# Lossless Compression Techniques

## 1. Run Length Coding

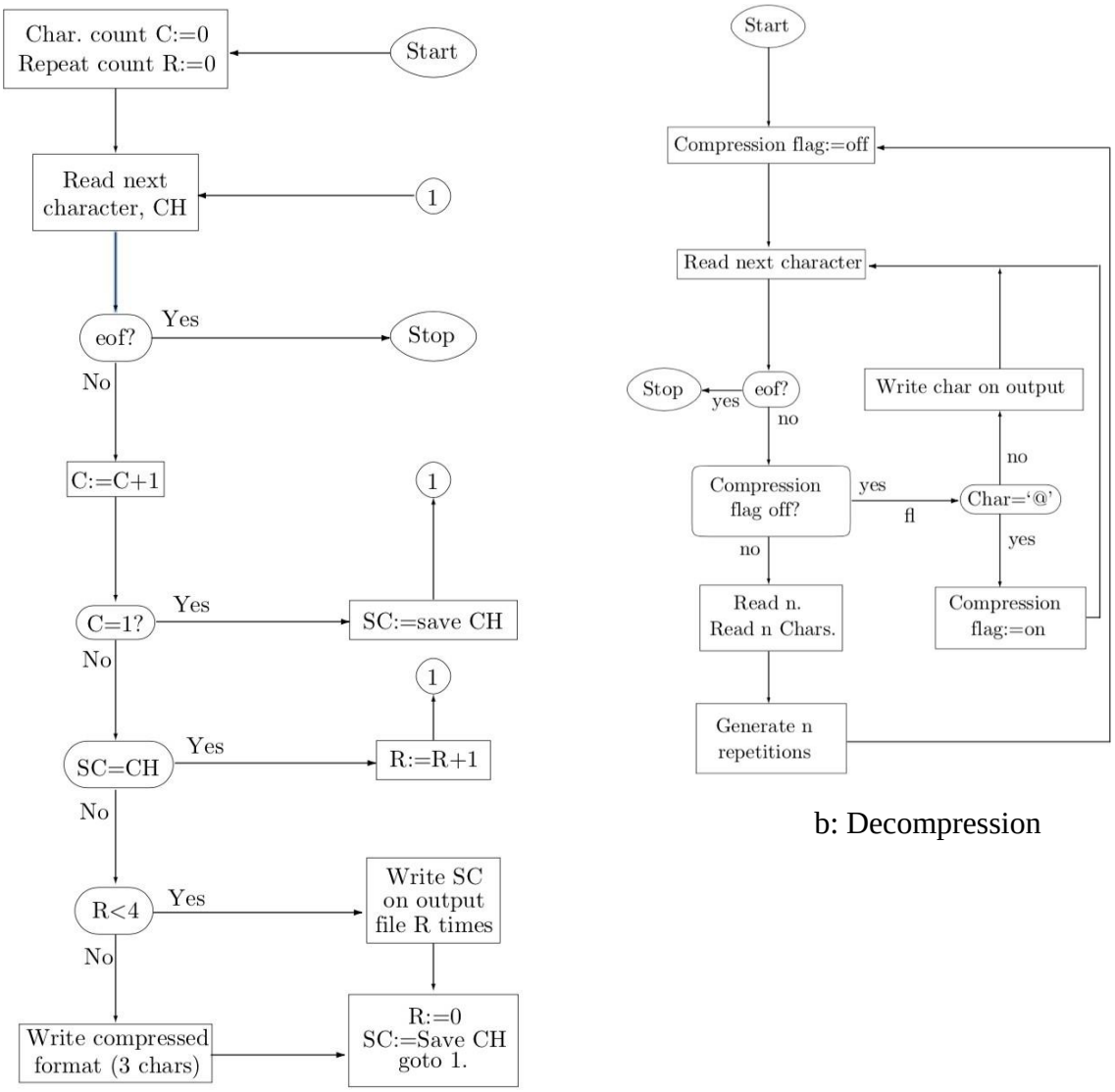
Is a simple and popular lossless data compression algorithm. The idea behind this approach to data compression is this*: If a data item* ***d*** *occurs* ***n*** *consecutive times in the input stream, replace the* ***n*** *occurrences with the single pair* ***nd****.* The ***n*** consecutive occurrences of a data item are called a run length of ***n***, and this approach to data compression is called run-length encoding or RLE. We apply this idea first to text compression and then to image compression.

