Notes on an Intro to Industrial Programming

As Illustrated by an LZW Implementation

Aaron Gorenstein

July 21, 2024

I'm exploring using more advanced projects for empowering post-baccealaurette self-studies. This is one such example. I have (had) grand plans to make *this very document* a resource to that end. Practically speaking, however, I think I need to do this more first. So this is more a collection of neat thing I'd want to use this LZW implementation to talk about.

Contents

1. Introduction

What Does This Give You? This is *not* going to be an introduction to how LZW works: there are plenty of good resources for that. This is a resource along two dimensions:

1. Those who feel like they understand that introduction, but have questions about what implementation "really looks like" as you get to the nitty-gritty of an algorithm;

2. Those with an undergraduate or introductory background in programming and are curious to see how a homework-level implementation (as might be found on Rosetta code) can be taken to a more professional level.

Why Trust Me? I have no meaningful prior experience in understanding or implementing compression algorithms. I have a few years of graduate study in computational complexity theory (so I have some background on the formal, mathematical side of things), I spent the better part of a decade on the Microsoft C++ compiler (so I'm familiar with applying more formal computer science to industry, as well as industrial C-or-C++ programming), and most recently am trying out distributed systems at MongoDB (so I have my finger on the pulse of cool hip things). I have a great interest in teaching computer science, which I've pursued throughout my education and now with intermittent volunteer opportunities (some of which were quite extensive).

The end result of the code here will *not* be competitive with, say, `gzip`. However, it will employ a considerable number of performance-focused implementation and design decisions. In this way it can be understood as an intermediate step between homework and "real program". Having not implemented `gzip` (which, to be clear, has a different algorithm), of course, to a degree that this is a useful stepping stone is taken on faith. However, the actual design decisions I'll highlight (various `struct` design decisions, for instance) I've all used in various "real" production code.

Why Write This? I enjoy studying computer science, in particular exploring the boundary between theory, academia, and industry, and sharing what I learn. This write-up is partly my own notes so I feel like I will "forever understand" LZW to a deep degree, as well as an avenue to talk about some less common programming tricks in-situ.

2. Design Overview

Let's review how LZW-encoding works:

- We're receiving a stream of bytes, and wish to emit a stream of bytes.

- As we read in our input, we're constantly finding the longest-matching-prefix that we've seen-so-far.
- Once we no longer have a previously-seen-prefix, we map this *new* string (which differs from a previously-seen-prefix by the single, additional, final byte) to a new integer key.
- We emit the key for the *previous* prefix.
- So, that is, we *don't* emit the just-created new key: we instead restart the prefix-match process, starting with that final byte we used last time. (Restarting with that newly-seen character is how the decoder is able to know which character it was.)
- Available features (complications) include bounding the number of prefixes we save in our dictionary, resetting our dictionary (to help in cases where the input dramatically changes character "halfway through" compression), and variable-width encoding of keys.

An apparently-canonical implementation can be found on the internet.

I'll introduce our implementation by comparison with the (single?) design that seems to pervade web resources. The listing in fig. 1, taken from the web-resources for Sedgwick[1] seems to express the common implementation. Of course it accurately implements (in this case) the simpler LZW variant with fixed-width (in this case, 12 bits) encoding. However, resources (mainly Wikipedia, but also that same Sedgwick site and the linked-to chapter on compression, here, and maybe even the original paper?) talks about variable-width.

To be clear, this is obviously a fine implementation to learn LZW—and the more general algorithmic considerations of compression—with. My presentation here is better understood as a *sequel* resource and topic: for those who want to learn more in an industrial perspective. So while we're about to list a number of deficiencies inherit in the code in fig. 1, it's *not* to its detriment.

- The operation is blocking: that is, it will run until termination. An alternative is some functionality that allows for partial processing, after which control is returned to the caller (which may, say, update a progress bar) which can then resume the next "chunk" of processing. This plays a particular role in stability: if you give your program a blank check for as much memory needed to decompress something, you may end up using a *lot* of memory. So it's nice to say "please decompress at most X bytes", or there-abouts.
- More generally there is the notion of "library support". We do have compression algorithms as stand-alone utilities, but you can also imagine a larger piece of software wanting to compress/decompress things in the course of its execution. The previous point is just one example: it may be nice to expose settings and behaviors programmatically, for other software packages to use.

