Representing Information

“Bit Juggling”

- Representing information using bits
- Number representations
- Some other bits
Motivations

- Computers Process Information
- Information is measured in bits
- By virtue of containing only “switches” and “wires” digital computer technologies use a binary representation of bits
- How do we use/interpret bits
- We need standards of representations for
  - Letters
  - Numbers
  - Colors/pixels
  - Music
  - Etc.

It isn’t a dream; the semester really has started.
Example: Sum of 2 dice

Information = $\log_2(\text{N original choices}/M \text{ new choices})$ bits

The average information provided by the sum of 2 dice:

$$i_{\text{ave}} = \sum_{i=2}^{12} \frac{M}{N} \log_2 \left( \frac{N}{M} \right) = -\sum_i p_i \log_2 (p_i) = 3.274 \text{ bits}$$
Show Me the Bits!

Can the sum of two dice REALLY be represented using 3.274 bits? If so, how?

The fact is, the average information content is a strict *lower-bound* on how small of a representation that we can achieve.

In practice, it is difficult to reach this bound. But, we can come very close.
Variable-Length Encoding

• Of course, we can use differing numbers of “bits” to represent each item of data. This is particularly useful if all items are *not* equally likely

• Equally likely items lead to fixed length encodings:
  – Ex: Encode a particular roll of 5?
  – {(1,4), (2,3), (3,2), (4,1)} which are equally likely if we use fair dice
  – Entropy = \(- \sum_{i=1}^{4} p(roll_i|\text{roll} = 5) \log_2 (p(roll_i|\text{roll} = 5)) = - \sum_{i=1}^{4} \frac{1}{4} \log_2 \left(\frac{1}{4}\right) = 2 \text{ bits}\)
  – 00 – (1,4), 01 – (2,3), 10 – (3,2), 11 – (4,1)

• Back to the original problem. Let’s use this encoding:

  2 - 10011   3 - 0101   4 - 011   5 - 001
  6 - 111    7 - 101   8 - 110   9 - 000
 10 - 1000   11 - 0100  12 - 10010
Variable-Length Encoding

• Taking a closer look

<table>
<thead>
<tr>
<th>Number</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10011</td>
</tr>
<tr>
<td>3</td>
<td>0101</td>
</tr>
<tr>
<td>4</td>
<td>011</td>
</tr>
<tr>
<td>5</td>
<td>001</td>
</tr>
<tr>
<td>6</td>
<td>111</td>
</tr>
<tr>
<td>7</td>
<td>101</td>
</tr>
<tr>
<td>8</td>
<td>110</td>
</tr>
<tr>
<td>9</td>
<td>000</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
</tr>
<tr>
<td>11</td>
<td>0100</td>
</tr>
<tr>
<td>12</td>
<td>10010</td>
</tr>
</tbody>
</table>

Unlikely rolls are encoded using more bits

More likely rolls use fewer bits

• Decoding

Example Stream: 1001100101011110011100101
Huffman Coding

- A simple *greedy* algorithm for approximating an entropy efficient encoding
  1. Find the 2 items with the smallest probabilities
  2. Join them into a new *meta* item whose probability is the sum
  3. Remove the two items and insert the new meta item
  4. Repeat from step 1 until there is only one item
Once the *tree* is constructed, label its edges consistently and follow the paths from the largest *meta* item to each of the real item to find the encoding.

2 - 10011  
3 - 0101  
4 - 011  
5 - 001  
6 - 111  
7 - 101  
8 - 110  
9 - 000  
10 - 1000  
11 - 0100  
12 - 10010

Huffman decoding tree
Encoding Efficiency

• How does this encoding strategy compare to the information content of the roll?

\[ b_{\text{ave}} = \frac{1}{36} 5 + \frac{2}{36} 4 + \frac{3}{36} 3 + \frac{4}{36} 3 + \frac{5}{36} 3 + \frac{6}{36} 3 \]
\[ + \frac{5}{36} 3 + \frac{4}{36} 3 + \frac{3}{36} 4 + \frac{2}{36} 4 + \frac{1}{36} 5 \]
\[ b_{\text{ave}} = 3.306 \]

• Pretty close. Recall that the lower bound was 3.274 bits. However, an efficient encoding (as defined by having an average code size close to the information content) is not always what we want!
Encoding Considerations

- Encoding schemes that attempt to match the information content of a data stream remove redundancy. They are *data compression* techniques.

