0x0039	9
0x3A	0x003A	:
0x3B	0x003B	;
0x3C	0x003C	<
0x3D	0x003D	=
0x3E	0x003E	>
0x3F	0x003F	?
0x40	0x0040	@
0x41	0x0041	A
0x42	0x0042	B

Unicode
16 bits long

Partial listings only!

Your book is wrong about the number of characters in the ASCII table.

To store the 4-character string "BAD!" in memory, the computer would store the binary representation of each individual character using the above 8-bit code.

BAD! = 00000010 00000001 00000100 10000001
B A D !

We have indicated above that the 8-bit numeric quantity 10000001 is interpreted as the character "!". However, as we mentioned earlier, the only way a computer knows that the 8-bit value 10000001 represents the symbol "!" and not the unsigned integer value 129 (128 + 1) or the signed integer value –1 (sign bit = negative, magnitude is 1) is by the context in which it is used. If these 8 bits are sent to a display device that expects to be given characters, then this value will be interpreted as an "!". If, on the other hand, this 8-bit value is sent to an arithmetic unit that adds unsigned numbers, then it will be interpreted as a 129 in order to make the addition operation meaningful.

To facilitate the exchange of textual information, such as word processing documents and electronic mail, between computer systems, it would be most helpful if everyone used the same code mapping. Fortunately, this is pretty much the case. Currently the most widely used code for representing characters internally in a computer system is called **ASCII**, an acronym for the *A*merican *S*tandard *C*ode for *I*nformation *I*nterchange. ASCII is an international standard for representing textual information in the majority of computers. It uses 8 bits to represent each character, so it is able to encode a total of $2^8 = 256$ different characters. These are assigned the integer values 0 to 255. However, only the numbers 32 to 126 have been assigned so far to printable characters. The remainder either are unassigned or are used for nonprinting control characters such as tab, form feed, and return. Figure 4.3 shows the ASCII conversion table for the numerical values 32–126.

However, a new code set called **UNICODE** is rapidly gaining popularity because it uses a 16-bit representation for characters rather than the 8-bit format of ASCII. This means that it is able to represent $2^{16} = 65{,}536$ unique characters instead of the $2^8 = 256$ of ASCII. It may seem like 256 characters are more than enough to represent all the textual symbols that we would ever need—for example, 26 uppercase letters, 26 lowercase letters, 10 digits, and a few dozen special symbols, such as +=–{}][\;'"?><.,;'%$#@. Add that all together and it still totals only about 100 symbols, far less than the 256 that can be represented in ASCII. However, that is true only if we limit our work to Arabic numerals and the Roman alphabet. The world grows more connected all the time—helped along by computers, networks, and the Web—and it is critically important that computers represent and exchange textual information using alphabets in addition to these 26 letters and 10 digits. When we start assigning codes to symbols drawn from alphabets such as Russian, Arabic, Chinese, Hebrew, Greek, Thai, Bengali, and Braille, as well as mathematical symbols and special linguistic marks such as tilde, umlaut, and accent grave, it becomes clear that ASCII does not have nearly enough room to represent them all. However, UNICODE, with space for over 65,000 symbols, is large enough to accommodate all these symbols and many more to come. In fact, UNICODE has defined standard code mappings for more

UTF-8 Illustrated

Bits	Last code point	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
7	U+007F	0xxxxxxx					
11	U+07FF	110xxxxx	10xxxxxx				
16	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx			
21	U+1FFFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx		
26	U+3FFFFFF	111110xx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	
31	U+7FFFFFFF	1111110x	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx

Character		Binary code	Binary UTF-8
$	U+0024	0100100	00100100
¢	U+00A2	00010100010	11000010 10100010
€	U+20AC	0010000010101100	11100010 10000010 10101100
𤭢	U+24B62	000100100101101100010	11110000 10100100 10101101 10100010

How about a text?

- Text in a computer has two alternative representations:

1. A fixed-length number representing the length of the text followed by the binary values of the characters in the text.
   - Ex: “ABC” =>

   00000011 01000001 01000001 01000001 (3 ‘A’ ‘B’ ‘C’)

2. Binary values of the characters in the text ended with a unique marker, like “00000000” which has no value in the ASCII table.
   - Ex: “ABC” =>

   01000001 01000001 01000001 00000000 (‘A’ ‘B’ ‘C’ END)

