Run-length encoding is a data compression algorithm that is supported by most bitmap file formats, such as **TIFF**, **BMP**, and **PCX**. RLE is suited for compressing any type of data regardless of its information content, but the content of the data will affect the compression ratio achieved by RLE. Although most RLE algorithms cannot achieve the high compression ratios of the more advanced compression methods, RLE is both easy to implement and quick to execute, making it a good alternative to either using a complex compression algorithm or leaving your image data uncompressed.

RLE works by reducing the physical size of a repeating string of characters. This repeating string, called a *run*, is typically encoded into two bytes. The first byte represents the number of characters in the run and is called the *run count*. In practice, an encoded run may contain 1 to 128 or 256 characters; the run count usually contains as the number of characters minus one (a value in the range of 0 to 127 or 255). The second byte is the value of the character in the run, which is in the range of 0 to 255, and is called the *run value*.

Uncompressed, a character run of 15 A characters would normally require 15 bytes to store:

```
AAAAAAAAAAAAAAAA
```

The same string after RLE encoding would require only two bytes:

```
15A
```

The 15A code generated to represent the character string is called an *RLE packet*. Here, the first byte, 15, is the run count and contains the number of repetitions. The second byte, A, is the run value and contains the actual repeated value in the run.
A new packet is generated each time the run character changes, or each time the number of characters in the run exceeds the maximum count. Assume that our 15-character string now contains four different character runs:

AAAAAAbbbXXXXXt

Using run-length encoding this could be compressed into four 2-byte packets:

6A3b5X1t

Thus, after run-length encoding, the 15-byte string would require only eight bytes of data to represent the string, as opposed to the original 15 bytes. In this case, run-length encoding yielded a compression ratio of almost 2 to 1.

Long runs are rare in certain types of data. For example, ASCII plaintext seldom contains long runs. In the previous example, the last run (containing the character t) was only a single character in length; a 1-character run is still a run. Both a run count and a run value must be written for every 2-character run. To encode a run in RLE requires a minimum of two characters worth of information; therefore, a run of single characters actually takes more space. For the same reasons, data consisting entirely of 2-character runs remains the same size after RLE encoding.

In our example, encoding the single character at the end as two bytes did not noticeably hurt our compression ratio because there were so many long character runs in the rest of the data. But observe how RLE encoding doubles the size of the following 14-character string:

Xtmprsqntwlfb

After RLE encoding, this string becomes:

1X1t1mlp1rl1s1q1l1n1l1t1w1l1f1b

RLE schemes are simple and fast, but their compression efficiency depends on the type of image data being encoded. A black-and-white image that is mostly white, such as the page of a book, will encode very well, due to the large amount of contiguous data that is all the same color. An image with many colors that is very busy in appearance, however, such as a photograph, will not encode very well. This is because the complexity of the image is expressed as a large number of different colors. And because of this complexity there will be relatively few runs of the same color.

Variants on Run-Length Encoding

There are a number of variants of run-length encoding. Image data is normally run-length encoded in a sequential process that treats the image data as a 1D stream, rather than as a 2D map of data. In sequential processing, a bitmap is encoded starting at the upper left corner and proceeding from left to right across each scan
line (the X axis) to the bottom right corner of the bitmap. But alternative RLE schemes can also be written to encode data down the length of a bitmap (the Y axis) along the columns), to encode a bitmap into 2D tiles or even to encode pixels on a diagonal in a zig-zag fashion. Odd RLE variants such as this last one might be used in highly specialized applications but are usually quite rare.

**Figure 9-2: Run-length encoding variants**

Another seldom-encountered RLE variant is a lossy run-length encoding algorithm. RLE algorithms are normally lossless in their operation. However, discarding data during the encoding process, usually by zeroing out one or two least significant bits in each pixel, can increase compression ratios without adversely affecting the appearance of very complex images. This RLE variant works well only with real-world images that contain many subtle variations in pixel values.

Make sure that your RLE encoder always stops at the end of each scan line of bitmap data that is being encoded. There are several benefits to doing so. Encoding only a simple scan line at a time means that only a minimal buffer size is required. Encoding only a simple line at a time also prevents a problem known as cross-coding.
Cross-coding is the merging of scan lines that occurs when the encoded process loses the distinction between the original scan lines. If the data of the individual scan lines is merged by the RLE algorithm, the point where one scan line stopped and another began is lost or, at least, is very hard to detect quickly.

Cross-coding is sometimes done, although we advise against it. It may buy a few extra bytes of data compression, but it complicates the decoding process, adding time cost. For bitmap file formats, this technique defeats the purpose of organizing a bitmap image by scan lines in the first place. Although many file format specifications explicitly state that scan lines should be individually encoded, many applications encode image data as a continuous stream, ignoring scan-line boundaries.

Have you ever encountered an RLE-encoded image file that could be displayed using one application but not using another? Cross-coding is often the reason. To be safe, decoding and display applications must take cross-coding into account and not assume that an encoded run will always stop at the end of a scan line.

