This Week’s Projects

- The topic this week: how information is stored inside a computer
- The kinds of things stored in a computer’s memory include:
  - numbers
  - text
  - drawings
  - photographs and other images
  - music and other audio
- Obviously a very big topic
  - we’ll focus on text
  - even more specifically, on strings of letters
    (and not worry about formatting)

Bits

- Computer memories are binary devices
- The smallest unit of memory is called a bit (for “binary digit”)
- Anything that has two states can potentially be used to store or transmit one bit of data
- Mechanical devices
  - beads on an abacus (up/down)
  - relay switches (open/closed)
  - paper tape or punch card (hole/not)

Magnetic devices
- core memories (clockwise/counter)
- magnetic tape

Electronic devices
- integrated circuits (0v / +5v)
- frequencies (low, high) in radio, telephone, ...

Optical devices
- use frequency or intensity of light
Bits

- We'll let computer engineers figure out the physical storage of bits.
- As computer scientists, all we care about is that the memory device or communication channel distinguishes between two states.
  - By convention, the states are labeled 0 and 1.
- Which state is 0 and which is 1 is arbitrary.
  - 0 = 0v, 1 = 5v.
  - 0 = off, 1 = on.
  - 0 = no interference, 1 = interference.
- We just have to be consistent...

Encoding

- To store information in a computer's memory, we have to encode it.
- An encoding is a pattern of 1s and 0s.
- Terminology: the pattern is a representation of the real-world object.

Data Representation

- A single bit can represent any type of data that has only two values.
  - true or false (e.g., answers on a test)
    - T = 1
    - F = 0
  - yes or no (e.g., votes on a referendum)
    - Y = 1
    - N = 0
  - male or female (e.g., gender on a survey)
    - F = 1
    - M = 0
- To represent a piece of data that can have more than two values, we need to use a pattern of two or more bits.

An Important Formula

- A set of \( k \) bits can represent up to \( 2^k \) different items.
  - 1 bit: 0, 1
  - 2 bits: 00, 01, 10, 11
  - 3 bits: 000, 001, 010, 011, 100, 101, 110, 111
- Example: the ASCII code (commonly used for text files) is a 7-bit code.
  - With 7 bits, there are \( 2^7 = 128 \) distinct combinations of 0s and 1s.
  - Sufficient to represent upper and lower case letters, digits, punctuation.
An Important Formula

- If a set has $n$ items, the number of bits required to represent an element of the set is $k = \lceil \log_2 n \rceil$.
- Each item in the set can then be assigned a unique pattern of $k$ bits.

Examples:
- 4 nucleotides in DNA (A, C, G, or T): 2 bits per letter
- A choice of 6 colors: 3 bits per color (two patterns unused)
- A lower case letter: 5 bits (6 patterns unused)

$k = \lceil \log_2 n \rceil$

Aside: Hexadecimal and Binary

- The binary representation for a string can be hard to read.
- It's a little easier to deal with if the codes are shown in hexadecimal (base 16):

```
01001001 00100111 01101101 00100000 01100001 01100110 01110010
01100001 01101001 01100100 00100000 01101111 01100110 00100000
01100011 01101111 01110111 01110011 00101110
```

The ASCII table from Wikipedia shows codes in binary, octal (base 8), and hexadecimal (base 16) as well as decimal.

Groups of Bits

- A byte is a collection of 8 bits.
- Before the 1970s the number of bits in a byte varied from system to system.
- By the 1980s the term had become standard.
- A central processing unit (CPU) operates on several bytes at a time.
- A word is a collection of two or more bytes.
- Typical word sizes are 32 bits (4 bytes) and 64 bits (8 bytes).

