Lzip

LZMA lossless data compressor
for Lzip version 1.16, 26 August 2014

by Antonio Diaz Diaz

Table of Contents

This manual is for Lzip (version 1.16, 26 August 2014).

1 Introduction

Lzip is a lossless data compressor with a user interface similar to the one of gzip or bzip2. Lzip is about as fast as gzip, compresses most files more than bzip2, and is better than both from a data recovery perspective. Lzip is a clean implementation of the LZMA (Lempel-Ziv-Markov chain-Algorithm) "algorithm".

The lzip file format is designed for long-term data archiving, taking into account both data integrity and decoder availability:

- The lzip format provides very safe integrity checking and some data recovery means. The lziprecover program can repair bit-flip errors (one of the most common forms of data corruption) in lzip files, and provides data recovery capabilities, including error-checked merging of damaged copies of a file.
- The lzip format is as simple as possible (but not simpler). The lzip manual provides the code of a simple decompressor along with a detailed explanation of how it works, so that with the only help of the lzip manual it would be possible for a digital archaeologist to extract the data from a lzip file long after quantum computers eventually render LZMA obsolete.
- Additionally lzip is copylefted, which guarantees that it will remain free forever.

A nice feature of the lzip format is that a corrupt byte is easier to repair the nearer it is from the beginning of the file. Therefore, with the help of lziprecover, losing an entire archive just because of a corrupt byte near the beginning is a thing of the past.

The member trailer stores the 32-bit CRC of the original data, the size of the original data and the size of the member. These values, together with the value remaining in the range decoder and the end-of-stream marker, provide a 4 factor integrity checking which guarantees that the decompressed version of the data is identical to the original. This guards against corruption of the compressed data, and against undetected bugs in lzip (hopefully very unlikely). The chances of data corruption going undetected are microscopic. Be aware, though, that the check occurs upon decompression, so it can only tell you that something is wrong. It can't help you recover the original uncompressed data.

Lzip uses the same well-defined exit status values used by bzip2, which makes it safer than compressors returning ambiguous warning values (like gzip) when it is used as a back end for other programs like tar or zutils.

The amount of memory required for compression is about 1 or 2 times the dictionary size limit (1 if input file size is less than dictionary size limit, else 2) plus 9 times the dictionary size really used. The option '`-0`' is special and only requires about 1.5 MiB at most. The amount of memory required for decompression is about 46 kB larger than the dictionary size really used.

Lzip will automatically use the smallest possible dictionary size for each file without exceeding the given limit. Keep in mind that the decompression memory requirement is affected at compression time by the choice of dictionary size limit.

When compressing, lzip replaces every file given in the command line with a compressed version of itself, with the name "original_name.lz". When decompressing, lzip attempts to guess the name for the decompressed file from that of the compressed file as follows:

filename.lz becomes filename

filename.tlz	becomes	filename.tar
anyothername	becomes	anyothername.out

(De)compressing a file is much like copying or moving it; therefore lzip preserves the access and modification dates, permissions, and, when possible, ownership of the file just as "cp -p" does. (If the user ID or the group ID can't be duplicated, the file permission bits S_ISUID and S_ISGID are cleared).

Lzip is able to read from some types of non regular files if the '`--stdout`' option is specified.

If no file names are specified, lzip compresses (or decompresses) from standard input to standard output. In this case, lzip will decline to write compressed output to a terminal, as this would be entirely incomprehensible and therefore pointless.

Lzip will correctly decompress a file which is the concatenation of two or more compressed files. The result is the concatenation of the corresponding uncompressed files. Integrity testing of concatenated compressed files is also supported.

Lzip can produce multi-member files and safely recover, with lziprecover, the undamaged members in case of file damage. Lzip can also split the compressed output in volumes of a given size, even when reading from standard input. This allows the direct creation of multivolume compressed tar archives.

Lzip is able to compress and decompress streams of unlimited size by automatically creating multi-member output. The members so created are large, about 64 PiB each.

2 Algorithm

There is no such thing as a "LZMA algorithm"; it is more like a "LZMA coding scheme". For example, the option '-0' of lzip uses the scheme in almost the simplest way possible; issuing the longest match it can find, or a literal byte if it can't find a match. Inversely, a much more elaborated way of finding coding sequences of minimum price than the one currently used by lzip could be developed, and the resulting sequence could also be coded using the LZMA coding scheme.

Lzip currently implements two variants of the LZMA algorithm; fast (used by option -0) and normal (used by all other compression levels).

The high compression of LZMA comes from combining two basic, well-proven compression ideas: sliding dictionaries (LZ77/78) and markov models (the thing used by every compression algorithm that uses a range encoder or similar order-0 entropy coder as its last stage) with segregation of contexts according to what the bits are used for.

