Representing and Manipulating Information: Bits, Ints, and Ops

Yipeng Huang

Rutgers University

February 20, 2023

Table of contents

Programming assignment 2: Queues, trees, and graphs

Programming Assignment 2 parts

1. bstLevelOrder: needs a queue (available in pa2/queue, will discuss today)
2. edgelist: will discuss today
3. isTree: needs DFS (stack)
4. solveMaze: needs BFS (queue)
5. mst: a greedy algorithm
6. findCycle: needs either DFS (stack) or BFS (queue)
7. matChainMul: another dynamic programming problem and prelude to integer operations

Programming assignment 2 & reading assignment

No quiz this week

1. Focus on PA2.

Programming assignment 2

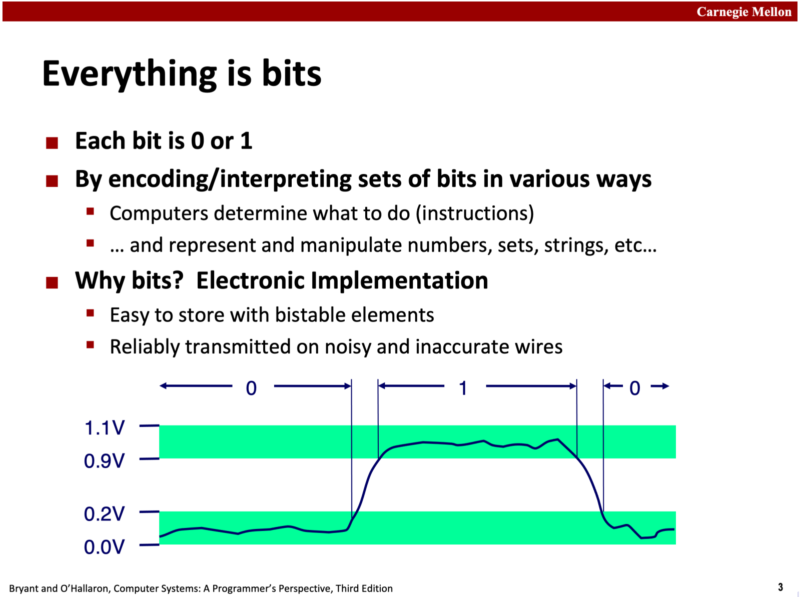
1. Due Friday 2/24.
2. More data structures: queues, BSTs, graphs; solidify managing memory.

Reading assignment: CS:APP Chapters 2.4

1. Preparation for next week
2. All about floating point numbers: A case studying in the design of an engineering standard.

Table of contents

Why binary



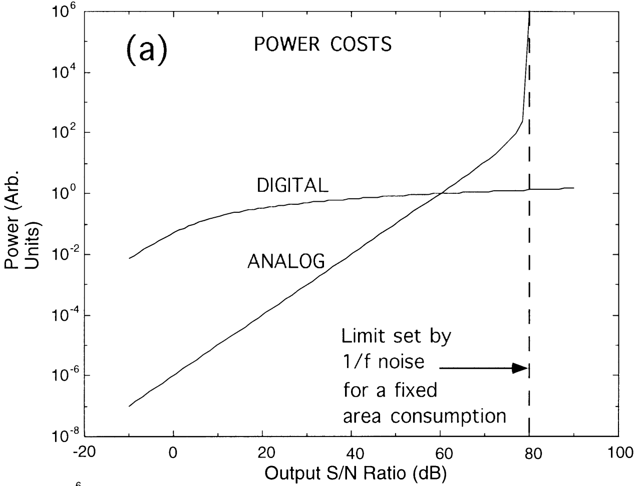
Everything is bits

- **Each bit is 0 or 1**
- **By encoding/interpreting sets of bits in various ways**
  - Computers determine what to do (instructions)
  - … and represent and manipulate numbers, sets, strings, etc…
- **Why bits? Electronic Implementation**
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Why binary

Figure: Rahul Sarpeshkar. Analog Versus Digital: Extrapolating from Electronics to Neurobiology. 1998.



Why binary

Digital encodings

Each doubling of either precision or range only needs one additional bit.

Analog encodings

Each doubling of either precision or range needs doubling of either area or power.

