Chapter 2

Image Compression

2.1 Introduction

The advantages of using data compression are pretty straightforward. It lets you store more stuff in the same space, and it lets you transfer that stuff in less time, or with less bandwidth. I used the word "stuff" and not "information" because, for pure data compression algorithms, the compressed file contains exactly the same amount of information as the original one. Of course, this is using the term "information" in a mathematical sense (normal people wouldn't consider a ZIP-compressed text file particularly "informative"). There is a distinction between information and data. For example, if someone sends you the same e-mail twice, you have two e-mails' worth of data, but just one e-mail's worth of information. Actually, you might consider that, by receiving the same e-mail twice, you have learned something new, so the repetition itself might be considered to add a little bit of information.

The fundamental idea behind digital data compression is to take a given representation of information (a chunk of binary data) and replace it with a different representation (another chunk of binary data) that takes up less space (space here being measured in binary digits, better known as bits), and from which the original information can later be recovered. If the recovered information is guaranteed to be exactly identical to the original, the compression method is described as "lossless". If the recovered information is not guaranteed to be exactly identical, the compression method is described as "lossy".

Let's start by looking at a simple lossless compression algorithm, called "run-length encoding", or RLE.

2.1.1 Run-length encoding

Let's say we have a text file consisting of the following characters:

```
AAAAAAAAAABBCDEEFFFFFFFFFFFFFFGGGGGGGGGGGGGGGGG
```

Run-length encoding replaces "runs" (that is, sequences of identical characters) with a single character, followed by the "length of the run" (the number of characters in that sequence), or vice-versa (first the length and then the character, the order isn't important as long as it's always the same). If we apply this algorithm to the text file above, we'll get something like this:

```
10A 2B 1C 1D 2E 13F 17G
```

We have just gone from 46 characters to 17, and that's using a plain text representation for the lengths. I used spaces between the pairs, and different colors for the lengths and characters, to make it easier to read, but naturally these wouldn't be necessary for a computer. In computer-generated files, each character typically requires one byte to be stored (in fact, "character" is commonly used to describe any value that's exactly one byte long, even if it's not in a text file), and a single byte can also be used to represent run lengths up to 255 characters long (256, if we also use the value "zero"), so the above could be encoded in just 14 bytes (7 lengths and 7 characters, each pair requiring 2 bytes).

But you may have noticed that not all sequences of characters became shorter using this new representation. For example, "BB" became "2B", which takes up the same two bytes. And "C" actually doubled in size (from one to two bytes), becoming "1C". It probably also occurred to you that runs of more than two identical characters are rare in human languages. So if you take a real sentence, such as:

```
Mary had a little lamb.
```

And run it through our RLE "compressor", you get something like this:

```
1M 1a 1r 1y 1  1h 1a 1d 1  1a 1  1L 1i 2t 1l 1e 1  1L 1a 1m 1b 1.
```

In other words, the only part that doesn't get bigger is the "tt" in "little", and even that just manages to stay the same size (2 bytes). So we've just gone from 23 bytes (23 characters) to 44 (remember, spaces in the original sentence must be encoded, too, otherwise they won't be there when we decompress it). Not exactly brilliant, as far as "compression" goes. However, if we have, say, an invoice, where a lot of lines are similar to this:

```
Item X ..........\ $100.00
```

Then maybe RLE can do something for us. After all, there are 60 dots between the item name and its price. Although "Item X" would double in size (from 6 to 12 bytes), and "$100.00" would also grow (though just from 7 to 10 bytes, since the "00" sequences wouldn't grow), the sequence of dots would go from 60 to 2 bytes. In other words, the complete line would go from 73 bytes to 24. That's a reduction to about 1/3rd of the original size. But if the same file contained a couple of paragraphs of "normal" text, with few or no repetitions, all our gains would be lost. So ideally, we'd like to have a way to compress only some parts of the file, while leaving others uncompressed.