Text Compression

- For very large strings we can save a lot of space by compressing the text.
- Today's topic (and the first project in the lab):
  - An alternative scheme for encoding letters.
  - Use shorter codes for more common letters.
- There are other algorithms for compressing files (including photos, images, and other types of data) but we'll focus on a simple plan for text only.
Sequence Databases

- To illustrate alternate codes we’ll use some examples from bioinformatics
- Recall from the slides on genetic algorithms that many important molecules in biochemistry are **polymers**
  - these are long chains of smaller molecules
  - DNA or RNA: chain of nucleotides
  - protein: chain of amino acids
- We can represent a polymer by using one letter for each element in the chain
  - a DNA sequence (e.g. a chromosome) is a string made of A, T, C, and G
  - a protein sequence is a string made from an alphabet of 20 letters (A, C, D, E, ...)

---

More Compact Codes

- For most text ASCII is a good choice
  - wide variety of letters and symbols
  - easily stored in computer memory (one letter = one byte)
- For sequence databases ASCII can be very inefficient
- Example:
  - DNA sequences have only A, C, G, and T
  - a special-purpose code that uses just 2 bits for each letter would require 1/4 as much memory
  - e.g. 575,000 bytes instead of 4,600,000 for the E. coli genome

### Nucleotide Code

<table>
<thead>
<tr>
<th>Nucleotide</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>00</td>
</tr>
<tr>
<td>C</td>
<td>01</td>
</tr>
<tr>
<td>G</td>
<td>10</td>
</tr>
<tr>
<td>T</td>
<td>11</td>
</tr>
</tbody>
</table>

---

Variable Length Codes

- Another way to save space is to create a code based on **letter frequencies**
- A **variable-length code** uses fewer bits for more common letters
- To construct a variable length code we need to know how often each letter occurs in the data
  - this table of letters and their frequencies was made by scanning protein sequences in a database of eukaryotes (plants and animals)

### Frequency Table

<table>
<thead>
<tr>
<th>Letter</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.069</td>
</tr>
<tr>
<td>C</td>
<td>0.021</td>
</tr>
<tr>
<td>D</td>
<td>0.050</td>
</tr>
<tr>
<td>E</td>
<td>0.069</td>
</tr>
<tr>
<td>F</td>
<td>0.038</td>
</tr>
<tr>
<td>G</td>
<td>0.063</td>
</tr>
<tr>
<td>H</td>
<td>0.015</td>
</tr>
<tr>
<td>K</td>
<td>0.023</td>
</tr>
<tr>
<td>L</td>
<td>0.096</td>
</tr>
<tr>
<td>M</td>
<td>0.041</td>
</tr>
<tr>
<td>N</td>
<td>0.056</td>
</tr>
<tr>
<td>P</td>
<td>0.059</td>
</tr>
<tr>
<td>Q</td>
<td>0.047</td>
</tr>
<tr>
<td>R</td>
<td>0.056</td>
</tr>
<tr>
<td>S</td>
<td>0.083</td>
</tr>
<tr>
<td>T</td>
<td>0.054</td>
</tr>
<tr>
<td>V</td>
<td>0.060</td>
</tr>
<tr>
<td>W</td>
<td>0.011</td>
</tr>
<tr>
<td>Y</td>
<td>0.031</td>
</tr>
</tbody>
</table>

---

*poly•mer* [ˈpæləmər]  
*noun* Chemistry  
a substance that has a molecular structure consisting chiefly or entirely of a large number of similar units bonded together; e.g., many synthetic organic materials used as plastics and resins.
Variable Length Codes

- A code based on these letter frequencies:

<table>
<thead>
<tr>
<th>Letter</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.069</td>
</tr>
<tr>
<td>B</td>
<td>0.021</td>
</tr>
<tr>
<td>C</td>
<td>0.050</td>
</tr>
<tr>
<td>D</td>
<td>0.069</td>
</tr>
<tr>
<td>E</td>
<td>0.058</td>
</tr>
<tr>
<td>F</td>
<td>0.038</td>
</tr>
<tr>
<td>G</td>
<td>0.063</td>
</tr>
<tr>
<td>H</td>
<td>0.025</td>
</tr>
<tr>
<td>I</td>
<td>0.048</td>
</tr>
<tr>
<td>J</td>
<td>0.059</td>
</tr>
<tr>
<td>K</td>
<td>0.056</td>
</tr>
<tr>
<td>L</td>
<td>0.094</td>
</tr>
<tr>
<td>M</td>
<td>0.023</td>
</tr>
<tr>
<td>N</td>
<td>0.041</td>
</tr>
<tr>
<td>O</td>
<td>0.058</td>
</tr>
<tr>
<td>P</td>
<td>0.047</td>
</tr>
<tr>
<td>Q</td>
<td>0.056</td>
</tr>
<tr>
<td>R</td>
<td>0.094</td>
</tr>
<tr>
<td>S</td>
<td>0.083</td>
</tr>
<tr>
<td>T</td>
<td>0.054</td>
</tr>
<tr>
<td>U</td>
<td>0.054</td>
</tr>
<tr>
<td>V</td>
<td>0.060</td>
</tr>
<tr>
<td>W</td>
<td>0.011</td>
</tr>
<tr>
<td>X</td>
<td>0.031</td>
</tr>
<tr>
<td>Y</td>
<td>0.031</td>
</tr>
<tr>
<td>Z</td>
<td>0.031</td>
</tr>
</tbody>
</table>

Note the most common letters are encoded with 4 bits, the least common with 6 bits.

Examples of sequences represented with this new code:

- MSLTK → 00100 1110 1111 0100 0111
- NWQEEL → 11011 101010 0000 1100 1100 1111

When these codes are stored in memory the bits are all pressed together and then divided into bytes.

- MSLTK → 00100111 01111010 00111xxx

Efficiency

- Does a variable-length code save any space?
  - It depends on whether the text being stored has the same distribution of letters used to generate the code.

  Example:
  - Suppose a strange type of RNA is expected to have 92.5% A's
  - The other three letters are all expected to occur 2.5% of the time
  - A variable length code based on these frequencies:
    - A → 1
    - C → 011
    - G → 00
    - T → 010
    (we’ll see how this code was generated later in this lecture)

- For a sequence with 10,000 bases that does in fact have 92.5% A's:
  - (9250 × 1) + (250 × 3) + (250 × 2) + (250 × 3) = 11,250 bits
    - A: 1
    - C: 011
    - G: 00
    - T: 010
    - Compare to 2 × 10,000 = 20,000 bits for the fixed-length code.
    - Compare to 8 × 10,000 = 80,000 bits for ASCII text file.

- But what if the text has a more typical distribution?
  - E. coli genes:
    - A = 24.2%, C = 24.5%, G = 27.3%, T = 24.0%
    - A set of 10,000 letters taken from this file would be encoded with
      - (2420 × 1) + (2450 × 3) + (2730 × 2) + (2400 × 3) = 20,310 bits
    - So using the wrong frequency might be worse than using a fixed-length code (but still better than 8-bit ASCII).
**Huffman Tree**

- The codes on the previous slide were defined by a **Huffman tree**
- A Huffman tree is a type of **binary tree**
  - circles represent *nodes*
  - connections between nodes are labeled with bits
  - the *root* is at the top of the diagram (no nodes above it)
  - *leaves* are at the bottom (no nodes below them)
  - there is one leaf for each letter in the alphabet
- The *path* from the root to a leaf defines the code for that letter

**Encoding a String of Letters**

- To encode a string just concatenate the codes of each letter
- Examples:
  - **ATG** ➔ 1
  - **GATTACA** ➔ 00

**Decoding a Sequence of Bits**

- Finding the string of letters defined by a bit sequence is known as **decoding**
- To decode a bit sequence:
  - start at the root of the tree
  - follow the path indicated by successive bits
  - when you reach a leaf write down the letter
  - if there are bits left repeat from step 1
- Examples:
  - **0101011** ➔ TAC
  - **0010011** ➔ GAGA