Binary Representation of Sound and Images

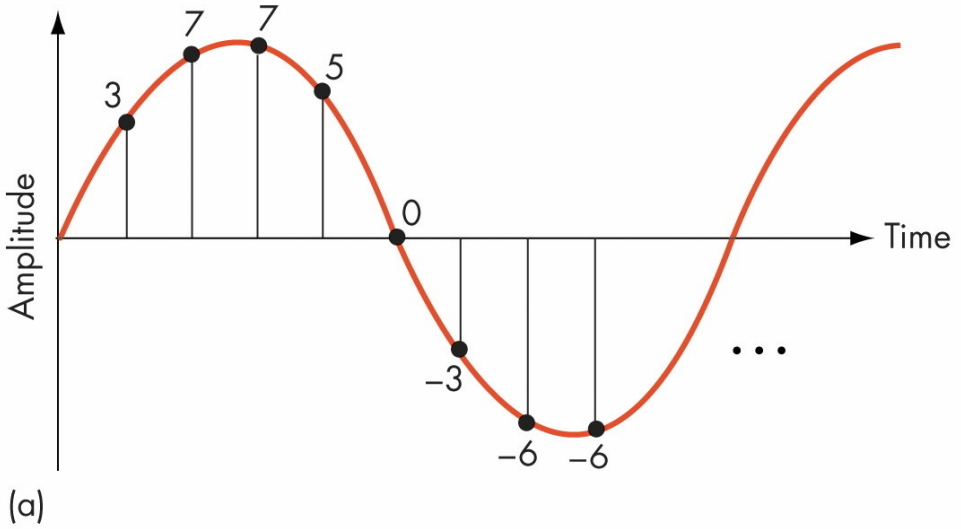
- Multimedia data is sampled to store a digital form, with or without detectable differences
- Representing sound data
  - Sound data must be digitized for storage in a computer
  - Digitizing means periodic sampling of amplitude values

Binary Representation of Sound and Images (continued)

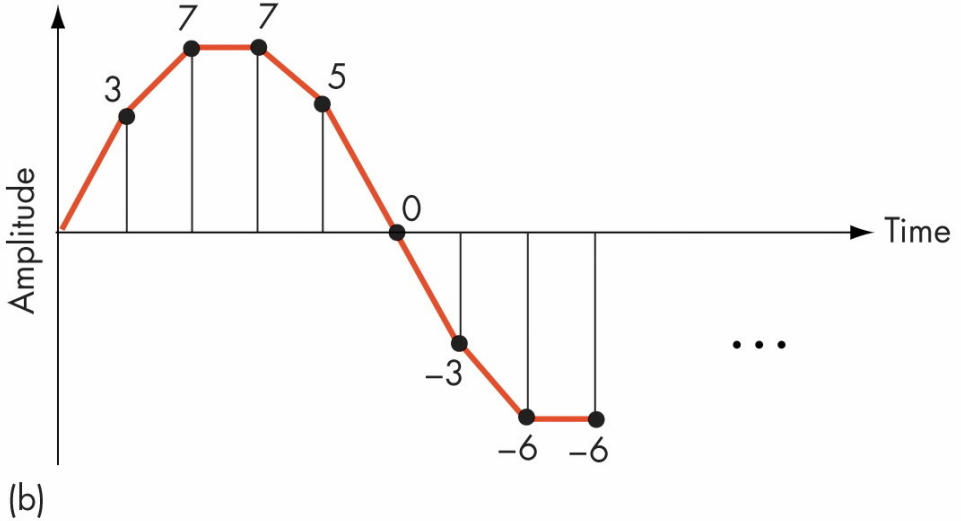
- From samples, original sound may be approximated
- To improve the approximation:
  - Sample more frequently (*increase sampling rate)*
  - Use more bits for each sample value *(↑ bit depth)*