Lzip is a two stage compressor. The first stage is a Lempel-Ziv coder, which reduces redundancy by translating chunks of data to their corresponding distance-length pairs. The second stage is a range encoder that uses a different probability model for each type of data; distances, lengths, literal bytes, etc.

Here is how it works, step by step:

1) The member header is written to the output stream.

2) The first byte is coded literally, because there are no previous bytes to which the match finder can refer to.

3) The main encoder advances to the next byte in the input data and calls the match finder.

4) The match finder fills an array with the minimum distances before the current byte where a match of a given length can be found.

5) Go back to step 3 until a sequence (formed of pairs, repeated distances and literal bytes) of minimum price has been formed. Where the price represents the number of output bits produced.

6) The range encoder encodes the sequence produced by the main encoder and sends the produced bytes to the output stream.

7) Go back to step 3 until the input data are finished or until the member or volume size limits are reached.

8) The range encoder is flushed.

9) The member trailer is written to the output stream.

10) If there are more data to compress, go back to step 1.

The ideas embodied in lzip are due to (at least) the following people: Abraham Lempel and Jacob Ziv (for the LZ algorithm), Andrey Markov (for the definition of Markov chains), G.N.N. Martin (for the definition of range encoding), Igor Pavlov (for putting all the above together in LZMA), and Julian Seward (for bzip2's CLI).

3 Invoking lzip

The format for running lzip is:

```
lzip [options] [files]
```

Lzip supports the following options:

'`-h`'
'`--help`' Print an informative help message describing the options and exit.

'`-V`'
'`--version`'
Print the version number of lzip on the standard output and exit.

'`-b bytes`'
'`--member-size=bytes`'
Set the member size limit to *bytes*. A small member size may degrade compression ratio, so use it only when needed. Valid values range from 100 kB to 64 PiB. Defaults to 64 PiB.

'`-c`'
'`--stdout`'
Compress or decompress to standard output. Needed when reading from a named pipe (fifo) or from a device. Use it to recover as much of the uncompressed data as possible when decompressing a corrupt file.

'`-d`'
'`--decompress`'
Decompress.

'`-f`'
'`--force`' Force overwrite of output files.

'`-F`'
'`--recompress`'
Force recompression of files whose name already has the '`.lz`' or '`.tlz`' suffix.

'`-k`'
'`--keep`' Keep (don't delete) input files during compression or decompression.

'`-m bytes`'
'`--match-length=bytes`'
Set the match length limit in bytes. After a match this long is found, the search is finished. Valid values range from 5 to 273. Larger values usually give better compression ratios but longer compression times.

'`-o file`'
'`--output=file`'
When reading from standard input and '`--stdout`' has not been specified, use '*file*' as the virtual name of the uncompressed file. This produces a file named '*file*' when decompressing, a file named '*file*`.lz`' when compressing, and several files named '*file*`00001.lz`', '*file*`00002.lz`', etc, when compressing and splitting the output in volumes.

'`-q`'
'`--quiet`' Quiet operation. Suppress all messages.

'`-s` *bytes*'
'`--dictionary-size=`*bytes*'
Set the dictionary size limit in bytes. Valid values range from 4 KiB to 512 MiB. Lzip will use the smallest possible dictionary size for each file without exceeding this limit. Note that dictionary sizes are quantized. If the specified size does not match one of the valid sizes, it will be rounded upwards by adding up to (*bytes* / 16) to it.

For maximum compression you should use a dictionary size limit as large as possible, but keep in mind that the decompression memory requirement is affected at compression time by the choice of dictionary size limit.

'`-S` *bytes*'
'`--volume-size=`*bytes*'
Split the compressed output into several volume files with names '`original_name00001.lz`', '`original_name00002.lz`', etc, and set the volume size limit to *bytes*. Each volume is a complete, maybe multi-member, lzip file. A small volume size may degrade compression ratio, so use it only when needed. Valid values range from 100 kB to 4 EiB.

'`-t`'
'`--test`' Check integrity of the specified file(s), but don't decompress them. This really performs a trial decompression and throws away the result. Use it together with '`-v`' to see information about the file.

'`-v`'
'`--verbose`'
Verbose mode.
When compressing, show the compression ratio for each file processed. A second '`-v`' shows the progress of compression.
When decompressing or testing, further -v's (up to 4) increase the verbosity level, showing status, compression ratio, dictionary size, trailer contents (CRC, data size, member size), and up to 6 bytes of trailing garbage (if any).

'`-0 .. -9`' Set the compression parameters (dictionary size and match length limit) as shown in the table below. Note that '`-9`' can be much slower than '`-0`'. These options have no effect when decompressing.

