Deflate compression is a lossless compression algorithm. It is a combination of the LZ77 compression algorithm and Huffman coding. It uses Huffman coding to determine if data is a literal or a reference pair. It also provides a means to store just literals. For the algorithm, you can use a custom Huffman tree or use the default provided one. Because this algorithm is the combination of LZ77 and Huffman coding, this document will refer to other documents about compressing and decompressing specifics dealing with LZ77 or Huffman coding. ZLIB will be discussed as a header since it is 2 bytes.

Like LZW, the bytes are read from left to right, but the bits are read from right to left. Deflate data tends to have a header to specify different settings such as how large the window should be for the back reference in the LZ77 algorithm. ZLIB is common for PNG files, so we will use that.

ZLIB HEADER

* Just as with LZW, the bytes are read from left to right, but the bits are read from right to left.
* There are 2 bytes for the header + Dictionary stuff
  + Dictionary Stuff refers to the additional bytes that can occur it the flag FDICT is set
  + We will only cover when FDICT is not set since PNG only deals with those.
* The first byte is the CMF byte
  + Compression Method and flags
  + Bits 0 to 3 are CM (Compression method)
    - CM = 8 is the deflate compression method which what this document cares about.
  + Bits 4 to 7 are CINFO (Compression Info)
    - For CM=8, this data refers to the size of the LZ77 window. It is the base-2 log of the window size - 8. Values above 7 are not allowed because 27+8 = 32
* The second byte is the FLG byte
  + Flag byte
  + Bits 0 to 4 are the FCHECK value
    - (CMF\*256 + FLG) must be a multiple of 31
  + Bit 5 is the FDICT flag
    - Specifies if a dictionary is present that should be read. It occurs directly after the FLG byte.
    - This must be 0 for deflate
  + Bits 6 to 7 are the FLEVEL value
    - Used by the deflate method to specify how it was compressed. Not necessary for decompression
    - 0 – Fastest Compression
    - 1 – Fast Compression
    - 2 – Default Compression
    - 3 – Maximum Compression
* ZLIB also contains an ADLER32 value after all the compressed data
  + This value is important for checking if the data is valid and for compressing.
  + The value is equal to B\*65536 + A
    - A = 1 + data[0] + data[1] + … + data[n] mod 65521
    - B = A[0] + A[1] + A[2] + A[3] + … + A[n] mod 65521
      * A[n] refers to the value of A up to data[n]
      * A[0] = 1 + data[0]
      * A[1] = 1 + data[1] + data[2]
      * So on
  + The value must be the same after decompressing
  + The ADLER32 checksum is 4 bytes and stored in big endian

DEFLATE

* The header for a deflate block is 3 bits
  + Bit 0 is the BFINAL flag
    - This flag determines if this block is the last block to be decompressed
  + Bits 1 and 2 are the BTYPE value
    - This determines what type of data is in this block.
    - 0 is no compression
    - 1 is compression with default Huffman tree
    - 2 is compression with dynamic Huffman tree
    - 3 is invalid
* No Compression
  + A block with no compression just contains literals.
  + First, you skip the bits to the next byte boundary.
  + There is a “header” for this block
    - This takes up 4 bytes
    - Bytes 0 and 1 are the LEN value
      * Specifies how many bytes to read
    - Bytes 2 and 3 are the NLEN value
      * The complement of LEN
  + After the first 4 bytes, you read LEN bytes of literal data.
* Compressed blocks
  + Both type 1 and 2 types of data blocks are compressed and decompressed in much the same way. The difference being the Huffman tree used.
  + The default tree will be discussed along with creating the dynamic tree.
  + The rules for reference pairs will be discussed as well.
  + Values 0 – 255 represent literal values
  + Value 256 represents the end of a block
  + Values 257 – 285 represent reference pair stuff
  + Note that some codes will never occur in actual data even if they are possible.
* For reference pairs, there is a table on how it works
  + Note that the extra bits get added to the lower length of that value.
  + The extra bits occur immediately after the value and should be interpreted as a number.
  + For Lengths:

|  |  |  |
| --- | --- | --- |
| Value | Extra Bits | Length |
| 257 | 0 | 3 |
| 258 | 0 | 4 |
| 259 | 0 | 5 |
| 260 | 0 | 6 |
| 261 | 0 | 7 |
| 262 | 0 | 8 |
| 263 | 0 | 9 |
| 264 | 0 | 10 |
| 265 | 1 | 11-12 |
| 266 | 1 | 13-14 |
| 267 | 1 | 15-16 |
| 268 | 1 | 17-18 |
| 269 | 2 | 19-22 |
| 270 | 2 | 23-26 |
| 271 | 2 | 27-30 |
| 272 | 2 | 31-34 |
| 273 | 3 | 35-42 |
| 274 | 3 | 43-50 |
| 275 | 3 | 51-58 |
| 276 | 3 | 59-66 |
| 277 | 4 | 67-82 |
| 278 | 4 | 83-98 |
| 279 | 4 | 99-114 |
| 280 | 4 | 115-130 |
| 281 | 5 | 131-162 |
| 282 | 5 | 163-194 |
| 283 | 5 | 195-226 |
| 284 | 5 | 227-257 |
| 285 | 0 | 258 |

