Welcome to CS 5!  Be sure to watch your head...
Welcome to CS 5! Be sure to watch your head...
Welcome back to CS 5!

Homework 4

Problem 0: Reading + response...

Problem 1: Convert to/from binary – and ternary!

Problem 2: Data and image compression (wow!)

due Sunday

due 6/22 (today!)
due Sunday
How high can we count... in 2015?

Cha(r)t survey of the day!

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?

Cha(r)t survey of the day!
How high can we count... in 2015?

Cha(r)t survey of the day!

What pop songs were #1 and #4 when Baby and Blank Space were #2 and #3?
How high can we count... in 2015?

Cha(r)t survey of the day!

What pop songs were #1 and #4 when Baby and Blank Space were #2 and #3?
Picobot + Recursion wrap-up!

few forget Picobot!
Bourton-on-the-water

Picobot + Recursion wrap-up!
Bourton-on-the-water
Bourton-on-the-water

town of 2000 people
Bourton-on-the-water's 1/9 model
has a level-2 model...
has a level-2 model...
and a level-3 model...
and a level-3 model...
and even a (very small!) level-4 model
and even a (very small!) level-4 model

Recursion Wrap-up!?
and even a (very small!) level-4 model

Recursion Wrap-up!?
We descend through CS!
Looking Back

Computing as composition

clay == functions

Looking Forward

Computing as representation

clay == data & bits

CS 101 Today...

Our top-10 list of binary jokes:

- On a scale of 1 to 10, how likely is it that this question is using binary?
  - ...4?
  - What’s a 4?

There are only 10 types of people in the world: Those who understand binary and those who don’t.
Binary Storage & Representation

The SAME bits can represent different pieces of data, depending on type.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Dec</th>
<th>Hex</th>
<th>Glyph</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010 0000</td>
<td>32</td>
<td>20</td>
<td>(blank) (\text{blank})</td>
</tr>
<tr>
<td>0010 0001</td>
<td>33</td>
<td>21</td>
<td>!</td>
</tr>
<tr>
<td>0010 0010</td>
<td>34</td>
<td>22</td>
<td>&quot;</td>
</tr>
<tr>
<td>0010 0011</td>
<td>35</td>
<td>23</td>
<td>#</td>
</tr>
<tr>
<td>0010 0100</td>
<td>36</td>
<td>24</td>
<td>$</td>
</tr>
<tr>
<td>0010 0101</td>
<td>37</td>
<td>25</td>
<td>%</td>
</tr>
<tr>
<td>0010 0110</td>
<td>38</td>
<td>26</td>
<td>&amp;</td>
</tr>
<tr>
<td>0010 0111</td>
<td>39</td>
<td>27</td>
<td>*</td>
</tr>
<tr>
<td>0010 1000</td>
<td>40</td>
<td>28</td>
<td>(</td>
</tr>
<tr>
<td>0010 1001</td>
<td>41</td>
<td>29</td>
<td>)</td>
</tr>
<tr>
<td>0010 1010</td>
<td>42</td>
<td>2A</td>
<td>*</td>
</tr>
<tr>
<td>0010 1011</td>
<td>43</td>
<td>2B</td>
<td>+</td>
</tr>
</tbody>
</table>

8 bits = 1 byte = 1 box

The same bits are in each container.

But why these bits?
The SAME bits represent an integer or a string, depending on type: `int` or `str`
The SAME bits represent an integer or a string, depending on type: \texttt{int} or \texttt{str}

<table>
<thead>
<tr>
<th>Binary</th>
<th>Dec</th>
<th>Hex</th>
<th>Glyph</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010 1111</td>
<td>47</td>
<td>2F</td>
<td>/</td>
</tr>
<tr>
<td>0011 0000</td>
<td>48</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>0011 0001</td>
<td>49</td>
<td>31</td>
<td>1</td>
</tr>
</tbody>
</table>

1 byte

8 bits

Identical bits are stored in each variable!