The bidimensional parameter space of LZMA can't be mapped to a linear scale optimal for all files. If your files are large, very repetitive, etc, you may need to use the '`--match-length`' and '`--dictionary-size`' options directly to achieve optimal performance. For example, '`-9m64`' usually compresses executables more (and faster) than '`-9`'.

Level	Dictionary size	Match length limit
-0	64 KiB	16 bytes
-1	1 MiB	5 bytes
-2	1.5 MiB	6 bytes
-3	2 MiB	8 bytes

-4	3 MiB	12 bytes
-5	4 MiB	20 bytes
-6	8 MiB	36 bytes
-7	16 MiB	68 bytes
-8	24 MiB	132 bytes
-9	32 MiB	273 bytes

'`--fast`'
'`--best`' Aliases for GNU gzip compatibility.

Numbers given as arguments to options may be followed by a multiplier and an optional '`B`' for "byte".

Table of SI and binary prefixes (unit multipliers):

Prefix	Value		Prefix	Value
k	kilobyte (10^3 = 1000)		Ki	kibibyte (2^10 = 1024)
M	megabyte (10^6)		Mi	mebibyte (2^20)
G	gigabyte (10^9)		Gi	gibibyte (2^30)
T	terabyte (10^12)		Ti	tebibyte (2^40)
P	petabyte (10^15)		Pi	pebibyte (2^50)
E	exabyte (10^18)		Ei	exbibyte (2^60)
Z	zettabyte (10^21)		Zi	zebibyte (2^70)
Y	yottabyte (10^24)		Yi	yobibyte (2^80)

Exit status: 0 for a normal exit, 1 for environmental problems (file not found, invalid flags, I/O errors, etc), 2 to indicate a corrupt or invalid input file, 3 for an internal consistency error (eg, bug) which caused lzip to panic.

4 File format

Perfection is reached, not when there is no longer anything to add, but when there is no longer anything to take away.
— Antoine de Saint-Exupery

In the diagram below, a box like this:

```
+---+
|   | <-- the vertical bars might be missing
+---+
```

represents one byte; a box like this:

```
+==============+
|              |
+==============+
```

represents a variable number of bytes.

A lzip file consists of a series of "members" (compressed data sets). The members simply appear one after another in the file, with no additional information before, between, or after them.