Decimal, binary, octal, and hexadecimal

Decimal	Binary	Octal	Hexadecimal
0	0b0000	0o0	0x0
1	0b0001	0o1	0x1
2	0b0010	0o2	0x2
3	0b0011	0o3	0x3
4	0b0100	0o4	0x4
5	0b0101	0o5	0x5
6	0b0110	0o6	0x6
7	0b0111	0o7	0x7

Decimal	Binary	Octal	Hexadecimal
8	0b1000	0o10	0x8
9	0b1001	0o11	0x9
10	0b1010	0o12	0xA
11	0b1011	0o13	0xB
12	0b1100	0o14	0xC
13	0b1101	0o15	0xD
14	0b1110	0o16	0xE
15	0b1111	0o17	0xF

In C, format specifiers for printf() and fscanf():

1. decimal: ’%d’
2. binary: none
3. octal: ’%o’
4. hexadecimal: ’%x’

Decimal, binary, octal, and hexadecimal

How to represent the range of unsigned char in each?

Unsigned char is one byte, 8 bits.

1. decimal: 0 to 255
2. binary: 0b0 to 0b11111111
3. octal: 0 to 0o377 (group by 3 bits)
4. hexadecimal: 0x00 to 0xFF (group by 4 bits)

Often encountered use of hexadecimal: RGB colors

Red, green, blue values ranging from 0-255

⬛	⬜	▩	???
#000000	#FFFFFF	#6A757C	#CC0033

Often encountered use of hexadecimal: RGB colors

Red, green, blue values ranging from 0-255

⬛	⬜	▨	🟥
#000000	#FFFFFF	#6A757C	#CC0033

Don’t confuse the bitstring vs. the interpreted value

The bitstring

11111111, 377, 255, FF

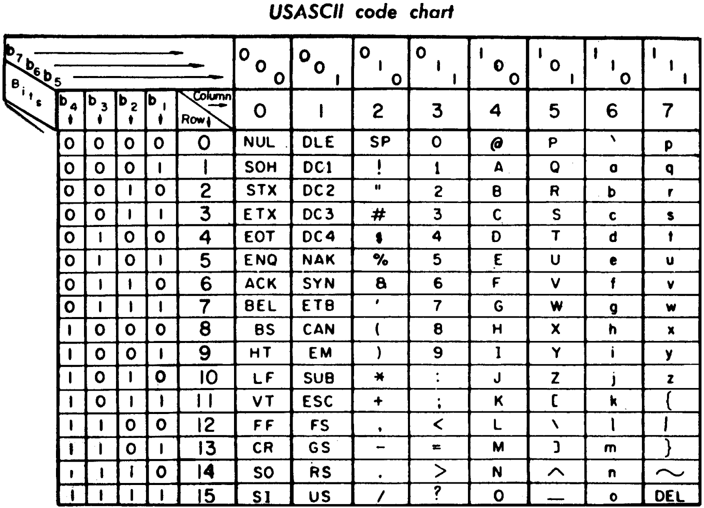
Interpretation of the value

To interpret the value of a bitstring, you need to know:

1. the radix, number base: 2, 8, 10, 16.
2. the representation of signed values: two’s complement.
3. size of the data type: char, short, int, long
4. decimal point

Representing characters

- char is a 1-byte, 8-bit data type.
- ASCII is a 7-bit encoding standard.
- "man ascii" to see Linux manual.
- Compile and run ascii.c to see it in action.
- Some interesting characters: 7 (bell), 10 (new line), 27 (escape).



USASCII code chart

Bits				$b_7 b_6 b_5$ →	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4	b_3	b_2	b_1	Column → Row ↓	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

Figure: ASCII character set. Image credit Wikimedia

Bitwise operations

Why are bitwise operations important?

- Network and UNIX settings using bit masks (e.g., umask)
- Hardware and microcontroller programming (e.g., Arduinos)
- Instruction set architecture encodings (e.g., ARM, x86)

Bitwise operations

~: bitwise NOT

unsigned char a = 128

$$\begin{aligned} a &= 0b1000\_0000 \\ \tilde{}a &= \tilde{}0b1000\_0000 \\ &= 0b0111\_1111 \\ &= 127 \end{aligned}$$

b	~b
0	1
1	0

Bitwise operations

&: bitwise AND

$$\begin{aligned} 3 \& 1 &= 0b11 \& 0b01 \\ &= 0b01 \\ &= 1 \end{aligned}$$

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

Bitwise operations

| : bitwise OR

$$\begin{aligned} 3|1 &= 0b11|0b01 \\ &= 0b11 \\ &= 3 \end{aligned}$$

$$\begin{aligned} 2|1 &= 0b10|0b01 \\ &= 0b11 \\ &= 3 \end{aligned}$$

a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise operations

^: bitwise XOR

$$\begin{aligned} 3 \wedge 1 &= 0b11 \wedge 0b01 \\ &= 0b10 \\ &= 2 \end{aligned}$$

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

`inplaceSwap.c`: Swapping variables without temp variables.

