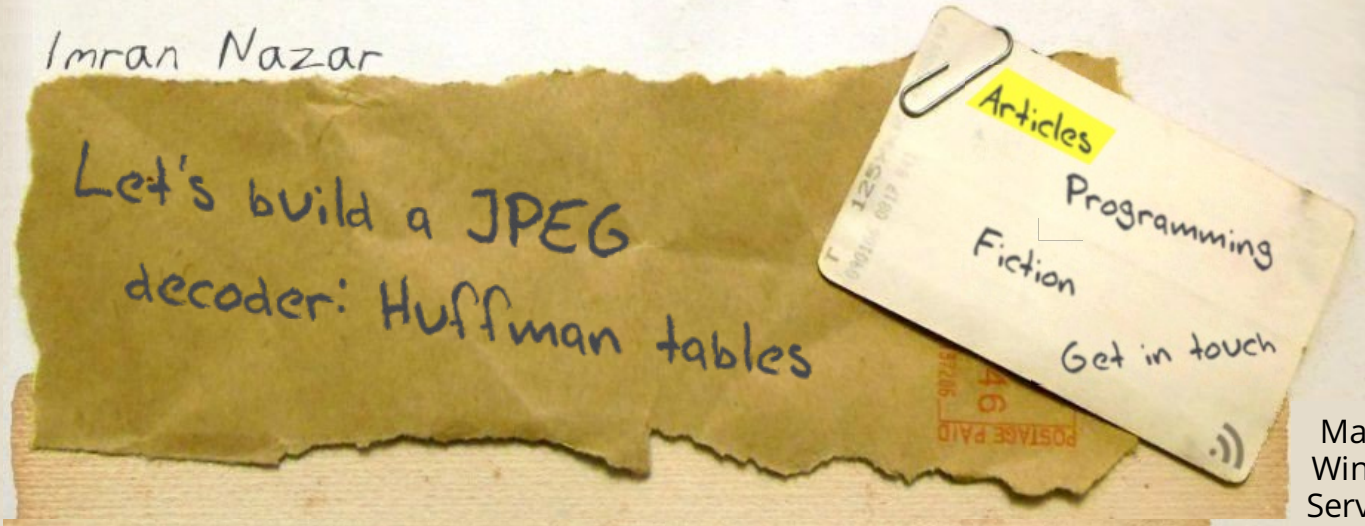

This article is part three of a series exploring the concepts and implementation of JPEG decoding; four parts are currently available, and others are expected to follow.

- Part 1: Concepts
- Part 2: File Structure
- Part 3: Huffman Tables
- Part 4: Frames and Bitstreams

Download the code: http://imrannazar.com/content/files/jpegparse.zip

In the previous part, I mentioned that most JPEG files employ an encoding technique on top of the image compression, in an attempt to remove any trace of redundant information from the image. The technique used by the most common JPEG encoding is an adaptation of one seen throughout the world of data compression, known as Huffman coding, so it's useful to explore in detail the structure and implementation of a Huffman decoder.

Because Huffman coding is the last thing performed by a JPEG encoder when saving an image file, it needs to be the first thing done by our decoder. This can be achieved in two ways:

- **Reconstitution**: The full image scan is decoded from its Huffman-coded state into a temporary placeholder, and further processing is performed on the temporary copy.
- **On-the-fly**: The encoded image scan is processed one code at a time, and each JPEG block is handled when enough information is available.

This article will take the second approach, to save memory and sacrifice time; full reconstitution can be implemented using the code built below in a very similar fashion.