Each member has the following structure:

```
+--+--+--+--+----+----+=============+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| ID string | VN | DS | Lzma stream | CRC32 |   Data size   |  Member size  |
+--+--+--+--+----+----+=============+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

All multibyte values are stored in little endian order.

'`ID string`'
: A four byte string, identifying the lzip format, with the value "LZIP" (0x4C, 0x5A, 0x49, 0x50).

'`VN (version number, 1 byte)`'
: Just in case something needs to be modified in the future. 1 for now.

'`DS (coded dictionary size, 1 byte)`'
: Lzip divides the distance between any two powers of 2 into 8 equally spaced intervals, named "wedges". The dictionary size is calculated by taking a power of 2 (the base size) and substracting from it a number of wedges between 0 and 7. The size of a wedge is (base_size / 16).
Bits 4-0 contain the base 2 logarithm of the base size (12 to 29).
Bits 7-5 contain the number of wedges (0 to 7) to substract from the base size to obtain the dictionary size.
Example: 0xD3 = 2^19 - 6 * 2^15 = 512 KiB - 6 * 32 KiB = 320 KiB
Valid values for dictionary size range from 4 KiB to 512 MiB.

'`Lzma stream`'
: The lzma stream, finished by an end of stream marker. Uses default values for encoder properties. See the chapter '`Stream format`' (see Chapter 5 [Stream format], page 10) for a complete description.

‘`CRC32 (4 bytes)`’
: CRC of the uncompressed original data.

‘`Data size (8 bytes)`’
: Size of the uncompressed original data.

‘`Member size (8 bytes)`’
: Total size of the member, including header and trailer. This field acts as a distributed index, allows the verification of stream integrity, and facilitates safe recovery of undamaged members from multi-member files.

5 Format of the LZMA stream in lzip files

The LZMA algorithm has three parameters, called "special LZMA properties", to adjust it for some kinds of binary data. These parameters are; '`literal_context_bits`' (with a default value of 3), '`literal_pos_state_bits`' (with a default value of 0), and '`pos_state_bits`' (with a default value of 2). As a general purpose compressor, lzip only uses the default values for these parameters.

Lzip also finishes the LZMA stream with an "End Of Stream" marker (the distance-length pair 0xFFFFFFFFU, 2), which in conjunction with the "member size" field in the member trailer allows the verification of stream integrity. The LZMA stream in lzip files always has these two features (default properties and EOS marker) and is referred to in this document as LZMA-302eos or LZMA-lzip.

The second stage of LZMA is a range encoder that uses a different probability model for each type of symbol; distances, lengths, literal bytes, etc. Range encoding conceptually encodes all the symbols of the message into one number. Unlike Huffman coding, which assigns to each symbol a bit-pattern and concatenates all the bit-patterns together, range encoding can compress one symbol to less than one bit. Therefore the compressed data produced by a range encoder can't be split in pieces that could be individually described.

It seems that the only way of describing the LZMA-302eos stream is describing the algorithm that decodes it. And given the many details about the range decoder that need to be described accurately, the source code of a real decoder seems the only appropriate reference to use.

What follows is a description of the decoding algorithm for LZMA-302eos streams using as reference the source code of "lzd", an educational decompressor for lzip files which can be downloaded from the lzip download directory. The source code of lzd is included in appendix A. Appendix A [Reference source code], page 16

5.1 What is coded

The LZMA stream includes literals, matches and repeated matches (matches reusing a recently used distance). There are 7 different coding sequences:

Bit sequence	**Name**	**Description**
0 + byte	literal	literal byte
1 + 0 + len + dis	match	distance-length pair
1 + 1 + 0 + 0	shortrep	1 byte match at latest used distance
1 + 1 + 0 + 1 + len	rep0	len bytes match at latest used distance
1 + 1 + 1 + 0 + len	rep1	len bytes match at second latest used distance
1 + 1 + 1 + 1 + 0 + len	rep2	len bytes match at third latest used distance
1 + 1 + 1 + 1 + 1 + len	rep3	len bytes match at fourth latest used distance

In the following tables, multi-bit sequences are coded in normal order, from MSB to LSB, except where noted otherwise.

Lengths (the '`len`' in the table above) are coded as follows:

Bit sequence	**Description**
0 + 3 bits	lengths from 2 to 9
1 + 0 + 3 bits	lengths from 10 to 17
1 + 1 + 8 bits	lengths from 18 to 273

The coding of distances is a little more complicated. LZMA divides the interval between any two powers of 2 into 2 halves, named slots. As possible distances range from 0 to (2^32 - 1), there are 64 slots (0 to 63). The slot number is context-coded in 6 bits. '`direct_bits`' are the remaining bits (from 0 to 30) needed to form a complete distance, and are calculated as (slot >> 1) - 1. If a distance needs 6 or more direct_bits, the last 4 bits are coded separately. The last piece (direct_bits for distances 4 to 127 or the last 4 bits for distances >= 128) is context-coded in reverse order (from LSB to MSB). For distances >= 128, the '`direct_bits - 4`' part is coded with fixed 0.5 probability.

Bit sequence	**Description**
slot	distances from 0 to 3
slot + direct_bits	distances from 4 to 127
slot + (direct_bits - 4) + 4 bits	distances from 128 to 2^32 - 1

5.2 The coding contexts

These contexts ('`Bit_model`' in the source), are integers or arrays of integers representing the probability of the corresponding bit being 0.

The indices used in these arrays are:

'`state`' A state machine ('`State`' in the source) with 12 states (0 to 11), coding the latest 2 to 4 types of sequences processed. The initial state is 0.

'`pos_state`'
Value of the 2 least significant bits of the current position in the decoded data.

'`literal_state`'
Value of the 3 most significant bits of the latest byte decoded.

'`len_state`'
Coded value of length (length - 2), with a maximum of 3. The resulting value is in the range 0 to 3.

In the following table, '`!literal`' is any sequence except a literal byte. '`rep`' is any one of '`rep0`', '`rep1`', '`rep2`' or '`rep3`'. The types of previous sequences corresponding to each state are:

State	**Types of previous sequences**
0	literal, literal, literal
1	match, literal, literal
2	rep or (!literal, shortrep), literal, literal
3	literal, shortrep, literal, literal

4	match, literal
5	rep or (!literal, shortrep), literal
6	literal, shortrep, literal
7	literal, match
8	literal, rep
9	literal, shortrep
10	!literal, match
11	!literal, (rep or shortrep)

The contexts for decoding the type of coding sequence are:

Name	**Indices**	**Used when**
bm_match	state, pos_state	sequence start
bm_rep	state	after sequence 1
bm_rep0	state	after sequence 11
bm_rep1	state	after sequence 111
bm_rep2	state	after sequence 1111
bm_len	state, pos_state	after sequence 110

The contexts for decoding distances are:

Name	**Indices**	**Used when**
bm_dis_slot	len_state, bit tree	distance start
bm_dis	reverse bit tree	after slots 4 to 13
bm_align	reverse bit tree	for distances >= 128, after fixed probability bits

There are two separate sets of contexts for lengths ('`Len_model`' in the source). One for normal matches, the other for repeated matches. The contexts in each Len_model are (see '`decode_len`' in the source):

Name	**Indices**	**Used when**
choice1	none	length start
choice2	none	after sequence 1
bm_low	pos_state, bit tree	after sequence 0
bm_mid	pos_state, bit tree	after sequence 10
bm_high	bit tree	after sequence 11

The context array '`bm_literal`' is special. In principle it acts as a normal bit tree context, the one selected by '`literal_state`'. But if the previous decoded byte was not a literal, two other bit tree contexts are used depending on the value of each bit in '`match_byte`' (the byte at the latest used distance), until a bit is decoded that is different from its corresponding bit in '`match_byte`'. After the first difference is found, the rest of the byte is decoded using the normal bit tree context. (See '`decode_matched`' in the source).

5.3 The range decoder

The LZMA stream is consumed one byte at a time by the range decoder. (See '`normalize`' in the source). Every byte consumed produces a variable number of decoded bits, depending on how well these bits agree with their context. (See '`decode_bit`' in the source).

The range decoder state consists of two unsigned 32-bit variables; `range` (representing the most significant part of the range size not yet decoded), and `code` (representing the current point within `range`). `range` is initialized to (2^32 - 1), and `code` is initialized to 0.

The range encoder produces a first 0 byte that must be ignored by the range decoder. This is done by shifting 5 bytes in the initialization of `code` instead of 4. (See the '`Range_decoder`' constructor in the source).

5.4 Decoding the LZMA stream

After decoding the member header and obtaining the dictionary size, the range decoder is initialized and then the LZMA decoder enters a loop (See '`decode_member`' in the source) where it invokes the range decoder with the appropriate contexts to decode the different coding sequences (matches, repeated matches, and literal bytes), until the "End Of Stream" marker is decoded.

6 A small tutorial with examples

WARNING! Even if lzip is bug-free, other causes may result in a corrupt compressed file (bugs in the system libraries, memory errors, etc). Therefore, if the data you are going to compress are important, give the '`--keep`' option to lzip and do not remove the original file until you verify the compressed file with a command like '`lzip -cd file.lz | cmp file -`'.

Example 1: Replace a regular file with its compressed version '`file.lz`' and show the compression ratio.

