More **bits** of CS

Too many bits? **Compress!**

---

Circuit design, part 1

I'd call this a KNOT gate...

This circuit was NOT, in fact, designed!

---

Below binary: **physical circuits**

---

**Hw #5 due Mon. 2/19**

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

---

**Lots of tutoring hrs - join in... !**
Bits' big idea

left-shifting by 1 doubles a value

Python:

```
42 << 1  # 84
```

Bitwise reason:

```
'101010'  # in binary, columns double in value leftward
```

right-shifting by 1 halves a value

Python:

```
42 >> 1  # 21
```

Bitwise reason:

```
'101010'  # in binary, columns halve in value rightward
```

Take-home:

I hope I don’t have to remember L vs R!

Aha! This can be implemented just with wiring!
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 → 011
01 10 → 010
01 11 → 100
10 00 → 010
10 01 → 011
10 10 → 100
10 11 → 101
11 00 → 011
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.
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'
    # Lots more rules - how many in all?

Notice that this code doesn't "understand" addition at all!
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]
    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'
    # what if we have to carry to the next column?
    if eS == '3' and eT == '9':
        return

Notice that this code doesn't "understand" addition at all!
```
Lab Debriefing &

hw5pr2.py
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'

    ntb( 42 )
    ntb( 21 ) + '0'
    ntb( 10 ) + '1'
    ntb( 5 ) + '0'
    ntb( 2 ) + '1'
    ntb( 1 ) + '0'
    ntb( 0 ) + '1'
    ''

    '1010101'
    out

These are awfully similar...
Lab Debriefing & hw5pr1.py

```
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?

In

```
ntb(42) + '0'
nntb(21) + '0'
nntb(10) + '1'
nntb( 5 ) + '0'
nntb( 2 ) + '1'
nntb( 1 ) + '0'
nntb( 0 ) + '1'
'101010'  `out`

out

fleek-ified!
Lab Debriefing & hw5pr1.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 & hw5pr1.py

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

What if you wanted base-3 input?! **Base-B input?**
Ariane 5

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

[IndexError] [TypeError] [HumanError]

16 bits — version 4

64 bits — version 5
How *far* can we count...?  

<table>
<thead>
<tr>
<th>with</th>
<th>1 bit</th>
<th>2 bits</th>
<th>3 bits</th>
<th>4 bits</th>
<th>7 bits</th>
<th>8 bits</th>
<th>N bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><em>1</em></td>
<td>11</td>
<td>111</td>
<td>1111</td>
<td>111111</td>
<td>1111111</td>
<td><em>1</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>127</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>255</td>
</tr>
</tbody>
</table>

I can see some patterns here – even with one eye closed!
How far back can we remember...?

List of most viewed YouTube videos

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 far back can we remember...?

Only briefly, of course...
Another overflow error!

Less worrisome, perhaps...

The "sign bit" has flipped to one. Thus, the number has become negative...!
Ariane 5

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

Is everything bits?

IndexError     TypeError     HumanError

16 bits 64 bits
version 4 version 5
Insight: Ancient Egyptian multiplication

Not sure - but surprisingly much is ...
Write the factors in two columns. Repeatedly **halve** the LEFT and **double** the RIGHT. (toss remainders...)

Pull out the RIGHT values where the LEFT values are **odd**.

**Sum** those values for the answer!

**Ancient Egyptian Multiplication**

**Example**

<table>
<thead>
<tr>
<th>halver</th>
<th>dbler</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>6</td>
</tr>
</tbody>
</table>

(a. should be 126)

Why does this work? a.k.a. RPM
Write the factors in two columns. 

Repeatedly **halve** the LEFT and **double** the RIGHT. (toss remainders...)

Pull out the RIGHT values where the LEFT values are **odd**.

**Sum** those values for the answer!

**Example**

<table>
<thead>
<tr>
<th>halver</th>
<th>dbler</th>
<th>(ans. should be 126)</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>2</td>
<td>48</td>
<td>96</td>
</tr>
<tr>
<td>1</td>
<td>96</td>
<td><strong>+ 96</strong></td>
</tr>
</tbody>
</table>

**Ans:** 126

**Quiz**

Try it!

<table>
<thead>
<tr>
<th>halver</th>
<th>dbler</th>
<th>(ans. ~ 165)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

**Ans:** ~ 165

<table>
<thead>
<tr>
<th>halver</th>
<th>dbler</th>
<th>(ans. ~ 240)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

**Extra:** Why does this always work? **Hint:** it's binary!
Ancient Egyptian Multiplication!

**AEM algorithm**

Write the factors in two columns.

Repeatedly **halve** the LEFT and **double** the RIGHT. (toss remainders...)

Pull out the RIGHT values where the LEFT values are **odd**.

**Sum** those values for the answer!

Extra: *Why does this always work?*  
**Hint:** it's binary!

---

**Example**

<table>
<thead>
<tr>
<th>halver</th>
<th>dbler</th>
<th>(ans. should be 126)</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>2</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>96</td>
<td>96</td>
</tr>
</tbody>
</table>

**Quiz**

Try it!

<table>
<thead>
<tr>
<th>halver</th>
<th>dbler</th>
<th>(ans. ~ 165)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>halver</th>
<th>dbler</th>
<th>(ans. ~ 240)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>
Although in ancient Egypt the concept of base 2 did not exist, the algorithm is essentially the same algorithm as [long multiplication](https://en.wikipedia.org/wiki/Long_multiplication) after the multiplier and multiplicand are converted to binary. The method as interpreted by conversion to binary is therefore still in wide use today as implemented by [binary multiplier circuits](https://en.wikipedia.org/wiki/Binary_multiplier) in modern computer processors.
Insight  Egyptian + Russian Multiplication

### Decimal

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>60</td>
</tr>
<tr>
<td>1</td>
<td>120</td>
</tr>
</tbody>
</table>

### Binary

\[
\begin{array}{c}
1111_{15} \\
\times \quad 1011_{11} \\
\hline
1111_{15} \\
11110_{30} \\
000000_{60} \\
\hline
1111000_{120} \\
+ \quad 1111000_{120} \\
\hline
10100101_{165}
\end{array}
\]
**Insight**  Egyptian + Russian Multiplication

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>10100</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1010</td>
</tr>
<tr>
<td>80</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>101000</td>
</tr>
<tr>
<td>160</td>
<td>160</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c}
10100_{12} \\
\times 1100_{12} \\
\hline
00000_{20} \\
000000_{40} \\
1010000_{80} \\
10100000_{160} \\
\hline
11110000_{240}
\end{array}
\]
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?
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?
Hw5: *images are just bits, too!*

Binary Image

Encoding as raw bits
one big string of 64 characters

"1010101001010101101010100101010110101010010101011010101001010101"
likely binary image...

and a reasonable candidate for compression
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!

Looks like the right 2% to keep!

How is it possible to throw away 98% of the image data!?
Most often... what's done?

**compressed** to 40kb

**original:** 2.3mb
compressed to 40kb

original: 2.3mb
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*?

"000000000000000011111111111111110000000000000000000000000000001111"
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

1 is the next digit

Again, there are 16 of them.

and so on...

7 bits: # of repeats

8-bit data block

28 zeros

4 ones

8-bit data block

8-bit data block

8-bit data block

00010000100100000001110010000100

00000000000000000000000000000000

11111111111111111111111111111111

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 """

def decompress( CI ):
    """ returns the binary image I from the run-length-encoded, "compressed" input, CI """
**def** `compress( I )`:

""" returns the RLE of the input binary image, I """