The types determine how to interpret the bits; the names don't matter at all…

over in memory…
Unicode + ASCII

In Python, `chr` and `ord` convert to/from Unicode + ASCII

<table>
<thead>
<tr>
<th>Binary</th>
<th>Dec</th>
<th>Hex</th>
<th>Glyph</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010 1111</td>
<td>47</td>
<td>2F</td>
<td>/</td>
</tr>
<tr>
<td>0011 0000</td>
<td>48</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>0011 0001</td>
<td>49</td>
<td>31</td>
<td>1</td>
</tr>
</tbody>
</table>

The SAME bits represent an integer or a string, depending on type: `int` or `str`
"Everything is bits"

The SAME bits represent an integer or a string, depending on type: \texttt{int} or \texttt{str}
This is why 'CS' < 'clear'!
Unsure
Binary Storage & Representation

8 bits = 1 byte = 1 box

The same bits are in each container.

The SAME bits can represent different pieces of data, depending on **type**.

- **str**
  - value: `' '*`
  - name:
- **int**
  - value: `42`
  - name:

But why **these** bits?
What is 42?

42

It's *not* this!
What is 42?

Syntax.

Value!
forty two
forty two

value

42

syntax

4
tens

2
ones
fifty two

Value
(semantics)

stuff we care about
(what things mean)

Syntax

stuff we use to communicate

42

tens

ones
forty two

Same Value!
forty two

Same Value!

thirty-twos  sixteens  eights  fours  twos  ones

but, a different syntax
thirty-twos  sixteens  eights  fours  twos  ones

with a binary syntax
forty two

\[
\begin{align*}
30 & \equiv 1 \\
16 & \equiv 0 \\
8 & \equiv 1 \\
4 & \equiv 0 \\
2 & \equiv 1 \\
1 & \equiv 0 \\
\end{align*}
\]

with a binary syntax
**Base 2**
"binary"

101010₂

**Base 10**
"decimal"

42₁₀

**Syntax**
the symbols used
(what things look like)

**Value**
stuff we care about
(what things mean)

forty two

the symbols used
(what things look like)
Base 2
"binary"

101010₂

Base 10
"decimal"

42₁₀

Different

Syntax
the symbols used
(what things look like)

Same!

Value
stuff we care about
(what things mean)

forty two

Syntax

Value

Base 2
"binary"

Base 10
"decimal"

101010₂

42₁₀

4 tens + 2 ones

123₁₀

1 hundred + 2 tens + 3 ones

writing 123 in binary…

each column represents the base’s next power

128's column
SixtyFour's column
ThirtyTwo's column
Sixteen's column
Eight's column
Four's column
Two's column
One's column

Tens column
Ones column

Tens column
Ones column

Hundred's column
Ones column

THIRTYTWOs col.
SIXTEENS col.
EIGHTS column
FOURs column
TWOs column
ONES column

THIRTYTWOs column
SIXTEENS column
EIGHTS column
FOURs column
TWOs column
ONES column
# Binary math

## Tables of One-Digit Facts

### Addition

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
</tbody>
</table>

### Multiplication

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>16</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>18</td>
<td>21</td>
<td>24</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
<td>32</td>
<td>36</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>40</td>
<td>45</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>12</td>
<td>18</td>
<td>24</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>48</td>
<td>54</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>14</td>
<td>21</td>
<td>28</td>
<td>35</td>
<td>42</td>
<td>49</td>
<td>56</td>
<td>63</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
<td>72</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>18</td>
<td>27</td>
<td>36</td>
<td>45</td>
<td>54</td>
<td>63</td>
<td>72</td>
<td>81</td>
</tr>
</tbody>
</table>

---

[http://www.youtube.com/watch?v=Nh7xapVB-Wk](http://www.youtube.com/watch?v=Nh7xapVB-Wk)
In binary, I'm an 11-eyed alien!