We can do this quite simply by defining a special "marker" that indicates the beginning or the end of each block of compressed text. For example, let's say we use an asterisk as our marker, and determine that compression is off by default and will only be turned on when there are three or more identical characters in a row. We now get (after compression):

```
X*[60].*\$100.00
```

That's just 17 bytes (the square brackets indicate that the number between them is stored as a value, and not as a sequence of decimal digits; the value "60" can be represented by a single byte, in a computer file). And if we run the previous sentence through our "enhanced RLE" compressor, we now get:

```
Mary had a little lamb.
```

There's no sequence that would benefit from RLE compression, so our compressor leaves it alone. No gain, but no loss, either. It's starting to look as if RLE can, at the very least, "pay for itself" (i.e., not make files bigger). But what if the original file contained an asterisk? If we just pass that asterisk directly into our compressed file, like any other character, that's going to confuse our decompressor (it's going to assume that the compression has been toggled, and misinterpret the rest of the file). We can deal with this in several ways, but the most generic one is to define a new rule: if our marker appears in the source data, that part of the data is always encoded (i.e., it's treated as compressible data). So, if we have the following source data:

`Figaro was the city's factotum*.`

And run it through our compressor, we get:

`Figaro was the city's factotum*[1]**.`

This increased the size from 32 bytes to 35, so, ideally, we would instead pick a different character to use as our "marker". As long as the encoder and decoder both use the same character and know the same rules, it will work. We could find out what is the rarest character in written English, and always use that, making the situation above extremely rare. But we might want to use our compressor on other data types (not just English text), so ideally it would scan the source file, find out the rarest character in it, and use that as the special marker (and list it in the file header, so the decoder would know which character to look for). Note that the method described above works for any number for markers in a row. For example, if you have a sequence of three markers (here represented by asterisks) in the source file:

***

And the compression toggle was already "on", those would be encoded as:

[3]

If the compression toggle was "off", then it would be turned on by one asterisk, and the sequence would be encoded as:

**[3]*

...optionally adding another marker at the end to turn compression back off (if the following data wasn't compressible). The following examples should clarify how the data would be encoded if the marker sequence appeared in the middle of compressible or incompressible data: The sequence `AAA**DDDD` would become `*[3]A[2]*[4]D`. The sequence `ABC**DEFG` would become `ABC*[2]**DEFG`.

How does the decoder know that the asterisk inside the compressed block is a character from the original text, and not another compression toggle marker? Because the previous byte was a run length. Counting from a marker, odd bytes are run lengths and even bytes are characters from the original data. The length and character always appear as pairs, so an asterisk can only be considered as a compression toggle marker if there is an even number of bytes between it and the previous marker. This means the decompressor will never confuse a "character" with a marker, even if they have the same byte value. It could, however, confuse a run length with a marker, as described below. We still have to deal with the case of a run length being identical to the value of our "special marker" byte. For example, if our marker was an asterisk, that has the ASCII byte value 42, then any sequence of

42 characters would cause an asterisk (i.e., a byte with the value 42) to be inserted in the compressed file. Later, the decoder would have no way to distinguish this asterisk from a "real" marker, since both would be bytes with the value 42, and both could appear in the same position relative to the previous marker. For example, if there was a sequence of 42 identical characters in our source file:

AAAHH !!???

This would be encoded as:

*[3]A[2]H [42]![3]?

However, the byte [42] happens to be the ASCII code for the asterisk, so the sequence above would be seen by the decoder as:

[3]A[2]H![3]?

In other words, the decoder would interpret the counter [42] as a compression toggle, and treat anything after it as uncompressed data, failing to decompress the exclamation and question marks, and adding a byte with the value 3 between them (a byte with the value 3 is actually interpreted as "end of text" by some text editors, which would make things even more confusing). To deal with this, the encoder needs to translate all run-length values (ex., using an indexed table, or some rule that the decoder recognizes), so that the run length of 42-character sequences is represented by another byte value (ex., zero). This means that although a byte can have 256 different values, our run length bytes can only use 255 different values, because one is reserved as our "special marker". This might seem like a small detail, but, if we ignored it, some files created by our compressor would later be impossible to decompress correctly. Note...