```
lzip -v file
```

Example 2: Like example 1 but the created '`file.lz`' is multi-member with a member size of 1 MiB. The compression ratio is not shown.

```
lzip -b 1MiB file
```

Example 3: Restore a regular file from its compressed version '`file.lz`'. If the operation is successful, '`file.lz`' is removed.

```
lzip -d file.lz
```

Example 4: Verify the integrity of the compressed file '`file.lz`' and show status.

```
lzip -tv file.lz
```

Example 5: Compress a whole floppy in /dev/fd0 and send the output to '`file.lz`'.

```
lzip -c /dev/fd0 > file.lz
```

Example 6: Decompress '`file.lz`' partially until 10 KiB of decompressed data are produced.

```
lzip -cd file.lz | dd bs=1024 count=10
```

Example 7: Decompress '`file.lz`' partially from decompressed byte 10000 to decompressed byte 15000 (5000 bytes are produced).

```
lzip -cd file.lz | dd bs=1000 skip=10 count=5
```

Example 8: Create a multivolume compressed tar archive with a volume size of 1440 KiB.

```
tar -c some_directory | lzip -S 1440KiB -o volume_name
```

Example 9: Extract a multivolume compressed tar archive.

```
lzip -cd volume_name*.lz | tar -xf -
```

Example 10: Create a multivolume compressed backup of a large database file with a volume size of 650 MB, where each volume is a multi-member file with a member size of 32 MiB.

```
lzip -b 32MiB -S 650MB big_db
```

7 Reporting bugs

There are probably bugs in lzip. There are certainly errors and omissions in this manual. If you report them, they will get fixed. If you don't, no one will ever know about them and they will remain unfixed for all eternity, if not longer.

If you find a bug in lzip, please send electronic mail to `lzip-bug@nongnu.org`. Include the version number, which you can find by running '`lzip --version`'.

Appendix A Reference source code

```
/*  Lzd - Educational decompressor for lzip files
    Copyright (C) 2013, 2014 Antonio Diaz Diaz.

    This program is free software: you have unlimited permission
    to copy, distribute and modify it.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
*/
/*
    Exit status: 0 for a normal exit, 1 for environmental problems
    (file not found, invalid flags, I/O errors, etc), 2 to indicate a
    corrupt or invalid input file.
*/

#include <algorithm>
#include <cerrno>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <stdint.h>
#include <unistd.h>


class State
  {
  int st;

public:
  enum { states = 12 };
  State() : st( 0 ) {}
  int operator()() const { return st; }
  bool is_char() const { return st < 7; }

  void set_char()
    {
    static const int next[states] = { 0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 4, 5 };
    st = next[st];
    }
  void set_match()     { st = ( st < 7 ) ? 7 : 10; }
  void set_rep()       { st = ( st < 7 ) ? 8 : 11; }
  void set_short_rep() { st = ( st < 7 ) ? 9 : 11; }
  };
```

