More **bits** of CS

Too many bits? Compress!

Below binary: **physical circuits**

---

Circuit design, part 1

I'd call this a KNOT gate...

This circuit was NOT, in fact, designed!

---

**Hw #4 due Mon. 2/24**

- **pr0 (reading)**  A bug and a crash!
- **pr1 (lab)**     binary ~ decimal
- **pr2**          conversion + compression
- **extra**        image processing...

---

**Lots of tutoring hrs - join in... !**
Next Thursday will be the first CS 5 in-class midterm

Un-warnings:

accommodations? Talk to me!
worries? concerns? Talk to me!
five problems, written
focus on semantics, not syntax

Suggestions:

go over in-class exercises and hwk problems
create a hand-written, single sided page of notes
consider small variations of old problems – and how they would change the solutions...

only 5 minutes? Try list comprehensions & LoL!
This Thursday and Next: no office hours

This week: Prof. Medero has a conflict

Available in my office (Olin 1269) tomorrow, 9-11am

Next week: Better to be available before the midterm!

Move hours to Wednesday 2/26, 1-3pm
Bits' big idea

left-shifting by 1 doubles a value

<table>
<thead>
<tr>
<th>Concept</th>
<th>Python</th>
<th>Bitwise reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>left-shift</td>
<td>$42 \ll 1$</td>
<td>'101010'</td>
</tr>
<tr>
<td>84</td>
<td></td>
<td>'1010100'</td>
</tr>
</tbody>
</table>

right-shifting by 1 halves a value

<table>
<thead>
<tr>
<th>Concept</th>
<th>Python</th>
<th>Bitwise reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>right-shift</td>
<td>$42 \gg 1$</td>
<td>'101010'</td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>'10101'</td>
</tr>
</tbody>
</table>
Bits' big idea

**Concept**

Left-shifting by 1 **doubles** a value

Right-shifting by 1 **halves** a value

**Python**

```
42 << 1
84
```

```
42 >> 1
21
```

**Bitwise reason**

Aha! This can be implemented just with wiring!

Purely mechanical

Do I halve to remember this?

No – just that columns are powers of two...
Adding strings?

```python
def add10(S, T):
    """ adds the *strings* S and T as decimal numbers """
```

S = '31'
T = '11'

s = '31'
t = '11'

is syntactic addition!

is circuit addition!
Adding strings?

```python
def add10(S,T):
    """ adds the *strings* S and T as decimal numbers ""
    if len(S) == 0: return T
    if len(T) == 0: return S
    eS = S[-1]  # eS ~ the "end of S"
    eT = T[-1]  # eT ~ the "end of T"
    if eS == '0' and eT == '1': return add10(S[:-1],T[:-1]) + '1'
    if eS == '1' and eT == '1': return add10(S[:-1],T[:-1]) + '2'
    if eS == '2' and eT == '1': return add10(S[:-1],T[:-1]) + '3'
    if eS == '3' and eT == '1': return add10(S[:-1],T[:-1]) + '4'
    # Lots more rules - how many in all?
```
Adding strings?

```
def add10(S,T):
    # adds the *strings* S and T as decimal numbers
    if len(S) == 0: return T
    if len(T) == 0: return S
    eS = S[-1]  # eS ~ the "end of S"
    eT = T[-1]  # eT ~ the "end of T"
    if eS == '0' and eT == '1': return add10(S[:-1],T[:-1]) + '1'
    if eS == '1' and eT == '1': return add10(S[:-1],T[:-1]) + '2'
    if eS == '2' and eT == '1': return add10(S[:-1],T[:-1]) + '3'
    if eS == '3' and eT == '1': return add10(S[:-1],T[:-1]) + '4'
```

Note that this function doesn't "understand" addition at all...!
Carrying on...

```
def add10(S,T):
    """ adds the *strings* S and T as decimal numbers
    """
    if len(S) == 0: return T
    if len(T) == 0: return S
    eS = S[-1]  # eS ~ the "end of S"
    eT = T[-1]  # eT ~ the "end of T"
    if eS == '0' and eT == '1': return add10(S[:-1],T[:-1]) + '1'
    if eS == '1' and eT == '1': return add10(S[:-1],T[:-1]) + '2'
    if eS == '2' and eT == '1': return add10(S[:-1],T[:-1]) + '3'
    if eS == '3' and eT == '1': return add10(S[:-1],T[:-1]) + '4'
    # what if we have to carry to the next column?
    if eS == '3' and eT == '9':
        return add10(add10(S[:-1],T[:-1]), '1') + '2'
```

hw4: addB
**All computation** is simply *functions of bits*

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Output, A+B</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>000</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>001</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>010</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>011</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>001</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>010</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>011</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>100</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>010</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>011</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>101</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>011</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>100</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>101</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>110</td>
</tr>
</tbody>
</table>

**This week:** you'll build this in Python.

Next week: you'll design this with wires.
Adding strings?

