# Algorithms Project:

## Data Compression and Decompression

In Data Compression or bit-rate reduction involves encoding information using fewer bits than the original representation. Compression can be either lossy or lossless. Lossless compression reduces bits by identifying and eliminating statistical redundancy. No information is lost in lossless compression. Lossy compression reduces bits by identifying marginally important information and removing it.

The Compression and Decompression involves many Algorithms. Some of them are:

* Lempel-Ziv-Storer-Szymanski(LZSS)
* Run Length Algorithm
* Burrows Wheeler Transform(BWT)
* Move To Front
* Huffman Algorithm

## Lempel-Ziv-Storer-Szymanski(LZSS)

### Encoding

LZSS is a dictionary encoding technique. Unlike Huffman coding, which attempts to reduce the average amount of bits required to represent a symbol, LZSS attempts to replace a string of symbols with a reference to a dictionary location for the same string. It is intended that the dictionary reference should be shorter than the string it replaces.

Encoding requires Following steps:-

Step 1. Initialize the dictionary to a known value.

Step 2. Read an original string that is the length of the maximum allowable match.

Step 3. Search for the longest matching string in the dictionary.

Step 4. If a match is found greater than or equal to the minimum allowable match length:

* Write the encoded flag, then the offset and length to the encoded output.
* Otherwise, write the original flag and the first original symbol to the encoded output.

Step 5. Shift a copy of the symbols written to the encoded output from the original string to the dictionary.

Step 6. Read a number of symbols from the original input equal to the number of symbols written in Step 4.

Step 7. Repeat from Step 3, until all the entire input has been encoded.

### Decoding

The LZSS decoding process is less resource intensive than the LZSS encoding process. The encoding process requires that the dictionary is searched for matches to the string to be encoding. Decoding an offset and length combination only requires going to a dictionary offset and copying the specified number of symbols. No searching is required.

Decoding requires following steps:-

Step 1. Initialize the dictionary to a known value.

Step 2. Read the encoded/not encoded flag.

Step 3. If the flag indicates an encoded string:

A. Read the encoded length and offset, then copy the specified number of symbols from the dictionary to the decoded output.

B. Otherwise, read the next character and write it to the decoded output.

Step 4. Shift a copy of the symbols written to the decoded output into the dictionary.

Step 5. Repeat from Step 2, until all the entire input has been decoded.

EXAMPLE:

Original text file – 54 bytes

Compressed text file – 17 bytes (31%)

It can also compress image

******

Original image -1.07 mb

Compressed image – 137 kb

## RUN LENGTH ALGORITHM

### Encoding

**Run-length encoding** (**RLE**) is a very simple form of 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: 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.

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

WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWWWWWWWWWWBWWWWWWWWWWWWWW

If we apply the run-length encoding (RLE) data compression algorithm to the above hypothetical scan line, we get the following:

12W1B12W3B24W1B14W

This is to be interpreted as twelve Ws, one B, twelve Ws, three Bs, etc.

The run-length code represents the original 67 characters in only 18. Of course, the actual format used for the storage of images is generally binary rather than [ASCII](http://en.wikipedia.org/wiki/ASCII) characters like this, but the principle remains the same.

RUNNING EXAMPLE :-

***Orignal text file – 26bytes***

***Compressed text file- 14 bytes***

## BURROW-WHEELER ALGORITHM

When a character string is transformed by the BWT, none of its characters change value. The transformation [permutes](http://en.wikipedia.org/wiki/Permutation) the order of the characters. If the original string had several substrings that occurred often, then the transformed string will have several places where a single character is repeated multiple times in a row. This is useful for compression, since it tends to be easy to compress a string that has runs of repeated characters by techniques such as move-to-front transform and [run-length encoding](http://en.wikipedia.org/wiki/Run-length_encoding).

For example:

|  |  |
| --- | --- |
| **Input** | SIX.MIXED.PIXIES.SIFT.SIXTY.PIXIE.DUST.BOXES |
| **Output** | TEXYDST.E.IXIXIXXSSMPPS.B..E.S.EUSFXDIIOIIIT |

The output is easier to compress because it has many repeated characters. In fact, in the transformed string, there are a total of six runs of identical characters: XX, SS, PP, .., II, and III, which together make 13 out of the 44 characters in it.

