A6 - Huffman Encoding
YEAH Hours
### Decimal - Binary - Octal - Hex - ASCII Conversion Chart

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Octal</th>
<th>Hex</th>
<th>ASCII</th>
<th>Decimal</th>
<th>Binary</th>
<th>Octal</th>
<th>Hex</th>
<th>ASCII</th>
<th>Decimal</th>
<th>Binary</th>
<th>Octal</th>
<th>Hex</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000000</td>
<td>000</td>
<td>0</td>
<td>NUL</td>
<td>32</td>
<td>00100000</td>
<td>040</td>
<td>20</td>
<td>SP</td>
<td>64</td>
<td>01000000</td>
<td>100</td>
<td>40</td>
<td>@</td>
</tr>
<tr>
<td>1</td>
<td>00000001</td>
<td>001</td>
<td>01</td>
<td>SOH</td>
<td>33</td>
<td>00100001</td>
<td>041</td>
<td>21</td>
<td>!</td>
<td>65</td>
<td>01000001</td>
<td>101</td>
<td>41</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>00000010</td>
<td>002</td>
<td>02</td>
<td>STX</td>
<td>34</td>
<td>00100010</td>
<td>042</td>
<td>22</td>
<td>&quot;</td>
<td>66</td>
<td>01000010</td>
<td>102</td>
<td>42</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>00000011</td>
<td>003</td>
<td>03</td>
<td>ETX</td>
<td>35</td>
<td>00100011</td>
<td>043</td>
<td>23</td>
<td>#</td>
<td>67</td>
<td>01000011</td>
<td>103</td>
<td>43</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>00000100</td>
<td>004</td>
<td>04</td>
<td>EOT</td>
<td>36</td>
<td>00100100</td>
<td>044</td>
<td>24</td>
<td>$</td>
<td>68</td>
<td>01000100</td>
<td>104</td>
<td>44</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td>00000101</td>
<td>005</td>
<td>05</td>
<td>ENQ</td>
<td>37</td>
<td>00100101</td>
<td>045</td>
<td>25</td>
<td>%</td>
<td>69</td>
<td>01000101</td>
<td>105</td>
<td>45</td>
<td>E</td>
</tr>
<tr>
<td>6</td>
<td>00000110</td>
<td>006</td>
<td>06</td>
<td>ACK</td>
<td>38</td>
<td>00100110</td>
<td>046</td>
<td>26</td>
<td>&amp;</td>
<td>70</td>
<td>01000110</td>
<td>106</td>
<td>46</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>00000111</td>
<td>007</td>
<td>07</td>
<td>BEL</td>
<td>39</td>
<td>00100111</td>
<td>047</td>
<td>27</td>
<td>'</td>
<td>71</td>
<td>01000111</td>
<td>107</td>
<td>47</td>
<td>G</td>
</tr>
<tr>
<td>8</td>
<td>00001000</td>
<td>010</td>
<td>08</td>
<td>BS</td>
<td>40</td>
<td>00101000</td>
<td>050</td>
<td>28</td>
<td>(</td>
<td>72</td>
<td>01010000</td>
<td>110</td>
<td>48</td>
<td>H</td>
</tr>
<tr>
<td>9</td>
<td>00001001</td>
<td>011</td>
<td>09</td>
<td>HT</td>
<td>41</td>
<td>00101001</td>
<td>051</td>
<td>29</td>
<td>)</td>
<td>73</td>
<td>01010001</td>
<td>111</td>
<td>49</td>
<td>I</td>
</tr>
<tr>
<td>10</td>
<td>00001010</td>
<td>012</td>
<td>0A</td>
<td>LF</td>
<td>42</td>
<td>00101010</td>
<td>052</td>
<td>2A</td>
<td>*</td>
<td>74</td>
<td>01010100</td>
<td>112</td>
<td>4A</td>
<td>J</td>
</tr>
<tr>
<td>11</td>
<td>00001011</td>
<td>013</td>
<td>0B</td>
<td>VT</td>
<td>43</td>
<td>00101011</td>
<td>053</td>
<td>2B</td>
<td>+</td>
<td>75</td>
<td>01010110</td>
<td>113</td>
<td>4B</td>
<td>K</td>
</tr>
<tr>
<td>12</td>
<td>00001100</td>
<td>014</td>
<td>0C</td>
<td>FF</td>
<td>44</td>
<td>00101100</td>
<td>054</td>
<td>2C</td>
<td>,</td>
<td>76</td>
<td>01011000</td>
<td>114</td>
<td>4C</td>
<td>L</td>
</tr>
<tr>
<td>13</td>
<td>00001101</td>
<td>015</td>
<td>0D</td>
<td>CR</td>
<td>45</td>
<td>00101101</td>
<td>055</td>
<td>2D</td>
<td>-</td>
<td>77</td>
<td>01011010</td>
<td>115</td>
<td>4D</td>
<td>M</td>
</tr>
<tr>
<td>14</td>
<td>00001110</td>
<td>016</td>
<td>0E</td>
<td>SO</td>
<td>46</td>
<td>00101110</td>
<td>056</td>
<td>2E</td>
<td>/</td>
<td>78</td>
<td>01011100</td>
<td>116</td>
<td>4E</td>
<td>N</td>
</tr>
<tr>
<td>15</td>