How does it work?

Don't confuse bitwise operators with logical operators

Bitwise operators

- ~
- &
- |
- ^

Logical operators

- !
- &&
- ||
- != (for bool type)

Table of contents

Representing negative and signed integers

Ways to represent negative numbers

1. Sign magnitude
2. 1s' complement
3. 2's complement

Representing negative and signed integers

Sign magnitude

Flip leading bit.

Representing negative and signed integers

1s′ complement

- Flip all bits
- Addition in 1s′ complement is sound
- In this encoding there are 2 encodings for 0
- -0: 0b1111
- +0: 0b0000

Representing negative and signed integers

2's complement

signed char	weight in decimal
00000001	1
00000010	2
00000100	4
00001000	8
00010000	16
00100000	32
01000000	64
10000000	-128

Table: Weight of each bit in a signed char type

- what is the most positive value you can represent? 127
- what is the most negative value you can represent? -128
- how to represent -1? 11111111
- how to represent -2? 11111110

Representing negative and signed integers

2’s complement

signed char	weight in decimal
00000001	1
00000010	2
00000100	4
00001000	8
00010000	16
00100000	32
01000000	64
10000000	-128

Table: Weight of each bit in a signed char type

- MSB: 1 for negative
- To make a number negative: flip all bits and add 1.
- Addition in 2’s complement is sound