In fact, a byte with the value zero (often called a NULL) is usually an excellent choice as a default marker, since we would probably never want to use zero as a run length anyway. However, if the source file we are trying to process contains a lot of null bytes, we might end up wasting a significant amount of space encoding those, so scanning the source for the rarest character is still the preferred solution. There is one last optimisation worth mentioning, which is related to sequences of exactly two identical characters. Since those sequences always take up two bytes (ex., AA or [2]A), the decision to compress them or not should (ideally) be based on whether they appear immediately after compressible data or not. For example: ABCDZZ would be encoded as ABCDZZ (i.e., the "ZZ" sequence would be treated as uncompressible). AAAAZZ be encoded as *[4]A[2]Z (i.e., the "ZZ" sequence would be treated as compressible). Although this makes no difference to the size of that sequence itself, it avoids adding unnecessary markers to define the start or end of compressed blocks.

Let's look at what we've achieved so far: we started with an algorithm that looked promising, realised it was no good for text (most text files would double in size), isolated the problem (normal text rarely has more than two identical letters in a row), came up with a solution (the marker to divide the data in compressed and non-compressed blocks), added a way to encode the marker itself when it appears in the source data, and added two extra optimisations (use the rarest character as the marker, and avoid needless compression toggles in the case of 2-character sequences).

Challenge Can you write an implementation of RLE?

Still, all the optimizations and special cases we added above were merely a way to

avoid making the "compressor" increase the size of the files. The actual compression (i.e., reduction in size) will only work when there are sequences of more than three identical characters, and in normal text that's relatively rare. Most large text files do have enough such sequences (ex., of spaces, or linefeed characters) to let our "enhanced RLE" compressor save a few bytes, but a 2% or 3% compression factor probably isn't worth the hassle of compressing and decompressing every single file whenever we want to use it. So, is RLE useless, except as a case study? No. There is one type of data where very long sequences of identical values are common. Images, specifically about high-contrast images with few colours, such as schematic drawings, logos, text scans (including most faxes) and cartoons. It's also common, in some Computer Graphics animations, to use a plain background (to be replaced later), meaning that hundreds or even thousands of sequential pixels have exactly the same colour.

Although another "family" of algorithms (which will be mentioned later) can provide compression at least as good as RLE even in those situations, RLE has the advantage of being very simple, and therefore very fast to execute. The TARGA (.TGA extension) image file format can use RLE compression, and is very popular with animators, since RLE-compressed TGA sequences can often be read from disk, decompressed, and shown on screen faster than uncompressed images. This is because CPUs and RAM are so much faster than hard drives that it's quicker to load, say, 500 kB of compressed data and decode it than it is to load 1000 kB of uncompressed data. Although a more advanced compression algorithm might get the size of that image down even further (ex., to 300 kB), the extra complexity would require more CPU cycles to decompress, resulting in overall lower performance. Also, virtually all image editing applications can read and write TGA files, which makes them a good choice for transferring frames from one program to another. Unfortunately some programs only support uncompressed TGA files.

Other image file formats that can use RLE compression include BMP, PCX, PackBits (a TIFF sub-format) and ILBM (an old format, used mainly in Amiga computers). Fax machines also use RLE (combined with another technique called entropy coding) to compress data before transmitting it.

Pixel color information is generally stored as a sequence of three values, representing the amounts of red, green and blue (RGB) that define it. Black is usually represented as [0] [0] [0], while white is represented as [255] [255] [255] (this is because 255 is the highest number that can be stored in a single byte; you can think of it as "100%"). Red is represented as [255] [0] [0], green is represented as [0] [255] [0], orange is represented as [255] [128] [0], and so on. If you're familiar with digital colour representation this should be trivial, and if you're not, you don't really need to worry about it now. The only thing to remember is that (in most digital image formats, including TGA) each pixel's colour is represented not by one, but by three numbers.