[1]The URL is here

```java
private static final int R = 256;        // number of input chars
private static final int L = 4096;       // number of codewords = 2^W
private static final int W = 12;         // codeword width

// snip

public static void compress() {
    String input = BinaryStdIn.readString();
    TST<Integer> st = new TST<Integer>();

    // since TST is not balanced, it would be better to insert in a different order
    for (int i = 0; i < R; i++)
        st.put("" + (char) i, i);

    int code = R+1;  // R is codeword for EOF

    while (input.length() > 0) {
        String s = st.longestPrefixOf(input);  // Find max prefix match s.
        BinaryStdOut.write(st.get(s), W);      // Print s's encoding.
        int t = s.length();
        if (t < input.length() && code < L)    // Add s to symbol table.
            st.put(input.substring(0, t + 1), code++);
        input = input.substring(t);            // Scan past s in input.
    }
    BinaryStdOut.write(R, W);
    BinaryStdOut.close();
}
```

Figure 1: A Textbook Implementation of LZW-Compress

- There is no debugging support. While the conceit is that this is a “finished” algorithm and presumably once we’re confident it its correctness we wouldn’t do ongoing development, in practice that rarely happens. We may want to add more logging, we may want to insert special hardware-specific subroutines, or something else. Having trace statements, assertions, and similar is important and keeps future development, tractable.

The LZW implementation we’ll develop here will have the following properties:

1. Variable-width encodings.
2. Clear-codes, in which the encoder can put a distinguished key in the stream that tells the dictionary to reset to the default.
3. An interface design—this influences the `encode` and `decode`—that takes into account some practical applications (memory usage, etc.).
4. A harness of the library into a simple CLI program that can be used to compress, decompress, etc.

```c
uint64_t bitwrite_buffer = 0;
uint32_t bitwrite_buffer_size = 0;
const uint32_t BITWRITE_BUFFER_MAX_SIZE = sizeof(bitwrite_buffer) * 8;

void bitwrite_buffer_push_bits(uint32_t v, uint8_t l) {
  ASSERT(bitwrite_buffer_size + l < BITWRITE_BUFFER_MAX_SIZE);
  uint32_t mask = (1 << l) - 1;
  bitwrite_buffer = (bitwrite_buffer << l) | (v & mask);
  bitwrite_buffer_size += l;
}

uint8_t bitwrite_buffer_pop_byte(void) {
  ASSERT(bitwrite_buffer_size >= 8);
  bitwrite_buffer_size -= 8;
  uint8_t b = bitwrite_buffer >> bitwrite_buffer_size;
  return b;
}
```

Figure 2: Bit Write Buffer Implementation

5. Extensive fuzzing and performance testing, and consequent (slight) performance focus.

The core implementation is under 600 lines of `C` code, and corresponding driver less than 300, totalling to fewer than 1,000 lines of code for the full implementation.

3. Implementation Details

3.1. Reading and Writing Bits and Bytes

We can typically only write bytes (that is, 8 bits) to a file at a time. If we encode a stream into 103 keys, each key being 9 bits long, we've set ourselves up for emitting 927 bits—obviously not neatly fitting into some number of bytes.

Solving this problem can be a motivating introduction to bitwise tricks and using those more "exotic" operators like `&`, `|`, and `<<`. We shall see that, with these tools, we can implement a way of *buffering* up unusual-length-values into a 64-bit value, and then "peel off" 8-bit chunks from that buffer as needed. The complete implementation is in fig. 2.

Our actual "buffer", `bitwrite_buffer`, is simply a 64-bit integer. We'll see over the course of our implementation that this basically restricts our key-length to 32-bit integers.

