Lossless compression

algorithms using

Data structures

TEAM MEMBERS:

* Rakshanda Bhure -159102168
* Rishi Kumar Ambwani -159102101
* Riya Sharma -159102102
* Rohan Juneja -159102103
* Saif Karigar -159102106
* Sakhsham Kamthan -159102107

# **Project Description:**

# **INTRODUCTION:**

## There are two major categories of compression algorithms: lossy and lossless.Lossy compression algorithm involve the reduction of a file’s size usually by removing small details that require a large amount of data to store at full fidelity.In lossy compression,it is impossible to restore the original file due to the removal of essential data.Lossless data compression is the size reduction of a file,such that a decompression function can restore the original file exactly with no loss of data.

## The basic principle that lossless compression algorithms work on is that any non random file will contain duplicated information that can be condensed using statistical modelling techniques that determine the probability of a character or phrase appearing.

# **CONTENTS:**

## In this project we discuss the algorithms and applications of Huffman,RLE and Lz77.

Huffman coding:   Huffman coding uses a specific method for choosing the representation for each symbol, resulting in a prefix code (sometimes called "prefix-free codes", that is, the bit string representing some particular symbol is never a prefix of the bit string representing any other symbol). Huffman coding is such a widespread method for creating prefix codes that the term "Huffman code" is widely used as a synonym for "prefix code" even when such a code is not produced by Huffman's algorithm.

Run-length encoding(RLE): **It** is a very simple form of lossless data compression in which *runs* of data (that is, sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run. This is most useful on data that contains many such runs. Consider, for example, simple graphic images such as icons, line drawings, and animations. It is not useful with files that don't have many runs as it could greatly increase the file size.

### Lz77: LZ77 algorithms achieve compression by replacing repeated occurrences of data with references to a single copy of that data existing earlier in the uncompressed data stream. A match is encoded by a pair of numbers called a *length-distance pair*, which is equivalent to the statement "each of the next *length* characters is equal to the characters exactly *distance* characters behind it in the uncompressed stream". (The "distance" is sometimes called the "offset" instead.)

**USES**

Lossless data compression is used ubiquitously in computing , from saving space on your personal computer to sending data over the web ,communicating over a secure shell, or viewing a PNG or GIF image.

The statistical models can then be used to generate codes for specific characters or phrases based on their probability of occurring, and assigning the shortest codes to the most common data. Such techniques include entropy encoding, run-length encoding and compression using a dictionary. Using these techniques and others, an 8 bit character or string of such characters could be represented with a few bits resulting in a large amount of redundant data being removed.

**LANGUAGE USED :** C Programming language

**COMPILER USED :** GCC Compiler

**DATA STRUCTURE USED:** Trees

**TEAM RESPONSIBILITY**

**PROJECT WORK**

* HUFFMAN CODING – Rakshanda Bhure and Riya Sharma
* RLE –Rishi Kumar Ambwani and Rohan Juneja
* Lz77 –-Saif Kaigar and Saksham Kamthan

**INDIVIDUAL WORK**

* GRAPHICS – Saif Karigar
* RESEARCH DEVELOPMENT – Riya Sharma
* DATA HANDLING – Saksham Kamthan
* DEVELOPMENT SRATEGY –Rohan Juneja
* PRESENTATION –Rakshanda Bhure
* OPERATIONS AND PRODUCTION –Rishi Kumar Ambwani

**MANAGEMENT**

* OVERALL MANAGEMENT –Riya Sharma and Rishi Kumar Ambwani
* CODING MANAGEMENT –Saif Karigar and Rakshanda Bhure
* INFORMATION GATHERING AND EXECUTION­ –Rohan Juneja and Saksham Kamthan

Huffman Coding

Huffman encoding is a way to assign binary codes to symbols that reduces the overall number of bits used to encode a typical string of those symbols.

For example, if you use letters as symbols and have details of the frequency of occurrence of those letters in typical strings, then you could just encode each letter with a fixed number of bits, such as in ASCII codes. You can do better than this by encoding more frequently occurring letters such as e and a, with smaller bit strings; and less frequently occurring letters such as q and x with longer bit strings.

Any string of letters will be encoded as a string of bits that are no-longer of the same length per letter. To successfully decode such as string, the smaller codes assigned to letters such as 'e' cannot occur as a prefix in the larger codes such as that for 'x'.

If you were to assign a code 01 for 'e' and code 011 for 'x', then if the bits to decode started as 011... then you would not know if you should decode an 'e' or an 'x'.

The Huffman coding scheme takes each symbol and its weight (or frequency of occurrence), and generates proper encodings for each symbol taking account of the weights of each symbol, so that higher weighted symbols have fewer bits in their encoding. (See the [WP article](http://en.wikipedia.org/wiki/Huffman_coding) for more information).

A Huffman encoding can be computed by first creating a tree of nodes:

* Create a leaf node for each symbol and add it to the [priority queue](https://rosettacode.org/wiki/Priority_queue).
* While there is more than one node in the queue:
  + Remove the node of highest priority (lowest probability) twice to get two nodes.
  + Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.
  + Add the new node to the queue.
* The remaining node is the root node and the tree is complete.

Traverse the constructed binary tree from root to leaves assigning and accumulating a '0' for one branch and a '1' for the other at each node. The accumulated zeros and ones at each leaf constitute a Huffman encoding for those symbols and

It is a compression algorithm used for loss-less data compression. Here’s the basic idea: each ASCII character is usually represented with 8 bits, but if we had a text filed composed of only the lowercase a-z letters we could represent each character with only 5 bits (i.e., 2^5 = 32, which is enough to represent 26 values), thus reducing the overall memory needed to store the data.