Importance of paying attention to limits of encoding

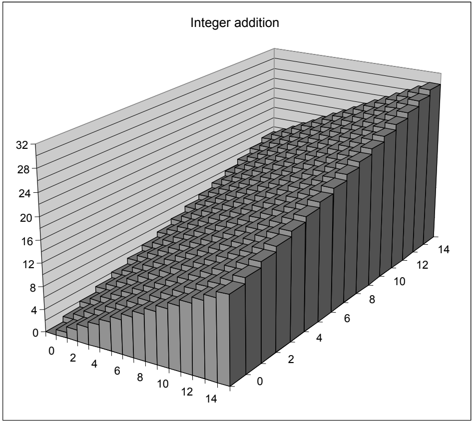


Figure: Image credit: CS:APP

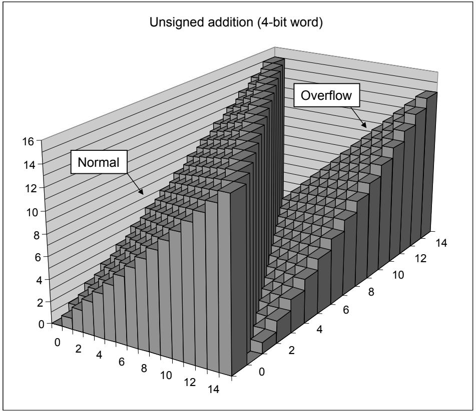


Figure: Image credit: CS:APP

Importance of paying attention to limits of encoding

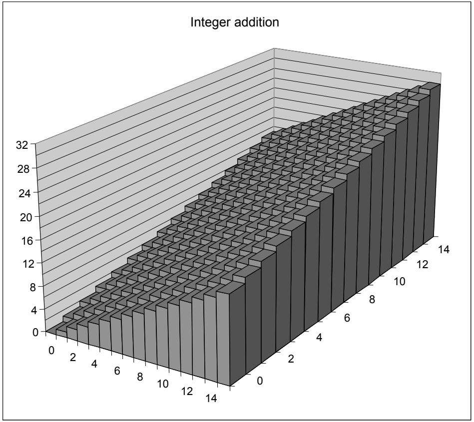


Figure: Image credit: CS:APP

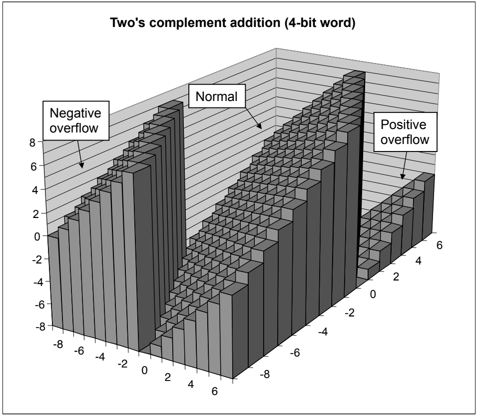


Figure: Image credit: CS:APP

`https://www.theatlantic.com/technology/archive/2014/12/how-gangnam-style-broke-youtube/383389/`

Table of contents

`matChainMul.c`: Minimum number of multiples needed for matrix chain multiplication

Learning objectives

- Review and master recursion.
- Array subsetting using pointer arithmetic.
- Using pass-by-reference to return computed results.
- A new algorithm that most classmates have not seen before.

Cost of multiplying matrices: the number of multiplies

- $A_{l \times m} \times B_{m \times n}$
- Needs $l \times m \times n$ number of multiplies
- (Well-kept secret: fewer multiplications possible, see Strassen's algorithm)

`matChainMul.c`: Minimum number of multiplies needed for matrix chain multiplication

$$A \times B \times C = \begin{bmatrix} a_{0,0} & a_{0,1} \end{bmatrix}_{1\times 2} \times \begin{bmatrix} b_{0,0} \\ b_{1,0} \end{bmatrix}_{2\times 1} \times \begin{bmatrix} c_{0,0} & c_{0,1} \end{bmatrix}_{1\times 2}$$

Parenthesization 1: 4+4 = 8 multiplies

$$\begin{aligned} A \times (B \times C) &= \begin{bmatrix} a_{0,0} & a_{0,1} \end{bmatrix}_{1\times 2} \times \begin{bmatrix} b_{0,0}c_{0,0} & b_{0,0}c_{0,1} \\ b_{1,0}c_{0,0} & b_{1,0}c_{0,1} \end{bmatrix}_{2\times 2} \\ &= \begin{bmatrix} \left(a_{0,0}b_{0,0}c_{0,0} + a_{0,1}b_{1,0}c_{0,0}\right) & \left(a_{0,0}b_{0,0}c_{0,1} + a_{0,1}b_{1,0}c_{0,1}\right) \end{bmatrix}_{1\times 2} \end{aligned}$$

Parenthesization 2: 2+2 = 4 multiplies

$$\begin{aligned} (A \times B) \times C &= \left(a_{0,0}b_{0,0} + a_{0,1}b_{1,0}\right) \times \begin{bmatrix} c_{0,0} & c_{0,1} \end{bmatrix}_{1\times 2} \\ &= \begin{bmatrix} \left(a_{0,0}b_{0,0} + a_{0,1}b_{1,0}\right)c_{0,0} & \left(a_{0,0}b_{0,0} + a_{0,1}b_{1,0}\right)c_{0,1} \end{bmatrix}_{1\times 2} \end{aligned}$$

`matChainMul.c`: Minimum number of multiplies needed for matrix chain multiplication

$A \times B \times C \times D$

First partitioning

- $A(BCD)$; but what is cost of finding (BCD)? Needs decomposition.
- $(AB)(CD)$
- $(ABC)D$; but what is cost of finding (ABC)? Needs decomposition.

Second partitioning

- $A(B(CD))$
- $A((BC)D)$
- $(AB)(CD)$
- $(A(BC))D$
- $((AB)C)D$

`toBin.c`: Printing the binary representation

- Shifting and masking
- Try modifying to print octal.

Bit shifting

$<< N$ Left shift by N bits

- multiplies by 2^N
- $2 << 3 = 0000\_0010_2 << 3 = 0001\_0000_2 = 16 = 2 * 2^3$
- $-2 << 3 = 1111\_1110_2 << 3 = 1111\_0000_2 = -16 = -2 * 2^3$

$>> N$ Right shift by N bits

- divides by 2^N
- $16 >> 3 = 0001\_0000_2 >> 3 = 0000\_0010_2 = 2 = 16/2^3$
- $-16 >> 3 = 1111\_0000_2 >> 3 = 1111\_1110_2 = -2 = -16/2^3$