Digression: Bounded-Size Buffers At the onset of my programming journey I would be very uncomfortable with this: having implicit resource limits felt very "vulnerable". A program hitting a resource limit (such as, in this case, secretly trying to encode too many bits at once in our buffer) can lead to hard-to-investigate bugs and a lot of pain.

```c
#ifndef IO_BUFFER_SIZE
#define IO_BUFFER_SIZE 4096
#endif
static uint8_t fwrite_buffer[IO_BUFFER_SIZE];
static int emit_buffer_next = 0;
void write_buffer_flush() {
  fwrite(fwrite_buffer, 1, emit_buffer_next, lzw_output_file);
  emit_buffer_next = 0;
}
void lzw_write_byte(uint8_t c) {
  if (emit_buffer_next == sizeof(fwrite_buffer)) {
    write_buffer_flush();
  }
  fwrite_buffer[emit_buffer_next++] = c;
}
```

Figure 3: Emitting Bytes

So I would have gone to great pains to make some kind of resizable bit-buffer that could handle any number of bits.

However, with experience I improved my intuition about which resource limits would actually be hit in development, and so I could feel more confident about "putting things off" for later. I also became more experienced with (or inured to?) investigating weird bugs and fixing them—still not something I sought out, but no longer a terror-in-the-dark. Lastly, you can see the most critical tool used here: `ASSERT` statements that will fail in the case the resource is exhausted. The flexibility to define "solving" the problem as "telling the developer about it, and failing" is a remarkably powerful one. Of course for commercial software it's unsatisfying, but for internal-facing projects like this it's great.

A maybe-unintuitive property is that we use the same emit bytebuffer in both the decode and encode cases. Maybe we can do something more optimally, then, in the decode/encode case, as we know that one direction of each isn't going to be variable-width.

Buffered IO This pattern of "empty the buffer when we can't fit the next thing" is pretty common. A common follow-on bug is forgetting to the flush the last thing.

Why use a `#define`? Partly for nostalgia-esque old-fashioned, I suppose. A lot of run-time options or flags or configuration can instead be compile-time. You can imagine, if we were allocating `fwrite_buffer` on the heap, we'd pass in the size as a runtime parameter. Here, however, we want to reserve the space for the buffer in the process-level memory, so we can pass the size as a *build* option: `gcc lzw.c -DIO_BUFFER_SIZE 10000`, for instance.

Why have a buffer at all? Semantically, we don't need the buffer at all: we can just call `fwrite` (or `fputc`) directly, one byte at a time. There is the general intuition that doing things in batches is “nicer” for computers and leads to better performance: indeed on my machine it lead to a considerable improvement.

It’s worth considering *why* (as in, what is the actual mechanism that lead to a speedup) in such a case. My initial hypothesis was that each `fputc` is a system call, and crossing that boundary is expensive. However, using `strace` (a utility that logs every system call) shows that the system itself *does* have a buffer (which is not too surprising) and so we actually have the same number of system calls.

This is a common challenge I’ve seen in performance work: people take the success of an optimization as proof that they’re hypothesis is write: for this situation I don’t know what actual mechanism changed to lead to a performance improvement, only that it’s considerable! This is a great homework problem, maybe. If I wanted to spend more time on this I’d next hypothesis something about dynamic linking or indirect calls, but again I don’t know.

Any gotchas? Yes, in particular we have to remember to flush this buffer, see what was do in `lzw_encode_end` in the listing at the end.

3.2. Interlude: Modules versus Objects

The state of our program is all globals. This is weird—why do we we do this? It avoid heap allocations, which are nice, at the cost that you can only do this one at a time. It’s not that bad—moreover, we can do more things compile-time.

3.3. State-Machine Based Dictionary

In the common internet example the dictionary is more-or-less literal. The TST object Sedgwick uses is some special-purpose tree that takes the head of the input remainder and will search it for as long as possible. To me this doesn’t feel natural (though maybe his approach is the better one, ultimately!) because there’s an aspect to how the `longestPrefixOf` method is what’s (behind the call-site) consuming the input.

(In particular, this doesn’t lend to a natural API extension that permits the user to specify “only encode the first N bytes”.)

Figure 4 shows how this code reads in input for encoding (compression), and in fig. 5 we see how taking the next bit (`c`) updates our `curr` node to either the next already-existing child in our tree, or allocate the new one.

So the dictionary is pretty important. Intuitively, that’s the thing we’re really generating, and using, a lot.

