More **bits** of CS

*Too many bits? Compress!*  
Below binary: **physical circuits**

---

**Hw #5 due Mon. 4/25**

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

---

**Jotto Corner**

<table>
<thead>
<tr>
<th>AM guess</th>
<th>my guess</th>
<th>HS guess</th>
<th>my guess</th>
</tr>
</thead>
<tbody>
<tr>
<td>party: 3</td>
<td>diner: 1</td>
<td>alien: 2</td>
<td>diner: 2</td>
</tr>
<tr>
<td>pears: 3</td>
<td>savvy: 1</td>
<td>ghost: 1</td>
<td>savvy: 1</td>
</tr>
<tr>
<td>??????: .</td>
<td>flock:</td>
<td>??????: .</td>
<td>flock:</td>
</tr>
</tbody>
</table>

---

**P/F vs T/F**

---

**Circuit design, part 1**

I'd call this a **KNOT** gate...

This circuit was NOT, in fact, designed!
def numToBin( N ):
    """ converts a decimal int to a binary string """

    if N==0:  return ''
    elif N%2==0:  return numToBin( N//2 ) + '0'
    elif N%2==1:  return numToBin( N//2 ) + '1'

    return ''

in 42

ntb( 42 )

ntb( 21 ) + '0'

ntb( 10 ) + '1'

ntb( 5 ) + '0'

ntb( 2 ) + '1'

ntb( 1 ) + '0'

ntb( 0 ) + '1'

out '101010'
def numToBin( N ):
    
    """ converts a decimal int to a binary string """

    if N==0: return ''
    else: return numToBin( N//2 ) + str(N%2)

What if you wanted base-3 output?!

base-B output?
def binToNum( S ):
    """ converts a binary string to a decimal int """
    if S=='':
        return 0
    elif S[-1]=='0':
        return 2*binToNum(S[:-1]) + 0
    elif S[-1]=='1':
        return 2*binToNum(S[:-1]) + 1
Lab Debriefing & hw5pr2.py

```python
def binToNum(S):
    ''' converts a binary string to a decimal int
    '''
    if S=='': return 0
    else: return 2*binToNum(S[:-1]) + int(S[-1])
```

What if you wanted base-3 input?!  
**base-B input?** saves the need for another if
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

42
84

42
21

in binary, columns double in value leftward

in binary, columns halve in value rightward

42
84

42
21
Bits' big idea

left-shifting by 1 doubles a value

right-shifting by 1 halves a value

Do I halve to remember this?

No – just that columns are powers of two...

Aha! This can be implemented just with wiring!

purely mechanical

Take-home

Concept

Python

Bitwise reason

42 << 1

84

'101010'

in binary, columns double in value leftward

'1010100'

42

'1010101'

21

'101011'