Digitization of an Analog Signal

(a) Sampling the Original Signal

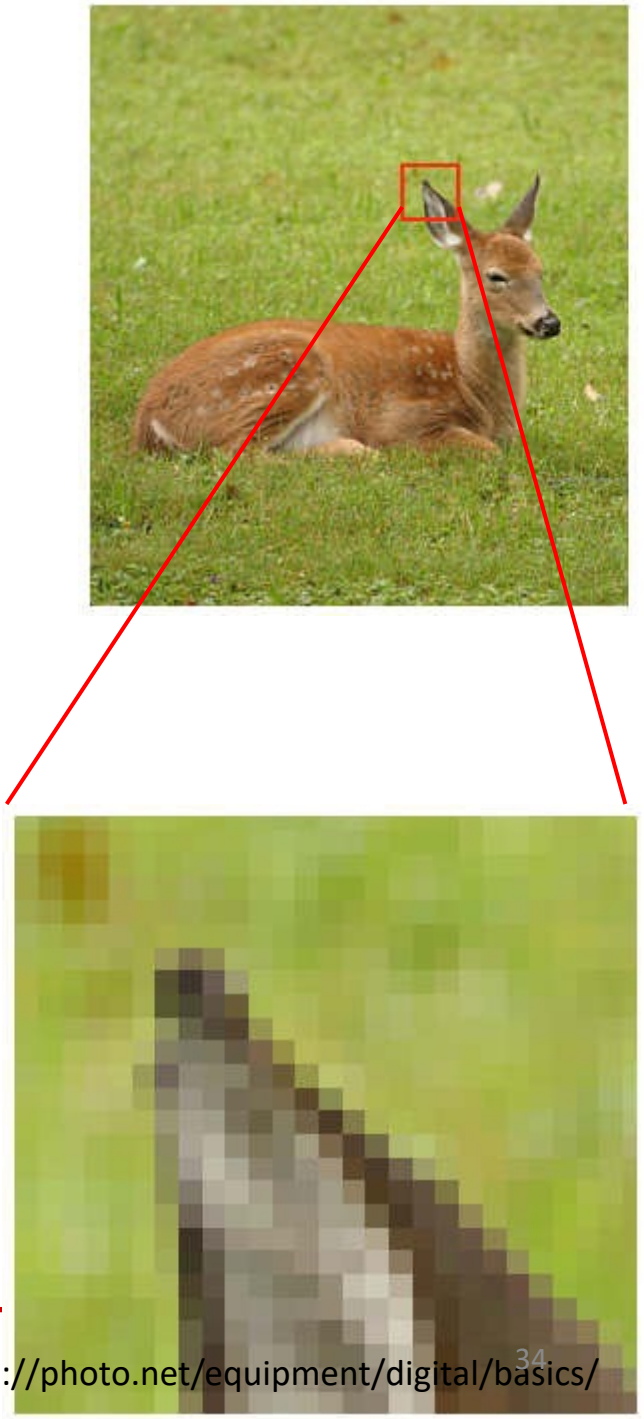


(b) Recreating the Signal from the Sampled Values



Binary Representation of Sound and Images (continued)

- Representing image data
  - Images are sampled by reading color and intensity values at even intervals across the image
  - Each sampled point is a pixel
  - Image quality depends on number of bits at each pixel
  - More image information: http://cat.xula.edu/tutorials/imaging/grayscale.php



http://photo.net/equipment/digital/basics/

The Reliability of Binary Representation

- Electronic devices are most reliable in a bistable environment
- Bistable environment
  - Distinguishing only two electronic states
    - Current flowing or not, or
    - Direction of flow
- Computers are bistable: hence binary representations