For example, the table of characters -> binary code could look like this:

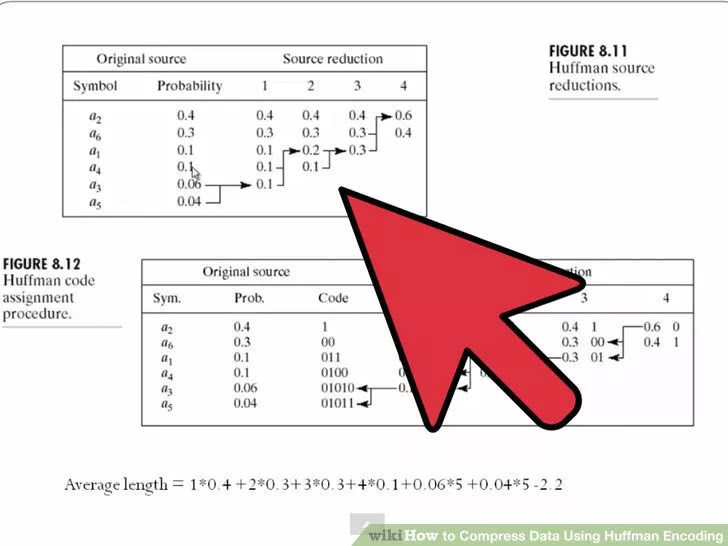
* a = 00000
* b = 00001
* c = 00010
* d = 00011
* e = 00100

And so on.This is the fixed-length representation, and it already creates a significant compression rate (around 35% in the example above).

A more efficient approach is to use a variable-length representation, where each character can have a different number of bits to represent it. More specifically we first analyze the frequency of each character in the text, and then we create a binary tree (called Huffman tree) giving a shorter bit representation to the most used characters, so that they can be reached faster. Notice that it must be a prefix tree (i.e., the code of every letter can’t be prefix to the code of any other letter) else the decompression wouldn’t work.   
Input is array of unique characters along with their frequency of occurrences and output is Huffman Tree.

[**How to Compress Data Using Huffman Encoding**](http://www.wikihow.com/Compress-Data-Using-Huffman-Encoding)

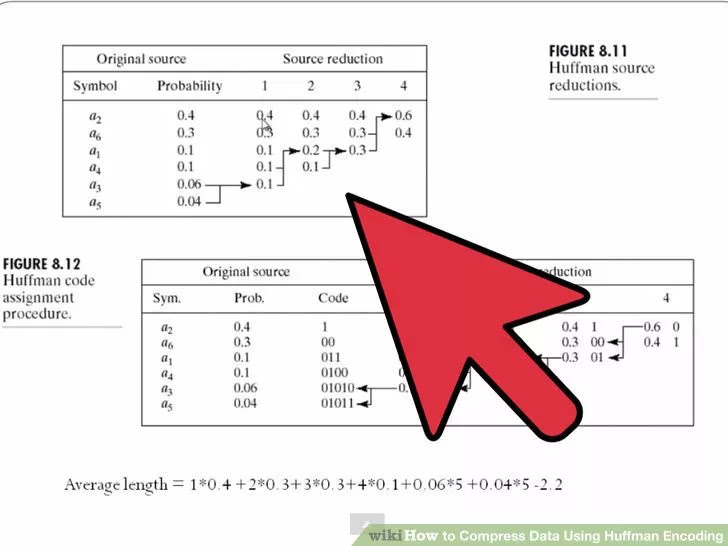
Huffman's algorithm is used to compress or encode data. Normally, each character in a text file is stored as eight bits (digits, either 0 or 1) that map to that character using an encoding called ASCII. A Huffman-encoded file breaks down the rigid 8-bit structure so that the most commonly used characters are stored in just a few bits ('a' could be "10" or "1000" rather than the ASCII, which is "01100001"). The least common characters, then, will often take up much more than 8 bits ('z' might be "00100011010"), but because they occur so rarely, Huffman encoding, on the whole, creates a much smaller file than the original.



**1**

**Count the frequency of each character in the file to be encoded.** Include a dummy character to mark the end of the file -- this will be important later. For now, call it the EOF (end of file) and mark it as having a frequency of 1.

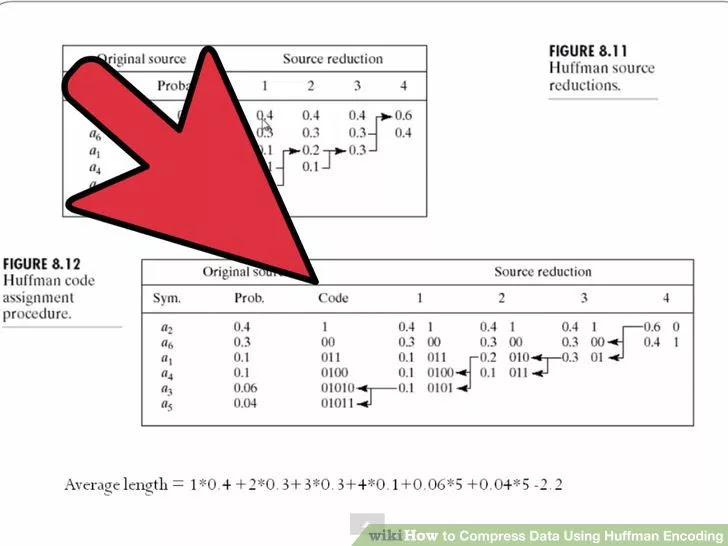
* For example, if you want to encode a text file reading "ab ab cab," you would have 'a' with frequency 3, 'b' with frequency 3, ' ' (space) with frequency 2, 'c' with frequency 1, and EOF with frequency 1.



**2**

**Store characters as tree nodes and put them into a priority queue.** You'll be building a big binary tree with each character as a leaf, so you should store the characters in a format such that they can become nodes of the tree. Place these nodes into a priority queue with each character's frequency as its node's priority.

* A binary tree is a data format where each piece of data is a node that can have up to one parent and two children. It's often drawn as a branching tree, hence the name.
* A queue is an aptly named data collection where the first thing to go into the queue is also the first thing to come out (like waiting in line). In a *priority* queue, the data are stored in order of their priority, so that the first thing to come out is the most urgent thing, the thing with the smallest priority, rather than the first thing enqueued.
* In the "ab ab cab" example, your priority queue would look like this: {'c':1, EOF:1, ' ':2, 'a':3, 'b':3}



**3**

**Begin to build your tree.** Remove (or *dequeue*) the two most urgent things from the priority queue. Create a new tree node to be the parent of these two nodes, storing the first node as its left child and the second as its right child. The priority of the new node should be the sum of the priorities of its child. Then enqueue this new node in the priority queue.