- However, sometimes our goal in encoding information is *increase redundancy*, rather than remove it. Why?
  - Makes the information easier to manipulate (fixed-sized encodings)
  - Makes the data stream resilient to noise (error detecting and correcting codes)

- Data compression allows us to store our entire music collection in a pocketable device

- Data redundancy enables us to store that *same* information *reliably* on a hard drive
An Example

Consider your favorite MP3:

American Idiot:

Audio CD: 176 s
Channels (stereo): x 2
samples / sec x 44100
bits / sample x 16

248,371,200 bits

MP3:

128 kbit/s
bits / byte x 8

2,821,829 bytes
22,574,632 bits

Compression Ratio: 248,371,200 / 22,574,632 = 11 : 1
Encoding

- Encoding describes the process of assigning representations to information.
- Choosing an appropriate and efficient encoding is a real engineering challenge (and an art).
- Impacts design at many levels:
  - Mechanism (devices, # of components used)
  - Efficiency (bits used)
  - Reliability (noise)
  - Security (encryption)
Fixed-Length Encodings

If all choices are *equally likely* (or we have no reason to expect otherwise), then a fixed-length code is often used. Such a code should use at least enough bits to represent the information content.

ex. Decimal digits $10 = \{0,1,2,3,4,5,6,7,8,9\}$

4-bit BCD (binary code decimal)

$$\log_2(10/1) = 3.322 < 4\text{ bits}$$

ex. ~84 English characters = \{A-Z (26), a-z (26), 0-9 (10), punctuation (8), math (9), financial (5)\}

7-bit ASCII (American Standard Code for Information Interchange)

$$\log_2(84/1) = 6.392 < 7\text{ bits}$$
Unicode

• ASCII is biased towards western languages. English in particular.
• There are, in fact, many more than 256 characters in common use:
  â, m, ö, ŋ, è, ¥, 擶, 敌, 
  力, छ, ञ, श, ञ, က, က
• Unicode is a worldwide standard that supports all languages, special characters, classic, and arcane
• Several encoding variants 16-bit (UTF-8)

  ASCII equiv range: 0xxxxxxxxxxxx

  Lower 11-bits of 16-bit Unicode 110yyyxyx 10xxxxxxx

  16-bit Unicode 1110zzzzz 10zyyyyyx 10xxxxxxx

  11110www 10wwzzzzz 10zyyyyyx 10xxxxxxx
Encoding Positive Numbers

It is straightforward to encode positive integers as a sequence of bits. Each bit is assigned a weight. Ordered from right to left, these weights are increasing powers of 2. The value of an n-bit number encoded in this fashion is given by the following formula:

\[ v = \sum_{i=0}^{n-1} 2^i b_i \]

2^11 \cdot 2^2 \cdot 2^8 \cdot 2^9 \cdot 2^6 \cdot 2^5 \cdot 2^4 \cdot 2^3 \cdot 2^2 \cdot 2^1 \cdot 2^0

\[ 0111110100000 \]

\[ 2^4 = 16 \]
\[ + 2^6 = 64 \]
\[ + 2^7 = 128 \]
\[ + 2^8 = 256 \]
\[ + 2^9 = 512 \]
\[ + 2^{10} = 1024 \]
\[ 2000_{10} \]
Some Bit Tricks

- You’ll need to get used to working in binary.
- Here are some helpful hints

1. Memorize the first 10 powers of 2

<table>
<thead>
<tr>
<th>(2^0)</th>
<th>(2^1)</th>
<th>(2^2)</th>
<th>(2^3)</th>
<th>(2^4)</th>
<th>(2^5)</th>
<th>(2^6)</th>
<th>(2^7)</th>
<th>(2^8)</th>
<th>(2^9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
</tr>
</tbody>
</table>
More Tricks with Bits

- You’ll need to get used to working in binary.
- Here are some helpful hints

2. Memorize the prefixes for powers of 2 that are multiples of 10

\[
\begin{align*}
2^{10} &= \text{Kilo} \quad (1024) \\
2^{20} &= \text{Mega} \quad (1024 \times 1024) \\
2^{30} &= \text{Giga} \quad (1024 \times 1024 \times 1024) \\
2^{40} &= \text{Tera} \quad (1024 \times 1024 \times 1024 \times 1024) \\
2^{50} &= \text{Peta} \quad (1024 \times 1024 \times 1024 \times 1024 \times 1024) \\
2^{60} &= \text{Exa} \quad (1024 \times 1024 \times 1024 \times 1024 \times 1024 \times 1024)
\end{align*}
\]
Even More Tricks with Bits

- You’ll need to get used to working in binary.
- Here are some helpful hints

3. When you convert a binary number to decimal, first break it down into clusters of 10 bits.
4. Then compute the value of the leftmost remaining bits (1) find the appropriate prefix (GIGA) (Often this is sufficient)
5. Compute the value of and add in each remaining 10-bit cluster
Other Helpful Clusters

Oftentimes we will find it convenient to cluster groups of bits together for a more compact representation. The clustering of 3 bits is called Octal. Octal is not that common today.

\[ v = \sum_{i=0}^{n-1} 8^i d_i \]