In the case of images with built-in transparency data, each pixel is represented by four values, where the fourth (called alpha) defines the opacity of that pixel. Here we will assume the image has no opacity channel, but the same principles would apply if it did (with 4 channels instead of 3). The alpha channel is usually the easiest one to compress, since most pixels tend to be 100% opaque or 100% transparent.

Figure 2.1: A sequence of orange pixels

Remember how RLE works? It "compresses" sequences of identical values. So what happens if we have, say, a completely orange image as the one presented in figure **??** presented in page 52? Each pixel will be represented by a sequence of 3 bytes with the values 255, 128 and 0 (or similar ones, depending on the exact tone of orange). [255][128][0][255][128][0] ... etc. This sequence will repeat itself thousands of times (307200 times, for a 640x480 image), but there will never actually be a sequence of three or more identical values in our data. In other words, our RLE algorithm won't do anything to it.

So, what can we do to make it work? We could rewrite the algorithm to use sequences of 24 bits (3 bytes) instead of 8 bits (1 byte), but there is a simpler (and generally more efficient) solution. Instead of sorting the pixel color data by pixel, we sort it by channel (a channel is each of the color components; in this case red, green and blue). So, instead of having a file with 307200 repetitions of [255] [128] [000], we have a file with 307200 repetitions of [255], followed by 307200 repetitions of [128], followed by 307200 repetitions of [0]. It can't encode runs of more than 255 identical values, but each time one of those runs is encoded, it shrinks from 255 bytes to 2. Each of the three sequences of 307200 identical values can be encoded in just 2410 bytes. The full pixel data goes from 900kB to just over 7kB. That's a reduction of more than 99%! This is a very special case (an image with a single color, essentially just a "background"), but sorting the data by channel will improve the RLE compressibility of nearly all images. Naturally, we must also update the decompressor so that it restores the original order (by interleaving the channels). Figure **??** presented in page **??** presents the concept of separating image channels and processing each channel. This example shows how some of the steps involved in data compression don't reduce the amount of data per se, but simply reorganize it (based on some knowledge about patterns in that data) in such a way that the compression steps are more efficient. This same reorganization wouldn't do anything for text or audio data, because there's no 3-byte pattern in the distribution of those types of data. It would just take longer and make the algorithm more complex, with no benefits. But when we apply a compression algorithm to a specific type of data, adapting it to that data's "typical" patterns can definitely pay off.

Naturally, RLE-compressed TGA files use the trick described above, as do most other image compression algorithms, even when they're not based on RLE (or when the data isn't in RGB, for that matter). As you have probably guessed, though, RLE by itself will never achieve high levels of compression on photographic images, or any other kind of image where there are no adjacent pixels with the same color. A typical RLE image compressor will reduce the size of an uncompressed 24-bit RGB photograph by less than 5% (ex., from 900 kiB to 860 kiB). Reducing the number of colors in the image improves the compression (the less colors there are, the more likely it will be that two adjacent pixels have the same color), but decreases the image quality. There are, however, some compression techniques that combine RLE with other algorithms, as we'll see below.