```
0001000010010000001110010000100
```

16 zeros  16 ones  28 zeros  4 ones

**def** `uncompress( CI )`:

""" returns the binary image I from the run-length-encoded, "compressed" input, CI """

```
0000000000000000111111111111111100000000000000000000000000001111
```

16 zeros  16 ones  28 zeros  4 ones
def compress( I ):
    """ returns the RLE of the input binary image, I """

the "compressed" image:
"0001000010010000001110010000100"

16 zeros 16 ones 28 zeros 4 ones

def uncompress( CI ):
    """ returns the binary image I from the run-length-encoded, "compressed" input, CI """

back to the original binary image
"000000000000000011111111111111110000000000000000000000001111"

16 zeros 16 ones 28 zeros 4 ones

what helper function might be useful here?
**Try it!**

Try writing the recursive function, \texttt{frontNum(S)}

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

\texttt{frontNum(S)} should return the \# of times the first element of the input \textit{S} appears consecutively \textbf{at the start} of \textit{S}:

Examples...

```python
>>> frontNum('111010')
4
>>> frontNum('00110010')
2
```

What are the \textbf{BEST} / \textbf{WORST} compression results you can get for an 8x8 input image (64 bits)?

**EXTRA!** How can you change our algorithm so that compressed images are always smaller than the originals?
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)?

- **shortest compressed representation**
  - only 8 bits total!

- **longest compressed representation**
  - aargh! 512 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)?

This is provably IMpossible!

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?
Binary images in practice...

Original Image

threshold too low

T = 78/255

threshold too high

T = 120/255

Adaptive T

Adaptive Threshold

No magic! (unfortunately!)

Position became something of a mania that, by the 1930s, reached its peak. The introduction of the luxury, or superliner. The luxury liners were mechanical and architectural wonders, referred to as "ships of state," and were often heavily subsidized by national governments and decorated to reflect that nation's culture.

T = 120/255
This landscape image is determined to contain a portrait document.

Intensity profiles

Lots of peaks == lots of text lines

Few peaks == few text lines

Transforming Technology:
Engineering for a Changing World

Right-side up?
This landscape image is determined to contain a portrait document.

Intensity profiles:
- Lots of peaks == lots of text lines
- Few peaks == few text lines}

Right-side up?

Lots of peaks == lots of text lines

Transforming Technology:
Engineering for a Changing World
It's all bits!

images, text, sounds, data, ...

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

'forty*two'

011001100110111101110010011101000111100100101010011101000111011101101111

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?
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**
   - **42**: 001001
   - **9**: 001001

2. **perform computation**
   - **101010**
   - **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?
Adding strings?

Multiplying by machine:

Doing anything by machine...

is circuit addition!

is syntactic multiplying!

is circuit interaction!

is syntactic interaction!

syntactic ~ meaning-free

means it can be done purely via surface syntax, which means it can be done without thinking...
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!
From gates to circuits...

What inputs make this circuit output 1?

What inputs make this circuit output 0?

Logisim
from circuit design...

next 2 weeks

...to a full computer!

Have an outstanding and fussituous week!(end)!

Why?!