<td>00001111</td>
<td>017</td>
<td>0F</td>
<td>SI</td>
<td>47</td>
<td>00101111</td>
<td>057</td>
<td>2F</td>
<td>\</td>
<td>79</td>
<td>01011110</td>
<td>117</td>
<td>4F</td>
<td>O</td>
</tr>
<tr>
<td>16</td>
<td>00010000</td>
<td>020</td>
<td>10</td>
<td>DLE</td>
<td>48</td>
<td>00110000</td>
<td>060</td>
<td>30</td>
<td>0</td>
<td>80</td>
<td>01100000</td>
<td>120</td>
<td>60</td>
<td>p</td>
</tr>
<tr>
<td>17</td>
<td>00010001</td>
<td>021</td>
<td>11</td>
<td>DC1</td>
<td>49</td>
<td>00110001</td>
<td>061</td>
<td>31</td>
<td>1</td>
<td>81</td>
<td>01100010</td>
<td>121</td>
<td>61</td>
<td>q</td>
</tr>
<tr>
<td>18</td>
<td>00010010</td>
<td>022</td>
<td>12</td>
<td>DC2</td>
<td>50</td>
<td>00110010</td>
<td>062</td>
<td>32</td>
<td>2</td>
<td>82</td>
<td>01100100</td>
<td>122</td>
<td>62</td>
<td>r</td>
</tr>
<tr>
<td>19</td>
<td>00010011</td>
<td>023</td>
<td>13</td>
<td>DC3</td>
<td>51</td>
<td>00110011</td>
<td>063</td>
<td>33</td>
<td>3</td>
<td>83</td>
<td>01100110</td>
<td>123</td>
<td>63</td>
<td>s</td>
</tr>
<tr>
<td>20</td>
<td>00010100</td>
<td>024</td>
<td>14</td>
<td>DC4</td>
<td>52</td>
<td>00110100</td>
<td>064</td>
<td>34</td>
<td>4</td>
<td>84</td>
<td>01101000</td>
<td>124</td>
<td>64</td>
<td>t</td>
</tr>
<tr>
<td>21</td>
<td>00010101</td>
<td>025</td>
<td>15</td>
<td>NAK</td>
<td>53</td>
<td>00110101</td>
<td>065</td>
<td>35</td>
<td>5</td>
<td>85</td>
<td>01101010</td>
<td>125</td>
<td>65</td>
<td>u</td>
</tr>
<tr>
<td>22</td>
<td>00010110</td>
<td>026</td>
<td>16</td>
<td>SYN</td>
<td>54</td>
<td>00110110</td>
<td>066</td>
<td>36</td>
<td>6</td>
<td>86</td>
<td>01101100</td>
<td>126</td>
<td>66</td>
<td>v</td>
</tr>
<tr>
<td>23</td>
<td>00010111</td>
<td>027</td>
<td>17</td>
<td>ETB</td>
<td>55</td>
<td>00110111</td>
<td>067</td>
<td>37</td>
<td>7</td>
<td>87</td>
<td>01101110</td>
<td>127</td>
<td>67</td>
<td>w</td>
</tr>
<tr>
<td>24</td>
<td>00011000</td>
<td>030</td>
<td>18</td>
<td>CAN</td>
<td>56</td>
<td>00111000</td>
<td>070</td>
<td>38</td>
<td>8</td>
<td>88</td>
<td>01110000</td>
<td>130</td>
<td>78</td>
<td>x</td>
</tr>
<tr>
<td>25</td>
<td>00011001</td>
<td>031</td>
<td>19</td>
<td>EM</td>
<td>57</td>
<td>00111001</td>
<td>071</td>
<td>39</td>
<td>9</td>
<td>89</td>
<td>01110010</td>
<td>131</td>
<td>79</td>
<td>y</td>
</tr>
<tr>
<td>26</td>
<td>00011010</td>
<td>032</td>
<td>2A</td>
<td>SUB</td>
<td>58</td>
<td>00111010</td>
<td>072</td>
<td>3A</td>
<td>:</td>
<td>90</td>
<td>01110100</td>
<td>132</td>
<td>7A</td>
<td>z</td>
</tr>
<tr>
<td>27</td>
<td>00011011</td>
<td>033</td>
<td>2B</td>
<td>ESC</td>
<td>59</td>
<td>00111011</td>
<td>073</td>
<td>3B</td>
<td>;</td>
<td>91</td>
<td>01110110</td>
<td>133</td>
<td>7B</td>
<td>{</td>
</tr>
<tr>
<td>28</td>
<td>00011100</td>
<td>034</td>
<td>2C</td>
<td>FS</td>
<td>60</td>
<td>00111100</td>
<td>074</td>
<td>3C</td>
<td>&lt;</td>
<td>92</td>
<td>01111000</td>
<td>134</td>
<td>7C</td>
<td>]</td>
</tr>
<tr>
<td>29</td>
<td>00011101</td>
<td>035</td>
<td>2D</td>
<td>GS</td>
<td>61</td>
<td>00111101</td>
<td>075</td>
<td>3D</td>
<td>=</td>
<td>93</td>
<td>01111010</td>
<td>135</td>
<td>7D</td>
<td>{</td>
</tr>
<tr>
<td>30</td>
<td>00011110</td>
<td>036</td>
<td>2E</td>
<td>RS</td>
<td>62</td>
<td>00111110</td>
<td>076</td>
<td>3E</td>
<td>&gt;</td>
<td>94</td>
<td>01111100</td>
<td>136</td>
<td>7E</td>
<td>~</td>
</tr>
<tr>
<td>31</td>
<td>00011111</td>
<td>037</td>
<td>2F</td>
<td>US</td>
<td>63</td>
<td>00111111</td>
<td>077</td>
<td>3F</td>
<td>?</td>
<td>95</td>
<td>01111110</td>
<td>137</td>
<td>7F</td>
<td>DEL</td>
</tr>
</tbody>
</table>
Problem