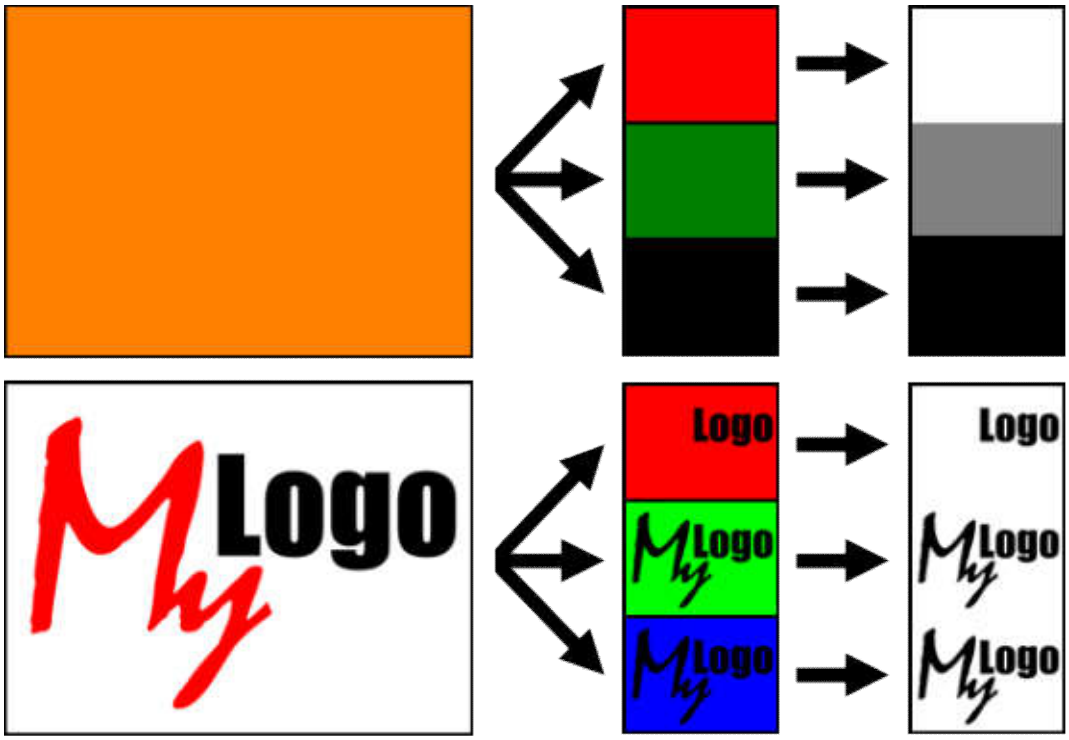

Figure 2.2: Sorting RGB channel data sequentially to improve RLE compression

Figure 2.3: Blue and orange pixel pattern

2.1.2 The Lempel-Ziv compressor family

In our example for the RLE image compressor we had a file where a sequence of three bytes - [255] [128] [0] - was repeated several thousand times. Since RLE wasn't able to compress that kind of pattern directly, we added another step to our algorithm, to sort the pixel color data in a different way. We were able to include that transformation in our algorithm because we already knew that RGB image data tends to follow a certain pattern (that repeats itself after 3 bytes). But what if we were compressing a file where the typical patterns have an unknown length, or where there are multiple patterns, with different lengths? Imagine we have an 8-bit RGB image where the first pixel is blue, the second one is orange, the third one is blue again, the fourth is orange, and so on as the image presented in figure **??** presented in page 53. In other words, let's say the (uncompressed) data for our image looks like this: [0][0][255] [255][128][0] [0][0][255] [255][128][0] ... etc.

This sequence might seem slightly compressible, since there is a sequence of three repeated values followed by a sequence of two repeated values that appears again and again in our file. However, having to constantly switch between compressed and uncompressed mode would actually make the file larger: [0] [0][0][255] [255] (5 bytes)

would be converted into *[3][0][2][255]* (6 bytes, including the markers). Applying our "RGB optimisation" (sorting the data by channel) wouldn't help, either. We'd get this: `R:` [0][255][0][255][0][255] ... `etc. G:` [0][128][0][128][0][128] ... `etc. B:` [255][0][255][0][255][0] ↩ ... `etc.`

Still no sequences of identical values, so RLE wouldn't be able to compress it. We could come up with a stranger way to reorganize the data, but what if we then got a file with sequences of three different pixels instead of two? We'd have to keep changing our compressor (and decompressor) to adapt it to the contents of each image. Not very practical.