For Back References:

|  |  |  |
| --- | --- | --- |
| Value | Extra Bits | Length |
| 0 | 0 | 1 |
| 1 | 0 | 2 |
| 2 | 0 | 3 |
| 3 | 0 | 4 |
| 4 | 1 | 5-6 |
| 5 | 1 | 7-8 |
| 6 | 2 | 9-12 |
| 7 | 2 | 13-16 |
| 8 | 3 | 17-24 |
| 9 | 3 | 25-32 |
| 10 | 4 | 33-48 |
| 11 | 4 | 49-64 |
| 12 | 5 | 65-96 |
| 13 | 5 | 97-128 |
| 14 | 6 | 129-192 |
| 15 | 6 | 193-256 |
| 16 | 7 | 257-384 |
| 17 | 7 | 385-512 |
| 18 | 8 | 513-768 |
| 19 | 8 | 769-1024 |
| 20 | 9 | 1025-1536 |
| 21 | 9 | 1537-2048 |
| 22 | 10 | 2049-3072 |
| 23 | 10 | 3073-4096 |
| 24 | 11 | 4097-6144 |
| 25 | 11 | 6145-8192 |
| 26 | 12 | 8193-12288 |
| 27 | 12 | 12289-16384 |
| 28 | 13 | 16385-24576 |
| 29 | 13 | 24577-32768 |

* Default Huffman Tree for compressed blocks

|  |  |  |
| --- | --- | --- |
| Value | Amount of Bits | Codes |
| 0 – 143 | 8 | 00110000 – 10111111 |
| 144 – 255 | 9 | 110010000 – 111111111 |
| 256 – 279 | 7 | 0000000 – 0010111 |
| 280 – 287 | 8 | 11000000 – 11000111 |

* + Note that for reference pairs, after reading a value representing a length value, you read the next 5 bits to get the back-reference value.
* Dynamic Huffman Tree
  + The data for the dynamic Huffman tree is stored as the first part of the block after the header.
  + The block starts with 3 values taking up 14 bits
    - The first 5 bits are for HLIT
      * The number of literal/length codes
      * The number of codes is 257 + HLIT
    - The next 5 bits are for HDIST
      * The number of distance codes
      * The number is 1 + HDIST
    - The next 4 bits are for HCLEN
      * The number of code length codes
      * The number is 4 + HCLEN
  + This block is quite complicated and relies on Huffman coding to get the Huffman tree for the data you want to decompress.
  + We first use the HCLEN value to read off code lengths.
    - Each code length is 3 bits
    - We read 4+HCLEN times
  + Each of the lengths refers to how long the code is for the following list
    - 16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15
    - Note that the code length 0 can occur
  + After obtaining all the lengths, we form the canonical Huffman code for the numbers 0-18.
    - Refer to the Huffman compression document
  + Next, we read data N times
    - N is the total number of values you must define.
    - N = (257 + HLIT) + (1 + HDIST)
  + For each of the values we compress, we follow the rules listed below:
    - 0 – 15: is a literal code length
    - 16: Copy the previous code length 3-6 times
      * Next 2 bits are the extra bits. Interpret like a number and add to 3 to find the actual value.
      * Then you copy and stuff
    - 17: Copy the code of 0 for 3 – 10 times
      * Next 3 bits are the extra bits. Interpret like a number and add to 3
    - 18: Copy the code of 0 for 11 – 138 times
      * Next 7 bits are the extra bits. Interpret like a number and add to 11
  + The code length of 0 means that the value does not occur in the data set.
  + The Literals and Length occur first then the Distance values after.
  + After defining the tree, the data come directly after.
  + The value for 256 (end of the block) must be defined otherwise, the data cannot be read.
  + When using a dynamic tree, the value for the back-reference is not just the 5 bits after a length value plus its extra bits.
    - The back-reference is Huffman coded as well.
  + Note that its possible to have no back-reference values and no length values.
* When decompressing, follow the Huffman coding rules to decode a value and then follow the LZ77 rules about copying data. Refer to those documents on how to do so.

DEFLATE COMPRESSION

* While it is better to stray away from this as you could be producing invalid data easily, it is good to know to save PNG files
* Compressing is simple as it follows the same flow as Huffman coding and LZ77 coding.
* The difference is that we must define a max back reference and separate our data into parts to maximize efficiency.
* For that reason, using either no compression or default compression is advised. Using dynamic trees comes with the disadvantage that your data could be inflating compared to the default tree.
  + Also, you must deal with the weird method of creating the Huffman code and what not.
* The requirement for ZLIB is that you have an ADLER-32 Checksum at the end of each ZLIB block.