When an encoder is encoding an image, an end-of-scan-line marker is placed in the encoded data to inform the decoding software that the end of the scan line has been reached. This marker is usually a unique packet, explicitly defined in the RLE specification, which cannot be confused with any other data packets. End-of-scan-line markers are usually only one byte in length, so they don't adversely contribute to the size of the encoded data.

Encoding scan lines individually has advantages when an application needs to use only part of an image. Let's say that an image contains 512 scan lines, and we need to display only lines 100 to 110. If we did not know where the scan lines started and ended in the encoded image data, our application would have to decode lines 1 through 100 of the image before finding the ten lines it needed. Of course, if the transitions between scan lines were marked with some sort of easily recognizable delimiting marker, the application could simply read through the encoded data, counting markers until it came to the lines it needed. But this approach would be a rather inefficient one.

Another option for locating the starting point of any particular scan line in a block of encoded data is to construct a scan-line table. A scan-line table usually contains one element for every scan line in the image, and each element holds the offset value of its corresponding scan line. To find the first RLE packet of scan line 10, all a decoder needs to do is seek to the offset position value stored in the tenth element of the scan-line lookup table. A scan-line table could also hold the number of bytes used to encode each scan line. Using this method, to find the first RLE packet of scan line 10, your decoder would add together the values of the first nine elements of the scan-line table. The first packet for scan line 10 would start at this byte offset from the beginning of the RLE-encoded image data.

Bit-, Byte-, and Pixel-Level RLE Schemes
The basic flow of all RLE algorithms is the same.

**Basic run-length encoding flow**

The parts of run-length encoding algorithms that differ are the decisions that are made based on the type of data being decoded (such as the length of data runs). RLE schemes used to encode bitmap graphics are usually divided into classes by the type of atomic (that is, most fundamental) elements that they encode. The three classes used by most graphics file formats are bit-, byte-, and pixel-level RLE.

*Bit-level RLE* schemes encode runs of multiple bits in a scan line and ignore byte and word boundaries. Only monochrome (black and white), 1-bit images contain a sufficient number of bit runs to make this class of RLE encoding efficient. A typical bit-level RLE scheme encodes runs of one to 128 bits in length in a single-byte packet. The seven least significant bits contain the run count minus one, and the most
significant bit contains the value of the bit run, either 0 or 1. A run longer than 128 pixels is split across several RLE-encoded packets.

**Byte-level RLE** schemes encode runs of identical byte values, ignoring individual bits and word boundaries within a scan line. The most common byte-level RLE scheme encodes runs of bytes into 2-byte packets. The first byte contains the run count of 0 to 255, and the second byte contains the value of the byte run. It is also common to supplement the 2-byte encoding scheme with the ability to store literal, unencoded runs of bytes within the encoded data stream as well.

In such a scheme, the seven least significant bits of the first byte hold the run count minus one, and the most significant bit of the first byte is the indicator of the type of run that follows the run count byte. If the most significant bit is set to 1, it denotes an encoded run. Encoded runs are decoded by reading the run value and repeating it the number of times indicated by the run count. If the most significant bit is set to 0, a literal run is indicated, meaning that the next run count bytes are read literally from the encoded image data. The run count byte then holds a value in the range of 0 to 127 (the run count minus one). Byte-level RLE schemes are good for image data that is stored as one byte per pixel.

**Pixel-level RLE** schemes are used when two or more consecutive bytes of image data are used to store single pixel values. At the pixel level, bits are ignored, and bytes are counted only to identify each pixel value. Encoded packet sizes vary depending upon the size of the pixel values being encoded. The number of bits or bytes per pixel is stored in the image file header. A run of image data stored as 3-byte pixel values encodes to a 4-byte packet, with one run-count byte followed by three run-value bytes. The encoding method remains the same as with the byte-oriented RLE.

**Figure 9-4: Bit-, byte-, and pixel-level RLE schemes**
It is also possible to employ a literal pixel run encoding by using the most significant bit of the run count as in the byte-level RLE scheme. Remember that the run count in pixel-level RLE schemes is the number of pixels and not the number of bytes in the run.

Earlier in this section, we examined a situation where the string "Xtmprszntwlfb" actually doubled in size when compressed using a conventional RLE method. Each 1-character run in the string became two characters in size. How can we avoid this negative compression and still use RLE?
Normally, an RLE method must somehow analyze the uncompressed data stream to determine whether to use a literal pixel run. A stream of data would need to contain many 1- and 2-pixel runs to make using a literal run efficient by encoding all the runs into a single packet. However, there is another method that allows literal runs of pixels to be added to an encoded data stream without being encapsulated into packets.