Instead, wouldn't it be better if the compressor could look for data patterns, regardless of their length, and replace the repetitions of those patterns with some sort of "counter", like RLE does for individual values?

Before going into specific details, let me call your attention to some examples of "compression" in our everyday language.

Instead of writing "Red, Green and Blue", we often write just "RGB". The same goes for things like HTML ("Hyper-Text Markup Language"), PC ("Personal Computer"), RAID ("Redundant Array of Inexpensive Disks"), and so on. Acronyms and initialisms are a form of data compression. However, they can't be "uncompressed" by themselves; the reader needs to know (or find out) their meaning by matching them to the "uncompressed" version.

In books, it's common to use a footnote to explain unusual terms or acronyms. Another option is to present the acronym only after it has been described. For example, the manual for a compositing program might include:

`All these effects are available when working with high dynamic range (HDR)images.`

And, from then on, always abbreviate the expression "high dynamic range" (because the authors assume people read the previous sentence, and still remember the meaning of that abbreviation): `To avoid clipping the highlights, make sure HDR mode is enabled.`

When someone says "I saw a gemsbok", you might need to reach for your dictionary or encyclopedia to learn that a gemsbok is a South-African antelope. So, in a way, the word "gemsbok" was also a "compressed" version of "species of antelope native to South Africa". As long as you remember that, you can use the "compressed" term (in other words, you only need to reach for the dictionary the first time you come across the term - unless you happen to forget it).

Abraham Lempel is a computer scientist and Jacob Ziv is an electrical engineer. Together, in 1977 they published an algorithm, called LZ77, based on some of the principles described above. A slightly different version was published in 1978, under the name LZ78. And perhaps its most famous variant was published in 1984 by Terry Welch, under the name LZW.

All of these algorithms (and several others, based on them) belong to a family known as "dictionary-based compressors". They work by replacing redundant (i.e., repeated) source data with references to its previous appearance (LZ77) or by explicit references to a "dictionary" compiled from all the data in the source file (LZ78).

For example, if we have this sentence: `The Flying Spaghetti Monster is real. I believe in↩ the Flying Spaghetti Monster.`

We can replace the second instance of the words "he Flying Spaghetti Monster"

(we can't include the "T" from the first word because one is uppercase and the other is lowercase) with a "length-distance pair" - a pair of numbers representing the distance to the previous occurrence of those words and the number of characters that are identical. In this example, that pair would be length = 27 & distance = 54. Again, we can use a special marker (represented here by an asterisk) to indicate that what follows is compressed data: `The Flying Spaghetti Monster is real. I believe in t`↩ *[27][54]*. When the decoder encounters that code, it will step back 54 characters and copy a 27-character sequence into the new position. So we've just compressed the words "he Flying Spaghetti Monster" (27 bytes) into two values (plus two markers), which take up only 4 bytes. This technique (of looking back and copying sequences using a length-distance pair) is called a "sliding window" algorithm. It's a special type of dictionary algorithm, where the dictionary is built along the way, and older sequences are forgotten (as they fall out of the "window"). If we chose to use a single byte to store the distance (of the length-distance pair), then we'd be limited to going back 256 characters, and there wouldn't be any point in keeping older sequences. If we chose to use a 2-byte value (known as a "short integer", or "int16") instead, then we could go back up to 65536 characters, and so on. The bigger the size of the window, the more efficient the compression will be, but it will require more RAM (system memory) to run.

Why 256 characters when 255 is the maximum value a byte can have? And why 65536 when 65535 is the maximum value a short integer can have? Simple: we can use the value 0 (zero) to represent 256 and 65536, respectively, since it would never be used otherwise. We don't necessary want to do that, though, we might prefer to save the value 0 to indicate that some error has occurred.

In a "pure" dictionary based encoder, the sequences are stored at the beginning of the compressed file, and remembered for all the duration of the compression and decompression. That allows the compressor to achieve better results with the same amount of RAM, but requires extra processing (to create the dictionary, the compressor needs to analyze all the source data before it starts to actually compress it).

