From bits to bytes to ints

- At some level everything is stored as either a zero or a one
  - A bit is a binary digit a byte is a binary term (8 bits)
  - We should be grateful we can deal with Strings rather than sequences of 0's and 1's.
  - We should be grateful we can deal with an int rather than the 32 bits that make an int

- Int values are stored as two's complement numbers with 32 bits, for 64 bits use the type long, a char is 16 bits
  - Standard in Java, different in C/C++
  - Facilitates addition/subtraction for int values
  - We don't need to worry about this, except to note:
    - `Integer.MAX_VALUE + 1 = Integer.MIN_VALUE`
    - `Math.abs(Integer.MIN_VALUE) != Infinity`

How are data stored?

- To facilitate Huffman coding we need to read/write one bit
  - Why do we need to read one bit?
  - Why do we need to write one bit?
  - When do we read 8 bits at a time? Read 32 bits at a time?

- We can't actually write one bit-at-a-time. We can't really write one char at a time either.
  - Output and input are buffered, minimize memory accesses and disk accesses
  - Why do we care about this when we talk about data structures and algorithms?
    - Where does data come from?

How do we buffer char output?

- Done for us as part of InputStream and Reader classes
  - InputStreams are for reading bytes
  - Readers are for reading char values
  - Why do we have both and how do they interact?
  - `Reader r = new InputStreamReader(System.in);`
  - Do we need to flush our buffers?

- In the past Java IO has been notoriously slow
  - Do we care about I? About O?
  - This is changing, and the java.nio classes help
    - Map a file to a region in memory in one operation

Buffer bit output

- To buffer bit output we need to store bits in a buffer
  - When the buffer is full, we write it.
  - The buffer might overflow, e.g., in process of writing 10 bits to 32-bit capacity buffer that has 29 bits in it
  - How do we access bits, add to buffer, etc.?

- We need to use bit operations
  - Mask bits – access individual bits
  - Shift bits – to the left or to the right
  - Bitwise and/or/negate bits
Representing pixels

- A pixel typically stores RGB and alpha/transparency values
  - Each RGB is a value in the range 0 to 255
  - The alpha value is also in range 0 to 255
  ```
  Pixel red = new Pixel(255,0,0,0);
  Pixel white = new Pixel(255,255,255,0);
  ```
- Typically store these values as int values, a picture is simply an array of int values
  ```
  void process(int pixel){
      int blue = pixel & 0xff;
      int green = (pixel >> 8) & 0xff;
      int red = (pixel >> 16) & 0xff;
  }
  ```

Bit masks and shifts

- Hexadecimal number: 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f
  - Note that f is 15, in binary this is 1111, one less than 10000
  - The hex number 0xff is an 8 bit number, all ones
- The bitwise & operator creates an 8 bit value, 0—255 (why)
  - 1&1 == 1, otherwise we get 0, similar to logical and
  - Similarly we have |, bitwise or

Bit operations revisited

- How do we write out all of the bits of a number
  ```
  /**
   * writes the bit representation of a int
   * to standard out
   */
  void bits(int val){
  }
  ```

Text Compression

- Input: String S
- Output: String S’
  - Shorter
  - S can be reconstructed from S’
Text Compression: Examples

“abcde” in the different formats

<table>
<thead>
<tr>
<th>Symbol</th>
<th>ASCII</th>
<th>Fixed length</th>
<th>Var. length</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>01100001</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>b</td>
<td>01100010</td>
<td>001</td>
<td>11</td>
</tr>
<tr>
<td>c</td>
<td>01100011</td>
<td>010</td>
<td>01</td>
</tr>
<tr>
<td>d</td>
<td>01100100</td>
<td>011</td>
<td>001</td>
</tr>
<tr>
<td>e</td>
<td>01100101</td>
<td>100</td>
<td>10</td>
</tr>
</tbody>
</table>

Huffman coding: go go gophers

<table>
<thead>
<tr>
<th>Symbol</th>
<th>ASCII</th>
<th>Huffman</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>103</td>
<td>1101111</td>
</tr>
<tr>
<td>o</td>
<td>011</td>
<td>000</td>
</tr>
<tr>
<td>p</td>
<td>012</td>
<td>1100011</td>
</tr>
<tr>
<td>h</td>
<td>104</td>
<td>1011001</td>
</tr>
<tr>
<td>e</td>
<td>0111</td>
<td>1001110</td>
</tr>
<tr>
<td>r</td>
<td>0121</td>
<td>1011111</td>
</tr>
<tr>
<td>s</td>
<td>0125</td>
<td>1101101</td>
</tr>
<tr>
<td>sp.</td>
<td>32</td>
<td>1000000</td>
</tr>
</tbody>
</table>

- Encoding uses tree:
  - 0 left/1 right
  - How many bits? 37!!
  - Savings? Worth it?

Huffman Coding

- D.A Huffman in early 1950’s
- Before compressing data, analyze the input stream
- Represent data using variable length codes
- Variable length codes though Prefix codes
  - Each letter is assigned a codeword
  - Codeword is for a given letter is produced by traversing the Huffman tree
  - Property: No codeword produced is the prefix of another
  - Letters appearing frequently have short codewords, while those that appear rarely have longer ones
- Huffman coding is optimal per-character coding method

Building a Huffman tree

- Begin with a forest of single-node trees (leaves)
  - Each node/tree/leaf is weighted with character count
  - Node stores two values: character and count
  - There are $n$ nodes in forest, $n$ is size of alphabet?

- Repeat until there is only one node left: root of tree
  - Remove two minimally weighted trees from forest
  - Create new tree with minimal trees as children,
    - New tree root's weight: sum of children (character ignored)

- Does this process terminate? How do we get minimal trees?
  - Remove minimal trees, hummm...
Building a tree

“A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”

Properties of Huffman coding

- Want to minimize weighted path length $L(T)$ of tree $T$
  - $L(T) = \sum_{i \in \text{Leaf}(T)} d_i w_i$
    - $w_i$ is the weight or count of each codeword $i$
    - $d_i$ is the leaf corresponding to codeword $i$
- How do we calculate character (codeword) frequencies?
- Huffman coding creates pretty full bushy trees?
  - When would it produce a “bad” tree?
- How do we produce coded compressed data from input efficiently?

Huffman Complexities

- How do we measure? Size of input file, size of alphabet
  - Which is typically bigger?
- Accumulating character counts: ______
  - How can we do this in $O(1)$ time, though not really
- Building the heap/priority queue from counts ______
  - Initializing heap guaranteed
- Building Huffman tree ______
  - Why?
- Create table of encodings from tree ______
  - Why?
- Write tree and compressed file ______

Writing code out to file

- How do we go from characters to encodings?
  - Build Huffman tree
  - Root-to-leaf path generates encoding
- Need way of writing bits out to file
  - Platform dependent?
  - Complicated to write bits and read in same ordering
- See BitInputStream and BitOutputStream classes
  - Depend on each other, bit ordering preserved
- How do we know bits come from compressed file?
  - Store a magic number
Decoding a message

Huffman coding: *go go gophers*

Huffman Tree 2

Other methods