ataata -> 01100001 01110100 01100001 01100001 01110100 01100001
\hspace{2cm} a \hspace{2cm} a \hspace{2cm} a

48 characters

Thought: Let’s represent ‘a’ with less characters!
Proposed Solution

Let’s arbitrarily represent ‘a’ with \(01\)

\[
\text{ataata} \rightarrow \begin{array}{cccc}
01 & 01110100 & 01 & 01 & 01110100 & 01 \\
\hline
a & a & a & a
\end{array}
\]

24 characters!

Why did we choose ‘a’?

How do we scale this?
Huffman encoding

Uses variable lengths for different characters to take advantage of their relative frequencies.

<table>
<thead>
<tr>
<th>Char</th>
<th>ASCII value</th>
<th>ASCII (binary)</th>
<th>Hypothetical Huffman</th>
</tr>
</thead>
<tbody>
<tr>
<td>' '</td>
<td>32</td>
<td>00100000</td>
<td>10</td>
</tr>
<tr>
<td>'a'</td>
<td>97</td>
<td>01100001</td>
<td>0001</td>
</tr>
<tr>
<td>'b'</td>
<td>98</td>
<td>01100010</td>
<td>01110100</td>
</tr>
<tr>
<td>'c'</td>
<td>99</td>
<td>01100011</td>
<td>001100</td>
</tr>
<tr>
<td>'e'</td>
<td>101</td>
<td>01100101</td>
<td>1100</td>
</tr>
<tr>
<td>'z'</td>
<td>122</td>
<td>01111010</td>
<td>00100011110</td>
</tr>
</tbody>
</table>
Huffman Tree

file.txt

bac aab b

Frequencies: { ' ':2, 'a':3, 'b':3, 'c':1, EOF:1}
Huffman compression

1. Count occurrences of each char in file
   \{ ' ': 2, 'a': 3, 'b': 3, 'c': 1, EOF: 1 \}

2a. Place chars, counts into priority queue

2b. Use PQ to create Huffman tree

3. Write logic to free the tree!

4. Traverse tree to find (char $\rightarrow$ binary) encoding map
   \{ ' ': 00, 'a': 11, 'b': 10, 'c': 010, EOF=011 \}

5. Convert to binary (For each char in file, look up binary rep in map)
   11 10 00 11 10 00 010 1 1 10 011 00
(1) Count occurrences