Convert these two binary numbers to decimal:

\[
\begin{array}{c|c|c|c|c|c}
32 & 16 & 8 & 4 & 2 & 1 \\
110011 & 10001000 \\
\end{array}
\]

Convert these two decimal numbers to binary:

\[
\begin{array}{c|c|c|c|c|c}
32 & 16 & 8 & 4 & 2 & 1 \\
28 & 101 & \\
101 & 10 & \\
\end{array}
\]

Add these two binary numbers:

\[
\begin{array}{c}
101101 \\
+ 1110 \\
\end{array}
\]

Multiply these binary numbers:

\[
\begin{array}{c}
101101 \\
* \quad 1110 \\
\end{array}
\]

**Extra!** Can you figure out the last binary digit (bit) of 53 without determining any other bits? The last two? 3?
Convert these two binary numbers to decimal:

- **110011**
  - $32 + 16 + 2 + 1 = 51$
- **10001000**
  - $128 + 8 = 136$

Convert these two decimal numbers to binary:

- **28**
  - **011100**
- **101_{10}**
  - **01100101**

Extra! Can you figure out the last binary digit (bit) of 53 without determining any other bits? The last two? 3?

We'll return to this in a bit...
Add these two binary numbers *WITHOUT* converting to decimal!

```
32 16  8  4  2  1

101101
  + 1110

1271
```

Hint: Do you remember this algorithm? It's the same!
Add these two binary numbers \textit{WITHOUT} converting to decimal!

\begin{align*}
\begin{array}{c}
1 \\
11 \\
\hline
101101 \\
+ \\
1110 \\
\hline
101101
\end{array}
\end{align*}

Hint: Do you remember this algorithm? It’s the same!

\[
\begin{array}{c}
1 \\
529 \\
+ \\
742 \\
\hline
1271
\end{array}
\]
Multiply these two binary numbers **WITHOUT** converting to decimal!

\[
\begin{array}{cccccc}
32 & 16 & 8 & 4 & 2 & 1 \\
\hline
101101 & & & & & \\
\times & & & & & \\
1110 & & & & & \\
\hline & 1110 & & & & \\
\end{array}
\]

<table>
<thead>
<tr>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>529</td>
<td>*</td>
<td>42</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1058 + 2116</td>
<td>&amp; &amp;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22218</td>
<td>&amp; &amp;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hint:
Do you remember this algorithm? It's the same!

**Goal**

630

A machine could - and probably **should** - be doing this!
Multiply these two binary numbers **WITHOUT** converting to decimal!

```
101101
* 1110
-----
000000
1011010
10110100
+ 101101000
-----
1001110110
```

**Goal**

```
529
* 42
---
1058
+ 2116
---
22218
```

"partial products"

A machine could - and probably **should** - be doing this!

Hint:
Do you remember this algorithm? It's the same!
There are 10 kinds of "people" in the universe:
those who know ternary,
those who don't, and
those who think this is a binary joke!
<table>
<thead>
<tr>
<th>Base</th>
<th>Conversion</th>
<th>Digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>101010</td>
<td>0, 1</td>
</tr>
<tr>
<td>3</td>
<td>1120</td>
<td>0, 1, 2</td>
</tr>
<tr>
<td>4</td>
<td>222</td>
<td>0, 1, 2, 3</td>
</tr>
<tr>
<td>5</td>
<td>132</td>
<td>0, 1, 2, 3, 4</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>0, 1, 2, 3, 4, 5</td>
</tr>
<tr>
<td>7</td>
<td>60</td>
<td>0, 1, 2, 3, 4, 5, 6</td>
</tr>
<tr>
<td>8</td>
<td>52</td>
<td>0, 1, 2, 3, 4, 5, 6, 7</td>
</tr>
<tr>
<td>9</td>
<td>46</td>
<td>0, 1, 2, 3, 4, 5, 6, 7, 8</td>
</tr>
<tr>
<td>10</td>
<td>42</td>
<td>0, 1, 2, 3, 4, 5, 6, 7, 8, 9</td>
</tr>
<tr>
<td>11</td>
<td>39</td>
<td>0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A</td>
</tr>
<tr>
<td>16</td>
<td>2A</td>
<td>0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F</td>
</tr>
</tbody>
</table>

**Hexadecimal**

All 42s!
| Base | Representation | Digits |重大事件
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td>42s!</td>
</tr>
<tr>
<td>2</td>
<td>101010</td>
<td>0, 1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1120</td>
<td>0, 1, 2</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>2A</td>
<td>0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F</td>
<td></td>
</tr>
</tbody>
</table>
Our Mascot, the Panda

Off base?

