Huffman Codes

Background: bits, bytes, words
Fixed length code: ASCII
Variable length codes
  letter frequency
  Huffman trees
Text compression examples
Reading: NTO 52

Computer Memory: Bits

- The smallest piece of data that can be stored in a computer memory is known as a bit
  - bit = “binary digit”
  - 0 or 1
  - can represent yes/no, true/false, up/down, and other binary values
- The abbreviation for bit is a lower case “b”
- For large values, the “b” is preceded by
  - K = kilo = 10³
  - M = mega = 10⁶
  - G = giga = 10⁹
- Examples:
  - 16Kb = 16 thousand bits (e.g. for an old memory chip)
  - 10Mbps = 10 million bits per second (e.g. for a network interface)

Computer Memory: Bytes and Words

- Memory capacity is usually described in terms of bytes
  - one byte = eight bits
  - abbreviation: upper case “B”
- Examples:
  - 512MB RAM
  - 80GB hard drive
- A computer’s central processing unit (CPU) has small internal memories known as registers
  - each register holds one word
  - a “32-bit” CPU has 4 bytes per word
- newer “64-bit” CPUs have 8 bytes per word
- larger word sizes typically mean higher computing power (e.g. IBM processor chips in Apple’s G4 and G5)

ASCII

- Strings are usually stored in memory by using one byte for each character
  - each letter is represented by an 8-bit binary number
- The ASCII code is the most common mapping from letters to numbers
- Examples:
  - hello → 01101000 01100101 01101100 01101100 01101111
  - ATG → 01000001 01010100 01000111

<table>
<thead>
<tr>
<th>00100000</th>
<th>00110000</th>
<th>0</th>
<th>01000000</th>
<th>#</th>
<th>01010000</th>
<th>P</th>
<th>01100000</th>
<th>~</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100001</td>
<td>1</td>
<td>00110001</td>
<td>1</td>
<td>01000001</td>
<td>A</td>
<td>01010001</td>
<td>Q</td>
<td>01100001</td>
</tr>
<tr>
<td>00100010</td>
<td>&quot;</td>
<td>00110010</td>
<td>2</td>
<td>01000010</td>
<td>B</td>
<td>01010010</td>
<td>R</td>
<td>01100010</td>
</tr>
<tr>
<td>00110011</td>
<td>#</td>
<td>00110011</td>
<td>3</td>
<td>01000011</td>
<td>C</td>
<td>01010010</td>
<td>S</td>
<td>01100010</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
A Trick

- Long strings of bits are hard to read, so codes are often written in hexadecimal (base 16).
- To convert an 8-digit binary number to hexadecimal break it into two 4-bit pieces and translate each piece.
- Examples:
  
<table>
<thead>
<tr>
<th>20</th>
<th>30</th>
<th>0</th>
<th>40</th>
<th>Θ</th>
<th>50</th>
<th>P</th>
<th>60</th>
<th>`</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>!</td>
<td>31</td>
<td>1</td>
<td>41</td>
<td>A</td>
<td>51</td>
<td>Q</td>
<td>61</td>
</tr>
<tr>
<td>22</td>
<td>`</td>
<td>32</td>
<td>2</td>
<td>42</td>
<td>B</td>
<td>52</td>
<td>R</td>
<td>62</td>
</tr>
<tr>
<td>23</td>
<td>#</td>
<td>33</td>
<td>3</td>
<td>43</td>
<td>C</td>
<td>53</td>
<td>S</td>
<td>63</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Compact Codes (cont’d)

- 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.
- The general formula for figuring out how many bits you need to represent $n$ different symbols is:
  $$b = \lceil \log_2 n \rceil$$
- For $n = 20$ letters we need:
  $$b = \lceil \log_2 20 \rceil = \lceil 4.3219 \rceil = 5$$

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).
- In many cases 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 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>

A Trick (cont’d)

- Here are the earlier two examples written in binary and hexadecimal:
  - `hello` → `01101000 01100101 01101100 01101100 01101111` → `68 65 6C 6C 6F`
  - `ATG` → `01000001 01010100 01000111` → `41 54 47`
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.
- Example: a code based on amino acid letters.
  - 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>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>0.025</td>
</tr>
<tr>
<td>I</td>
<td>0.048</td>
</tr>
<tr>
<td>K</td>
<td>0.059</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>P</td>
<td>0.058</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>