```
enum {
  min_dictionary_size = 1 << 12,
  max_dictionary_size = 1 << 29,
  literal_context_bits = 3,
  pos_state_bits = 2,
  pos_states = 1 << pos_state_bits,
  pos_state_mask = pos_states - 1,

  len_states = 4,
  dis_slot_bits = 6,
  start_dis_model = 4,
  end_dis_model = 14,
  modeled_distances = 1 << (end_dis_model / 2),         // 128
  dis_align_bits = 4,
  dis_align_size = 1 << dis_align_bits,

  len_low_bits = 3,
  len_mid_bits = 3,
  len_high_bits = 8,
  len_low_symbols = 1 << len_low_bits,
  len_mid_symbols = 1 << len_mid_bits,
  len_high_symbols = 1 << len_high_bits,
  max_len_symbols = len_low_symbols + len_mid_symbols + len_high_symbols,

  min_match_len = 2,                                    // must be 2

  bit_model_move_bits = 5,
  bit_model_total_bits = 11,
  bit_model_total = 1 << bit_model_total_bits };

struct Bit_model
  {
  int probability;
  Bit_model() : probability( bit_model_total / 2 ) {}
  };

struct Len_model
  {
  Bit_model choice1;
  Bit_model choice2;
  Bit_model bm_low[pos_states][len_low_symbols];
  Bit_model bm_mid[pos_states][len_mid_symbols];
  Bit_model bm_high[len_high_symbols];
  };
```

```
class CRC32
  {
  uint32_t data[256];           // Table of CRCs of all 8-bit messages.

public:
  CRC32()
    {
    for( unsigned n = 0; n < 256; ++n )
      {
      unsigned c = n;
      for( int k = 0; k < 8; ++k )
        { if( c & 1 ) c = 0xEDB88320U ^ ( c >> 1 ); else c >>= 1; }
      data[n] = c;
      }
    }

  void update_buf( uint32_t & crc, const uint8_t * const buffer,
                   const int size ) const
    {
    for( int i = 0; i < size; ++i )
      crc = data[(crc^buffer[i])&0xFF] ^ ( crc >> 8 );
    }
  };

const CRC32 crc32;


typedef uint8_t File_header[6]; // 0-3 magic, 4 version, 5 coded_dict_size

typedef uint8_t File_trailer[20];
                        //  0-3  CRC32 of the uncompressed data
                        //  4-11 size of the uncompressed data
                        // 12-19 member size including header and trailer

class Range_decoder
  {
  uint32_t code;
  uint32_t range;

public:
  Range_decoder() : code( 0 ), range( 0xFFFFFFFFU )
    {
    for( int i = 0; i < 5; ++i ) code = (code << 8) | get_byte();
    }

  uint8_t get_byte() { return std::getc( stdin ); }
```

```
int decode( const int num_bits )
  {
  int symbol = 0;
  for( int i = 0; i < num_bits; ++i )
    {
    range >>= 1;
    symbol <<= 1;
    if( code >= range ) { code -= range; symbol |= 1; }
    if( range <= 0x00FFFFFFU )                          // normalize
      { range <<= 8; code = (code << 8) | get_byte(); }
    }
  return symbol;
  }

int decode_bit( Bit_model & bm )
  {
  int symbol;
  const uint32_t bound = ( range >> bit_model_total_bits ) * bm.probability;
  if( code < bound )
    {
    range = bound;
    bm.probability += (bit_model_total - bm.probability) >> bit_model_move_bits;
    symbol = 0;
    }
  else
    {
    range -= bound;
    code -= bound;
    bm.probability -= bm.probability >> bit_model_move_bits;
    symbol = 1;
    }
  if( range <= 0x00FFFFFFU )                            // normalize
    { range <<= 8; code = (code << 8) | get_byte(); }
  return symbol;
  }

int decode_tree( Bit_model bm[], const int num_bits )
  {
  int symbol = 1;
  for( int i = 0; i < num_bits; ++i )
    symbol = ( symbol << 1 ) | decode_bit( bm[symbol] );
  return symbol - (1 << num_bits);
  }

int decode_tree_reversed( Bit_model bm[], const int num_bits )
  {
  int symbol = decode_tree( bm, num_bits );
```