The Huffman algorithm

The concept behind Huffman coding and other entropy-based schemes is similar to the concept behind the substitution cipher: each unique character in an input is transformed into a unique output character. The simplest example is the Caesar substitution, which can be represented in tabular form as follows:

```
A => D
B => E
C => F
...
Y => B
Z => C

This is an example of a Caesar cipher
Wklv lv dq hadpsoh ri d Fdhvdu flskhu
```

An improvement on the standard substitution cipher can be made by noting the relative frequency of

characters in the input, and designing a table that contains shorter codes as substitutes for these characters, than for rarer ones. Taking a look at the frequency of letters in the above example, with their ASCII representations included, we can produce a table of increasing unique codes such as the following:

Character	ASCII	Frequency	Code
Space	00100000	7	00
a	01100001	5	01
e	01100101	4	100
i	01101001	3	1010
s	01110011	3	1011
h	01101000	2	11000
p	01110000	2	11001
r	01110010	2	11010
C	01000011	1	110110
T	01010100	1	110111
c	01100011	1	111000
f	01100110	1	111001
l	01101100	1	111010
m	01101101	1	111011
n	01101110	1	111100
o	01101111	1	111101
x	01110111	1	111110

Table 1: Frequency of characters in the string "This is an example of a Caesar cipher"

Substituting these codes for the characters in the original text, it can be seen how the encoded data is much smaller than the original.

```
This is an example of a Caesar cipher
01010100 01101000 01101001 01110011 00100000 01101001 01110011 00100000
01100001 01101110 00100000 01100101 01110111 01100001 01101101 01110000
01101100 01100101 00100000 01101111 01100110 00100000 01100001 00100000
01000011 01100001 01100101 01110011 01100001 01110010 00100000 01100011
01101001 01110000 01101000 01100101 01110010

110111 11000 1010 1011 00 1010 1011 00 01 111100 00 100 111110 01 111011
11011 111010 100 00 111101 111001 00 01 00 110110 01 100 1011 01 11010
00 111000 1010 11001 11000 100 11010

54 68 69 73 20 69 73 20 61 6E 20 65 77 61 6D 70 6C 65 20 6F 66 20 61 20
43 61 65 73 61 72 20 63 69 70 68 65 72
DF 15 65 58 F8 4F 9E F7 D4 3D E4 4D 99 6E 8E 2B 38 9A
```

The main disadvantage of the Huffman coding method is that the table of codes needs to be stored alongside the compressed data: in the above example, the red string of encoded bytes would be meaningless without the corresponding frequency table. The table of codes and their corresponding characters can be recorded in full, but there is a more space-efficient way to save the codes, if attention is paid to the pattern of their occurrence. Two things are of note here: firstly that the codes increase in length, but also that within a group of the same length, codes are sequential. This means the code table can be written down as:

```
2 codes of length two  , starting at 00
1 code  of length three, starting at 100
2 codes of length four , starting at 1010
3 codes of length five , starting at 11000
9 codes of length six  , starting at 110110
```

A careful eye on the codes themselves can yield further improvements on how much space it takes to record the encoding table. If we take a look at the codes in conjunction with the list of code length above, we can start counting as follows.

```
00 (zero)
01 (one)
Next code would be 10 (two)
100 (four)
Next code would be 101 (five)
1010 (ten)
1011 (eleven)
Next code would be 1100 (twelve)
11000 (twenty four)
11001 (twenty five)
11010 (twenty six)
Next code would be 11011 (twenty seven)
110110 (fifty four)
110111 (fifty five)
111000 (fifty six)
111001 (fifty seven)
111010 (fifty eight)
111011 (fifty nine)
111100 (sixty)
111101 (sixty one)
111110 (sixty two)
```

In every case, when the requisite number of codes has been counted for the given code length, all that is needed is to double the counter and continue for the next code length. In other words, there is no need to record the "starting at" part of the code lengths list above, since it can be inferred by starting at zero. The final code list therefore looks as follows.

```
2 codes of length two
1 code  of length three
2 codes of length four
3 codes of length five
9 codes of length six

The above codes correspond to the following characters, in this order:
Space,a,e,i,s,h,p,r,C,T,c,f,l,m,n,o,x
```

The JPEG DHT segment

A JPEG file's Huffman tables are recorded in precisely this manner: a list of how many codes are present of a given length (between 1 and 16), followed by the meanings of the codes in order. This information is held in the file's "Define Huffman Table" (DHT) segments, of which there can be up to 32, according to the JPEG standard.

