More **bits** of CS

Too many bits? **Compress!**

Below binary: **physical circuits**

---

**Lab Debriefing & hw4pr2.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'
    else:
        return numToBin(N//2) + '1'
```

---

**Bits' big idea**

- **left-shifting by 1 doubles** a value
  - `42 << 1` 84
  - `1010100`

- **right-shifting by 1 halves** a value
  - `42 >> 1` 21
  - `10101`

---

**How high can we count...?**

<table>
<thead>
<tr>
<th>N bits</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>1</td>
</tr>
<tr>
<td>2 bits</td>
<td>11</td>
</tr>
<tr>
<td>3 bits</td>
<td>111</td>
</tr>
<tr>
<td>4 bits</td>
<td>1111</td>
</tr>
<tr>
<td>7 bits</td>
<td>111111</td>
</tr>
<tr>
<td>8 bits</td>
<td>1111111</td>
</tr>
<tr>
<td>31 bits</td>
<td>255</td>
</tr>
</tbody>
</table>
**Insight** Ancient Egyptian Multiplication

AEM/RPM algorithm

Write the factors in two columns. 
Repeatedly **half** the LEFT and double the RIGHT. (just remainders...) 
Pull out the RIGHT values where the LEFT values are **odd**. 
**Sum** those values for the answer!

**Why does this work?**

a.k.a. Russian Peasants' Multiplication

Example

**Insight** AEM algorithm

### Decimal

<table>
<thead>
<tr>
<th>21</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>24</td>
</tr>
<tr>
<td>2</td>
<td>48</td>
</tr>
<tr>
<td>1</td>
<td>96</td>
</tr>
</tbody>
</table>

### Binary

110, 6

0000, 12

11000, 24

000000, 8

+ 1100000, 96

1111110, 126

Although in ancient Egypt the concept of base 2 did not exist, the algorithm is essentially the same algorithm as **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 in modern computer processors**.

**Hw4:** images are just bits, too!

**Hw4:** **lossless** binary image compression

Binary Image

10101010
01010101
10101010
01010101
10101010
01010101

Encoding as raw bits

one big string of 64 characters

"1010101010101010101010101010101010101010101010101010101010101010"

Binary Image

Encoding as raw bits

one big string of 64 characters

"0000000000000000000000000000000000000000000000000000000000000000001111"

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

same-data streaks

compress

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

```
00010000
```

7 bits?

8-bit total data block

What if you need a larger # of repeats?
Try it!

**frontNum(S)** should return the number of times the first element of the argument `S` appears consecutively **at the start** of `S`:

Try writing the recursive function, **frontNum(S)**

```python
def frontNum(S):
    # 1 base case:
    if len(S) <= 1:
        return
    # or 2 base cases:
    elif S[0] == ____ :
        return
    else:
        return
```

Examples...

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

**EXTRA!** Can you change our algorithm so that compressed images are always smaller than the originals?
It's all bits!

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

'forty*two'

011001100110111101110010011101000111100100101010
011101000111011101101111

All computation boils down to manipulating bits!

All computation is simply functions of bits

binary inputs A and B

output, A+B

If len(S) == 0: return T
if len(T) == 0: return S

eS = S[-1]  # S ~ the "end of S"  
eT = T[-1]  # T ~ the "end of T"

if eS == '0' and eT == '1': return add10(S[:-1], T[:-1]) + '1'
elif eS == '1' and eT == '0': return add10(S[:-1], T[:-1]) + '2'
elif eS == '2' and eT == '1': return add10(S[:-1], T[:-1]) + '3'
elif eS == '3' and eT == '1': return add10(S[:-1], T[:-1]) + '4'
# what if we have to carry to the next column?
elif eS == '3' and eT == '9':
    return

In a computer, each bit is represented as a voltage (1 is +3v and 0 is 0v)

(1) set input voltages
(2) perform computation
(3) read output voltages

Computation is simply the deliberate combination of those voltages!

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

Carrying on...

S = '23'
T = '19'

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'
    elif eS == '1' and eT == '0': return add10(S[:-1], T[:-1]) + '2'
    elif eS == '2' and eT == '1': return add10(S[:-1], T[:-1]) + '3'
    elif eS == '3' and eT == '1': return add10(S[:-1], T[:-1]) + '4'
    # what if we have to carry to the next column?
    elif eS == '3' and eT == '9':
        return

hw4: addB
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!
### Quiz

**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!*

<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></td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>96</td>
<td>+ 96</td>
</tr>
<tr>
<td></td>
<td></td>
<td>126</td>
</tr>
</tbody>
</table>

**Example**

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>

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