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

Examples:
- MSLTK → 00100110110110000001
- NWQEL → 1101110101000001100110011001111

Variable Length Codes (cont'd)

- 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>101011</td>
</tr>
<tr>
<td>D</td>
<td>0011</td>
</tr>
<tr>
<td>E</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>11010</td>
</tr>
<tr>
<td>G</td>
<td>1001</td>
</tr>
<tr>
<td>H</td>
<td>00101</td>
</tr>
<tr>
<td>I</td>
<td>0001</td>
</tr>
<tr>
<td>K</td>
<td>0111</td>
</tr>
<tr>
<td>L</td>
<td>0111</td>
</tr>
<tr>
<td>M</td>
<td>00100</td>
</tr>
<tr>
<td>N</td>
<td>11011</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>
</tbody>
</table>

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

Examples:
- MSLTK → 00100110110110000001
- NWQEL → 1101110101000001100110011001111

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:
    
    | Letter | Code |
    |--------|------|
    | A      | 01 |
    | C      | 011 |
    | G      | 00 |
    | T      | 010 |

  We’ll see how this code was generated later in this lecture.
Efficiency (cont’d)

- For a text with 10,000 letters that does in fact have 92.5% A’s:
  \[(9250 \times 1) + (250 \times 3) + (250 \times 2) + (250 \times 3) = 11,250 \text{ bits}\]
- 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 expected to take
      \[(2420 \times 1) + (2450 \times 3) + (2730 \times 2) + (2400 \times 2) = 20,310 \text{ bits}\]

- So using the wrong frequency might be worse than using a fixed-length code

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:
  - 0101011 → TAC
  - 0010001 → 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.

\[
\begin{array}{cccccccc}
A & C & G & O & E & F & H & M \\
.089 & .021 & .030 & .028 & .038 & .025 & .051 & .021
\end{array}
\]

- Repeat until there is only one node left in the list:
  - Remove the first two items (call them \( n_1 \) and \( n_2 \)).
  - Make an interior tree node \( n \) with \( n_1 \) and \( n_2 \) as children.
  - The frequency of \( n \) is the sum of frequencies of \( n_1 \) and \( n_2 \).
  - Label the connection to \( n_1 \) with \( 0 \) and the connection to \( n_2 \) with \( 1 \).
  - Insert \( n \) back into the list (keeping the list sorted by frequency).

Generating a Huffman Tree (cont’d)

- The next two steps of the example:

\[
\begin{array}{cccccccc}
M & H & Y & F & N & \cdots & S & L \\
.023 & .025 & .031 & .038 & .041 & \cdots & .083 & .094
\end{array}
\]

- The next two steps of the example:

\[
\begin{array}{cccccccc}
M & H & Y & F & N & \cdots & S & L \\
.023 & .025 & .031 & .038 & .041 & \cdots & .083 & .094
\end{array}
\]

Group Project

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

\[
\begin{array}{cccc}
T & C & G & A \\
.025 & .025 & .025 & .025
\end{array}
\]

- Do we end up with the tree shown earlier?
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
t• how do we know where the third letter starts?
• using our sample DNA code this string represents "TACT"
• with ASCII it's easy -- letter i starts $8 \times i$ 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
• Huffman codes are not often used in applications, but they might be used to compress a file for long-term storage or to send over the internet
• 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:
  % more aa.txt
  A 0.069
  C 0.021
  D 0.050
  ...
  % huffman -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.unshift` inserts an item at the front of `a`

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

PriorityQueue Class (cont’d)

- 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 `unshift` method
  - instead of putting the item at the front of the array, this new method will scan the array to find the correct location so the array remains sorted

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

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.unshift(node)
  end
  return q
end
```

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

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

- `self` means “the Array that implements this priority queue”
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.unshift(n)
enormal
```

- 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.

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

```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
  ```text
  hello, hello, I'm in a place called vertigo
  ```
  The second “hello” can be replaced with a “pointer” to the first one:
  ```text
  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)
  ```text
  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):
  ```text
  uncompressed: 5,998,130 bytes
  compressed: 1,669,141
  27.8%
  ```
- PDF document (mostly binary data)
  ```text
  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 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.

Review

- The main topics for today:
  - **Encoding** translates letters into sequences of bits.
  - **Decoding** recovers letters from bit sequences.
  - ASCII codes (8 bits per character) are the default choice for text files.
  - Text can be **compressed** by using alternative codings.
  - 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.