### Encoding

The transform is done by sorting all rotations of the text in lexicographic order, then taking the last column. For example, the text "^BANANA|" is transformed into "BNN^AA|A" through these steps (the red | character indicates the 'EOF' pointer):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Transformation** | | | | |
| **Input** | **All Rotations** | **Sorting All Rows in Alphabetical Order by their first letters** | **Taking Last Column** | **Output Last Column** |
| ^BANANA| | ^BANANA|  |^BANANA  A|^BANAN  NA|^BANA  ANA|^BAN  NANA|^BA  ANANA|^B  BANANA|^ | **A**NANA|^B  **A**NA|^BAN  **A**|^BANAN  **B**ANANA|^  **N**ANA|^BA  **N**A|^BANA  **^**BANANA|  **|**^BANANA | ANANA|^**B**  ANA|^BA**N**  A|^BANA**N**  BANANA|**^**  NANA|^B**A**  NA|^BAN**A**  ^BANANA**|**  |^BANAN**A** | BNN^AA|A |

### Decoding

|  |  |  |  |
| --- | --- | --- | --- |
| **Inverse Transformation** | | | |
| **Input** | | | |
| BNN^AA|A | | | |
| **Add 1** | **Sort 1** | **Add 2** | **Sort 2** |
| B  N  N  ^  A  A  |  A | A  A  A  B  N  N  ^  | | BA  NA  NA  ^B  AN  AN  |^  A| | AN  AN  A|  BA  NA  NA  ^B  |^ |
| **Add 3** | **Sort 3** | **Add 4** | **Sort 4** |
| BAN  NAN  NA|  ^BA  ANA  ANA  |^B  A|^ | ANA  ANA  A|^  BAN  NAN  NA|  ^BA  |^B | BANA  NANA  NA|^  ^BAN  ANAN  ANA|  |^BA  A|^B | ANAN  ANA|  A|^B  BANA  NANA  NA|^  ^BAN  |^BA |
| **Add 5** | **Sort 5** | **Add 6** | **Sort 6** |
| BANAN  NANA|  NA|^B  ^BANA  ANANA  ANA|^  |^BAN  A|^BA | ANANA  ANA|^  A|^BA  BANAN  NANA|  NA|^B  ^BANA  |^BAN | BANANA  NANA|^  NA|^BA  ^BANAN  ANANA|  ANA|^B  |^BANA  A|^BAN | ANANA|  ANA|^B  A|^BAN  BANANA  NANA|^  NA|^BA  ^BANAN  |^BANA |
| **Add 7** | **Sort 7** | **Add 8** | **Sort 8** |
| BANANA|  NANA|^B  NA|^BAN  ^BANANA  ANANA|^  ANA|^BA  |^BANAN  A|^BANA | ANANA|^  ANA|^BA  A|^BANA  BANANA|  NANA|^B  NA|^BAN  ^BANANA  |^BANAN | BANANA|^  NANA|^BA  NA|^BANA  ^BANANA|  ANANA|^B  ANA|^BAN  |^BANANA  A|^BANAN | ANANA|^B  ANA|^BAN  A|^BANAN  BANANA|^  NANA|^BA  NA|^BANA  ^BANANA|  |^BANANA |
| **Output** | | | |
| ^BANANA| | | | |

Example :-

Normally BWT increases size but when used with other algorithm, it give brilliant compression.

Like we use Run Length Algorithm after BWT

## MOVE TO FRONT ALGORITHM

### Encoding

The move-to-front transform is an encoding of data(typically a stream of bytes) designed to improve the performance of entropy encoding techniques of compression. When efficiently implemented, it is fast enough that its benefits usually justify including it as an extra step in data compression algorithm

The main idea is that each symbol in the data is replaced by its index in the stack of “recently used symbols”. For example, long sequences of identical symbols are replaced by as many zeroes, whereas when a symbol that has not been used in a long time appears, it is replaced with a large number. Thus at the end the data is transformed into a sequence of integers; if the data exhibits a lot of local correlations, then these integers tend to be small.