### A. RLE Text Compression

Just replacing 2._all_is_too_well with 2._a2_is_t2_we2 will not work. Even the string 2._a2l_is_t2o_we2l does not solve this problem. One way to solve this problem is to precede each repetition with a special escape character. If we use the character @ as the escape character, then the string 2._a@2l_is_t@2o_we@2l can be decompressed unambiguously. However, this string is longer than the original string, because it replaces two consecutive letters with three characters. We have to adopt the convention that only four or more repetitions of the same character will be replaced with a repetition factor. The main problems with this method are the following:

1. In English text there are not many repetitions. There are many "doubles", but a "triple" is rare.
2. The character "@" may be part of the text in the input stream, in which case a different escape character must be chosen. Sometimes the input stream may contain every possible character in the alphabet. Figure 2a is a flowchart for such a simple run-length text compressor.

After reading the first character, the count is 1 and the character is saved. Subsequent characters are compared with the one already saved and, if they are identical to it, the repeat-count is incremented. When a different character is read, the operation depends on the value of the repeat count. If it is small, the saved character is written on the compressed file and the newly read character is saved. otherwise, an "@" is written, followed by the repeat-count and the saved character. Decompression is also straightforward. It is shown in Figure 1.3b. When an "@" is read, the repetition count ***n*** and the actual character are immediately read, and the character is written ***n*** times on the output stream.




a: Compression

b: Decompression

Figure 2: RLT text compression and decompression

To get an idea of the compression ratios produced by RLE, we assume a string of N characters that needs to be compressed. We assume that the string contains M repetitions of average length L each. Each of the M repetitions is replaced by 3 characters (escape, count, and data), so the size of the compressed string is: $N - M(L - 3)$ and the compression factor is: $N/(N - M(L - 3))$.

Examples: $N = 1000$, $M = 10$, $L = 4$ yield a compression factor of $1000/[1000 - 10(4 - 3)] = 1.01$. A better result is obtained in the case $N = 1000$, $M = 50$, $L = 10$, where the factor is $1000/[1000 - 50(10 - 3)] = 1.538$.

## B. RLE Image Compression

RLE can be used to compress grayscale images. Each run of pixels of the same intensity (gray level) is encoded as a pair (run length, pixel value). The run length usually occupies one byte, allowing for runs of up to 255 pixels. The pixel value occupies several bits, depending on the number of gray levels (typically between 4 and 8 bits).

RLE is a natural candidate for compressing graphical data. A digital image consists of small dots called pixels. Each pixel can be either one bit, indicating a black or a white dot, or several bits, indicating one of several colours or shades of gray. We assume that the pixels are stored in an array called a bitmap in memory, so the bitmap is the input stream for the image. Pixels are normally arranged in the bitmap in scan lines, so the first bitmap pixel is the dot at the top left corner of the image, and the last pixel is the one at the bottom right corner.

Compressing an image using RLE is based on the observation that if we select a pixel in the image at random, there is a good chance that its neighbours will have the same colour. The compressor thus scans the bitmap row by row, looking for runs of pixels of the same colour. If the bitmap starts, e.g., with 17 white pixels, followed by 1 black one, followed by 55 white ones, etc., then only the numbers 17, 1, 55, need be written on the output stream. The compressor assumes that the bitmap starts with white pixels. If this is not true, then the bitmap starts with zero white pixels, and the output stream should start with 0. The resolution of the bitmap should also be saved at the start of the output stream.

The size of the compressed stream depends on the complexity of the image. The more detail, the worse the compression. However, Figure 3 shows how scan lines go through a uniform area. A line enters through one point on the perimeter of the area and exits through another point, and these two points are not “used” by any other scan lines. It is now clear that the number of scan lines traversing a uniform area is roughly equal to half the length (measured in pixels) of its perimeter. Since the area is uniform, each scan line contributes one number to the output stream. The compression ratio of a uniform region therefore roughly equals: 2×half the length of the perimeter /total number of pixels in the region = perimeter /area.

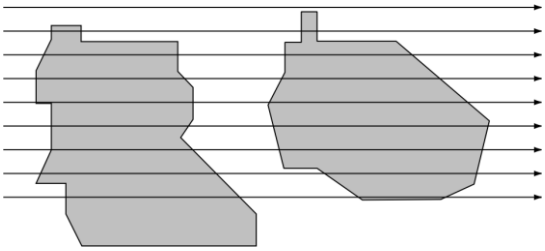


Figure 3: Uniform area and scan lines

Example: An 8-bit deep grayscale bitmap that starts with12, 12, 12, 12, 12, 12, 12, 12, 12, 35, 76, 112, 67, 87, 87, 87, 5, 5, 5, 5, 5, 5, 1, . . . is compressed into 9 ,12,35,76,112,67, 3 ,87, 6 ,5,1,. . . , where the bold numbers indicate counts. The problem is to distinguish between a byte containing a grayscale value (such as 12) and one containing a count (such as 9 ). Here are some solutions