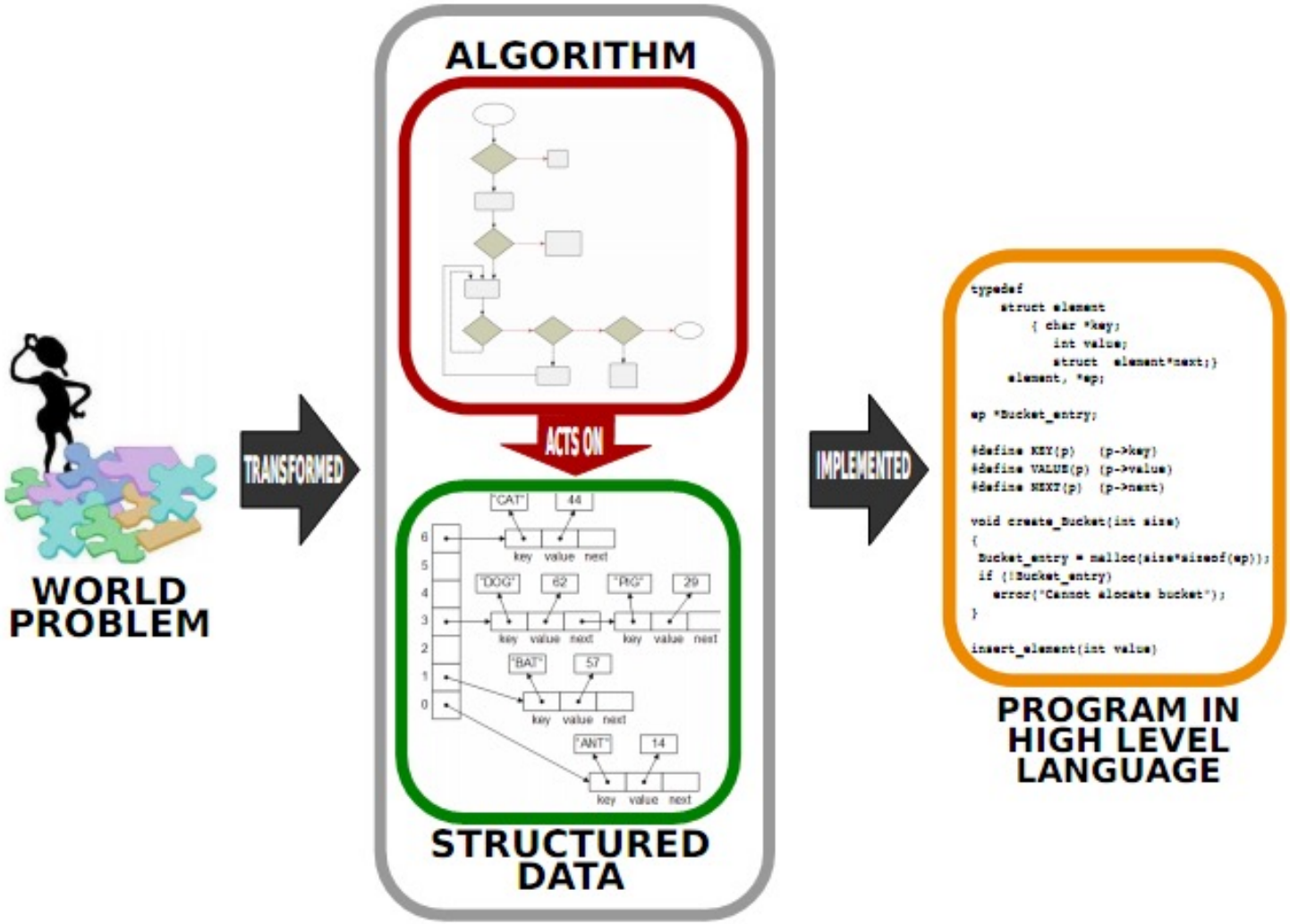
Further reading

https://pp4e-book.github.io/chapters/ch3_data_representation.html

Now

- Basic data & structured data

Design of a solution



What is data?

- **Data**: Information to be processed to solve a problem.
- Identify the data for the following example problems:
  - Find all wheat growing areas in a terrestrial satellite image.
  - Given the homework, lab and examination grades of a class, calculate the letter grades.
  - Alter the amplitude of a sound recording for various frequencies.
  - Extrapolate China's population for the year 2040 based on the change in the population growth rate up to this time.
  - Compute the launch date and the trajectory for a space probe so that it will pass by the outermost planets in the closest proximity.
  - Compute the layout of the internals of a CPU so that the total wiring distance is minimized.
  - Find the cheapest flight plan from A to B, for given intervals for arrival and departure dates.
  - Simulate a war between two land forces, given (i) the attack and the defense plans, (ii) the inventories and (iii) other attributes of both forces.

What is data?

- CPU can only understand two types of data:
  - Integers,
  - Floating points.
- The following are not directly understandable by a CPU:
  - Characters (‘a’, ‘A’, ‘2’, …)
  - Strings (“apple”, “banana”, …)
  - Complex Numbers
  - Matrices
  - Vectors
- But, programming languages can implement these data types.

Basic Data Types

- Integers
  - Full support from the CPU
  - Fast processing
- Floating Points
  - Support from the CPU with some precision loss
  - Slower compared to integer processing due to “interpretation”

Problems due to precision loss

- What has precision loss is misleading: 0.9 has precision loss whereas 0.9375 does not!
  - (0.9375 = 0.5 + 0.25 + 0.125 + 0.0625)
  - (½ + ¼ + 1/8 + 1/16) = 0.1111
  - Imprecise representations are round-offs!
  - If you have a lot of round-offs, you might be in trouble!

Problems due to precision loss

- 1.0023 – 1.0567
  - Result: -0.05440000000000004
- 1000.0023 – 1000.0567
  - Result: -0.05439999999986903
- Why?
  - Since the floating point representation is based on shifting the bits in the mantissa, the following are not equivalent in a PC
- π = 3.1415926535897931….
  - sin(π) should be zero
  - But it is not: $1.2246467991473532 \times 10^{-16}$

Problems due to precision loss

- Associativity in mathematics:
  - $(a+b) + c = a + (b + c)$
  - This does not hold in floating-point arithmetic:

set $a = 1234.567$, $b = 45.67834$ and $c = 0.0004$:
$(a + b) + c$ will produce 1280.2457399999998,
$a + (b + c)$ will produce 1280.2457400000001.

So what to do with precision loss, then?

- If you can transform the problem to the integer domain, do so.
  - Refrain from using floating points as much as you can.
- Use the most precise type of floating point of the high level language you are using.
  - Use only less precision floating points if you are short in memory.

So what to do with precision loss, then?