* The priority queue now looks like this: {' ':2, new node:2, 'a':3, 'b':3}

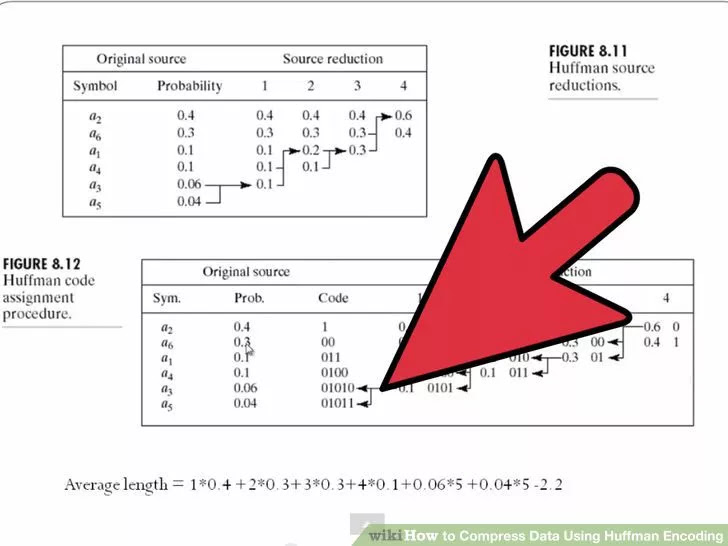
**4**

**Finish building your tree:** repeat the above step until there is only one node in the queue. Note that in addition to the nodes you created for the characters and their frequencies, you will also be dequeuing, turning into trees, and re-enqueueing parent nodes, nodes that are already themselves trees.

* When you're finished, the last node in the queue will be the *root* of the encoding tree, with all the other nodes branching off from it.
* The most frequently used characters will be the leaves closest to the top of the tree, while the rarely used characters will be positioned at the bottom of the tree, farther away from the root.

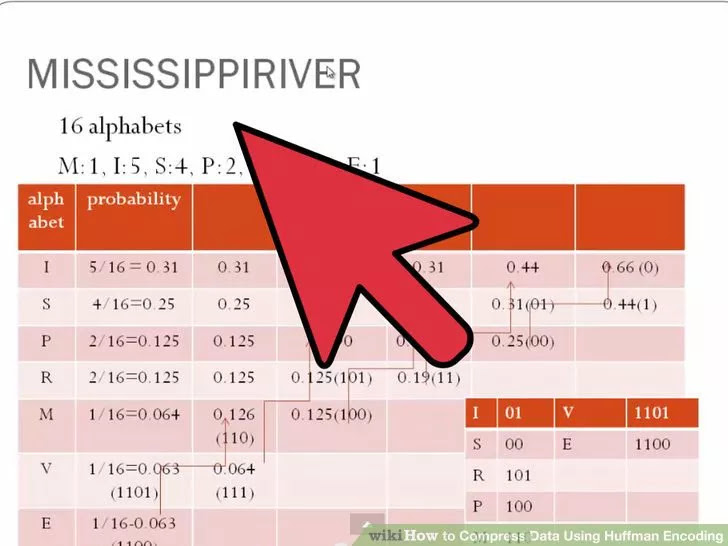
**5**

**Create an *encoding map.*** Walk through the tree to reach each character. Every time you visit a node's left child, that's a '0'. Every time you visit a node's right child, that's a '1'. When you get to a character, store the character with the sequence of 0s and 1s that it took to get there. This sequence is what the character will be encoded as in the compressed file. Store the characters and their sequences in a map.

* For example, start at the root. Visit the root's left child, and then visit that node's left child. Since the node you're at now doesn't have any children, you've reached a character. This is ' '. Since you walked left twice to get here, the encoding for ' ' is "00".
* For this tree, the map will look like this: {' ':"00", 'a':"10", 'b':"11", 'c':"010", EOF:"011"}.
* 

**6**

* **In the output file, include the encoding map as a header.** This will allow the file to be decoded.



**7**

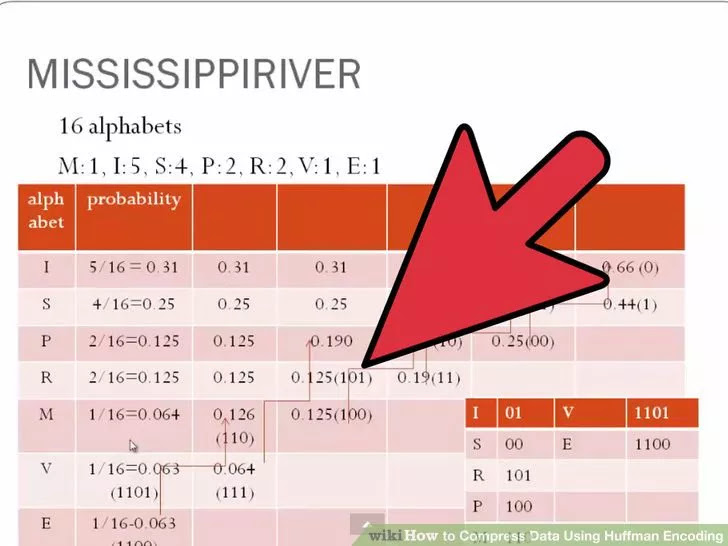
**Encode the file.** For each character in the file to be encoded, write the binary sequence you've stored in the map. Once you've finished encoding the file, make sure to add the EOF to the end.

* For the file "ab ab cab", the encoded file will look like this: "1011001011000101011011".
* Files are stored as bytes (8 bits, or 8 binary digits). Because the Huffman Encoding algorithm doesn't use the 8-bit format, encoded files will often not have lengths that are multiples of 8. The remaining digits will be filled in with 0s. In this case, two 0s would be added at the end of the file, which looks like another space. This could be a problem: how would the decoder know when to stop reading? However, because we included an end-of-file character, the decoder will get to this and then stop, ignoring anything else that's been added on after.