Base 12 –
"Duodecimal Society"
"Dozenal Society"

Base 20:
Americas

Olmec base-20 numbers
E. Mexico, ~ 300 AD

Base 27:
New Guinea

Telefol is a language spoken by the Telefol people in Papua New Guinea, notable for possessing a base-27 numeral system.

Base 60 – Ancient Sumeria

Some of these bases are still echoing around...
But *why* binary?
Ten symbols is too many!

A computer has to differentiate \textit{physically} among all its possibilities.

\begin{center}
\begin{tabular}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{tabular}
\end{center}

ten symbols \sim ten different voltages

\textcolor{red}{This is too difficult to replicate billions of times}\textcolor{red}{\hspace{0.5cm}}(engineering)

What digits are these?

\textcolor{red}{Ouch!}
Ten symbols is too many!

A computer has to differentiate physically among all its possibilities.

```
0 1 2 3 4 5 6 7 8 9
```

ten symbols \sim ten different voltages

This is too difficult to replicate billions of times

What digits are these?
Two symbols is easiest!

A computer has to differentiate **physically** among all its possibilities.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ten symbols ~ ten different voltages

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

two symbols ~ two different voltages

What digits are these?

Easy!
Two symbols is easiest!

A computer has to differentiate **physically** among all its possibilities.

ten symbols ~ ten different voltages

two symbols ~ two different voltages

What digits are these?
Ternary computers?

50 of these *Setun* ternary machines were made at Moscow U. ~ 1958

This project was discontinued in 1970... *though not because of the ternary design!*
Back to bits...
Reasoning ~ Value vs. Syntax

What does `left-shifting` do to the `value` of a `decimal` #?

What does `right-shifting` do to the `value` of a `decimal` #?

Python bitwise operators: the "shifts"

- `<<` - left-shift
- `>>` - right-shift
Reasoning, *bit by bit*

**left-shift by 1**

- 11
- 110

3 << 1

- 6

What does *left-shifting* do to the value of a binary #?

**left-shift by 2**

- 11
- 1100

3 << 2

- 12

**right-shift by 1**

- 101010
- 10101

42 >> 1

- 21

- 1010

42 >> 2

- ?

What does *right-shifting* do to the value of a binary #?
Old Microsoft systems interview question, #42:

42. Give a fast way to multiply a number by 7.
In today's processors shifts, and, or, add, and subtract are all very fast, whereas multiplying, dividing, and mod are relatively slow.

With this in mind, how could we compute these expressions using only fast operations, maybe in combination?

\[
\begin{align*}
\text{N} / / 4 & \\
\text{N} \times 7 & \\
\text{N} \times 17 & \\
\text{N} \% 16 & \text{ extra fleek!}
\end{align*}
\]
In today's processors shifts, and, or, add, and subtract are all *very fast*, whereas multiplying, dividing, and mod are relatively *slow*.

With this in mind, how could we compute these expressions using *only fast* operations, maybe in combination?

<table>
<thead>
<tr>
<th>Expression</th>
<th>Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/4</td>
<td>N &gt;&gt; 2</td>
</tr>
<tr>
<td>N*7</td>
<td>(N&lt;&lt;3) - N</td>
</tr>
<tr>
<td>N*17</td>
<td>(N&lt;&lt;4) + N</td>
</tr>
<tr>
<td>N%16</td>
<td>N-((N&gt;&gt;4)&lt;&lt;4) extra fleek!</td>
</tr>
</tbody>
</table>
Back to bits... 

not the original name...
**Extra!**  Can you figure out the **last binary digit** (bit) of 53 without determining any earlier bits?  The last **two**?  **three**?  All of them?