1. If the image is limited to just 128 grayscales, we can devote one bit in each byte to indicate whether the byte contains a grayscale value or a count.
2. If the number of grayscales is 256, it can be reduced to 255 with one value reserved as a flag to precede every byte with a count. If the flag is, say, 255, then the sequence above becomes: 255, 9, 12, 35, 76, 112, 67, 255, 3, 87, 255, 6, 5, 1,....
3. A group of m pixels that are all different is preceded by a byte with the negative value −m. The sequence above is encoded by: 9, 12, −4, 35, 76, 112, 67, 3, 87, 6, 5, ?, 1,... (the value of the byte with ? is positive or negative depending on what follows the pixel of 1).

Three more points should be mentioned:

1. Since the run length cannot be 0, it makes sense to write the [run length minus one] on the output stream. Thus the pair (3, 87) denotes a run of four pixels with intensity 87. This way, a run can be up to 256 pixels long.
2. In color images it is common to have each pixel stored as three bytes, In such a case, runs of each color should be encoded separately. The pixels (171, 85, 34), (172, 85, 35), (172, 85, 30), and (173, 85, 33) should be separated into the three sequences (171, 172, 172, 173, . . .), (85, 85, 85, 85, ), and (34, 35, 30, 33, ). Each sequence should be run-length encoded separately. This means that any method for compressing grayscale images can be applied to color images as well.
3. It is preferable to encode each row of the bitmap individually. Thus if a row ends with four pixels of intensity 87 and the following row starts with 9 such pixels, it is better to write ..., 4, 87, 9, 87, on the output stream rather than

..., 13, 87,.... It is even better to write the sequence ..., 4, 87, eol, 9, 87,..., where “eol” is a special end-of-line code.

Disadvantage of image RLE: When the image is modified, the run lengths normally have to be completely redone. The RLE output can sometimes be bigger than pixel by-pixel storage (i.e., an uncompressed image, a raw dump of the bitmap) for complex pictures. A good, practical RLE image compressor should be able to scan the bitmap by rows, columns, or in zigzag (Figure4 a, b) and it may automatically try all three ways on every bitmap compressed to achieve the best compression.

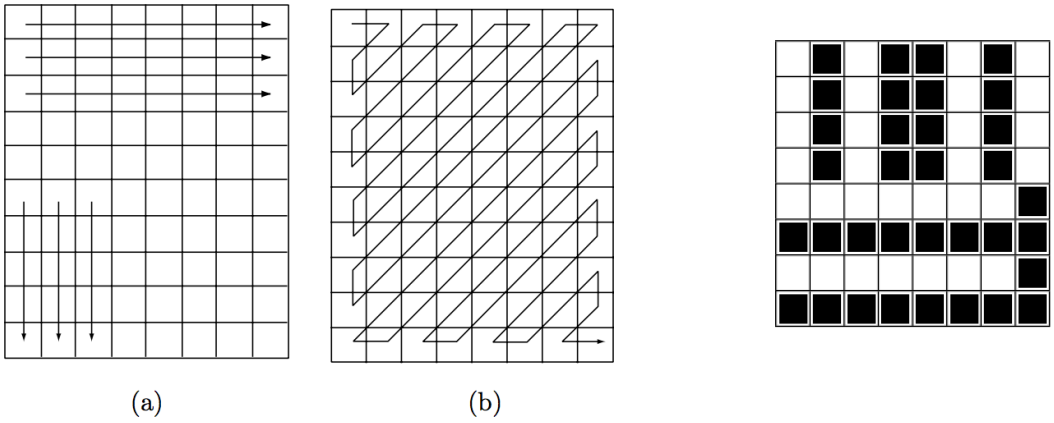



Figure 4: RLE Scanning.

**H.W/** Given the 8×8 bitmap of Figure above, use RLE to compress it, first row by row, then column by column. Describe the results in detail.

## 2. Golomb-Rice Coding

Golomb coding is a lossless data compression method invented by Solomon W. Golomb. Symbols following a **geometric distribution** will have a Golomb code as an optimal prefix code, making Golomb coding highly suitable for situations in which the occurrence of small values in the input stream is significantly more likely than large values. The Golomb codes belong to a family of codes designed to encode integers with the assumption that the larger an integer, the lower its probability of occurrence. The simplest code for this situation is the unary code. The unary code for a positive integer ***N*** is simply ***N*** 1s followed by a 0. Thus, the code for 4 is 11110, and the code for 7 is 11111110.