**Generating a Huffman Tree**

- There is a simple and elegant algorithm for generating a Huffman tree
- Start by making a list that contains a leaf node for each letter
  - include the letter's frequency with the node:
  - sort the nodes so the least frequent letters are at the front of the list:
Generating a Huffman Tree (cont’d)

- Repeat until there is only one node left in the list:
  - remove the first two items (call them $n1$ and $n2$)
  - make a new node (call it $n$) that will be an interior tree node:
    - $n1$ and $n2$ will be the children of $n$
    - the frequency of $n$ is the sum of frequencies of $n1$ and $n2$
    - label the connection to $n1$ with 0 and the connection to $n2$ with 1
  - insert $n$ back into the list (keeping the list sorted by frequency)

In-Class Project

- Let’s do one as a group, using the data from the strange RNA
- Here’s the initial list of leaf nodes:

Disadvantages

- It’s difficult to access letters in the middle of the text encoded with a variable length code
- Example: suppose we have a string like 0101011010
  - how do we know where the third letter starts?
  - using our example DNA code this string represents “TACT”
  - with ASCII it’s easy -- letter i starts 8 * i bits into the string
  - with a Huffman code we have to decode the first i-1 letters -- the C is the 011 following 010 and 1
- Another drawback: we have to know (or assume) the letter frequency before we can encode the text
  - if we use the wrong frequency we don’t save any space
**Review of Huffman Trees**

- Some things to know about Huffman trees:
  - Frequent letters appear near the root
  - Infrequent letters are further from the root
  - Each letter has a unique path, so each letter has its own code
- Given a tree diagram, you should be able to
  - Encode a string (write the sequence of bits for the string)
  - Decode a bit sequence (determine which letters the sequence represents)

**Implementation**

- The program to use for the Huffman tree projects on this week's lab is `huffman.rb`
- Has methods for making a tree and encoding and decoding strings
- You will also need files that define letter frequencies
  - The file named `aafreq.txt` has the frequencies of the 20 amino acid letters
  - Another file named `ntx.txt` has the “extreme DNA” frequencies

**Implementation**

- The methods you will use for the lab are in a module named Huffman
- After you load the program, you can call these methods
  - The word Huffman needs to be included with the method name

```ruby
>> load "huffman.rb" Note: don’t forget to type the include statement -- see the lab for more info
>> include Huffman

>> freq = Huffman.readFrequencies("ntx.txt")
=> {"A"=>0.925, "C"=>0.025, "G"=>0.025, "T"=>0.025}

>> freq["A"]
=> 0.925
>> freq["T"]
=> 0.025
```

A call to `readFrequencies` creates an associative array (“hash”) with the frequency of each letter

**Implementation**

- A class named `Node` captures all the information the program needs to represent a single node in a Huffman tree

```ruby
>> t1 = Node.new("T", 0.025)
=> T: 0.025
>> t2 = Node.new("C", 0.025)
=> C: 0.025

>> Node.combine(t2,t1)
=> ( 0.050 C: 0.025 T: 0.025 )

Node.combine is a constructor that makes a new node from two existing nodes
```
Priority Queue

- The key to this program is the data structure used to hold the tree nodes
  - initialize with leaf nodes for each letter
  - as the program runs two nodes are removed and replaced by a new interior node
- The data structure is an array that is always sorted
  - removing two items always removes the two lowest frequency nodes
  - inserting an item always places it in the correct location so the list remains sorted
- The priority queue introduced in the slides on ELIZA will work here, too
  - queue here means "line" -- first in, first out
  - high priority items -- in this case low frequency nodes -- cut to the front of the line

Main Loop

- Now that we have a PriorityQueue class the main loop of the Huffman tree algorithm is trivial:

```
def Huffman.initQueue(a)
  q = PriorityQueue.new
  a.each do |x,f|
    node = Node.new(x,f)
    q.insert(node)
  end
  return q
end
```

```
T 0.025
C 0.025
G 0.025
A 0.925
```

