Data Compression

Review: data representation
Compressing text
  letter frequency
  Huffman codes
Reading: CSO 1.8

Aside: Hexadecimal and Binary

- The binary representation for a string can be hard to read
  - this is the binary pattern for an eleven-letter message:
    01001001 00100000 01101100 01101111 01100110 01100101 00100000
    01010010 01110101 01100010 01111001
  - It’s a little easier to deal with if the codes are shown in hexadecimal (base 16):
    49 20 6C 6F 76 65 20 52 75 62 79
  - The table at Wikipedia shows both formats:

<table>
<thead>
<tr>
<th>Binary</th>
<th>Oct</th>
<th>Dec</th>
<th>Hex</th>
<th>Glyph</th>
</tr>
</thead>
<tbody>
<tr>
<td>01001001</td>
<td>040</td>
<td>32</td>
<td>20</td>
<td>SF</td>
</tr>
<tr>
<td>01000101</td>
<td>141</td>
<td>69</td>
<td>45</td>
<td>A</td>
</tr>
<tr>
<td>01001100</td>
<td>142</td>
<td>70</td>
<td>46</td>
<td>B</td>
</tr>
<tr>
<td>01001101</td>
<td>143</td>
<td>71</td>
<td>47</td>
<td>C</td>
</tr>
<tr>
<td>01010000</td>
<td>144</td>
<td>72</td>
<td>48</td>
<td>D</td>
</tr>
<tr>
<td>01010001</td>
<td>145</td>
<td>73</td>
<td>49</td>
<td>E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary</th>
<th>Oct</th>
<th>Dec</th>
<th>Hex</th>
<th>Glyph</th>
</tr>
</thead>
<tbody>
<tr>
<td>01001001</td>
<td>040</td>
<td>32</td>
<td>20</td>
<td>SF</td>
</tr>
<tr>
<td>01000101</td>
<td>141</td>
<td>69</td>
<td>45</td>
<td>A</td>
</tr>
<tr>
<td>01001100</td>
<td>142</td>
<td>70</td>
<td>46</td>
<td>B</td>
</tr>
<tr>
<td>01001101</td>
<td>143</td>
<td>71</td>
<td>47</td>
<td>C</td>
</tr>
<tr>
<td>01010000</td>
<td>144</td>
<td>72</td>
<td>48</td>
<td>D</td>
</tr>
<tr>
<td>01010001</td>
<td>145</td>
<td>73</td>
<td>49</td>
<td>E</td>
</tr>
</tbody>
</table>

Aside: Hexadecimal and Binary

- There is a simple trick for converting binary to hexadecimal and vice versa
  - Binary → Hex
    - make groups of 4 bits (starting on the right)
    - convert each group of 4 using the table at right
    - example: 11011100 → 6C
  - Hex → Binary
    - use the table to convert each individual digit
    - example: 1A → 00011010
Aside: Hexadecimal and Binary

You will often see binary numbers written in hexadecimal:
- software keys, checksums (used for security and for error correction), and many other items are the result of algorithms applied to binary bit patterns.
- the result of the algorithm is shown in hex.

Representing Other Data

- An important formula introduced earlier specifies the number of bits required to represent a member of a set of $n$ items:
  $$k = \lceil \log_2 n \rceil$$
- each item in the set can be assigned a unique pattern of $k$ bits.
- Examples:
  - a choice of 6 colors: 3 bits per color (two patterns unused)
  - a lower case letter: 5 bits (6 patterns unused)

Sequence Databases

- To illustrate alternate codes we’ll use some examples from bioinformatics.
- 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, ...).

Sequence Databases

- There are many sequence databases on the internet:
  - the National Center for Biotechnology Information (NCBI) is a branch of the National Library of Medicine.
  - the latest “RefSeq” collection has over 5 million sequences from 5,000 organisms.
- The data can be downloaded as a set of text files.
- Sequences (and other data) are encoded as ASCII strings.

```
% mysql -h teleost
mysql> use zfish
mysql> select accession, aaseq from genes limit 1;
+-----------+---------------------------------------------+
<table>
<thead>
<tr>
<th>accession</th>
<th>aaseq</th>
</tr>
</thead>
<tbody>
<tr>
<td>NW_633960</td>
<td>MQHLKITPSQGAFGLMVGLTVLRLEFAPPPRCGVF...</td>
</tr>
</tbody>
</table>
```
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