**Decoding**

**1**

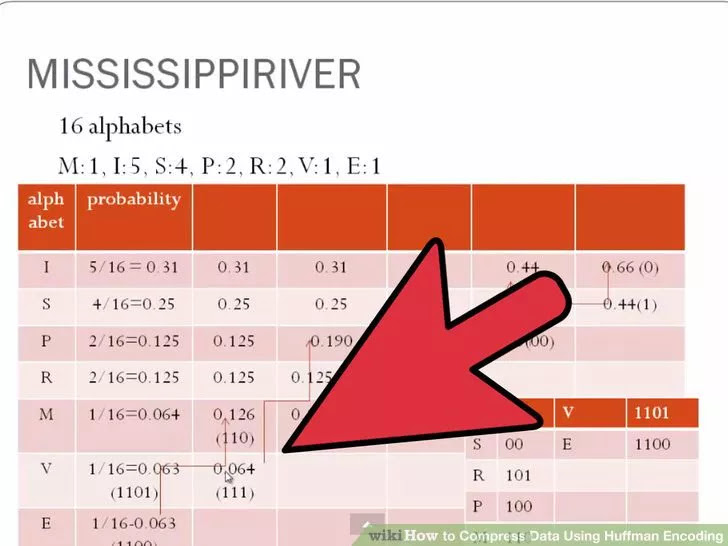
**Read in a Huffman-encoded file.** First, read the header, which should be the encoding map. Use this to build a decoding tree in the same way you built the tree you used to encode the file. The two trees should be identical.



**2**

**Read in the binary one digit at a time.** Traverse the tree as you read: if you read in a '0', go to the left child of the node you're at, and if you read in a '1', go to the right child. When you reach a leaf (a node without any children), you've arrived at a character. Write the character into the decoded file.

* Because of the way the characters are stored in the tree, the codes for each character have a *prefix property*, so that no character's binary encoding can ever occur at the start of another character's encoding. The encoding for each character is totally unique. This makes decoding much easier.



**3**

**Repeat until you reach the EOF.** Congratulations! You've decoded the file

## The Huffman Algorithm

So far, we've gone over the basic principles we'll need for the Huffman algorithm, both for encoding and decoding, but we've had to guess at what would be the best way of actually encoding the characters. For our simple text string, it wasn't too hard to figure out a decent encoding that saved a few bits. But in the general case, it might be hard to figure out a good solution, let alone the best possible solution.   
  
The Huffman algorithm is a so-called "greedy" approach to solving this problem in the sense that at each step, the algorithm chooses the best available option. It turns out that this is sufficient for finding the best encoding.   
  
The basic idea behind the algorithm is to build the tree bottom-up. First, every letter starts off as part of its own tree and the trees are ordered by the frequency of the letters in the original string. Then the two least-frequently used letters are combined into a single tree, and the frequency of that tree is set to be the combined frequency of the two trees that it links together.   
  
For instance, if we started out with two characters that showed up once, L and T, in our sample string, they would be recombined into a new tree that has a "supernode" that links to both L and T, and has a frequency of 2:

X, 2

/ \

/ \

L, 1 T, 1

This new tree is reinserted into the list of trees in its sorted position. The process is then repeated, treating trees with more than one element the same as any other trees except that their frequencies are the sum of the frequencies of all of the letters at the leaves. (This is just the sum of the left and right children of any node because each node stores the frequency information about its own children.) The process completes when all of the trees have been combined into a single tree -- this tree will describe a Huffman compression encoding.   
  
Essentially, a tree is built from the bottom up -- we start out with 256 trees (for an ASCII file) -- and end up with a single tree with 256 leaves along with 255 internal nodes (one for each merging of two trees, which takes place 255 times). The tree has a few interesting properties -- the frequencies of all of the internal nodes combined together will give the total number of bits needed to write the encoded file (except the header). This property comes from the fact that at each internal node, a decision must be made to go left or right, and each internal node will be reached once for each time a character beneath it shows up in the text of the document.   
  
To go from plain text to compressed text, you would have to do a traversal of the tree and store the path to reach each leaf node as a string of bits (0 for going left, 1 for going right) and associate that bit with the particular character at the leaf. Once this is done, converting a plain text file into a compressed file is just a matter of replacing each letter with an appropriate bit string and then handling the possibility of having some extra bits that need to be written (this is discussed more fully in the implementation notes). Notice that two different data structures likely need to be used here -- a list of trees, and those binary trees themselves. It might make sense to use several data structures such as:

struct tree\_t

{

tree\_t \*left;

tree\_t \*right;

char character;

};

to store the tree elements (note that if left and right are NULL, then we would know that the node is a leaf and that character stores a valid char of interest) and

struct list\_t

{

list\_t \*next;

int total\_frequency;

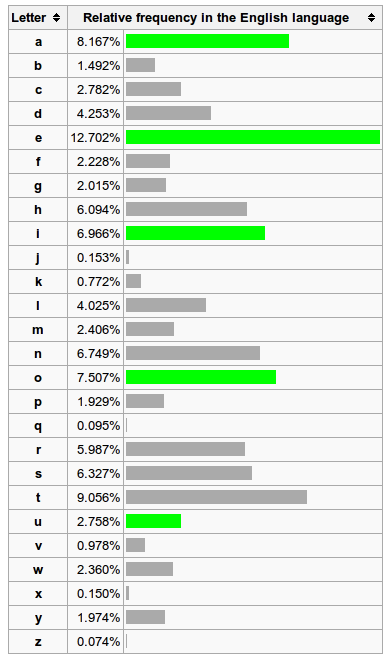
tree\_t \*tree;

}

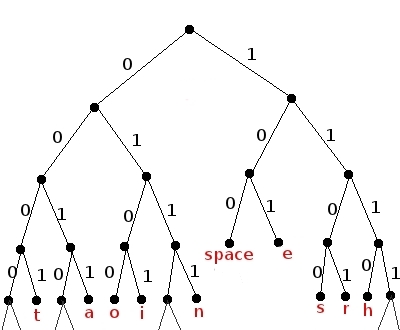
to store the list of trees that still need to be merged together as a [linked list](http://www.cprogramming.com/tutorial/lesson15.html).

Real Life Example of Huffman

The frequency of the letters in the English language (according to Wikipedia) is the following:



Now the algorithm to create the Huffman tree is the following:

* Create a forest with one tree for each letter and its respective frequency as value
* Join the two trees with the lowest value, removing each from the forest and adding instead the resulting combined tree
* Repeat until there’s only one tree left
* The Huffman tree for the a-z letters (and the space character) using the frequency table above would look like this (you would need to e xpand to the lower branches to see all the letters):

We can traverse the tree and create a table with all the letters and their respective binary codes. It would look like this

**RLE{Run Length Encoding}**

**Run-length encoding** (**RLE**) is a very simple form of [lossless data compression](https://en.wikipedia.org/wiki/Lossless_data_compression) in which *runs* of data (that is, sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run. This is most useful on data that contains many such runs. Consider, for example, simple graphic images such as icons, line drawings, and animations. It is not useful with files that don't have many runs as it could greatly increase the file size.