Multiplying by machine:

Doing anything by machine...

is circuit addition!

is syntactic addition!

is circuit multiplying!

is syntactic multiplying!

syntactic ~ meaning-free

means it can be done purely via surface syntax, which means it can be done without thinking...
How high can we count...?

<table>
<thead>
<tr>
<th>Number of Bits</th>
<th>Binary Representation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2 bits</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>3 bits</td>
<td>111</td>
<td>7</td>
</tr>
<tr>
<td>4 bits</td>
<td>1111</td>
<td>15</td>
</tr>
<tr>
<td>7 bits</td>
<td>1111111</td>
<td>127</td>
</tr>
<tr>
<td>8 bits</td>
<td>11111111</td>
<td>255</td>
</tr>
<tr>
<td>N bits</td>
<td>2^n - 1</td>
<td></td>
</tr>
<tr>
<td>31 bits</td>
<td>2^31 - 1</td>
<td></td>
</tr>
</tbody>
</table>
How high can we count... in 2015?

List of most viewed YouTube videos

Top videos

<table>
<thead>
<tr>
<th>Rank</th>
<th>Video name</th>
<th>Uploader / artist</th>
<th>Views (as of September 29, 2015)</th>
<th>Upload date</th>
<th>Notes</th>
</tr>
</thead>
</table>

51 bits
How high can we count... in 2015!

List of most viewed YouTube videos

From Wikipedia, the free encyclopedia

This list of most viewed YouTube videos consists of the 30 most viewed videos of all time as derived from YouTube charts. Videos that YouTube suspects have had their view counts manipulated are not included in this list. View counts are based on the YouTube website; many of the videos are music videos that play through YouTube's partner site, Vevo, and YouTube view counts will lag those of Vevo by a few days.[1]

As of September 2015, nine music videos have received over 1 billion views, with the top video, "Gangnam Style", exceeding 2 billion views.

Top videos

<table>
<thead>
<tr>
<th>Rank</th>
<th>Video name[A]</th>
<th>Uploader / artist</th>
<th>Views (as of September 29, 2015)</th>
<th>Upload date</th>
<th>Notes</th>
</tr>
</thead>
</table>
Other overflow errors...

Less worrisome, perhaps...

The "sign bit" has flipped to one. Thus, the number has become negative...!
Hw4: images are just bits, too!

old pixel at 42,42 has

- red = 1 (out of 255)
- green = 36 (out of 255)
- blue = 117 (out of 255)

new pixel at 42,42 has

- red = 254 (out of 255)
- green = 219 (out of 255)
- blue = 138 (out of 255)

how many bits represent each color channel?
Hw4: *images are just bits, too!*

---

Binary Image

Encoding as raw bits

```
10101010
01010101
10101010
01010101
10101010
01010101
10101010
01010101
```

especially *binary images*

"1010101001010101101010100101010110101010010101011010101001010101"
likely *compressible* image...
Too many pixels... too little time + space!

image compression is everywhere!
Too many pixels... too little time + space!

image compression is everywhere!

How is it possible to throw away 98% of the image data?!
One solution!

We throw away 98% of the image area!

How is it possible to throw away 98% of the image data!? 

Looks like the right 2% to keep!
More often... what's done?

*compressed* to 40kb

*original*: 2.3mb
compressed to 40kb

original: 2.3mb
Hw4: *lossless* binary image compression

Binary Image

```
00000000
00000000
11111111
11111111
00000000
00000000
00000000
00001111
```

Encoding as raw bits

```
00000000
00000000
11111111
11111111
00000000
00000000
00000000
00001111
```
likely compressible image...
Hw4: **lossless** binary image compression

If our images tend to have **long streaks of unchanging data**, how might we represent it more efficiently, **but still in binary**?

```
"00000000000000001111111111111111000000000000000000001111"
```
Hw4: lossless image compression

One possible algorithm:

bit #repeats

Any problems with this?
Hw4: lossless image compression

0 is the first digit
and there are 1,098,188 of them.

It's ambiguous! this could just be a huge number of 0 pixels!

our algorithm:

bit #repeats

could be misinterpreted!
fixed-width compression

We need fixed-width blocks:

```
bit  #repeats
```

1 bit fill 7 bits for the # of repeats

8-bits total
If you use 7 bits to hold the # of consecutive repeats, what is the largest number of bits that one block can represent?