3.4. Data Structure Design

The structure is pretty exotic. Behold this diagram:

Short String Optimization We save a lot of memory with that crazy union thing. This is definitely a case where we have to decide on a heuristic: in industry figuring out

```c
size_t lzw_encode(size_t l) {
  for (;;) {
    int c = lzw_read_byte();
    if (c == EOF) {
      lzw_encode_end();
      break;
    }
    if (lzw_next_char(c) != NEXT_CHAR_CONTINUE) // begin processing...
```

Figure 4: The control-flow driving compression

```c
int lzw_next_char(uint8_t c) {
  lzw_node_p next = children_set_find(&curr->children, c);
  if (next) {
    curr = next;
    return NEXT_CHAR_CONTINUE;
  }
  if (lzw_max_key && lzw_next_key >= lzw_max_key) {
    DTRACE(DB_STATE, "APPEND(%#x)\tMAXKEY\n", c);
    return NEXT_CHAR_MAX;
  }
  // Else, allocate a new leaf in our tree.
  // Note curr is NOT updated: we still need to emit the "old" prefix in this case.
  const uint32_t k = lzw_next_key++;
  next = children_set_allocate(&curr->children, c, k);
  lzw_data[k].parent = next;
  return NEXT_CHAR_NEW;
}
```

Figure 5: Evolving our dictionary object

the right way to "define success" (including how to update that defition!) is the most important thing.

3.5. Feature: Clear Codes

This was a bear to get the coordination right: I hadn't anticipated the subtle issue about increasing the length as if it's a new key.

It's also weird that it counts as bytes we encode/decode.

3.6. Using our Features: EMA

Learned that trick from my SAT solver. It's actually really nice! From my motivating input it definitely worked. Dream: avoid having to emit clear codes at all. I was really close, but coordinating the implicit state was even worse. Maybe that's a homework problem?

3.7. Debugging Support

See the trace statements? Articulating the streams as emitting keys, and emitting logical bits, and then emitted physical bytes, was very useful for debugging. Supplementary tools: parse and compose log statements.

4. Testing: Fuzzing

Fuzzing was a lot of fun! It was more a quick way to determine that nothing regressed, but also found some awful edge-cases with clear-codes. Frustration: lack of tooling for futzing with command line parameters, basically.

5. Performance

I don't get hardware counters. There's encoding speed-and-memory-usage, decoding speed-and-memory-usage, and compression ratio. With some futzing I did "ok" compared to gzip on our motivating input. Still much slower, not sure why: presumably data structure is just bad, but would want hardware counters.

Attribution Problem in Perfomance I worked on a project where our software supported multiple workloads, call them A, B, C, etc. One workload, P, was both *extremely popular* and also *very resource intensive*. A single process could be working on multiple workloads at once. A common refrain I had to push back against was "oh, every single crash/resource-exhaustion is attributable to P". On the one hand that's a reasonable starting point, but tautologically *every* trace of a process is going to include a P: it was by far the most common one. I'm sure there's some cutesy statistical phrasing for this behavior, but this was a surprising example of heuristics biting people—often the issue was some other, less-exercised workload. (Nevermind the higher-level issues of load-balancing etc.)

5.1. Compression Ratio

This is trying to be the "textbook definition", so except up to EMA stuff I didn't expect much improvement. That EMA really helps is very cool.

5.2. Throughput

Need hardware counters! I can continue to guess what's wrong, but it's just not productive (or satisfying).

6. What's Next?

- Getting hardware counters to help inform performance work.
- It's not clear if, e.g., setting the max key to 1024 would basically induce an off-by-one in our max-length (we should only need 10 bits for each symbol, but we may accidentally get up to 11).
- I'd like to actually use this as a library for another project. That may mean making it more "ergonomic" (and better) to use the EMA stuff (as in, provide it for those users). It may also, sadly, mean that I should change this from a module to an object, so we actually have multiple such threads going on at once, haha.

7. Industrial Comparison

A homework assignment is to compare this to a real compression algorithm.