<table>
<thead>
<tr>
<th>Octal - base 8</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>2</td>
</tr>
<tr>
<td>011</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>110</td>
<td>6</td>
</tr>
<tr>
<td>111</td>
<td>7</td>
</tr>
</tbody>
</table>

- 3 \cdot 8^2 = 1536
- 2 \cdot 8^1 = 16
- 0 \cdot 8^0 = 0

\[ 0 + 3 \cdot 8^3 + 2 \cdot 8^2 + 7 \cdot 8^1 + 11 \cdot 8^0 = 2000_{10} \]
One Last Clustering

Clusters of 4 bits are used most frequently. This representation is called **hexadecimal**. The hexadecimal digits include 0-9, and A-F, and each digit position represents a power of 16.

\[ v = \sum_{i=0}^{n-1} 16^i d_i \]

0x7d0

Hexadecimal - base 16

| 0000 - 0 | 1000 - 8 |
| 0001 - 1 | 1001 - 9 |
| 0010 - 2 | 1010 - a |
| 0011 - 3 | 1011 - b |
| 0100 - 4 | 1100 - c |
| 0101 - 5 | 1101 - d |
| 0110 - 6 | 1110 - e |
| 0111 - 7 | 1111 - f |

Hexadecimal notation: 01111110100000

\[ 0*16^0 = 0 \]
\[ + 13*16^1 = 208 \]
\[ + 7*16^2 = 1792 \]
\[ 2000_{10} \]
Signed-Number Representations

There are also schemes for representing signed integers with bits. One obvious method is to encode the sign of the integer using one bit. Conventionally, the most significant bit is used for the sign. This encoding for signed integers is called the SIGNED MAGNITUDE representation.

\[ v = -1^s \sum_{i=0}^{n-2} 2^i b_i \]

\[ \begin{array}{cccccccc}
S & 2^9 & 2^8 & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
\end{array} \]

2000  -2000

Even though this approach seems straightforward, it is not used that frequently in practice (with one important exception).
2's Complement Integers

The 2’s complement representation for signed integers is the most commonly used signed-integer representation. It is a simple modification of unsigned integers where the most significant bit is considered negative.

8-bit 2’s complement example:

\[ v = -2^{n-1}b_{n-1} + \sum_{i=0}^{n-2} 2^{i}b_{i} \]

\[ 11010110 = -2^7 + 2^6 + 2^4 + 2^2 + 2^1 \]
\[ = -128 + 64 + 16 + 4 + 2 = -42 \]
Why 2’s Complement?

If we use a two’s complement representation for signed integers, the same binary addition mod $2^n$ procedure will work for adding positive and negative numbers (don’t need separate subtraction rules). The same procedure will also handle unsigned numbers!

When using signed magnitude representations, adding a negative value really means to subtract a positive value. However, in 2’s complement, adding is adding regardless of sign. In fact, you NEVER need to subtract when you use a 2’s complement representation.

Example:

$$\begin{align*}
55_{10} &= 00110111_2 \\
+ 10_{10} &= 00001010_2 \\
65_{10} &= 01000001_2 \\
55_{10} &= 00110111_2 \\
-10_{10} &= 11110110_2 \\
45_{10} &= 100101101_2
\end{align*}$$
2’s Complement Tricks

- **Negation** – changing the sign of a number
  - First complement every bit (i.e. $1 \rightarrow 0$, $0 \rightarrow 1$)
  - Add 1
  
  Example: $20 = 00010100$, $-20 = 11101011 + 1 = 11101100$

- **Sign-Extension** – aligning different sized 2’s complement integers

  16-bit version of $42 = 0000\ 0000\ 0010\ 1010$

  8-bit version of $-2 = \underbrace{1\ 1\ 1\ 1\ 1\ 1\ 1\ 1}$ $\uparrow$

  $1\ 1\ 1\ 1\ 1\ 1\ 1\ 0$
CLASS EXERCISE

10’s-complement Arithmetic
(You’ll never need to borrow again)

Step 1) Write down two 3-digit numbers that you want to subtract

Step 2) Form the 9’s-complement of each digit in the second number (the subtrahend)

Step 3) Add 1 to it (the subtrahend)

Step 4) Add this number to the first

Step 5) If your result was less than 1000,
form the 9’s complement again and add 1
and remember your result is negative
else
subtract 1000

What did you get? Why weren’t you taught to subtract this way?

Helpful Table of the 9’s complement for each digit:

<table>
<thead>
<tr>
<th>Digit</th>
<th>9’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>
Summary

1) Selecting the encoding of information has important implications on how this information can be processed, and how much space it requires.

2) Computer arithmetic is constrained by finite representations, this has advantages (it allows for complement arithmetic) and disadvantages (it allows for overflows, numbers too big or small to be represented).

3) Bit patterns can be interpreted in an endless number of ways, however important standards do exist
   - Two’s complement