in binary, columns halve in value rightward
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?
```

S

'|31'

T

'|11'

is syntactic addition!

is circuit addition!
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]
    eT = T[-1]
    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 increment(add10(S, T))

s = '23'
t = '19'

hw5: addB

S T
'23' '19'
All computation is simply functions of bits

Binary inputs A and B

Output, A+B

00 00 000
00 01 001
00 10 010
00 11 011
01 00 001
01 01 010
01 10 011
01 11 100
10 00 100
10 01 101
10 10 011
10 11 110
11 00 101
11 01 100
11 10 101
11 11 110

This week: you'll build this in Python.

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

is circuit addition!

is syntactic addition!

syntactic ~ meaning-free

Multiplying by machine:

is circuit multiplying!

is syntactic multiplying!

Doing anything by machine...

is circuit interaction!

is syntactic interaction!

means it can be done purely via surface syntax, which means it can be done without thinking...
Ariane 5

This week's reading: *bits can be vital*

IndexError  TypeError  HumanError

16 bits  64 bits

version 4  version 5
How high can we count...?

<table>
<thead>
<tr>
<th>with</th>
<th>1 bit</th>
<th>1 bit</th>
<th>1 bit</th>
<th>1 bit</th>
<th>1 bit</th>
<th>1 bit</th>
<th>1 bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 bits</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>3 bits</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
</tr>
<tr>
<td>4 bits</td>
<td>1111</td>
<td>1111</td>
<td>1111</td>
<td>1111</td>
<td>1111</td>
<td>1111</td>
<td>1111</td>
</tr>
<tr>
<td>7 bits</td>
<td>1111111</td>
<td>1111111</td>
<td>1111111</td>
<td>1111111</td>
<td>1111111</td>
<td>1111111</td>
<td>1111111</td>
</tr>
<tr>
<td>8 bits</td>
<td>11111111</td>
<td>11111111</td>
<td>11111111</td>
<td>11111111</td>
<td>11111111</td>
<td>11111111</td>
<td>11111111</td>
</tr>
<tr>
<td>N bits</td>
<td>(2^{N-1})</td>
<td>(2^{N-1})</td>
<td>(2^{N-1})</td>
<td>(2^{N-1})</td>
<td>(2^{N-1})</td>
<td>(2^{N-1})</td>
<td>(2^{N-1})</td>
</tr>
<tr>
<td>31 bits</td>
<td>(2^{31}-1)</td>
<td>(2^{31}-1)</td>
<td>(2^{31}-1)</td>
<td>(2^{31}-1)</td>
<td>(2^{31}-1)</td>
<td>(2^{31}-1)</td>
<td>(2^{31}-1)</td>
</tr>
</tbody>
</table>

I can see some patterns here – even with one eye closed!
How high can we count... in 2015?

<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>
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.

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>

only briefly, of course...
Other overflow errors...

Less worrisome, perhaps...

The "sign bit" has flipped to one. Thus, the number has become negative...!
Counting sheep, xkcd style...

How many bits?
Hw5: *images are just bits, too!*  

**old** pixel at 42,42 has

- red = 1 (out of 255) = $2^8 - 1$
- green = 36 (out of 255)
- blue = 117 (out of 255)

**new** pixel at 42,42 has

(guesses as to what this transformation was?)

how many **bits** represent each color channel?
Hw5: *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?
Hw5: *images are just bits, too!*

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

especially *binary images*
likely compressible image... home!
Too many pixels... too little time + space!

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

Too many pixels... too little time + space!

Image compression is everywhere!
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
Hw5: lossless binary image compression

Binary Image

Encoding as raw bits
one big string of 64 characters

00000000
00000000
11111111
11111111
00000000
00000000
00000000
00001111

same-data streaks
likely **compressible** image...

Lots of similar pixels!
Hw5: **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**?

"000000000000000011111111111111110000000000000000000001111"
Hw5: lossless image compression

One possible algorithm:

```
bit #repeats
```

Any problems with this?
Hw5: 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

0 is the first digit
There are 16 of them.

1 is the next digit
Again, there are 16 of them.

7 bits: # of repeats

28 zeros

4 ones

and so on...

8-bit data block
8-bit data block
8-bit data block
8-bit data block

00010000100100000001110010000100

0000000000000000

1111111111111111

0000000000000000

0000000000000000

0000000000001111

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

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

a binary image, I

"000000000000000000000000000000000000000001111111111111111111111111111111"

42 zeros

31 ones

"010101010100111111"

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

"000000000000000011111111111111111111111111111111111111111111111111111100000000000011111111111"

12 zeros 20 ones 21 zeros 11 ones

compress( I )

the "compressed" output returned by compress(I)

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

    "00000000000011111111111111111111000000000000000000011111111111"
    12 zeros
    20 ones
    21 zeros
    11 ones

    "000011001001010000001010110001011"
    12 zeros
    20 ones
    21 zeros
    11 ones

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

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

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

1 base case:
len(S) == 0:
return

elif:
return

else:
return

or 2 base cases:
len(S) == 1:
return

If S == '' or S == '1'
or S == '0'

If the first two bits DO match....

If the first two bits DON'T match....
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!

Anyone see why this is NOT QUITE the worst-compressible image?

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!

images, text, sounds, data, ...

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

'forty*two'

011001101101110111001001111010001111001001010100111011101101111

9 ASCII characters
8 bits each
9*8 == 72 bits total

All computation boils down to manipulating bits!
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!

But what's this green thing?

(1) set input voltages

ADDER circuit

101010

42

9

001001
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!

But what's this green thing?

(1) set input voltages

(2) perform computation
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!
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

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?

001
000
001
010
011
100
101
110
111

What inputs make this circuit output 0?

Each AND performs a separate task... what are those tasks?

Logisim Evolution
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!