A. Core Library Implementation

```c
#include <assert.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "lzw.h"

typedef struct lzw_node_tag *lzw_node_p;

typedef struct lzw_children_set_tag {
```

```c
  bool use_array;
  struct {
    int index;
    struct {
      uint8_t key;
      lzw_node_p value;
    } immediate[4];
  } local;
  lzw_node_p *all;
} lzw_children_set_t;

// we want to build a mapping from keys to data-strings.
typedef struct lzw_node_tag {
  lzw_children_set_t children;
  uint32_t key;
} lzw_node_t, *lzw_node_p;

typedef struct {
  uint8_t *data;
  uint32_t len;
  lzw_node_p parent;
} lzw_data_t;

lzw_data_t *lzw_data = NULL;
lzw_node_p root = NULL;
lzw_node_p curr = NULL;

uint32_t lzw_length = 0;
uint32_t lzw_next_key = 0;
uint32_t lzw_max_key = 0;

FILE *lzw_input_file;
FILE *lzw_output_file;

uint64_t lzw_bytes_written = 0;
uint64_t lzw_bytes_read = 0;

// There's an implicit invariant that our lzw_length can never be more than 32,
// and this means our buffers always need at least 2 uses before overflowing.
// Some of our iteration depends on that (in particular write_key, which
// unconditionally enbuffers something before trying to drain it).
uint64_t bitread_buffer = 0;
uint32_t bitread_buffer_size = 0;
const uint32_t BITREAD_BUFFER_MAX_SIZE = sizeof(bitread_buffer) * 8;
uint64_t bitwrite_buffer = 0;
uint32_t bitwrite_buffer_size = 0;
const uint32_t BITWRITE_BUFFER_MAX_SIZE = sizeof(bitwrite_buffer) * 8;

// Before we get too much into executable code,
```

```c
// we want to express the different modes we can run in.
// I.e., debug levels
#ifdef NDEBUG
#define DTRACE(k, ...)
#define ASSERT(x)
#define DEBUG_STMT(x)
#else
enum {
  DB_STATE,
  DB_BYTE_STREAM,
  DB_KEY_STREAM,
  DB_DICTIONARY,
  DB_MAX,
};
int DB_KEYS_SET[DB_MAX] = {};
#define DTRACE(k, ...)                                              \
  if (DB_KEYS_SET[k]) {                                             \
    fprintf(stderr, __VA_ARGS__);                                   \
  }
#define ASSERT(x) assert(x)
#define DEBUG_STMT(x) x
#endif

void lzw_set_debug_string(const char *s) {
#ifdef NDEBUG
#else
  for (int i = 0; i < strlen(s); i++) {
    switch (s[i]) {
    case 's':
      DB_KEYS_SET[DB_STATE] = 1;
      break;
    case 'b':
      DB_KEYS_SET[DB_BYTE_STREAM] = 1;
      break;
    case 'k':
      DB_KEYS_SET[DB_KEY_STREAM] = 1;
      break;
    case 'd':
      DB_KEYS_SET[DB_DICTIONARY] = 1;
      break;
    }
  }
#endif
}

#ifndef NDEBUG
static const char *asbits(uint64_t v, uint8_t l) {
  static char format[(1 << 8 * sizeof(uint8_t)) + 1];
  // Emit bits backwards
```

```c
  for (int i = 0; i < l; i++) {
    format[l - (i + 1)] = v % 2 ? '1' : '0';
    v >>= 1;
  }
  format[l] = '\0';
  return format;
}
#endif

lzw_node_p children_set_find(lzw_children_set_t *s, uint8_t k) {
  if (!s->use_array) {
    int max_index = s->local.index;
    for (int i = 0; i < max_index; i++) {
      if (s->local.immediate[i].key == k) {
        return s->local.immediate[i].value;
      }
    }
    return NULL;
  }
  return s->all[k];
}

lzw_node_p children_set_allocate(lzw_children_set_t *s, uint8_t c, uint32_t k) {