<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>

More Compact Codes

- A special code for protein sequences would need 5 bits per letter
  - there are 20 amino acid letters
  - $2^4 = 16$, so 4 bits don’t provide enough combinations
  - $2^5 = 32$ combinations
    
    $$ k = \lceil \log_2 20 \rceil = \lceil 4.3219 \rceil = 5 $$

  - A = 00000, C = 00001, D = 00010, etc

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 local database of eukaryotes (plants and animals)

<table>
<thead>
<tr>
<th>Letter</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.069</td>
</tr>
<tr>
<td>G</td>
<td>0.063</td>
</tr>
<tr>
<td>M</td>
<td>0.023</td>
</tr>
<tr>
<td>S</td>
<td>0.083</td>
</tr>
<tr>
<td>C</td>
<td>0.021</td>
</tr>
<tr>
<td>H</td>
<td>0.045</td>
</tr>
<tr>
<td>N</td>
<td>0.041</td>
</tr>
<tr>
<td>T</td>
<td>0.054</td>
</tr>
<tr>
<td>D</td>
<td>0.050</td>
</tr>
<tr>
<td>I</td>
<td>0.048</td>
</tr>
<tr>
<td>P</td>
<td>0.059</td>
</tr>
<tr>
<td>V</td>
<td>0.060</td>
</tr>
<tr>
<td>E</td>
<td>0.069</td>
</tr>
<tr>
<td>K</td>
<td>0.059</td>
</tr>
<tr>
<td>Q</td>
<td>0.047</td>
</tr>
<tr>
<td>W</td>
<td>0.011</td>
</tr>
<tr>
<td>F</td>
<td>0.038</td>
</tr>
<tr>
<td>L</td>
<td>0.094</td>
</tr>
<tr>
<td>R</td>
<td>0.056</td>
</tr>
<tr>
<td>Y</td>
<td>0.031</td>
</tr>
</tbody>
</table>

Variable Length Codes

- A code based on these letter frequencies:

<table>
<thead>
<tr>
<th>Letter</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>1001</td>
</tr>
<tr>
<td>D</td>
<td>0011</td>
</tr>
<tr>
<td>E</td>
<td>0011</td>
</tr>
<tr>
<td>F</td>
<td>1101</td>
</tr>
<tr>
<td>H</td>
<td>0011</td>
</tr>
<tr>
<td>I</td>
<td>0011</td>
</tr>
<tr>
<td>K</td>
<td>1111</td>
</tr>
<tr>
<td>L</td>
<td>1111</td>
</tr>
<tr>
<td>M</td>
<td>1010</td>
</tr>
<tr>
<td>N</td>
<td>1101</td>
</tr>
<tr>
<td>P</td>
<td>0110</td>
</tr>
<tr>
<td>Q</td>
<td>0000</td>
</tr>
<tr>
<td>R</td>
<td>0101</td>
</tr>
<tr>
<td>S</td>
<td>1110</td>
</tr>
<tr>
<td>T</td>
<td>0100</td>
</tr>
<tr>
<td>V</td>
<td>1000</td>
</tr>
<tr>
<td>W</td>
<td>1100</td>
</tr>
</tbody>
</table>

- least common
- most common
Variable Length Codes

- A code based on these letter frequencies:

<table>
<thead>
<tr>
<th>Letter</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1011</td>
</tr>
<tr>
<td>G</td>
<td>1001</td>
</tr>
<tr>
<td>M</td>
<td>00100</td>
</tr>
<tr>
<td>S</td>
<td>1110</td>
</tr>
<tr>
<td>C</td>
<td>101011</td>
</tr>
<tr>
<td>H</td>
<td>00101</td>
</tr>
<tr>
<td>N</td>
<td>11011</td>
</tr>
<tr>
<td>T</td>
<td>0100</td>
</tr>
<tr>
<td>D</td>
<td>0011</td>
</tr>
<tr>
<td>I</td>
<td>0001</td>
</tr>
<tr>
<td>P</td>
<td>0110</td>
</tr>
<tr>
<td>V</td>
<td>1000</td>
</tr>
<tr>
<td>E</td>
<td>1100</td>
</tr>
<tr>
<td>K</td>
<td>0111</td>
</tr>
<tr>
<td>Q</td>
<td>0000</td>
</tr>
<tr>
<td>W</td>
<td>101010</td>
</tr>
<tr>
<td>F</td>
<td>11010</td>
</tr>
<tr>
<td>L</td>
<td>1111</td>
</tr>
<tr>
<td>R</td>
<td>0101</td>
</tr>
<tr>
<td>Y</td>
<td>10100</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 1110 1111 0111