So, let's see what our new algorithms can do with the image examples mentioned above. First with the pure orange background as the one presented in figure **??** presented in page 52 (which RLE was able to handle quite well, bringing the size down to about 7kB). The source data (RGB byte values) looks like this: [255] [128]↩ [0] [255] [128] [0] ... `etc.` (for a total of 307200 pixels, or 921600bytes)

Using our LZ77-style sliding window technique, we can replace those 921600 bytes with this: [255][128][0] * [921597][3] *

It might sound a bit confusing to say that the decoder must go back 3 bytes and then copy 921597 bytes. But if you think of this operation as taking place one byte at a time, you'll realize that it's perfectly possible, since new bytes are being inserted after our original three, and those are the source for the next copy operations.

We can't store the number 921597 as a single byte (the maximum would be 255 - assuming we don't reassign zero), and we can't even store it as an short integer (the maximum would be 65535), so let's assume we use a 4-byte value (known as an "integer", or "int32"), that can store values from 0 to 4294967295.

This means we have three source data bytes, followed by a 1-byte marker, fol-

Figure 2.4: Pattern with 16 white and 32 black pixels

lowed by a 1-byte length, followed by a 4-byte distance, followed by another 1-byte marker (which isn't really necessary, since there's no data after it, but let's include it anyway. This adds up to 10 bytes. [1]

If we use an LZ78-style compressor instead, our compressed data would look something like the listing 2.1 presented in page 56:

Listing 2.1: Dictionary Representation of Pure Orange Image in LZ78

```
Dictionary: [X][255][128][0]
Reference list: [921597][X]
```

Where X is a code generated by the encoder. It's harder to measure the exact size here, since the file must use different types of markers to separate codes from symbols, lengths from references, and the dictionary itself from the reference list, but overall the results for such a simple pattern would be very similar to the sliding window version (in fact, both types of algorithm are mathematically equivalent).

Now let's look at the second case, where the color of the pixels alternates between blue and orange (which we couldn't compress at all, using RLE). The (uncompressed) source data looks like this: [0][0][255] [255][128][0] [0][0][255] [255][128][0] ...etc. (for a total of 921600 bytes) for the image presented in **??** presented in page 53

If we apply our LZ77-style compressor to it, we get this: [0][0][255][255][128][0]*[921594][6]* ↩
Which can be stored in 13 bytes. The length-distance pairs can be concatenated, of course. For example, to store an image made from a pattern of sequences of 16 white pixels followed 32 black pixels like the one presented in figure **??** presented in page 56, we could do this: [255]*[47][1]*[0]*[95][1][921456][144]* That's a total of 20 bytes. When that data is processed by the decompressor, the first length-distance pair adds 47 bytes with the value 255, which (added to our first byte) describe 16 white pixels (3 bytes per pixel, RGB). The second length-distance pair adds 95 zeros to our existing zero, thus describing 32 black pixels.

The third length-distance pair takes all the bytes created by the first two (a total of 144 bytes) and repeats them for a further 921456, to make a grand total of 921600 bytes (the total number of bytes in an uncompressed 8-bits-per-channel RGB 640x480 image). Given these results, it's easy to see why the LZ algorithms are the basis of most modern lossless data compressors. Naturally, the more complex the pattern(s), the bigger the file will get. The patterns described above were all relatively short and simple (hence the spectacular compression ratios), but even on a plain text file (which RLE would hardly be able to compress at all), LZ algorithms

[1] Although this might seem a lot better than the result we got from our RLE compressor, the comparison is unfair, because we limited the RLE compressor to using a single byte to store the length for each sequence, and we are allowing the LZ compressor to use 4-byte integers. If we had allowed our RLE encoder to use 4-byte values, it would have been able to compress the raw image data into 15 bytes; not as good as the LZ algorithm, but pretty close.