RLE may also be used to refer to an early graphics file format supported by [CompuServe](https://en.wikipedia.org/wiki/CompuServe) for compressing black and white images, but was widely supplanted by their later [Graphics Interchange Format](https://en.wikipedia.org/wiki/Graphics_Interchange_Format). RLE also refers to a little-used image format in [Windows 3.x](https://en.wikipedia.org/wiki/Windows_3.x), with the extension **rle**, which is a Run Length Encoded Bitmap, used to compress the Windows 3.x startup screen.

Typical applications of this encoding are when the source information comprises long substrings of the same character or binary digit.

Example of RLE

For example, consider a screen containing plain black text on a solid white background. There will be many long runs of white [pixels](https://en.wikipedia.org/wiki/Pixel) in the blank space, and many short runs of black pixels within the text. A hypothetical [scan line](https://en.wikipedia.org/wiki/Scan_line), with B representing a black pixel and W representing white, might read as follows:

WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWWWWWWWWWWBWWWWWWWWWWWWWW

With a run-length encoding (RLE) data compression algorithm applied to the above hypothetical scan line, it can be rendered as follows:

12W1B12W3B24W1B14W

This can be interpreted as a sequence of twelve Ws, one B, twelve Ws, three Bs, etc.

The run-length code represents the original 67 characters in only 18. While the actual format used for the storage of images is generally binary rather than [ASCII](https://en.wikipedia.org/wiki/ASCII) characters like this, the principle remains the same. Even binary data files can be compressed with this method; file format specifications often dictate repeated bytes in files as padding space. However, newer compression methods such as [DEFLATE](https://en.wikipedia.org/wiki/DEFLATE) often use [LZ77](https://en.wikipedia.org/wiki/LZ77)-based algorithms, a generalization of run-length encoding that can take advantage of runs of strings of characters (such as BWWBWWBWWBWW).

Run-length encoding can be expressed in multiple ways to accommodate data properties as well as additional compression algorithms. For instance, one popular method encodes run lengths for runs of two or more characters only, using an "escape" symbol to identify runs, or using the character itself as the escape, so that any time a character appears twice it denotes a run. On the previous example, this would give the following:

WW12BWW12BB3WW24BWW14

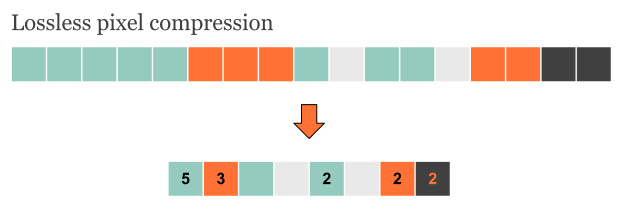
This would be interpreted as a run of twelve Ws, a B, a run of twelve Ws, a run of three Bs, etc. In data where runs are less frequent, this can significantly improve the compression rate.

One other matter is the application of additional compression algorithms. Even with the runs extracted, the frequencies of different characters may be large, allowing for further compression; however, if the run lengths are written in the file in the locations where the runs occurred, the presence of these numbers interrupts the normal flow and makes it harder to compress.

**Introduction**

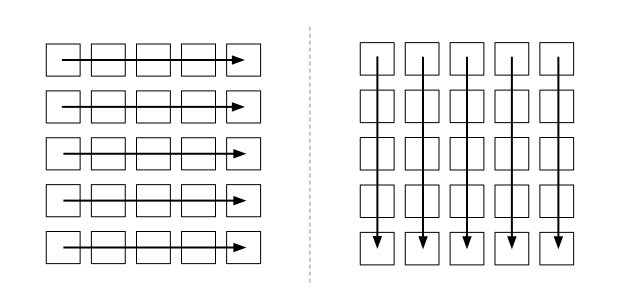
[Run-length encoding](http://www.stoimen.com/blog/2012/01/09/computer-algorithms-data-compression-with-run-length-encoding/) is a data compression algorithm that helps us encode large runs of repeating items by only sending one item from the run and a counter showing how many times this item is repeated. Unfortunately this technique is useless when trying to compress natural language texts, because they don’t have long runs of repeating elements. In the other hand RLE is useful when it comes to image compression, because images happen to have long runs pixels with identical color.

As you can see on the following picture we can compress consecutive pixels by only replacing each run with one pixel from it and a counter showing how many items it contains.

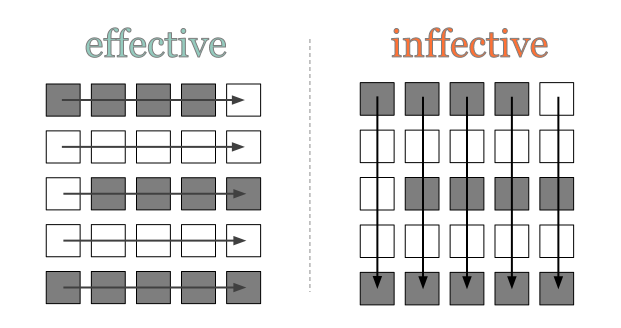
[](http://www.stoimen.com/blog/wp-content/uploads/2012/05/1.LosslessRLEforImages.png)Although lossless RLE can be quite effective for image compression, it is still not the best approach!

In this case we can save only counters for pixels that are repeated more than once. Such the input stream “aaaabbaba” will be compressed as “[4]a[2]baba”.

Actually there are several ways run-length encoding can be used for image compression. A possible way of compressing a picture can be either row by row or column by column, as it is shown on the picture below.

[](http://www.stoimen.com/blog/wp-content/uploads/2012/05/2.RowbyRowandColbyCol.png)Row by row or column by column compression.

The problem in practice is that sometimes compressing row by row may be effective, while in other cases the same approach is very ineffective. This is illustrated by the image below.

