Data compression is an easy concept to understand. It's simply any technique that lets you take information in one format and convert it to a format that requires less memory or disk space. However, data compression is a relatively new topic (it was impractical before computers came along), and it can be very confusing since many compression algorithms have their basis in mathematics and information theory.

As we examine the topic of data compression, we'll cover a bit of background about the two types of data compression techniques. Then we'll look at a method called Run Length Encoding (RLE). While RLE is not as sophisticated as other methods currently in use, it's a good scheme to study for newcomers to the subject of data compression. Finally, we'll present a program that implements the RLE algorithm.

Two data compression types: logical and physical

Almost all data compression techniques fall into one of two basic categories: logical or physical. The primary distinction between these two categories is that logical compression techniques know in advance and take advantage of the nature of the input data stream—physical compression techniques do not.

Logical compression

To get a feeling for how logical compression works, let's look at two logical compression techniques. First, let's suppose you want to compress a BASIC database file. It's pretty good bet that not all the fields in each record are filled. With this in mind, you can replace unused byte positions with special compression characters. This type of compression is cheap, although not universally applicable because it requires prior knowledge of the file structure.

An example of another logical compression technique is compression. ASCII files that contain no extended ASCII characters—those characters with ordinal values greater than 127. In this case, the right bit of each character is 0, so you can shift the entire input data stream by 1 bit for each character, thus reducing the file size by 12.5 percent. Again, you must know beforehand that the input data stream contains no extended ASCII characters. You can use numerous methods to determine whether each character is a special compression character or not.

Physical compression

Unlike logical compression, physical compression requires no prior knowledge of the data in the input stream and thus has much broader application. We'll concentrate on physical compression techniques in the remainder of this article.

Null suppression was one of the first physical compression techniques to find widespread use. This technique scans the input stream for contiguous blocks of blank or null characters and replaces each block with one ordered pair of characters—a special compression character and a count character. The IBM PBBC/BBCN transmission protocol still employs this technique to increase transmission throughput.

You can explore several other physical compression techniques, including Bmpacking, Run Length Encoding, Huffman Encoding, and Adaptive Encoding. Just as with sorting algorithms, compression techniques can be optimized performance when you use them correctly and deleterious performance when you use them incorrectly. Correct—on

In this issue

• Saving space with data compression ..........1
• Accent yourself ..................................5
• An input problem on fast machines ..........6
• Writing your own matter() and _matheend() functions ..............................................8
• The site not-time math library exceptions ....8
• New, improved probability functions ..........12
• Source code listings ..................................13

A Publication of The Gold Group
Saving space with data compression

this case—means choosing the proper compression tech-
nique that yields the smallest file in the least amount of time.
On occasion, the best method is actually a combination of
techniques. That is, you pipe the output from one method
into the input of another method. Although the end results
are mediocre, they can be very attractive.

You can use several metrics to evaluate the effective-
ness of data compression techniques. One typical measure
is compression ratio, which is the compression ratio—
the length of the input stream divided by the length of the
output stream. A high compression ratio (greater than 1.5)
means the data are highly compressed; a low compression
ratio (1.0 to 1.5) means the data are somewhat compressed;
a compression ratio less than or equal to 1 means compression
failed, and the output stream is greater than or the same size
as the input stream. Typically, you must balance your desired
compression ratio against the execution speed of the al-
gerine; since, as a general rule, execution time increases along
with the compression ratio.

Run Length Encoding

Now that we've covered the basics of data compression, let's
examine an actual compression technique called Run Length
Encoding (RLE). RLE is a variant of the Null suppression
algorithm. The RLE algorithm scans the input data stream
for runs of matching characters. Upon finding four or more
equivalent characters in a row—a run— it replaces them with
a special compression character and a count character.
The RLE algorithm responds with runs of four or more
characters because it can save space by encoding runs of
only two or three characters. This limit exists because the
compression technique requires exactly three characters to
represent a run in compressed form.
Before looking at any code, let's manually work through an example. Suppose you have the data

A, B, C, D, E, F, E, E, E, E

As you can see, the last two groups of characters—the D and E groups—both contain four or more identical characters, which makes them candidates for RLE compression. After applying RLE with the special compression character 0x00, the compressed output appears as

A, B, C, D, 0x00, 2, E, 0x00, 4

Notice that RLE replaces each compressible run with a three-character sequence made up of the original character, the compression character 0x00, and the number of additional occurrences of the original character in the run. In this case, RLE reduced the number of bytes from 15 to 6, and realized a 60 percent decrease in size and a compression ratio of 2.5:1.