Consider an RLE scheme that uses three bytes, rather than two, to represent a run. The first byte is a *flag value* indicating that the following two bytes are part of an encoded packet. The second byte is the *count value*, and the third byte is the *run value*. When encoding, if a 1-, 2-, or 3-byte character run is encountered, the character values are written directly to the compressed data stream. Because no additional characters are written, no overhead is incurred.

**RLE scheme with three bytes**

![Diagram of RLE scheme with three bytes]

When decoding, a character is read; if the character is a flag value, the run count and run values are read, expanded, and the resulting run written to the data stream. If the character read is not a flag value, it is written directly to the uncompressed data stream.

There are two potential drawbacks to this method:

- The minimum useful run-length size is increased from three characters to four. This could affect compression efficiency with some types of data.
- If the unencoded data stream contains a character value equal to the flag value, it must be compressed into a 3-byte encoded packet as a run length of one. This prevents erroneous flag values from occurring in the compressed data stream. If many of these flag value characters are present, poor compression
will result. The RLE algorithm must therefore use a flag value that rarely occurs in the uncompressed data stream.

**Vertical Replication Packets**

Some RLE schemes use other types of encoding packets to increase compression efficiency. One of the most useful of these packets is the *repeat scan line packet*, also known as the *vertical replication packet*. This packet does not store any real scan-line data; instead, it just indicates a repeat of the previous scan line. Here's an example of how this works.

Assume that you have an image containing a scan line 640 bytes wide and that all the pixels in the scan line are the same color. It will require 10 bytes to run-length encode it, assuming that up to 128 bytes can be encoded per packet and that each packet is two bytes in size. Let's also assume that the first 100 scan lines of this image are all the same color. At 10 bytes per scan line, that would produce 1000 bytes of run-length encoded data. If we instead used a vertical replication packet that was only one byte in size (possibly a run-length packet with a run count of 0) we would simply run-length encode the first scan line (10 bytes) and follow it with 99 vertical replication packets (99 bytes). The resulting run-length encoded data would then only be 109 bytes in size.

If the vertical replication packet contains a count byte of the number of scan lines to repeat, we would need only one packet with a count value of 99. The resulting 10 bytes of scan-line data packets and two bytes of vertical replication packets would encode the first 100 scan lines of the image, containing 64,000 bytes, as only 12 bytes--a considerable savings. Illustrates 1- and 2-byte vertical replication packets.

**RLE scheme with 1- and 2-byte vertical replication packets**

Unfortunately, definitions of vertical replication packets are application dependent. At least two common formats, *WordPerfect Graphics Metafile (WPG)* and *GEM Raster (IMG)*, employ the use of repeat scan line packets to enhance data compression.
performance. WPG uses a simple 2-byte packet scheme, as previously described. If the first byte of an RLE packet is zero, then this is a vertical replication packet. The next byte that follows indicates the number of times to repeat the previous scan line.

The GEM Raster format is more complicated. The byte sequence, 00h 00h FFh, must appear at the beginning of an encoded scan line to indicate a vertical replication packet. The byte that follows this sequence is the number of times to repeat the previous scan line minus one.

NOTE:

Many of the concepts we have covered in this section are not limited to RLE. All bitmap compression algorithms need to consider the concepts of cross-coding, sequential processing, efficient data encoding based on the data being encoded, and ways to detect and avoid negative compression.

RLE c++ Code sample:

```cpp
// RLE algorithm.cpp : Defines the entry point for the console application.
//
#include "stdafx.h" //work with txt files only
#include "iostream"
#include "fstream"
using namespace std;

struct compress
{
    char runValue;
    int counter;
    compress(int x=0, int y=1):runValue(x),counter(y){} //constructor
};

int _tmain(int argc, _TCHAR* argv[])
{
    fstream infile1, infile2, outFile;
    struct compress zip;
    char cur,
    char next;
    char fName[100]="";

    cout<<"please enter file name :";
    cin>>fName;

    inFile1.open(fName, ios::in);
    inFile1.unsetf(ios::skipws);
    inFile2.open(fName, ios::in);
    inFile2.unsetf(ios::skipws);
    outFile.open("compressed.txt", ios::out);
    outFile.unsetf(ios::skipws);
    while(1){
    }
```
infile1>>cur;
if(infile1.fail()) break;
infile2>>next;
infile2>>next;
if(infile2.fail()) break;
while(1)
{
    if(cur!=next)
    {
        outfile<<"1"<<cur; // handled error
        infile1>>cur;
        infile2>>next;
        if(infile2.fail()) break;
    }
    if(cur==next)
    {
        while(cur==next)
        {
            zip.counter++;
            infile1>>cur;
            infile2>>next;
            if(infile2.fail()) break;
        }
        zip.runValue=cur;
        outfile<<zip.counter<<zip.runValue;
        zip.counter=1;
        infile1>>cur;
        infile2>>next;
        if(infile2.fail()) break;
    }
    // end of first while
    // end of file
    infile1.close();
    infile2.close();
    outfile.close();
    cout<<"compression operion completed.\n";
    return 0;
}