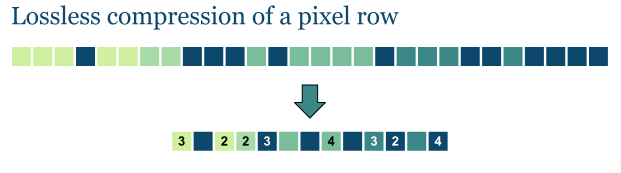
[](http://www.stoimen.com/blog/wp-content/uploads/2012/05/3.EffectiveandIneffectiveCompression.png)Sometimes image compression may be done only after some preprocessing that can help us understand the best compression approach!

Obviously run-length encoding is a very good approach when compressing images, however when we talk about big images with millions of pixels it’s somehow natural to come with some lossy compression.

**Overview**

Lossy RLE is a very suitable algorithm when it comes to images, because in most of the cases large images do appear to have big spaces of identical pixel colors, i.e. when the half of the picture is the blue sky. By using lossy compression we can skip very short runs.

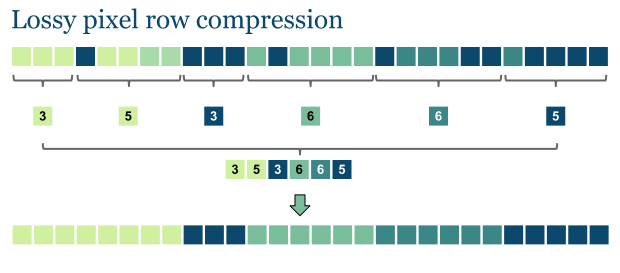
First we’ve to say how long will be the shortest run that we will keep in the compression. For instance if 3 is the shortest run, then runs of 2 consecutive elements will be skipped.

[](http://www.stoimen.com/blog/wp-content/uploads/2012/05/4.LossessImageRow.png)Lossless compression of a pixel row in some cases can be very inefective!

Of course if we set the shortest run to be only one element long, this will make our compression completely lossless, which isn’t very effective. However when we talk about millions of pixels even runs of three or more elements are very short, so it’s up to the developer to decide how long will be the shortest run.

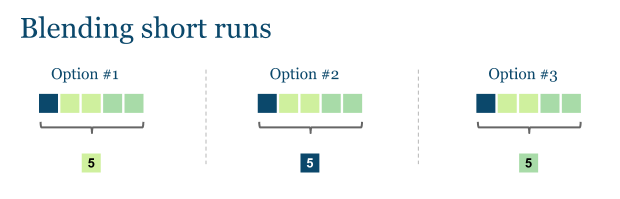
Some Examples

Let’s first define the shortest run that we will keep untouched to be at least three element long.

[](http://www.stoimen.com/blog/wp-content/uploads/2012/05/5.LossyImageRow.png)We can lose some information that is invisbile to the eye.

The above image is compressed more effectively than the lossless pixel row from the previous picture.

The thing is how to merge short runs. For instance the following three runs have to be blended into one color run.

[](http://www.stoimen.com/blog/wp-content/uploads/2012/05/6.BlendingShortRuns.png)We must chose how to blend short runs!

We can choose the middle color (option #1) or not, but this will always depend on the picture and it will be effective in some cases and ineffective in other.

**RLE Pseudocode**

Given a binary image of dimension n x m, with a background pixel intensity of 0 and foreground intensity of 1.

set color to 0

set count to 0

for each pixel in the image

if current pixel not equal to color

write count

set color to current pixel color

set count to 1

else

increment count by 1

if count not equal to 0

write count // record last run

**Complexity**

In general lossless RLE compelxity is linear – O(n) where n is the number of items from the input stream. Even with the small modification above the complexity remains linear. However we can modify the compression in a slightly different manner (in order to get the middle value from consecutive short runs). This will somehow affect the complexity of the algorithm, of course.

**Application**

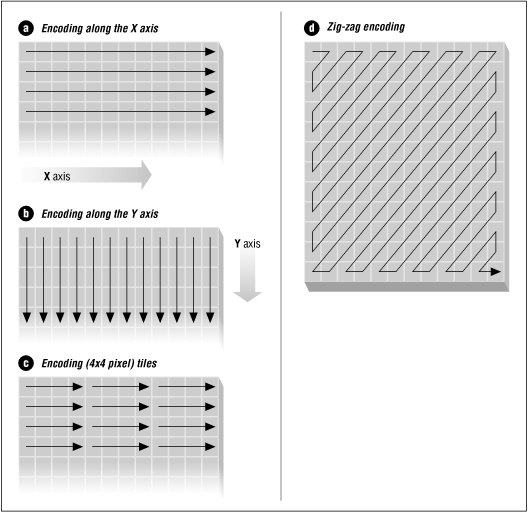
Run-length encoding isn’t a very effective option when compressing texts, but for images where long runs of the identical pixels happen to occur it is quite useful.. This is most useful on data that contains many such runs. Consider, for example, simple graphic images such as icons, line drawings, and animations. It is not useful with files that don't have many runs as it could greatly increase the file size.

Image Compression with RLE

**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 (shown in [Figure 9-2](http://www.fileformat.info/mirror/egff/ch09_03.htm#X058-9-C09-FG-2), a). But alternative RLE schemes can also be written to encode data down the length of a bitmap (the Y axis) along the columns (shown in [Figure 9-2](http://www.fileformat.info/mirror/egff/ch09_03.htm#X058-9-C09-FG-2), b), to encode a bitmap into 2D tiles (shown in [Figure 9-2](http://www.fileformat.info/mirror/egff/ch09_03.htm#X058-9-C09-FG-2), c), or even to encode pixels on a diagonal in a zig-zag fashion (shown in [Figure 9-2](http://www.fileformat.info/mirror/egff/ch09_03.htm#X058-9-C09-FG-2), d). 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 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.

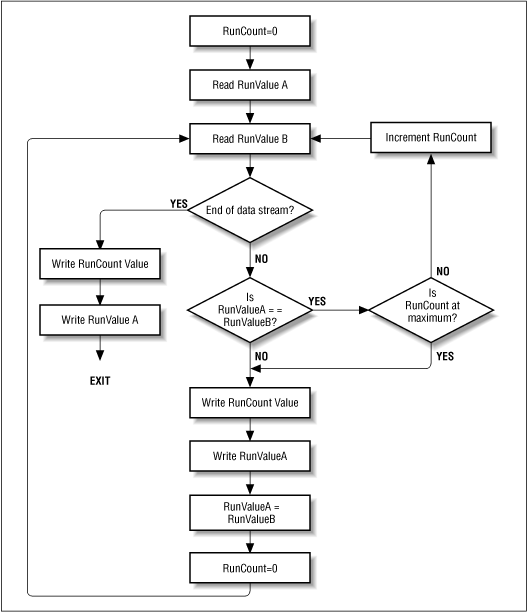
Encoding scan lines individually has advantages when an application needs to use only part of an image. Let's say that an image contains 512 scan lines, and we need to display only lines 100 to 110. If we did not know where the scan lines started and ended in the encoded image data, our application would 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, as illustrated in [Figure 9-3](http://www.fileformat.info/mirror/egff/ch09_03.htm#X058-9-C09-FG-3).