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
<td>(17-16)</td>
<td>(12-15)</td>
<td>(7-14)</td>
<td>(1-15)</td>
<td>(0-16)</td>
<td>(11-16)</td>
</tr>
<tr>
<td>17</td>
<td>255</td>
<td>65535</td>
<td>32767</td>
<td>16383</td>
<td>8191</td>
<td>4095</td>
</tr>
</tbody>
</table>
Lab 4: *Computing in binary*

This first step of **left-to-right** conversion into binary is tricky to program... *Why?*

You mean *aside* from having to think in binary?
Lab 4: Computing in binary

This first step of left-to-right conversion into binary is tricky to program... *Why?*

It's tricky to find the largest power needed...
Lab 4: *Computing in binary*

Let's run right-to-left!

What does the fact that 141 is ODD tell us?!
# Lab 4: Computing in binary

<table>
<thead>
<tr>
<th>base 10</th>
<th>base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>141</td>
<td>10001101</td>
</tr>
<tr>
<td>141</td>
<td>10001101</td>
</tr>
<tr>
<td>70</td>
<td>10001101</td>
</tr>
<tr>
<td>35</td>
<td>10001101</td>
</tr>
<tr>
<td>21</td>
<td>10001101</td>
</tr>
<tr>
<td>16</td>
<td>10001101</td>
</tr>
<tr>
<td>4</td>
<td>10001101</td>
</tr>
<tr>
<td>2</td>
<td>10001101</td>
</tr>
</tbody>
</table>

What does the fact that 141 is ODD tell us?!

Let's run right-to-left!

141 = 10001101

answer
Lab 4: *Computing in binary*

Why does this work?!

Let's run right-to-left!

What does the fact that 141 is ODD tell us?!
Converting to binary ~ starting from the right!

<table>
<thead>
<tr>
<th>...</th>
<th>32s</th>
<th>16s</th>
<th>8s</th>
<th>4s</th>
<th>2s</th>
<th>1s</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>16s</td>
<td>8s</td>
<td>4s</td>
<td>2s</td>
<td>1s</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>8s</td>
<td>4s</td>
<td>2s</td>
<td>1s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>4s</td>
<td>2s</td>
<td>1s</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

in the end, we need "53"-worth of value

bits!

value remaining

top-level reality!

"next"-level reality...

53
in the end, we need "53"-worth of value

Converting to binary ~ starting from the right!

top-level reality!
"next"-level reality...
bits!
value remaining
in the end, we need "53"-worth of value

Extra! Can you figure out the last binary digit (bit) of 53 without determining any earlier bits? The last two? three? All of them?
Lab 4: Computing in binary

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 10 1</td>
<td></td>
</tr>
<tr>
<td>141</td>
<td></td>
</tr>
</tbody>
</table>

141 = 10001101

What does the fact that 141 is ODD tell us?!

Let's run right-to-left!

141 = 10001101

128 64 32 16 8 4 2 1
Lab 4: Computing in binary

You'll write these right! (to-left)

numToBinary( N )  binaryToNum( S )

decimal syntax, N  we need to represent binary numbers with strings

n2b(141)  b2n('10001101')

Right-to-left works!
def numToBinary( N ):
    if N == 0:
        return ''
    elif N%2 == 0:
        return numToBinary(     ) + empty string means 0
    else:
        return numToBinary(     ) + empty string means 0

If N is even, what is the final bit?
If N is odd, what is the final bit?

How much VALUE is left to convert!?
### Bits & Binary

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
</tr>
<tr>
<td>16</td>
<td>10000</td>
</tr>
<tr>
<td>21</td>
<td>10101</td>
</tr>
<tr>
<td>42</td>
<td>101010</td>
</tr>
<tr>
<td>127</td>
<td>11111111</td>
</tr>
</tbody>
</table>

#### Shifting Bits

- **Shifting bits left** (`<< 1`)
  - What's `101000`?  
  - What's `1000`?  

