

Teaching guide: Run-length encoding (RLE)

This teaching guide is designed to help you teach Run-length encoding from our GCSE Computer Science specification (8525). It is not prescriptive; it simply gives you some teaching ideas that you can adapt to the needs of your students.

Run-length encoding (RLE) is a form of lossless compression

RLE is a simple method of compressing data by specifying the number of times a character or pixel colour repeats followed by the value of the character or pixel. The aim is to reduce the number of bits used to represent a set of data. Reducing the number of bits used means that it will take up less storage space and be quicker to transfer.

There are many different compression methods. Compression methods can be categorised as being *lossless* or *lossy*. With a lossless compression method the original data can be restored *exactly* as it was before it was decompressed, whereas when a lossy compression method is used some of the original data is lost during the compression process and cannot be restored when the data is decompressed.

There are times when lossy compression is the best choice – either when a significantly smaller file size is desired or if the data loss is not noticeable, eg there are many sounds that people cannot hear and if compression results in the loss of some of these sounds then the data loss is not important.

Lossy compression can often compress the data further than lossless compression (ie lossy compression results in fewer bits being used).

For text files and programs it is critical that not even a single bit of data is lost and therefore a lossless compression method *must* be used.

RLE for text data

The process involves going through the text and counting the number of consecutive occurrences of each character (called "a run"). The number of occurrences of the character and the character itself are then stored in pairs.

Example:

aaaabbbbbbbbcdddd

There are 16 characters in the example so 16 bytes (assuming Extended ASCII is being used) are needed to store these characters in an uncompressed format:

Example as ASCII:

97 97 97 97 98 98 98 98 98 98 99 100 100 100 100 100

RLE can be used to store that same data using fewer bytes. There are four consecutive occurrences of the character 'a' followed by six consecutive occurrences of 'b', one 'c' and finally five consecutive occurrences of 'd'. This could be written as the following sequence of count-character pairs:

Run-length encoding for the above example:

(4, a) (6, b) (1, c) (5, d)

So, if the text `aaaabbbbbbcdddd` is compressed using RLE, storing each count in binary in a single byte, we would end up with:

04 97 06 98 01 99 05 100

As we can see, this compressed version only requires 8 bytes – a reduction from the original 16 bytes.

Text that contains characters with a high frequency of repetition can be compressed efficiently. Text with few repetitions, ie more random text, will not compress well and can even result in the compressed version using *more* storage space than the uncompressed version.

A possible solution to this is to use a special byte value that 'flags' when a run is going to occur. This does mean though that any run of bytes automatically has an additional byte added to it. When there is no flag, the next byte(s) are taken as their face value and with a run of 1.

Example

Uncompressed text

aaaaaaaaabbbbbbbecececececececdccccccccccccccdec b (46 bytes)

RLE count-character pairs

(10, a) (6, b) (1, e) (1, c) (1, e) (1, c) (1, e) (1, c) (1, e)
(1, c) (1, e) (1, c) (1, e) (1, c) (15, d) (1, e) (1, c) (1, b)

No Flag RLE

10 97 06 98 01 101 01 99 01 101 01 99 01 101 01 99 01 101 01
99 01 101 01 99 01 101 01 99 15 100 01 101 01 99 01 98 (36 bytes)

Flag (value 255) RLE

255 10 97 255 06 98 101 99 101 99 101 99 101 99 101 99 101 99
255 15 100 101 99 98 (24 bytes)

An algorithm to decompress an RLE sequence stored in the array RLE would look like this:

```
index ← 0
WHILE index < LEN(RLE)
  IF RLE[index] = 255 THEN
    count ← RLE[index + 1]
    character ← CODE_TO_CHAR(RLE[index + 2])
    FOR i ← 1 TO count
      OUTPUT character
    ENDFOR
    index ← index + 3
  ELSE
    OUTPUT CODE_TO_CHAR(RLE[index])
    index ← index + 1
  ENDIF
ENDWHILE
```

RLE for bitmapped image data

RLE for bitmapped image compression works in the same way as text compression. The effectiveness depends on the image being compressed; images with long runs of pixels of the same colour will provide a higher compression ratio (uncompressed size / compressed size) than images that have frequently changing colours.

There are several ways of using RLE on bitmapped images: bit level RLE, byte level RLE and pixel level RLE.

Bit Level RLE (for black and white images)

Bit level RLE is effective when one bit is used to represent the colour of each pixel, ie in a monochrome image. Here a single byte represents the value of a pixel and the run length. Within the 8 bits, the left-most bit identifies the colour (eg 0 = white and 1 = black) the next 7 bits identify the run length (runs that are longer than 127 need to be broken down into a number of 127 long runs plus a run to represent the pixels 'left over').