**Figure 9-3: 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.

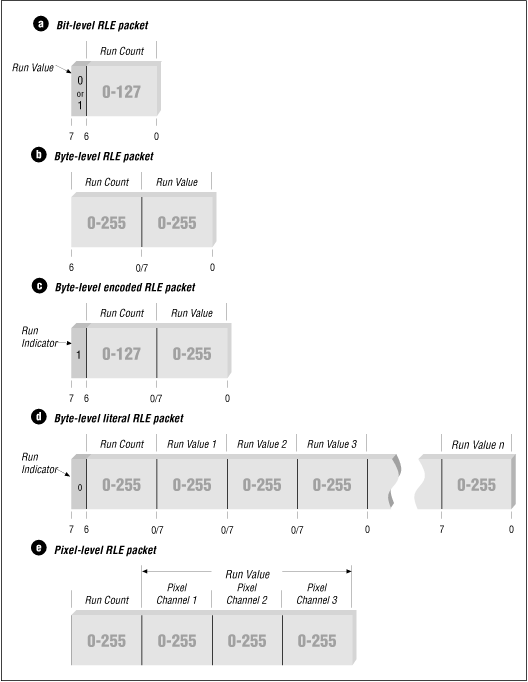
*Bit-level RLE* schemes encode runs of multiple bits in a scan line and ignore byte and word boundaries. Only monochrome (black and white), 1-bit images contain a sufficient number of bit runs to make this class of RLE encoding efficient. A typical bit-level RLE scheme encodes runs of one to 128 bits 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 (shown in [Figure 9-4](http://www.fileformat.info/mirror/egff/ch09_03.htm#X058-9-C09-FG-4), a). 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 (shown in [Figure 9-4](http://www.fileformat.info/mirror/egff/ch09_03.htm#X058-9-C09-FG-4), b). If the most significant bit is set to 1, it denotes an encoded run (shown in [Figure 9-4](http://www.fileformat.info/mirror/egff/ch09_03.htm#X058-9-C09-FG-4), c). 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 (shown in [Figure 9-4](http://www.fileformat.info/mirror/egff/ch09_03.htm#X058-9-C09-FG-4), d). 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 (shown in [Figure 9-4](http://www.fileformat.info/mirror/egff/ch09_03.htm#X058-9-C09-FG-4), e). The encoding method remains the same as with the byte-oriented RLE.

**Figure 9-4: Bit-, byte-, and pixel-level RLE schemes**



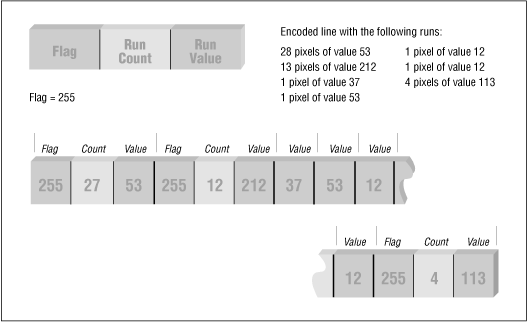
It is also possible to employ a literal pixel run encoding by using the most significant bit of the run count as in the byte-level RLE scheme. Remember that the run count in pixel-level RLE schemes is the number of pixels and not the number of bytes in the run.

Earlier in this section, we examined a situation where the string "Xtmprsqzntwlfb" 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 (shown in [Figure 9-5](http://www.fileformat.info/mirror/egff/ch09_03.htm#X058-9-C09-FG-5)). 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.

**Figure 9-5: 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**

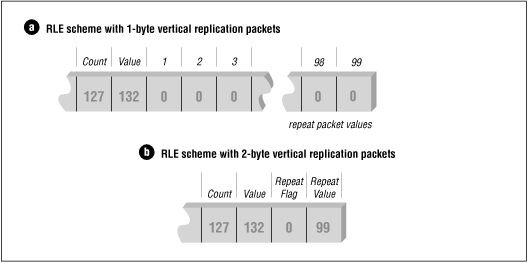
Some RLE schemes use other types of encoding packets to increase compression efficiency. One of the most useful of these packets is the *repeat scan line packet*, also known as the *vertical replication packet*. This packet does not store any real scan-line data; instead, it just indicates a repeat of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

[Figure 9-6](http://www.fileformat.info/mirror/egff/ch09_03.htm#X058-9-C09-FG-6) illustrates 1- and 2-byte vertical replication packets.

**Figure 9-6: 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)](http://www.fileformat.info/format/wpg/egff.htm) and [GEM Raster (IMG)](http://www.fileformat.info/format/gemraster/egff.htm), 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.**

**LZ-77 Compression**

1. **Introduction-**

LZ77 and LZ78 are the two lossless data compression algorithms published in papers by Abraham Lempel and Jacob Ziv in 1977 and 1978. They are also known as LZ1 and LZ2 repsectively.

These two algorithms form the basis for many variations including LZW,LZSS,LZMA and others.

Besides their academic influence, these algorithms formed the basis of several ubiquitous compression schemes, inclusing GIF, ZIP, etc.

1. **Concept-**

Lempel-Ziv 77(LZ77) algorithm is the first Lempel-Ziv compression algorithm for sequential data compression. The dicitonary is a portion of the previously encoded sequence. The encoder examines the input sequence through a sliding window.

The window consists of two parts:

1. A search buffer that contains the next portion of the recently encoded sequence,
2. A look-ahead buffer that contains the next portion of the sequence to be encoded.