- If you have two numbers that are magnitude-wise incomparable, you are likely to lose the contribution of the smaller one. That will yield unexpected results when you repeat the addition in a computational loop where the looping is so much that the accumulation of the smaller is expected to become significant. It will not.
- The contrary happens too. Slight inaccuracies accumulate in loops to significant magnitudes and yield non-sense values.
- You better use well known, decent floating point libraries instead of coding floating point algorithms by yourself.

Integers in Python

- Python provides `int` type.

```
>>> type(3)
<type 'int'>
>>> type(3+4)
<type 'int'>
>>>
```

- For big integers, Python has the `long` type:

```
>>> type(3L)
<type 'long'>
>>> type(3L+4L)
<type 'long'>
>>>
```

`int` is limited by the hardware whereas `long` type is unlimited in Python.

Floating Points in Python

- Python provides `float` type.

```
>>> type(3.4)
<type 'float'>
>>>
```

```
>>> 3.4+4.3
7.7
>>> 3.4 / 4.3
0.7906976744186046
```

Simple Operations with Numerical Values in Python

Operator	Operator Type	Description
+	Binary	Addition of two operands
-	Binary	Subtraction of two operands
-	Unary	Negated value of the operand
*	Binary	Multiplication of two operands
/	Binary	Division of two operands
**	Binary	Exponentiation of two operands (Ex: x**y = x^y)

- **abs**(x): Absolute value of x.
- **round**(Float): Rounded value of float.
- **int**(Number), **long**(Number), **float**(Number): Conversion between numerical values of different types.

Characters

- We need characters to represent textual data:
  - Characters: ‘A’, …, ‘Z’, ‘0’, .., ‘9’, ‘a’, …, ‘z’ … etc.
- Computer uses the ASCII table for converting characters to binary numbers:

SYN	00010110	,	00101100	B	01000010	X	01011000	n	01101110
ETB	00010111	-	00101101	C	01000011	Y	01011001	o	01101111
CAN	00011000	.	00101110	D	01000100	Z	01011010	p	01110000
EM	00011001	/	00101111	E	01000101	(	01011011	q	01110001
SUB	00011010	0	00110000	F	01000110	\	01011100	r	01110010
ESC	00011011	1	00110001	G	01000111	)	01011101	s	01110011
FS	00011100	2	00110010	H	01001000	^	01011110	t	01110100
GS	00011101	3	00110011	I	01001001	_	01011111	u	01110101
RS	00011110	4	00110100	J	01001010	‘	01100000	v	01110110
US	00011111	5	00110101	K	01001011	a	01100001	w	01110111
SPC	00100000	6	00110110	L	01001100	b	01100010	x	01111000
!	00100001	7	00110111	M	01001101	c	01100011	y	01111001
"	00100010	8	00111000	N	01001110	d	01100100	z	01111010
#	00100011	9	00111001	O	01001111	e	01100101	}	01111011
$	00100100	:	00111010	P	01010000	f	01100110	|	01111100
%	00100101	;	00111011	Q	01010001	g	01100111	}	01111101
&	00100110	<	00111100	R	01010010	h	01101000	~	01111110
’	00100111	=	00111101	S	01010011	i	01101001	DEL	01111111
(	00101000	>	00111110	T	01010100	j	01101010		
)	00101001	?	00111111	U	01010101	k	01101011		
*	00101010	@	01000000	V	01010110	l	01101100		
+	00101011	A	01000001	W	01010111	m	01101101		

A portion of the ASCII table.

Characters in Python

- Python does not have a separate data type for characters!
- However, one character strings can be treated like characters:
  - **ord**(One_Char_String) : returns the ASCII value of the character in One_Char_String.
    - Ex: ord("A") returns 65.
  - **chr**(ASCII_value) : returns the character in a string that has the ASCII value ASCII_value:
    - Ex: chr(66) returns "B"

Boolean Values

- The CPU often needs to compare numbers, or data:
  - $3 >^{?} 4$
  - $125 =^{?} 1000/8$
  - $3 \leq^{?} 12345.34545/12324356.0$
- We have the truth values for representing the answers to such comparisons:
  - If correct: TRUE, True, true, T, 1
  - If not correct: FALSE, False, false, F, 0

Boolean Values in Python

- Python provides the `bool` data type for boolean values.
- `bool` data type can take `True` or `False` as values.

```
>>> 3 > 4
False
>>> type(3 > 4)
<type 'bool'>
```

```
>>> True == 2
False
>>> True == 1
True
>>> False == 0
True
>>> False == 1
False
```

Try this:

```
>>> True == bool(2)
```

not operator:

```
not True
not 4 > 3
```