```
    int reversed_symbol = 0;
    for( int i = 0; i < num_bits; ++i )
      {
      reversed_symbol = ( reversed_symbol << 1 ) | ( symbol & 1 );
      symbol >>= 1;
      }
    return reversed_symbol;
    }

  int decode_matched( Bit_model bm[], const int match_byte )
    {
    Bit_model * const bm1 = bm + 0x100;
    int symbol = 1;
    for( int i = 7; i >= 0; --i )
      {
      const int match_bit = ( match_byte >> i ) & 1;
      const int bit = decode_bit( bm1[(match_bit<<8)+symbol] );
      symbol = ( symbol << 1 ) | bit;
      if( match_bit != bit )
        {
        while( symbol < 0x100 )
          symbol = ( symbol << 1 ) | decode_bit( bm[symbol] );
        break;
        }
      }
    return symbol & 0xFF;
    }

  int decode_len( Len_model & lm, const int pos_state )
    {
    if( decode_bit( lm.choice1 ) == 0 )
      return decode_tree( lm.bm_low[pos_state], len_low_bits );
    if( decode_bit( lm.choice2 ) == 0 )
      return len_low_symbols +
             decode_tree( lm.bm_mid[pos_state], len_mid_bits );
    return len_low_symbols + len_mid_symbols +
           decode_tree( lm.bm_high, len_high_bits );
    }
  };


class LZ_decoder
  {
  unsigned long long partial_data_pos;
  Range_decoder rdec;
  const unsigned dictionary_size;
  uint8_t * const buffer;       // output buffer
```

```
  unsigned pos;                  // current pos in buffer
  unsigned stream_pos;           // first byte not yet written to stdout
  uint32_t crc_;

  void flush_data();

  uint8_t get_byte( const unsigned distance ) const
    {
    unsigned i = pos - distance - 1;
    if( pos <= distance ) i += dictionary_size;
    return buffer[i];
    }

  void put_byte( const uint8_t b )
    {
    buffer[pos] = b;
    if( ++pos >= dictionary_size ) flush_data();
    }

public:
  LZ_decoder( const unsigned dict_size )
    :
    partial_data_pos( 0 ),
    dictionary_size( dict_size ),
    buffer( new uint8_t[dictionary_size] ),
    pos( 0 ),
    stream_pos( 0 ),
    crc_( 0xFFFFFFFFU )
    { buffer[dictionary_size-1] = 0; }          // prev_byte of first byte

  ~LZ_decoder() { delete[] buffer; }

  unsigned crc() const { return crc_ ^ 0xFFFFFFFFU; }
  unsigned long long data_position() const { return partial_data_pos + pos; }

  bool decode_member();
  };


void LZ_decoder::flush_data()
  {
  if( pos > stream_pos )
    {
    const unsigned size = pos - stream_pos;
    crc32.update_buf( crc_, buffer + stream_pos, size );
    errno = 0;
    if( std::fwrite( buffer + stream_pos, 1, size, stdout ) != size )
```

```
      { std::fprintf( stderr, "Write error: %s.\n", std::strerror( errno ) );
        std::exit( 1 ); }
    if( pos >= dictionary_size ) { partial_data_pos += pos; pos = 0; }
    stream_pos = pos;
    }
  }


bool LZ_decoder::decode_member()               // Returns false if error
  {
  Bit_model bm_literal[1<<literal_context_bits][0x300];
  Bit_model bm_match[State::states][pos_states];
  Bit_model bm_rep[State::states];
  Bit_model bm_rep0[State::states];
  Bit_model bm_rep1[State::states];
  Bit_model bm_rep2[State::states];
  Bit_model bm_len[State::states][pos_states];
  Bit_model bm_dis_slot[len_states][1<<dis_slot_bits];
  Bit_model bm_dis[modeled_distances-end_dis_model];
  Bit_model bm_align[dis_align_size];
  Len_model match_len_model;
  Len_model rep_len_model;
  unsigned rep0 = 0;                    // rep[0-3] latest four distances
  unsigned rep1 = 0;                    // used for efficient coding of
  unsigned rep2 = 0;                    // repeated distances
  unsigned rep3 = 0;
  State state;

  while( !std::feof( stdin ) && !std::ferror( stdin ) )
    {
    const int pos_state = data_position() & pos_state_mask;
    if( rdec.decode_bit( bm_match[state()][pos_state] ) == 0 )  // 1st bit
      {
      const uint8_t prev_byte = get_byte( 0 );
      const int literal_state = prev_byte >> ( 8 - literal_context_bits );
      Bit_model * const bm = bm_literal[literal_state];
      if( state.is_char() )
        put_byte( rdec.decode_tree( bm, 8 ) );
      else
        put_byte( rdec.decode_matched( bm, get_byte( rep0 ) ) );
      state.set_char();
      }
    else
      {
      int len;
      if( rdec.decode_bit( bm_rep[state()] ) != 0 )             // 2nd bit
        {
```