The following is what the encoder examines as the input sequence through a sliding window:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Search buffer | Look ahead buffer |  |  |
| ..a | c c a b r a c a d | a b r a r r a r r | a | c |

3. Algorithm(Encoding)-

1. To encode the sequence in the look-ahead buffer, the encoder moves a search pointed back through the search buffer until it encounters a match to the first symbol in the look-ahead buffer.
2. The distance of the pointer from the look-ahead buffer is called the *offset.*The encoder then examines the symbols following the symbol at the pointer location to see if they match consecutive symbols in the look-ahead buffer.
3. The number of consecutive symbols in the search buffer that match consecutive symbols in the look-ahead buffer, starting with the first symbol, is called the *length of the match*. The encoder searches the search buffer for the longest match.
4. Once the longest match has been found, the encoder encodes it with a triple <*o, l, c*>, where *o* is the offset, *l* is the length of the match, and *c* is the codeword corresponding to the symbol in the look-ahead buffer that follows the match.
5. For example, in the diagram above, the longest match is the first ‘a’ of the search buffer. The offset *o* in this case is 7, the length of the match *l* is 4, and the symbol in the look-ahead buffer following the match is ‘r’.
6. The reason for sending the third element in the triple is to take care of the situation where no match for the symbol in the look-ahead buffer can be found in the search buffer.
7. In this case, the offset and match-length values are set to 0, and the third element of the triple is the code for the symbol itself.
8. The algorithm looks like the following:

while( lookAheadBuffer not empty )  
{  
    get a pointer (position, match) to the longest match   
    in the window for the lookAheadBuffer;  
  
    output a (position, length, char()) triple;  
    shift the window length+1 characters along;  
}

**4.Decoding –**

1. For the decoding process, it is basically a table loop-up procedure and can be done by reversing the encoding procedures.
2. We employ the same buffer sized *n* characters, and then use its first (***N***-*n*) spaces to hold the previously decoded characters, where ***N*** is the size of the window (sum of the size of the look-ahead buffer and the search buffer) used in the encoding process.
3. If we break up each triple that we encounter back into its components:- position offset *o*, match length *l*, and the last symbol of the incoming stream *c*, we extract the match string from buffer according to *o*, and thus obtain the original content.
4. What has been shifted out of the buffer is sent to the output. The process continues until all code words have been decoded, and the decoding is then complete.

**5.** Shortcomings of LZ77-

1. One of the main limitations of the LZ77 algorithm is that it uses only a small window into previously seen text, which means it continuously throws away valuable dictionary entries because they slide out of the dictionary.
2. The longest match possible is roughly the size of the look-ahead buffer; many of the matches actually existing in the file may actually be much longer.
3. The sliding window makes the algorithm biased toward exploiting recency in the text, and this is not necessarily a loss.
4. One obvious way to get around this problem is to increase the size of the search and look-ahead buffers.
5. However, changing these parameters will drastically increase the CPU time needed for compression.
6. Since string comparisons between the search buffer phrases and the look-ahead buffer proceed sequentially, the runtime here will increase in direct proportion to the length of the look-ahead buffer.
7. Also, changing the size of the search buffer will only marginally affect the efficiency because we are still using an algorithm that relies on recency to perform compression.

**6. Example-**

AN LZ77 decoding of the triple<7,4,C®> is shown below

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | C | C | A | B | R | A | C | A | D |

Initial State

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | C | C | A | B | R | A | C | A | D |

Move back 7

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | C | C | A | B | R | A | C | A | D | A |

Copy 1

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | C | C | A | B | R | A | C | A | D | A | B |

Copy 2

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | C | C | A | B | R | A | C | A | D | A | B | R |

Copy 3

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | C | C | A | B | R | A | C | A | D | A | B | R | A |

Copy 4

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | C | C | A | B | R | A | C | A | D | A | B | R | A | R |

**Pseudocode**

The pseudocode is a reproduction of the LZ77 compression algorithm sliding window.

begin

fill view from input

while (view is not empty) do

begin

find longest prefix p of view starting in coded part

i := position of p in window

j := length of p

X := first char after p in view

output(i,j,X)

add j+1 chars

end

end

**Example**

The calculation of the LZ77-based factorization of the string aacaacabcabaaac is illustrated.

The table shows the calculation of the LZ77 factorization using a dictionary buffer of size 12 and a preview buffer of size 9. In the far right column is from top to bottom read the output of the algorithm (0, 0, "a") (1, 1, "c") (3, 4, "b") (3, 3, "a") (12, 3, "$"). The position is relative to the right edge of the dictionary buffer, this must be considered when decoding.

The buffers operate on the principle of a sliding window, i.e. to be compressed data stream is pushed right into the buffer. As noted in the algorithm, the shift is to the length of the match found in the dictionary, and a further position. This means that redundant triples be avoided as new characters are usually always taken individually in the dictionary. In the example, so the third triple (0, 0, "c") should be incorporated, what the compression ratio, however, deteriorated significantly. The matches are green and marked to be moved string in red. It is important to note that more and more a character is shifted, was found to be in accordance to new characters do not have to double encode.

Example of a LZ77 compression sliding window

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Example of a LZ77 compression sliding window** | | | | | | | | | | | | | | | | | | | | | | | |
| **Line** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **Output** |
| 1 | (Empty) | | | | | | | | | | | | a | a | c | a | a | c | a | b | c | a | (0,0,"**a**") |
| 2 | (Empty) | | | | | | | | | | | a | a | c | a | a | c | a | b | c | a | b | (1,1,"**c**") |
| 3 | (Empty) | | | | | | | | | a | a | c | a | a | c | a | b | c | a | b | a | a | (3,4,"**b**") |
| 4 | (Empty) | | | | a | a | c | a | a | c | a | b | c | a | b | a | a | a | c | (Empty) | | | (3,3,"**a**") |
| 5 | a | a | c | a | a | c | a | b | c | a | b | a | a | a | c | (Empty) | | | | | | | (12,3,"**$**") |
| **finished** | | | | | | | | | | | | | | | | | | | | | | | |

The first popular characters is unknown, so that the first "a" is added to (0, 0, "a"). In the 2nd line "a" can already be read from the dictionary buffer (marked in green) so that "c" is accepted as the new character. In the 3rd line is a special case of the LZ77 algorithm can be seen as the matching string extends into the preview window, shown in the example by green text on a red background. Line 4 and 5 are equivalent to deal with the first two. Except that last a triple $ is next inserted character, since the text is fully compressed and there is no next character

C(r)