You can apply this method to all types of data because it makes no assumptions about the type or format of the data. However, you might wonder what happens if a 0x00 character—the compression character in this example—actually appears in your file as data. You can handle this simply by outputting the 0x00 character followed by a 0 count byte. This works because the RLE decompression algorithm outputs a 0x00 character whenever it finds a 0 byte following a 0x00 character in the compressed file.

An example compression program: RLE-C

To demonstrate how RLE data compression works, we coded the RLE-C program in Listing A on page 19. This program accepts the names of the input and output files as command line arguments and compresses the data stream from the input file into the output file. As a precaution, we coded RLE-C to verify that the output file already exists. RLE-C reports the error and terminates.

RLE-C contains three functions: main(), compress(), and uncompress(). After the main() function opens the input and output files, it remains in a while loop that calls compress() once for each character in the input stream. compress() is a state machine with five unique states: FirstLine, ShortChar, ShortRun, LongChar, and LongRun. For each state, compress() takes a different set of actions and returns to the caller. Moreover, each time compress() returns, it emits a new value that the caller will in turn write to the output file (except, of course, in the case of EOF).

The FirstLine state

The FirstLine state calls uncompress()—or, when the value of State equals FstLine—compress() reads a character from the input file into the variable Prev and sets State to ShortChar. Then, compress() returns Prev to the while loop (in main()), which writes Prev to the output file. compress() enters the FirstLine state only at the beginning of the input file and each time after reading a 0x00 character from the input file.

The LongChar state

compress() enters the LongChar state each time it moves a long character from the input stream directly to the output stream. The LongChar state determines whether this long character is the first in a run of four or more identical characters (a large run), the first in a run of two or three characters (a small run), or not part of a run at all. Since neither is the most complex state (in compress()), we're going to examine its instructions in detail.

The first step (compress()) takes in this state is testing Prev's (the previous character's) value to see if it was the RepeatCharacter (0x00) or a regular character. If Prev equals Repeat and State equals LongChar, compress() knows that the previous iteration stored Prev as a byte, not as the RLE repeat identifier. In this case, compress() identifies the 0x00 character as data by emitting a 0.

If Prev is a regular character (not equal to Repeat), compress() must determine if the next character from the input file matches Prev. If the characters don't match, compress() simply returns the newly read character and remains in the LongChar state. However, if the characters match, compress() must determine if any preceding characters matched—in other words, if there is a run of repeating characters in the file.

When compress() encounters a run of characters, it determines if it's a small run (four or less characters) or a large run (four or more characters). If it's a small run, compress() sets State to ShortRun and emits the character in Prev. However, if the run of four or more characters, compress() sets State to LongRun and emits a Repeat character.

The SmallRun state

Whenever compress() finds a run of two or three matching characters, it sets State to ShortRun and emits the number of repeating characters. It then enters the ShortRun state. In the ShortRun state, compress() examines the variable Prev and determines if it is a regular character or a Repeat character. compress() switches to the ShortChar state if Prev is a regular character, or the ShortRun state if Prev is a Repeat character. compress() enters the ShortChar state if Prev is equal to Repeat, or the ShortRun state if Prev is a regular character.

The SmallChar state

Whenever compress() finds a run of two or three matching characters, it sets State to ShortChar and emits the number of repeating characters. It then enters the ShortChar state. In the ShortChar state, compress() examines the variable Prev and determines if it is a regular character or a Repeat character. compress() switches to the ShortChar state if Prev is a regular character, or the ShortRun state if Prev is a Repeat character. compress() enters the ShortRun state if Prev is equal to Repeat, or the ShortChar state if Prev is a regular character.