Map<int, int> buildFrequencyTable(istream& input)

Take as input an istream containing the file to compress, then hands back a Map associating each character in the file with its frequency.

bac aab b

{ ' ':2, 'a':3, 'b':3, 'c':1, EOF:1 }
HuffmanNode* buildEncodingTree(Map<int, int> freqTable)

Take as input a Map associating each character in the file with its frequency containing the file to compress, then hands back a Huffman encoding tree
HuffmanNode

```c
HuffmanNode* {
    int character;    // character being represented by this node
    int count;       // number of occurrences of that character
    HuffmanNode* zero;  // 0 (left) subtree (nullptr if empty)
    HuffmanNode* one;   // 1 (right) subtree (nullptr if empty)
}
```

The character field is declared as type `int`, but you should think of it as a `char`. The character field can take one of three types of values:

- `char` value
- `PSEUDO_EOF` which represents the pseudo-EOF value
- `NOT_A_CHAR` which represents something that isn't actually a character
(2a) Place occurrences into PQueue

Stanford PQueue
pq.enqueue(value, priority)

Map: {' ':2, 'a':3, 'b':3, 'c':1, EOF:1}

PQueue: [ EOF:1, 'c':1, ' ':2, 'a':3, 'b':3]
(2b) Build tree
The Huffman tree tells you the binary encodings to use.

- example: 'b' is 10
- example: 'c' is 010

 `{ ' ':00, 'a':11, 'b':10, 'c':010, EOF:011}`
void encodeData(istream input,
               Map<int, string> encodingMap,
               obistream output)

Take as input an **istream** of text to compress, a **Map** associating each character with the bit sequence to use to encode it, then writes everything to the **obitstream**.
obitstream: Writes one bit at a time to output.

| void writeBit(int bit) | Writes a single bit (must be 0 or 1) |

- obitstream also contains the members from ostream.
  - open, read, write, fail, close
(4) Encode the file

- Based on the preceding tree, we have the following encodings:
  \{' ':00, 'a':11, 'b':10, 'c':010, EOF:011\}

- The text "ab ab cab" would be encoded as:

<table>
<thead>
<tr>
<th>char</th>
<th>'a'</th>
<th>'b'</th>
<th>' '</th>
<th>'a'</th>
<th>'b'</th>
<th>' '</th>
<th>'c'</th>
<th>'a'</th>
<th>'b'</th>
<th>EOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td>11</td>
<td>10</td>
<td>00</td>
<td>11</td>
<td>10</td>
<td>00</td>
<td>010</td>
<td>11</td>
<td>10</td>
<td>011</td>
</tr>
</tbody>
</table>

- Overall: 1110001110000101110011, (22 bits, ~3 bytes)
Decompressing

Wait... Don’t you need delimiters?!?

1011010001101011011011
b a c _ a c a

– Read each bit one at a time.
– If it is 0, go left; if 1, go right.
– If you reach a leaf, output the character there and go back to the tree root.

● Output:
  bac aca
Decompressing II

How do we know what the map is for decompressing?

Include the mapping in the file itself!

\{32:2, 97:3, 98:3, 99:1, 256:1\}

*Hint: Maps can easily be read and written to/from streams using << and >> operators*
(5) Decode the file

```c
void decodeData(ibitstream input,
    HuffmanNode* encodingTree,
    ostream out)
```

Take as input an `ibitstream` of bits, a pointer `encodingTree` to the root of an encoding tree, then writes everything to `out`
\textbf{ibitstream}: Reads one bit at a time from input.

\begin{center}
\begin{tabular}{|l|p{12cm}|}
\hline
\textbf{int readBit()} & Reads a single 1 or 0; returns -1 at end of file \\
\hline
\end{tabular}
\end{center}

- \textit{ibitstream} also contains the members from \textit{istream}.
  - \texttt{open, read, write, fail, close}
void compress(istream& input, obitstream& output)

This is the overall compression function; in this function you should compress the given input file into the given output file. You will take as parameters an input file that should be encoded and an output bit stream to which the compressed bits of that input file should be written. You should:

- Read the input file one character at a time,
- Build an encoding of its contents
- Write a compressed version of that input file, including a header, to the specified output file.

This function should be built on top of the other encoding functions and should call them as needed
void decompress(ibitstream& input, ostream& output)

This function should do the opposite of compress:

- Read the bits from the given input file one at a time, including your header packed inside the start of the file
- Write the original contents of that file to the file specified by the output parameter.
Good luck Huffman encoding!

David A. Huffman