An example will shed some light on how the transform works. Imagine instead of bytes, we are encoding values in a-z. We wish to transform the following sequence:

bananaaa

By convention, the list is initially (abcdefghijklmnopqrstuvwxyz). The first letter in the sequence is b, which appears at index 1 (the list is indexed from 0 to 25). We put a 1 to the output stream:

1

The b moves to the front of the list, producing (bacdefghijklmnopqrstuvwxyz). The next letter is a, which now appears at index 1. So we add a 1 to the output stream. We have:

1,1

and we move back the letter a to the top of the list. Continuing this way, we find that the sequence is encoded by:

1,1,13,1,1,1,0,0

|  |  |  |
| --- | --- | --- |
| **Iteration** | **Sequence** | **List** |
| **b**ananaaa | 1 | (abcdefghijklmnopqrstuvwxyz) |
| b**a**nanaaa | 1,1 | (bacdefghijklmnopqrstuvwxyz) |
| ba**n**anaaa | 1,1,13 | (abcdefghijklmnopqrstuvwxyz) |
| ban**a**naaa | 1,1,13,1 | (nabcdefghijklmopqrstuvwxyz) |
| bana**n**aaa | 1,1,13,1,1 | (anbcdefghijklmopqrstuvwxyz) |
| banan**a**aa | 1,1,13,1,1,1 | (nabcdefghijklmopqrstuvwxyz) |
| banana**a**a | 1,1,13,1,1,1,0 | (anbcdefghijklmopqrstuvwxyz) |
| bananaa**a** | 1,1,13,1,1,1,0,0 | (anbcdefghijklmopqrstuvwxyz) |
| Final | 1,1,13,1,1,1,0,0 | (anbcdefghijklmopqrstuvwxyz) |

Normally it doesn’t change the size but it is used in different algorithm to compress the data such as bzip2, deflate.

## HUFFMAN ALGORITHM

Huffman coding is an entropy encoding algorithm used for lossless data compression. The term refers to the use of VARIABLE- LENGTH CODING.

### Encoding

The technique works by creating a binary tree of nodes. These can be stored in a regular array, the size of which depends on the number of symbols, n. A node can be either a leaf node or an internal node. Initially, all nodes are leaf nodes, which contain the **symbol** itself, the **weight** (frequency of appearance) of the symbol and optionally, a link to a **parent** node which makes it easy to read the code (in reverse) starting from a leaf node. Internal nodes contain symbol **weight**, links to **two child nodes** and the optional link to a **parent** node. As a common convention, bit '0' represents following the left child and bit '1' represents following the right child. A finished tree has up to n leaf nodes and n-1 internal nodes. A Huffman tree that omits unused symbols produces the most optimal code lengths.

The simplest construction algorithm uses a [priority queue](http://en.wikipedia.org/wiki/Priority_queue) where the node with lowest probability is given highest priority:

1. Create a leaf node for each symbol and add it to the priority queue.
2. While there is more than one node in the queue:
   1. Remove the two nodes of highest priority (lowest probability) from the queue
   2. Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.
   3. Add the new node to the queue.
3. The remaining node is the root node and the tree is complete.

Since efficient priority queue data structures require O(log *n*) time per insertion, and a tree with *n* leaves has 2*n*−1 nodes, this algorithm operates in O(*n*log *n*) time, where *n* is the number of symbols.

If the symbols are sorted by probability, there is a [linear-time](http://en.wikipedia.org/wiki/Linear-time) (O(*n*)) method to create a Huffman tree using two [queues](http://en.wikipedia.org/wiki/Queue_(data_structure)), the first one containing the initial weights (along with pointers to the associated leaves), and combined weights (along with pointers to the trees) being put in the back of the second queue. This assures that the lowest weight is always kept at the front of one of the two queues:

1. Start with as many leaves as there are symbols.
2. Enqueue all leaf nodes into the first queue (by probability in increasing order so that the least likely item is in the head of the queue).
3. While there is more than one node in the queues:
   1. Dequeue the two nodes with the lowest weight by examining the fronts of both queues.
   2. Create a new internal node, with the two just-removed nodes as children (either node can be either child) and the sum of their weights as the new weight.
   3. Enqueue the new node into the rear of the second queue.
4. The remaining node is the root node; the tree has now been generated.

### Decoding

 The process of decompression is simply a matter of translating the stream of prefix codes to individual byte values, usually by traversing the Huffman tree node by node as each bit is read from the input stream (reaching a leaf node necessarily terminates the search for that particular byte value). Before this can take place, however, the Huffman tree must be somehow reconstructed.