The example below shows a section of 32 pixels from a monochrome bitmap:



The uncompressed representation would be

```
11111100000000001111111111111111
```

When compressed using bit level RLE, the first six pixels are black so the left-most bit in the first byte of the encoding would be a 1 (underlined in the example below) followed by the 7-bit binary number representing 6.

```
10000110
```

The next 10 pixels are white (represented by the underlined 0 in the example below) and so the next byte in the encoding would be:

```
00001010
```

The complete encoding including the final 16 black pixels is:

```
10000110 00001010 10010000
```

The original bitmap used 32 bits whereas the bit level RLE of the bitmap uses only 24 bits. This compression works due to the relatively long sequences of same colour pixels.

Byte level RLE (for 256-colour images)

Each pixel (colour) is represented by a single byte giving 256 possibilities. The colours themselves would be held within a colour palette table which would be stored in the image file. The colour palette is arbitrary and determined by factors such as hardware, image composition or file type. The image data is then represented by pairs of bytes representing the run length and then the index of the run colour in the colour palette table.

Example:



6 black, 10 yellow, 16 blue

If the colour table for the example above specified black as 16, yellow as 126 and blue as 20 then the corresponding bytes would be:

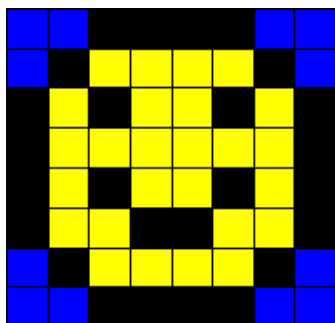
```
00000110 00010000 00001010 01111110 00010000 00010100
```

As with text compression a flag byte could be used to prevent negative compression.

Pixel level RLE (16.7 million colour images)

Each pixel is represented by three bytes, for an RGB bitmap. Each pair would consist of a run length byte, followed by the three bytes that represent the pixel colour.

The metadata for a bitmap will include the number of rows and number of columns (in this case 8x8), so it is safe for the RLE to 'run over' a row, ie reading from the top-left pixel, the image can be encoded as two blue pixels, four black pixels, three blue pixels, and so on.



RGB colour values			
	R	G	B
	0	0	255
	0	0	0
	255	255	0

This image uses three bytes per pixel and so the total UNCOMPRESSED file size (ignoring metadata) is $8 \times 8 \times 3 = 192$ bytes.

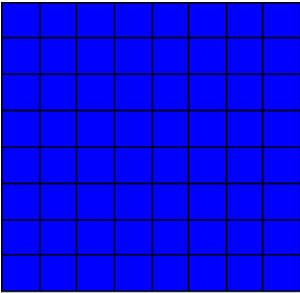
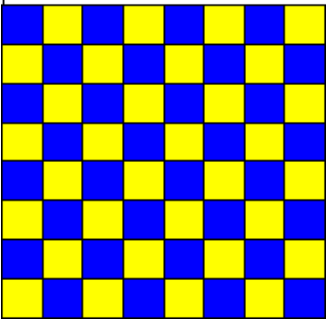
Example RLE for the 8 x 8 face shown above

For presentation purposes, the data is shown as three sets of 4 bytes per row, in the actual file this would simply be a continuous stream of data.

Count	R	G	B	Count	R	G	B	Count	R	G	B
2	0	0	255	4	0	0	0	3	0	0	255
1	0	0	0	4	255	255	0	1	0	0	0
1	0	0	255	1	0	0	0	1	255	255	0
1	0	0	0	2	255	255	0	1	0	0	0
1	255	255	0	2	0	0	0	6	255	255	0
2	0	0	0	1	255	255	0	1	0	0	0
2	255	255	0	1	0	0	0	1	255	255	0
2	0	0	0	2	255	255	0	2	0	0	0
2	255	255	0	1	0	0	0	1	0	0	255
1	0	0	0	4	255	255	0	1	0	0	0
3	0	0	255	4	0	0	0	2	0	0	255

Each cell in the table requires one byte hence the total bytes required for the RLE of the bitmap is $11 \times 12 = 132$ bytes. This results in a file size that is about two-thirds (actually 68.75%) of the original size.

Best and worst cases for RLE

<p>Best case</p> 	<p>The longer the run lengths the better the compression, therefore if we only had one colour we could specify the entire 64 pixel image with just 4 bytes.</p> <table border="1" data-bbox="679 465 1142 546"><thead><tr><th>Count</th><th>R</th><th>G</th><th>B</th></tr></thead><tbody><tr><td>64</td><td>0</td><td>0</td><td>255</td></tr></tbody></table> <p>Total bytes required = 4 bytes</p>	Count	R	G	B	64	0	0	255
Count	R	G	B						
64	0	0	255						
<p>Worst case</p> 	<p>The shorter the run lengths the less compression we achieve. The worst case is that the colour changes every pixel.</p> <p>Each pixel takes exactly 4 bytes EVERY time, because there are no runs (or alternatively you can think of all the runs as being of length 1).</p> <p>Total bytes required = $8 \times 8 \times 4 = 256$ bytes</p>								

File sizes

The actual file size of compressed data will often be larger than indicated by the examples shown in this document as there will also be metadata stored in the file (along with the character/pixel data).