More **bits** of CS

Too many **bits**?  **Compress!**

Below binary: **physical circuits**

---

**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><strong>party</strong>: 3</td>
<td>diner: 1</td>
<td><strong>alien</strong>: 2</td>
<td>diner: 2</td>
</tr>
<tr>
<td><strong>pears</strong>: 3</td>
<td>savvy: 1</td>
<td><strong>ghost</strong>: 1</td>
<td>savvy: 1</td>
</tr>
<tr>
<td>??????: .</td>
<td>flock: ?</td>
<td>????: .</td>
<td>flock: ?</td>
</tr>
</tbody>
</table>

---

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

**pr0 (reading)**  A bug and a crash!

**pr1 (lab)**  binary ~ decimal

**pr2**  conversion + compression

**extra**  image processing...

---

**Lots of tutoring hrs - join in... !**
Lab Debriefing & hw5pr2.py

```python
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'
```

These are awfully similar...
def numToBin( N ):
    """ converts a decimal int to a binary string """
    if N==0:
        return ''
    else:
        return numToBin( N//2 ) + str(N%2)
Lab Debriefing & hw5pr2.py

```python
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
```

Again, awfully similar...
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 *

Python

42 << 1
84

Bitwise reason

in binary, columns double in value leftward

Aha! This can be implemented

just with wiring!

purely mechanical

AI!

right-shifting by 1

* halves a value *

Take-home

42 >> 1
21

42
84

'101010'

'in binary, columns halve in value rightward'

'1010100'

'1010101'

'101010'

'10101'

42

84

21
Bits' big idea

**Concept**

- **left-shifting by 1**
  - **doubles** a value
  - Python: `42 << 1`  
    - 84

- **right-shifting by 1**
  - **halves** a value
  - Python: `42 >> 1`  
    - 21

**Python**

- `42 << 1`  
  - `84`

- `42 >> 1`  
  - `21`

**Bitwise reason**

- '101010' to '1010100'
- '101010' to '10101'

**Do I halve to remember this?**

**Take-home**

- Aha! This can be implemented just with wiring!

**purely mechanical**

- 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'
```

is syntactic addition!

is circuit addition!
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'
    # Lots more rules - how many in all?
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]
    eT = T[-1]
    eS ~ the "end of S"  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...!

is syntactic addition!

is circuit addition!
Carrying on...

```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'
    # what if we have to carry to the next column?
    if eS == '3' and eT == '9':
        return add10('1', add10('...'))
```

```
hw5: addB
S '23'
T '19'
```

```
\[
\begin{array}{c}
S \\ + T \\
\hline
\text{\textcolor{red}{1}} \\
\hline
\text{\textcolor{red}{2}}
\end{array}
\]
```
All computation is simply functions of bits.

Binary inputs A and B:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>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? is circuit addition!

Multiplying by machine: is circuit multiplying!

Doing anything by machine... is circuit interaction!

syntactic ~ meaning-free

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>Number of Bits</th>
<th>Binary Number</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>2.1 billion</td>
</tr>
</tbody>
</table>

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

List of most viewed YouTube videos

From Wikipedia, the free encyclopedia

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>
<tbody>
<tr>
<td>1.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</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...

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)
- **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!*

Binary Image

Encoding as raw bits

one big string of 64 characters

"1010101001010101101010100101010110101010010101011010101001010101"
likely compressible image...
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!

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

We throw away 98% of the image area!

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

**compressed** to **40kb**

**original:** **2.3mb**
Hw5: *lossless* binary image compression

Binary Image

Encoding as raw bits

```plaintext
00000000
00000000
11111111
11111111
00000000
00000000
00000000
00011111
```

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

"0000000000000000111111111111111100000000000000000000000000001111"
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.

0100001100000111001100

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

our algorithm:
bite #repeats

could be misinterpreted!
We need **fixed-width** blocks:

- **bit**
- **#repeats**

<table>
<thead>
<tr>
<th>bit</th>
<th>#repeats</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit fill</td>
<td>7 bits for the # of repeats</td>
</tr>
</tbody>
</table>

---

**fixed-width** compression
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

"000000000000000000000000000000000000000011111111111111111111111111111111"

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

"0000000000001111111111111111111100000000000011111111111"

12 zeros 20 ones 21 zeros 11 ones

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

"0000000000001111111111111111111100000000000000000000011111111111"

12 zeros 20 ones 21 zeros 11 ones

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:

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

frontNum('1111010')
4
frontNum('00110010')
2
```

or 2 base cases:

1 base case:

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

or 2 base cases:

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)?

- **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)?

shortest compressed representation

longest compressed representation

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!

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

'forty*two'

01100110110111101110010011101000111100100101010011101000111011101101111

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
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?

42

(1) set input voltages

101010

(2) perform computation

ADDER circuit

9

001001

110011
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!"

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** 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?
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!