8-bit total data block

1 bit: the initial pixel

7 bits: # of repeats

What if you need a larger # of repeats?
def compress(I):
    """ returns the RLE of the input binary image, I """

a binary image, I

"000000000000000000000000000000000000000011111111111111111111111111111111"

42 zeros

31 ones

"001010101010011111"

42, in binary

31, in binary

the "compressed" output returned by compress(I)
def compress( I ):
    """ returns the RLE of the input binary image, I """

a 64-bit binary image, I

"0000000000001111111111111111111100000000000000000000011111111111"

12 zeros  20 ones  21 zeros  11 ones

compress( I )

"000001100000101000001101111111"

the "compressed" output returned by compress(I)

Then, discuss ...

What helper function would be useful for compress?

What's an image I whose compressed output gets larger, not smaller? (Aargh!)

• What are the BEST-compressible / WORST-compressible 64-bit images?

• How could you improve the algorithm so that it always compresses?!!
def compress( I ):
    """ returns the RLE of the input binary image, I """

    a binary image, I

"""000000000000111111111111111111110000000000000000000011111111111""

12 zeros  20 ones  21 zeros  11 ones

"""0000110010010100000010101100010111"

12  20  21  11

the "compressed" image returned from compress( I )
**Use this!**

frontNum(S) returns the # of times the first element of the input S appears consecutively **at the start** of S:

- frontNum('111010') = 4
- frontNum('00110010') = 2

```python
def frontNum(S):
    if len(S) <= 1:
        return
    elif len(S) == 0:
        return
    else:
        return
```

- If S == '' or S == '1' or S == '0'
- If the first two bits **DO** match....
- If the first two bits **DON'T** match....

**BEST / WORST images?**
What are the **BEST** and the **WORST** compression results you can get for an 8x8 image input (64 bits)?

How could we improve this compression algorithm so that *all images* compress to smaller than the originals? That is, how can we make compression always **work**?
What are the **BEST** and the **WORST** compression results you can get for an 8x8 image input (64 bits)?

How could we improve this compression algorithm so that all images compress to smaller than the originals? That is, how can we make compression always work?
What are the BEST and the WORST compression results you can get for an 8x8 image input (64 bits)?

**BEST**
only 8 bits total!

**WORST**
aargh! 512 bits!

Impossible! *Provably!*

How could we improve this compression algorithm so that *all images* compress to smaller than the originals? That is, how can we make compression always work?
It's all bits!

All computation boils down to manipulating bits!

even the string 'forty*two' is represented as a sequence of bits...

'forty*two'

0110011001101111011100100111010001111001
00101010
011101000111011101101111

9 ASCII characters
8 bits each
9*8 == 72 bits total
In a computer, each bit is represented as a **voltage** (1 is +5v and 0 is 0v)

Computation is simply the **deliberate combination** of those voltages!

42

101010

(1) set input voltages

But what's this green thing?

ADDER circuit

9

001001

51
In a computer, each bit is represented as a **voltage** *(1  is  +5\text{v} \text{ and } 0 \text{ is  } 0\text{v})*

Computation is simply the **deliberate combination** of those voltages!

\begin{itemize}
  \item \textbf{42} \hspace{1cm} 101010 \hspace{1cm} (1) \textit{set input voltages}
  \item \textbf{9} \hspace{1cm} 001001
  \item \textbf{But what's this green thing?}
\end{itemize}

\textbf{ADDER circuit}

\begin{itemize}
  \item (2) \textit{perform computation}
  \item \textbf{110011}
\end{itemize}
In a computer, each bit is represented as a **voltage** (1 is +5v and 0 is 0v)

Computation is simply the **deliberate combination** of those voltages!

Richard Feynman: "Computation is just a physics experiment that always works!"

**ADDER circuit**

1. Set input voltages
2. Perform computation
3. Read output voltages

But what's this green thing?
Our building blocks: *logic gates*

AND outputs 1 only if *ALL* inputs are 1

OR outputs 1 if *ANY* input is 1

NOT reverses its input

These circuits are *physical* functions of bits...

... and *all* mathematical functions can be built from them!
Our building blocks: logic gates

**AND** outputs 1 only if **ALL** inputs are 1

**OR** outputs 1 if **ANY** input is 1

**NOT** reverses its input

**AND** | **OR** | **NOT**
---|---|---
[AND symbol] | [OR symbol] | [NOT symbol]

**ALL** circuits are combinations of bits...

...and *all* mathematical functions can be built from them!
From gates to *circuits*...

What inputs make this circuit output 1?

What inputs make this circuit output 0?

Each AND performs a separate task... what are those tasks?
From gates to circuits...

Designing our own circuits...

What circuit outputs 1 for these four inputs?

... and outputs 0 for these four inputs?!
from circuit design...

next 2 weeks

...to a full computer!