  lzw_node_p r = calloc(1, sizeof(lzw_node_t));
  r->key = k;
  if (!s->use_array) {
    int n = s->local.index++;
    if (n < 4) {
      s->local.immediate[n].key = c;
      s->local.immediate[n].value = r;
      return r;
    }
    s->use_array = true;
    s->all = calloc(256, sizeof(lzw_node_p));
    for (int i = 0; i < n; i++) {
      s->all[s->local.immediate[i].key] = s->local.immediate[i].value;
    }
  }
  s->all[c] = r;
  return r;
}

enum { NEXT_CHAR_CONTINUE = 0, NEXT_CHAR_MAX = 1, NEXT_CHAR_NEW = 2 };
const uint32_t lzw_clear_code = 256;

// The primary action of this table is to ingest
// the next byte, and maintain the correct encoding
// information for the implicit string seen-so-far.
// That's captured in this function:
```

```c
int lzw_next_char(uint8_t c) {
  lzw_node_p next = children_set_find(&curr->children, c);
  if (next) {
    DTRACE(DB_STATE, "APPEND(%#x)\t\tSTATE\t%u\t%u\n", c, curr->key, next->key);
    curr = next;
    return NEXT_CHAR_CONTINUE;
  }
  // we have reached the end of the string.
  if (lzw_max_key && lzw_next_key >= lzw_max_key) {
    DTRACE(DB_STATE, "APPEND(%#x)\tMAXKEY\n", c);
    return NEXT_CHAR_MAX;
  }
  // Create the new fields for the new node
  const uint32_t k = lzw_next_key++;
  next = children_set_allocate(&curr->children, c, k);
  const uint32_t l = curr->key == -1 ? 0 : lzw_data[curr->key].len;
  uint8_t *data = calloc(l + 1, sizeof(uint8_t));
  if (l) {
    memcpy(data, lzw_data[curr->key].data, l * sizeof(uint8_t));
  }
  data[l] = c;
  lzw_data[k].data = data;
  lzw_data[k].len = l + 1;
  lzw_data[k].parent = next;
  DTRACE(DB_DICTIONARY, "DICT\tADD\t%u\t%u\t%u\t%#x\n", k, curr->key, l + 1, c);
  DTRACE(DB_STATE, "APPEND(%#x)\t\tNEWSTATE %u\n", c, k);
  return NEXT_CHAR_NEW;
}

// We also have book-keeping of when we have to
// update the length
static size_t lzw_data_size(void) {
  return (1 << lzw_length) * sizeof(lzw_data_t);
}

void lzw_len_update() {
  DTRACE(DB_STATE, "INCLENGTH %d->%d\n", lzw_length, lzw_length + 1)
  size_t old_length = lzw_data_size();
  lzw_length++;
  size_t new_length = lzw_data_size();
  ASSERT(old_length * 2 == new_length);
  lzw_data = realloc(lzw_data, new_length);
  memset((char *)lzw_data + old_length, 0, old_length);
}

void free_dictionary(lzw_node_p t) {
  if (!t)
    return;
  if (t->children.use_array) {
```

```
    for (uint16_t i = 0; i < 256; ++i) {
      free_dictionary(t->children.all[i]);
    }
    free(t->children.all);
  } else {
    int n = t->children.local.index;
    for (int i = 0; i < n; i++) {
      free_dictionary(t->children.local.immediate[i].value);
    }
  }
  free(t);
}

void lzw_destroy_state(void) {
  free_dictionary(root);
  curr = NULL;
  root = NULL;
  for (uint32_t i = 0; i < lzw_next_key; ++i) {
    if (lzw_data[i].data)
      free(lzw_data[i].data);
  }
  if (lzw_data) {
    free(lzw_data);
    lzw_data = NULL;
  }
  lzw_next_key = 0;
  ASSERT((bitread_buffer & ((1 << bitread_buffer_size) - 1)) == 0);
  ASSERT(bitwrite_buffer_size == 0);
}

void bitwrite_buffer_push_bits(uint32_t v, uint8_t l) {
  ASSERT(bitwrite_buffer_size + l < BITWRITE_BUFFER_MAX_SIZE);
  uint32_t mask = (1 << l) - 1;