Nodes with the lowest frequency are at the front of the queue

File Compression in Practice

- File compression applications are very common
  - Stuffit, others for Mac and PC
  - zip, compress for Unix (including Linux and OS/X)
- File compression also works on music, images, and many other types of data
  - jpeg, gif, and other image formats are compressed from original image data
- Example: the compress program for Unix systems

```
% ls -l NC_000913.gbs
-rw-r--r-- 1 conery 5998130 May 10 12:33 NC_000913.gbs
% compress NC_000913.gbs
% ls -l NC_000913.gbs.Z
-rw-r--r-- 1 conery 1669141 May 10 12:33 NC_000913.gbs.Z
```
Lempel-Zev Compression Algorithm

- The `compress` program uses a method known as Lempel-Zev
- This algorithm discovers common patterns in the text as it works its way through the document -- no need to define letter frequencies
- Example: suppose a document has the string
  hello, hello, I’m in a place called vertigo
- The second “hello” can be replaced with a “pointer” to the first one:
  hello, [*,7]I’m ...
- the 7 here means “use 7 letters from the place pointed to by *”
- Ordinary English documents can be compressed by a factor of two or three
- Special documents with many repeated substrings can be compressed much further

Some Examples

- A research paper (mostly English, with lots of markup symbols)
  - uncompressed: 40,773 bytes
  - compressed: 17,991
  - reduced version is 44.1% the size of the original
- E.coli feature table (English text, with very regular structure and lots of repeated phrases):
  - uncompressed: 5,998,130 bytes
  - compressed: 1,669,141
  - 27.8%
- PDF document (mostly binary data)
  - uncompressed: 5,417,058 bytes
  - compressed: 5,060,755
  - 93.4% -- very little reduction in size

Compressing DNA

- The table below shows results from random strings of RNA made by a program that generated artificial strings with 92.5% A and 2.5% C, G, and T
  - with 90% A's there should be many long runs of A's
  - we can expect lots of opportunities for LZ to discover repeated substrings

<table>
<thead>
<tr>
<th>Length</th>
<th>Uncompressed</th>
<th>2-bit Code</th>
<th>Huffman</th>
<th>LZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>1,001</td>
<td>250</td>
<td>141</td>
<td>154</td>
</tr>
<tr>
<td>10,000</td>
<td>10,001</td>
<td>2,500</td>
<td>1,406</td>
<td>985</td>
</tr>
<tr>
<td>100,000</td>
<td>100,001</td>
<td>25,000</td>
<td>14,063</td>
<td>8,156</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1,000,001</td>
<td>250,000</td>
<td>140,625</td>
<td>75,439</td>
</tr>
</tbody>
</table>

Two Final Notes

- **Encoding** is not the same as **encryption**
- In everyday English a “code” is used to protect private or sensitive information
- In math and computer science a code is simply a pattern of bits
- If we want the pattern to hide information we use an encryption algorithm to scramble the code

Programmers also use the word “code” to mean “program”, as in “Download the Ruby code from ...”
Two Final Notes

- Data compression for images or music often throws away information
  - e.g. MP3’s are “good enough” for ring tones or iPods
  - JPEG files are OK for photos on web pages
  - users are often willing to sacrifice music or image quality to save space

- The text compression algorithms described here allow us to recover all the information in the original document -- no data is lost

Review

- The main topics for today:
  - **Encoding** translates symbols (letters, digits, bases, ...) into sequences of bits
  - **Decoding** recovers symbols from bit sequences
  - ASCII codes (8 bits per character) are the default choice for text files
  - text can be **compressed** by using alternate encodings
  - special-purpose codes can be designed for an application (e.g. 2-bit code for DNA)
  - **Variable-length codes** are based on **letter frequencies**
  - the **Huffman tree** algorithm can be used to generate variable-length codes

- You should be able to
  - encode or decode a string using ASCII
  - encode or decode a string given a drawing of a Huffman tree
  - create a Huffman tree for a small alphabet given a table of letter frequencies (you’ll get some experience with this algorithm in this week’s lab)