Golomb coding uses a tuneable parameter ***M*** to divide an input value ***N*** into two parts: ***q***, the results of a division by ***M*** and ***r***, the reminder. The quotient is sent in a unary coding, followed by the reminder in truncated binary encoding. When ***M* = 1**, Golomb coding is equivalent to unary coding. The Golomb-Rice code is a

special case of Golomb code. In this algorithm, if the ***M*** parameter is a power of 2, it becomes equivalent to the simpler Rice encoding. The description of the Golomb-Rice code can be given as follows:

1. Fix the parameter *M* to an integer value.
2. For *N*, the number to be encoded, find
    1. quotient = *q* = int[*N*/*M*]
    2. remainder = *r* = *N* modulo *M*
3. Generate Codeword
    1. The Code format : <Quotient Code><Remainder Code>, where
    2. Quotient Code (in unary coding)
        1. Write a *q*-length string of 1 bits (alternatively, of 0 bits)
        2. Write a 0 bit (respectively, a 1 bit)
    3. Remainder Code (in truncated binary encoding)
        1. LET $b = \lfloor \log_2(M) \rfloor$
            1. If $r < 2^b - M$ code *r* in binary representation using $b-1$ bits.
            2. If $r \geq 2^b - M$ code the number $r + 2^b - M$ in binary representation using *b* bits.

**Example:**

Set M = 10, Thus $b = \lceil log_(10 \rceil = 4$, the cut off is $2^b - M = 6$

| Encoding of quotient part | |
|---|---|
| ***q*** | **output bits** |
| 0 | 0 |
| 1 | 10 |
| 2 | 110 |
| 3 | 1110 |
| 4 | 11110 |
| 5 | 111110 |
| 6 | 1111110 |
| ⋮ | ⋮ |
| N | $\underbrace{111\cdots1110}_{N}$ |

| Encoding of remainder part | | | |
|---|---|---|---|
| ***r*** | **offset** | **binary** | **output bits** |
| 0 | 0 | 0000 | 000 |
| 1 | 1 | 0001 | 001 |
| 2 | 2 | 0010 | 010 |
| 3 | 3 | 0011 | 011 |
| 4 | 4 | 0100 | 100 |
| 5 | 5 | 0101 | 101 |
| 6 | 12 | 1100 | 1100 |
| 7 | 13 | 1101 | 1101 |
| 8 | 14 | 1110 | 1110 |
| 9 | 15 | 1111 | 1111 |

For example, with a Rice-Golomb encoding of parameter $M$ = 10, the decimal number 42 would first be split into $q$ = 4,$r$ = 2, and would be encoded as qcode($q$),rcode($r$) = qcode(4),rcode(2) = 11110,010

**Examples**. Choosing $m = 3$ produces $c = 2$ and the three remainders 0, 1, and 2. We compute $2^2 - 3 = 1$, so the first remainder is coded in $c - 1 = 1$ bit to become 0, and the remaining two are coded in two bits each ending with $11_2$, to become 10 and 11. Selecting $m = 5$ results in $c = 3$ and produces the five remainders 0 through 4. The first three ($2^3 - 5 = 3$) are coded in $c - 1 = 2$ bits each, and the remaining two are each coded in three bits ending with $111_2$. Thus, 00, 01, 10, 110, and 111. The following simple rule shows how to encode the $c$-bit numbers such that the last of them will consist of $c$ 1's. Denote the largest of the $(c-1)$-bit numbers by $b$, then construct the integer $b+1$ in $c-1$ bits, and append a zero on the right. The result is the first of the $c$-bit numbers and the remaining ones are obtained by incrementing.

Table 2.10 shows some examples of $m$, $c$, and $2^c - m$, as well as some Golomb codes for $m = 2$ through 13.