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 DNA 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:
  \[(9250 \times 1) + (250 \times 3) = 11,250 \text{ bits}\]

  - A: 1
  - C: 011
  - G: 00
  - T: 010
  - compare to 2 \times 10,000 = 20,000 bits for the fixed-length code.
  - compare to 8 \times 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
      \[(2420 \times 1) + (2450 \times 3) + (2730 \times 2) + (2400 \times 2) = 20,310 \text{ bits}\]

- 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
  - GATTACA

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:
  - 010111 → TAC
  - 001001 → 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:
    - A: 1
    - T: 010
    - C: 011
    - G: 00
  - sort the nodes so the least frequent letters are at the front of the list:
    - W: 0.01
    - C: 0.021
    - M: 0.023
    - H: 0.025
    - Y: 0.031
    - F: 0.038
    - N: 0.041
    - S: 0.083
    - L: 0.094
- 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)

Note the list shrinks by one node on each step (remove two, insert one)
Generating a Huffman Tree (cont’d)

- The next two steps of the example:

Group Project

- Let’s do one as a group, using the data from the strange DNA
- 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 x 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

To generate the trees and codes used in this lecture I wrote a program in Ruby

- input: file containing letters and frequencies; one or more strings to encode
- output: binary digits of the encoding, plus (optionally) the code table

Example:

```bash
% more aa.txt
A 0.069
C 0.021
D 0.050
...
% huffman.rb -f aa.txt MSKT
MSKT => 00100111001110100
```

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
- One name for this type of structure is priority queue
  - queue here means “line” -- first in, first out
  - high priority items -- in this case low frequency nodes -- cut to the front of the line

PriorityQueue Class

Ruby makes it very easy to write a class for priority queue objects
  - the key idea is that priority queues are very similar to arrays

- Arrays in Ruby have two methods that do almost what we want:
  - `a.shift` removes the first element of array `a`
  - `a.insert` inserts an item at an arbitrary location in `a`

```ruby
>> a = [1,2,3,4,5]
=> [1, 2, 3, 4, 5]
>> a.shift
=> 1
>> a
=> [2, 3, 4, 5]
>> a.insert(6,0)
=> [6, 2, 3, 4, 5]
```

Extra Credit

Ruby allows us to write our own new class, named `PriorityQueue`
  - we will tell Ruby that `PriorityQueue` objects are just like `Array` objects
  - then all we have to do is write the code for a new `insert` method that only allows
    the new item to go at a location defined by its priority
  - this new method will scan the array to find the correct location so the array remains sorted

```ruby
class PriorityQueue < Array
  def insert(node)
    ... end
  end
end
```
PriorityQueue Class (cont’d)  

- The code below assumes the object being inserted (called node here) has a method name \texttt{freq} that returns the frequency
- Any object that has this method can be inserted into a PriorityQueue

```ruby
class PriorityQueue < Array
  def insert(node)
    i = 0
    while (i < self.length)
      if node.freq < self[i].freq
        break
      else
        i += 1
      end
    end
    super(i, node)  
  return self
  end
end
```

Extra Credit

- \texttt{self} means "the Array that implements this priority queue"
- \texttt{super} means "the insert method in the Array class"

Initializing the Queue

- Here is a method that initializes the queue by creating a leaf node for each letter in an array and inserting the node into the queue:

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

Main Loop

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

```ruby
f = readFrequencies(file)
pq = initQueue(f)
while pq.length > 1
  n1 = pq.shift
  n2 = pq.shift
  n = TreeNode.combine(n1,n2)
  pq.insert(n)
end
```

- Note that since PriorityQueue is a type of Array our object pq can use the length and shift methods already defined for the Array class

Extra Credit

- \texttt{initQueue(a)}
- \texttt{pq = PriorityQueue.new}
- \texttt{def initQueue(a)}
- \texttt{a.each do |x,f| node = TreeNode.new(x,f) q.insert(node) end return q end}

File Compression in Practice

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

```bash
% 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 DNA 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** (translating letters to bit sequences) is not the same as **encryption**
  - encryption is used to hide information, to prevent others from learning sensitive or secret data
  - often strings are encrypted by generating binary numbers, but encrypted strings may just be other strings of letters
  - example: Caesar cipher (A ⇔ N, B ⇔ O, ... M ⇔ Z)
- Data compression, e.g. for images or music, often throws away information
  - e.g. MP3’s are “good enough” for ring tones or iPods
  - GIF files are OK for line drawings
  - 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 strings -- no data is lost.
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)