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:
    - Infinity + 1 = -Infinity
    - Math.abs(-Infinity) > 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?

```java
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
  
  ```java
  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

  ```java
  void process(int pixel){
      int blue = pixel & 0xff;
      int green = (pixel >> 8) & 0xff;
      int red = (pixel >> 16) & 0xff;
  }
  ```
Bit masks and shifts

```c
void process(int pixel){
    int blue  = pixel & 0xff;
    int green = (pixel >> 8) & 0xff;
    int red   = (pixel >> 16) & 0xff;
}
```

- **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

```c
/**
 * 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

Enodings
ASCII: 8 bits/character
Unicode: 16 bits/character

<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>

“abcde” in the different formats
ASCII:
01100001 01100010 01100011 01100100 01100101
Fixed:
000001010011100
Var:
000110100110

Encodings
ASCII: 8 bits/character
Unicode: 16 bits/character
Huffman coding: *go go gophers*

<table>
<thead>
<tr>
<th>ASCII</th>
<th>3 bits</th>
<th>Huffman</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>103</td>
<td>1100111</td>
</tr>
<tr>
<td>o</td>
<td>111</td>
<td>1101111</td>
</tr>
<tr>
<td>p</td>
<td>112</td>
<td>1110000</td>
</tr>
<tr>
<td>h</td>
<td>104</td>
<td>1101000</td>
</tr>
<tr>
<td>e</td>
<td>101</td>
<td>1100101</td>
</tr>
<tr>
<td>r</td>
<td>114</td>
<td>1110010</td>
</tr>
<tr>
<td>s</td>
<td>115</td>
<td>1110011</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 1950s
- 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”
Building a tree

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

📝 Count occurrence of all occurring character  \( O(\ N) \)

📝 Build priority queue  \( O(\ A) \)

✔️ Build Huffman tree  \( A \ log \ A \)

✔️ Create Table of codes from tree  \( O(\ ) \)

✖️ Write Huffman tree and coded data to file  \( O(\ ) \)
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?
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

01100000100001001101
Decoding

- Read in tree data \( O(\quad) \)

- Decode bit string with tree \( O(\quad) \)
Huffman coding: *go go gophers*

- **ASCII** 3 bits Huffman

<table>
<thead>
<tr>
<th>ASCII</th>
<th>3 bits</th>
<th>Huffman</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>103</td>
<td>1100111</td>
</tr>
<tr>
<td>o</td>
<td>111</td>
<td>1101111</td>
</tr>
<tr>
<td>p</td>
<td>112</td>
<td>1100000</td>
</tr>
<tr>
<td>h</td>
<td>104</td>
<td>1101000</td>
</tr>
<tr>
<td>e</td>
<td>101</td>
<td>1100101</td>
</tr>
<tr>
<td>r</td>
<td>114</td>
<td>1110010</td>
</tr>
<tr>
<td>s</td>
<td>115</td>
<td>1110011</td>
</tr>
<tr>
<td>sp.</td>
<td>32</td>
<td>1000000</td>
</tr>
</tbody>
</table>

- **choose two smallest weights**
  - combine nodes + weights
  - Repeat
  - Priority queue?

- **Encoding uses tree:**
  - 0 left/1 right
  - How many bits?
Huffman Tree 2

- "A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS"
  - E.g. "A SIMPLE" ⇔ "10101101001000101001110011100000"

![Huffman Tree Diagram]
Other methods

- **Adaptive Huffman coding**
- **Lempel-Ziv algorithms**
  - Build the coding table on the fly while reading document
  - Coding table changes dynamically
  - Protocol between encoder and decoder so that everyone is always using the right coding scheme
  - Works well in practice (compress, gzip, etc.)
- **More complicated methods**
  - Burrows-Wheeler (bunzip2)
  - PPM statistical methods