| $m$ | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $c$ | 1 | 2 | 2 | 3 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| $2^c - m$ | 0 | 1 | 0 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

| $m/n$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 0\|0 | 0\|1 | 10\|0 | 10\|1 | 110\|0 | 110\|1 | 1110\|0 | 1110\|1 | 11110\|0 | 11110\|1 | 111110\|0 | 111110\|1 | 1111110\|0 |
| 3 | 0\|0 | 0\|10 | 0\|11 | 10\|0 | 10\|10 | 10\|11 | 110\|0 | 110\|10 | 110\|11 | 1110\|0 | 1110\|10 | 1110\|11 | 11110\|0 |
| 4 | 0\|00 | 0\|01 | 0\|10 | 0\|11 | 10\|00 | 10\|01 | 10\|10 | 10\|11 | 110\|00 | 110\|01 | 110\|10 | 110\|11 | 11110\|00 |
| 5 | 0\|00 | 0\|01 | 0\|10 | 0\|110 | 0\|111 | 10\|00 | 10\|01 | 10\|10 | 10\|110 | 10\|111 | 110\|00 | 110\|01 | 110\|10 |
| 6 | 0\|00 | 0\|01 | 0\|100 | 0\|101 | 0\|110 | 0\|111 | 10\|00 | 10\|01 | 10\|100 | 10\|101 | 10\|110 | 10\|111 | 110\|00 |
| 7 | 0\|00 | 0\|010 | 0\|011 | 0\|100 | 0\|101 | 0\|110 | 0\|111 | 10\|00 | 10\|010 | 10\|011 | 10\|100 | 10\|101 | 10\|110 |
| 8 | 0\|000 | 0\|001 | 0\|010 | 0\|011 | 0\|100 | 0\|101 | 0\|110 | 0\|111 | 10\|000 | 10\|001 | 10\|010 | 10\|011 | 10\|100 |
| 9 | 0\|000 | 0\|001 | 0\|010 | 0\|011 | 0\|100 | 0\|101 | 0\|110 | 0\|1110 | 0\|1111 | 10\|000 | 10\|001 | 10\|010 | 10\|011 |
| 10 | 0\|000 | 0\|001 | 0\|010 | 0\|011 | 0\|100 | 0\|101 | 0\|1100 | 0\|1101 | 0\|1110 | 0\|1111 | 10\|000 | 10\|001 | 10\|010 |
| 11 | 0\|000 | 0\|001 | 0\|010 | 0\|011 | 0\|100 | 0\|1010 | 0\|1011 | 0\|1100 | 0\|1101 | 0\|1110 | 0\|1111 | 10\|000 | 10\|001 |
| 12 | 0\|000 | 0\|001 | 0\|010 | 0\|011 | 0\|1000 | 0\|1001 | 0\|1010 | 0\|1011 | 0\|1100 | 0\|1101 | 0\|1110 | 0\|1111 | 10\|000 |
| 13 | 0\|000 | 0\|001 | 0\|010 | 0\|0110 | 0\|0111 | 0\|1000 | 0\|1001 | 0\|1010 | 0\|1011 | 0\|1100 | 0\|1101 | 0\|1110 | 0\|1111 |

Table 2.10: Some Golomb Codes for $m = 2$ Through 13.

Note: In this example, c = b which is defined above.

**Example 1:**

The source of information A generates the symbols {A0, A1, A2, A3 and A4} with the corresponding probabilities {0.4, 0.3, 0.15, 0.1 and 0.05}. Encoding the source symbols using binary encoder and Golomb encoder gives:

| Source Symbol | $P_i$ | Binary Code | Golomb Code |
|---|---|---|---|
| A0 | 0.4 | 000 | 0 |
| A1 | 0.3 | 001 | 10 |
| A2 | 0.15 | 010 | 110 |
| A3 | 0.1 | 011 | 1110 |
| A4 | 0.05 | 100 | 1111 |
| $L_{avg}$ | H = 2.0087 | 3 | 2.05 |

The Entropy of the source is

$$H = -\sum_{i=0}^{4} Pi \log_2 Pi = 2.0087 \text{ bit/symbol}$$

Since we have 5 symbols ($5<8=2^3$), we need 3 bits at least to represent each symbol in binary (fixed-length code). Hence the average length of the binary code is

$$Lavg = \sum_{i=0}^{4} Pi\, li = 3\,(0.4 + 0.3 + 0.15 + 0.1 + 0.05) = 3 \text{ bit/symbol}$$

Thus the efficiency of the binary code is

$$\eta = \frac{H}{Lavg} = \frac{2.0087}{3} = 67\%$$

The average length of the Golomb code is

$$Lavg = \sum_{i=0}^{4} Pi\, li = 0.4 * 1 + 0.3 * 2 + 0.15 * 3 + 0.1 * 4 + 0.05 * 4 = 2.05 \text{ bit/symbol}$$

Thus the efficiency of the Golomb code is

$$\eta = \frac{H}{Lavg} = \frac{2.0087}{2.05} = 98\%$$

This example demonstrates that the efficiency of the Golomb encoder is much higher than that of the binary encoder.