The SmallRun state

Whenever compress() finds a run of two or three matching characters, it sets State to ShortRun and emits the number of repeating characters. It then enters the ShortRun state. In the ShortRun state, compress() examines the variable Prev and determines if it is a regular character or a Repeat character. compress() switches to the ShortChar state if Prev is a regular character, or the ShortRun state if Prev is a Repeat character. compress() enters the ShortChar state if Prev is equal to Repeat, or the ShortRun state if Prev is a regular character.
to have the largest possible range of output bits, you should normalize input counts before subtracting). In this way, you can represent repeat counts from 3 to 255 by mapping them to the range 0 to 255. Of course, you must account for this in the decompression algorithm by adding 2 to each. Before expanding the run back to its original length.

The SendNextChar state

Compress() enters the SendNextChar state after existing a previous three-character encoded run. At this point, the variable C holds the last character read from the input stream or the character ending the run by non-matching. For the SendNextChar state, Compress() sets state to Sendchar, Penc = C, and prints the character in C. On the subsequent iteration, Decompress() reverts C to the last character matching Penc.

The program continues switching between states according to the above rules until it finds the end of the input file. At that time, it closes the input and output files and exits.

An example decompression

program UNRLECl

Similar to RLEC, the UNRLECl program is listed 18 on page 14. It accepts the input and output filenames as command line arguments and decompresses the data stream from the input file into the output file. Also like RLECl, UNRLECl won't overwrite existing files. By simply pasting all UNRLECl, you can tell that decompression is a much simpler task than compression.

UNRLECl consists of an encode() function and a Decompress() function, which performs all decompression tasks. After opening the input and output files, encode() encodes the input stream into the output stream. Decompress() is a state machine just like the Compress() function in RLECl. However, Decompress() is much simpler and has only two states: FirstTime and RepeatChar.

Decompress() starts in the FirstTime state, writing all input characters except Input 00001 directly to the output file. When Decompress() encounters a repeat character, it switches to the RepeatChar state. The RepeatChar state reads the next input character to see if it's a '0'. If it is, Decompress() writes a 000001 to the output file, otherwise, it returns to the RepeatChar state and continues reading the input file. In this way, Decompress() always switches back to the FirstTime state after processing the RepeatChar state.

Some afterthoughts

The RLE implementation we've presented in this article—while very effective—is just one of many RLE implementations used throughout the computer industry. Some use repeat characters other than 00001, some have special escape characters with application-specific meanings, and some compress data streams built conceptually of 4-bit nibbles or 16-bit words instead of 8-bit bytes.

Our algorithm's strong point is the way it differentiates between compressible runs (four or more characters) and non-compressible runs (two or three characters). Consequently, it can achieve very high compression ratios. However, there is a remote possibility you may encounter files with high concentrations of 00001 characters. Since each 00001 in the input stream results in two characters in the output file, such files may result in low compression ratios.

If you're a real stickler for performance, you may alter RLECl and UNRLECl to monitor the populations of all 256 ASCII characters in the input stream and dynamically choose the repeat character as the one occurring least frequently in the input stream. As long as the compression and decompression algorithms agree on when to switch from one repeat character to another (based on the data in the original input stream), such an approach will give you a nearly bullet-proof RLE algorithm. Of course, this enhancement will increase your algorithm's average compression rate as the expense of execution time. Both end you have to decide when enough is enough—do the advantages gained by further enhancements to the algorithm justify slower program execution?

Conclusion

Since understanding the concepts behind data compression algorithms is often easier than implementing them, we hope you can use RLECl and UNRLECl to begin your exploration of data compression techniques.

Gary Condon is the president of Infinity Design Concepts Inc (IDC) and codveloper of the ZIP file format. For information on IDC's products—which include NARC1M, UNIXRLECl, and ZM—call (949) 360-2314.

Got a hot tip?

Do you have a tip, trick, or code snippet you’d like to share with other readers? If so, send it in. We may use your idea as the basis for an article. If we publish your idea, we'll give you a byline and pay you $25 or more, depending on its scope and usefulness. Send all submissions, which become property of The Cebbs Group, to:

- **Editor-in-Chief**
  - Inside-Microsoft C
  - The Cebbs Group
  - 9420 Bunaan Parkway, Suite 300
  - Louisville, KY 40022