As seen above, data encoded by the Huffman algorithm ends up recorded as a series of codes wedged together in a bit-stream; this also applies to the image scan in a JPEG file. A simple routine for reading codes from the bit stream may look like this:

Pseudocode for reading a Huffman-coded value

```
Code = 0
Length = 0
Found = False

Do
    Code = Code << 1
    Code = Code | (The next bit in the stream)
    Length = Length + 1
    If ((Length, Code) is in the Huffman list) Then
        Found = True
    End If
While Found = False
```

In order to facilitate this algorithm, the Huffman codes should be stored in a way that allows us to determine if a code is in the map at a given length. The canonical way to represent a Huffman code list is as a binary tree, where the sequence of branches defines the code and the depth of the tree tells us how long the code is. The C++ STL abstracts this out for us, into the `map` construct.

Since there are up to 32 possible Huffman tables that can be defined in a JPEG file, our implementation will require 32 `map`s to be available. It's also worth defining at this point how the DHT segment handler will be called by the `parseSeg` method developed in the previous part of this series.

jpeg.h: DHT data definition

```
class JPEG {
    private:
        // Defines a tuple of length and code, for use in the Huffman maps
        typedef std::pair<int, u16> huffKey;

        // The array of Huffman maps: (length, code) -> value
        std::map<huffKey, u8> huffData[32];

        // DHT segment handler
        int DHT();
};
```

jpeg.cpp: Passing control to the segment handler

```
int JPEG::parseSeg()
{
    ...

    switch (id) {
        // The DHT segment defines a Huffman table. The handler should
        // read exactly as many bytes from the file as are in the
        // segment; if not, something's gone wrong
        case 0xFFC4:
            size = READ_WORD() - 2;
            if (DHT() != size) {
                printf("Unexpected end of DHT segment\n");
                return JPEG_SEG_ERR;
            }
            break;

        ...
    }

    return JPEG_SEG_OK;
}
```

jpeg.cpp: DHT segment handler, builds a Huffman map

```
int JPEG::DHT()
{
    int i, j;

    // A counter of how many bytes have been read
    int ctr = 0;

    // The incrementing code to be used to build the map
    u16 code = 0;

    // First byte of a DHT segment is the table ID, between 0 and 31
    u8 table = fgetc(fp);
    ctr++;

    // Next sixteen bytes are the counts for each code length
    u8 counts[16];
    for (i = 0; i < 16; i++) {
        counts[i] = fgetc(fp);
        ctr++;
    }

    // Remaining bytes are the data values to be mapped
    // Build the Huffman map of (length, code) -> value
    for (i = 0; i < 16; i++) {
        for (j = 0; j < counts[i]; j++) {
            huffData[table][huffKey(i + 1, code)] = fgetc(fp);
            code++;
            ctr++;
        }
        code <<= 1;
    }

    // Once the map has been built, print it out
    printf("Huffman table #%02X:\n", table);

    std::map<huffKey, u8>::iterator iter;
    for (iter = huffData[table].begin();
         iter != huffData[table].end();
         iter++
    ) {
        printf("    %04X at length %d = %02X\n",
            iter->first.second, iter->first.first, iter->second);
    }

    return ctr;
}
```

As with the previous part, the JPEG class can be instantiated with a filename; if this is done, the above code will produce output along the following lines:

```
Found segment at file position 177: Huffman table
Huffman table #00:
    0000 at length 2 = 04
    0002 at length 3 = 02
    0003 at length 3 = 03
    0004 at length 3 = 05
    0005 at length 3 = 06
    0006 at length 3 = 07
    000E at length 4 = 01
    001E at length 5 = 00
    003E at length 6 = 08
    007E at length 7 = 09
```

Next time: Reading the bitstream

Once the Huffman maps have been built for a JPEG file, the image scan can be decoded for further processing. In the next part, I'll take a look at the Huffman decoding of the scan, in the wider context of reading blocks from the image and examining the process through which they are transformed.

Imran Nazar <tf@imrannazar.com>, Feb 2013.

Article dated: 24th Feb 2013