- **Shifting bits right** (`>> 1`)
  - Maximum value of 4 bits? 
  - Maximum value of 7 bits? 
  - Maximum value of `N` bits?!
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>
</tbody>
</table>

- I can see some patterns here – even with one eye closed!

15 bits

31 bits
Counting sheep, xkcd style...

1...2...

1,306...1,307...

32,767...-32,768...

-32,767...-32,766...

How many bits?
Ariane 5

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

IndexError  

TypeError  

HumanError

16 bits  

64 bits

version 4  

version 5

a much less critical error...
How high can we count... in 2015?

<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>
Other overflow errors...

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

Pixel at 42,42 has

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

How many bits represent each color channel?
All images are just bits

pixel at 42,42 has

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

how many bits represent each color channel?

red = 00000001 (8 bits)
grn = 00100100 (8 bits) "24 bit color"
blu = 01110101 (8 bits)
hw4pr2: *images are just bits, too!*

**Binary Image**

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

**Encoding as raw bits**

one big string of 64 characters

```
"1010101001010101101010100101010110101010010101011010101001010101"
```
Too many pixels... too little time + space!

Image compression is everywhere!
How is it possible to throw away 98% of the image data!? 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
compressed to 40kb

original: 2.3mb
hw4pr2: **lossless** image compression

Binary Image

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

Encoding as raw bits

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

same-data streaks
a very 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*?

"000000000000000011111111111111110000000000000000000001111"
One possible algorithm:

```
bit #repeats
```

Any problems with this?
0100001100000111001100

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!

0100001100000011100011100

our algorithm:

**bit #repeats**

could be misinterpreted!

1,098,188 zeros!
### fixed-width compression

<table>
<thead>
<tr>
<th>Bit</th>
<th># repeats</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>28 zeros</td>
</tr>
<tr>
<td></td>
<td>4 ones</td>
</tr>
</tbody>
</table>

We need **fixed-width** blocks:

- 1 bit fill
- 7 bits for the # of repeats

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

\[
\begin{array}{c}
\text{8-bit total data block} \\
\text{7 bits: # of repeats} \\
\text{1 bit: the initial pixel}
\end{array}
\]

*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

"00101010100111111"

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 """
```

What helper function would be useful for `compress`?

What's an image $I$ whose compressed output \textit{gets larger, not smaller}? (Aargh!)

- What are the BEST-compressible / WORST-compressible 64-bit images?
- How could you \textit{improve} the algorithm so that it always \textit{compresses}?!?
def compress( I ):
    """ returns the RLE of the input binary image, I """

a binary image, IQuiz

"0000000000001111111111111111111100000000000000000000011111111111"

12 zeros 20 ones 21 zeros 11 ones

"000011001001010100000010101110001011"

12 20 21 11

the "compressed" image returned from compress( IQuiz )
**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
    elif len(S) == 1:
        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)?

**BEST**

only 8 bits total!

**WORST**

aargh! 512 bits!

Anyone see why this is NOT QUITE the worst-compressable 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!

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'

01100110 01101111 01110010 01110100 01111001 00101010 01110100 01110111 01101111

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

All computation boils down to manipulating bits!
Adding strings?

Multiplying by machine:

Doing anything by machine...

\textbf{syntactic} \sim \text{meaning-free}

\textbf{syntactic} multiplying!

\textbf{syntactic} interaction!

\textbf{circuit} multiplying!

\textbf{circuit} interaction!

means it can be done purely via \textbf{surface syntax}, which means it can be done \textit{without thinking}...
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

42

101010

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?

---

42

101010

(1) set input voltages

(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!

42 \[\downarrow\] 101010 \(\xrightarrow{(1)}\) set input voltages

9 \[\uparrow\] 001001

110011 \(\xrightarrow{(2)}\) perform computation

\[\uparrow\]

But what’s this green thing?

**ADDER circuit**

\[\downarrow\]

51 \(\xrightarrow{(3)}\) read output voltages

Richard Feynman: "Computation is just a physics experiment that always works!"
When traveling, always insist on *bitwise* accommodations ... !

See you at lab – *in just a bit!*