```
  if( rdec.decode_bit( bm_rep0[state()] ) != 0 )          // 3rd bit
    {
    unsigned distance;
    if( rdec.decode_bit( bm_rep1[state()] ) == 0 )        // 4th bit
      distance = rep1;
    else
      {
      if( rdec.decode_bit( bm_rep2[state()] ) == 0 )      // 5th bit
        distance = rep2;
      else
        { distance = rep3; rep3 = rep2; }
      rep2 = rep1;
      }
    rep1 = rep0;
    rep0 = distance;
    }
  else
    {
    if( rdec.decode_bit( bm_len[state()][pos_state] ) == 0 ) // 4th bit
      { state.set_short_rep(); put_byte( get_byte( rep0 ) ); continue; }
    }
  state.set_rep();
  len = min_match_len + rdec.decode_len( rep_len_model, pos_state );
  }
else
  {
  rep3 = rep2; rep2 = rep1; rep1 = rep0;
  len = min_match_len + rdec.decode_len( match_len_model, pos_state );
  const int len_state = std::min( len - min_match_len, len_states - 1 );
  const int dis_slot =
    rdec.decode_tree( bm_dis_slot[len_state], dis_slot_bits );
  if( dis_slot < start_dis_model ) rep0 = dis_slot;
  else
    {
    const int direct_bits = ( dis_slot >> 1 ) - 1;
    rep0 = ( 2 | ( dis_slot & 1 ) ) << direct_bits;
    if( dis_slot < end_dis_model )
      rep0 += rdec.decode_tree_reversed( bm_dis + rep0 - dis_slot - 1,
                                         direct_bits );
    else
      {
      rep0 += rdec.decode( direct_bits - dis_align_bits ) << dis_align_bits;
      rep0 += rdec.decode_tree_reversed( bm_align, dis_align_bits );
      if( rep0 == 0xFFFFFFFFU )           // Marker found
        {
        flush_data();
        return ( len == min_match_len );  // End Of Stream marker
```

```
            }
          }
        }
      state.set_match();
      if( rep0 >= dictionary_size || rep0 >= data_position() )
        { flush_data(); return false; }
      }
    for( int i = 0; i < len; ++i ) put_byte( get_byte( rep0 ) );
    }
  }
flush_data();
return false;
}


int main( const int argc, const char * const argv[] )
  {
  if( argc > 1 )
    {
    std::printf( "Lzd %s - Educational decompressor for lzip files.\n",
                 PROGVERSION );
    std::printf( "Study the source to learn how a lzip decompressor works.\n"
                 "See the lzip manual for an explanation of the code.\n"
                 "It is not safe to use lzd for any real work.\n"
                 "\nUsage: %s < file.lz > file\n", argv[0] );
    std::printf( "Lzd decompresses from standard input to standard output.\n"
                 "\nCopyright (C) 2014 Antonio Diaz Diaz.\n"
                 "This is free software: you are free to change and redistribute it.\n"
                 "There is NO WARRANTY, to the extent permitted by law.\n"
                 "Report bugs to lzip-bug@nongnu.org\n"
                 "Lzd home page: http://www.nongnu.org/lzip/lzd.html\n" );
    return 0;
    }

  for( bool first_member = true; ; first_member = false )
    {
    File_header header;
    for( int i = 0; i < 6; ++i ) header[i] = std::getc( stdin );
    if( std::feof( stdin ) || std::memcmp( header, "LZIP", 4 ) != 0 )
      {
      if( first_member )
        { std::fprintf( stderr, "Bad magic number (file not in lzip format)\n" );
          return 2; }
      break;
      }
    if( header[4] != 1 )
      {
```

```
      std::fprintf( stderr, "Version %d member format not supported.\n",
                    header[4] );
      return 2;
      }
    unsigned dict_size = 1 << ( header[5] & 0x1F );
    dict_size -= ( dict_size / 16 ) * ( ( header[5] >> 5 ) & 7 );
    if( dict_size < min_dictionary_size || dict_size > max_dictionary_size )
      { std::fprintf( stderr, "Invalid dictionary size in member header\n" );
        return 2; }

    LZ_decoder decoder( dict_size );
    if( !decoder.decode_member() )
      { std::fprintf( stderr, "Data error\n" ); return 2; }

    File_trailer trailer;
    for( int i = 0; i < 20; ++i ) trailer[i] = std::getc( stdin );
    unsigned crc = 0;
    for( int i = 3; i >= 0; --i ) { crc <<= 8; crc += trailer[i]; }
    unsigned long long data_size = 0;
    for( int i = 11; i >= 4; --i ) { data_size <<= 8; data_size += trailer[i]; }
    if( crc != decoder.crc() || data_size != decoder.data_position() )
      { std::fprintf( stderr, "CRC error\n" ); return 2; }
    }

  if( std::fclose( stdout ) != 0 )
    { std::fprintf( stderr, "Can't close stdout: %s.\n", std::strerror( errno ) );
      return 1; }
  return 0;
  }
```

Concept index