  bitwrite_buffer = (bitwrite_buffer << l) | (v & mask);
  bitwrite_buffer_size += l;
}

uint8_t bitwrite_buffer_pop_byte(void) {
  ASSERT(bitwrite_buffer_size >= 8);
  bitwrite_buffer_size -= 8;
  uint8_t b = bitwrite_buffer >> bitwrite_buffer_size;
  return b;
}

#ifndef IO_BUFFER_SIZE
#define IO_BUFFER_SIZE 4096
#endif
static uint8_t fwrite_buffer[IO_BUFFER_SIZE];
static int emit_buffer_next = 0;
```

```
void write_buffer_flush() {
  fwrite(fwrite_buffer, 1, emit_buffer_next, lzw_output_file);
  emit_buffer_next = 0;
}
void lzw_write_byte(uint8_t c) {
  if (emit_buffer_next == sizeof(fwrite_buffer)) {
    write_buffer_flush();
  }
  fwrite_buffer[emit_buffer_next++] = c;
}

// Reading v from "left to right", we
// emit the l bits of v.
void write_key(uint32_t v, uint8_t l) {
  ASSERT((v & ((1 << l) - 1)) == v); // v doesn't have extra bits
  DTRACE(DB_KEY_STREAM, "EMITKEY(%d):\t\t%d\t%s\n", l, v, asbits(v, l));
  bitwrite_buffer_push_bits(v, l);
  while (bitwrite_buffer_size >= 8) {
    uint8_t c = bitwrite_buffer_pop_byte();
    DTRACE(DB_BYTE_STREAM, "EMITBYTE(encode):\t%#x\t%s\n", c, asbits(c, 8));
    lzw_write_byte(c);
    lzw_bytes_written++;
  }
}

bool key_requires_bigger_length(uint32_t k) { return k >= (1 << lzw_length); }
bool update_length() {
  if (key_requires_bigger_length(lzw_next_key)) {
    lzw_len_update();
    return true;
  }
  return false;
}

void lzw_init() {
  root = (lzw_node_p)calloc(1, sizeof(lzw_node_t));
  curr = root;
  root->key = -1;
  lzw_length = 1;
  lzw_next_key = 0;
  lzw_data = calloc(1 << lzw_length, sizeof(lzw_data_t));

#ifndef NDEBUG
  int old_state_key = DB_KEYS_SET[DB_STATE];
  DB_KEYS_SET[DB_STATE] = 0;
  int old_dict_key = DB_KEYS_SET[DB_DICTIONARY];
  DB_KEYS_SET[DB_DICTIONARY] = 0;
#endif
  for (uint16_t i = 0; i < 256; ++i) {
```

```c
    lzw_next_char(i);
    update_length();
  }
#ifndef NDEBUG
  DB_KEYS_SET[DB_STATE] = old_state_key;
  DB_KEYS_SET[DB_DICTIONARY] = old_dict_key;
#endif

  ASSERT(lzw_clear_code == lzw_next_key);
  lzw_next_key++; // reserve 256 for the clear-code.
  update_length();

  bitread_buffer = 0;
  bitread_buffer_size = 0;

  bitwrite_buffer = 0;
  bitwrite_buffer_size = 0;

  lzw_bytes_read = 0;
  lzw_bytes_written = 0;
}

static uint8_t fread_buffer[IO_BUFFER_SIZE];
static int read_buffer_next = 0;
static int read_buffer_max = 0;
uint32_t lzw_read_byte(void) {
  if (read_buffer_next == read_buffer_max) {
    read_buffer_max =
        fread(fread_buffer, 1, sizeof(fread_buffer), lzw_input_file);
    read_buffer_next = 0;
  }
  if (read_buffer_max == 0) {
    return EOF;
  }
  return fread_buffer[read_buffer_next++];
}

bool input_eof(void) { return read_buffer_max == 0 && feof(lzw_input_file); }

size_t lzw_encode(size_t l) {
  size_t i = 0;
  for (;;) {
    int c = lzw_read_byte();
    if (c == EOF) {
      DTRACE(DB_BYTE_STREAM, "READBYTE(encode):\t%d\n", c);
      DTRACE(DB_STATE, "lzw_encode:eof\n");
      lzw_encode_end();
      break;
    }
```

```c
      i++;
      DTRACE(DB_BYTE_STREAM, "READBYTE(encode):\t%#x\t%s\n", c, asbits(c, 8));
      if (lzw_next_char(c) != NEXT_CHAR_CONTINUE) {
        write_key(curr->key, lzw_length);
        curr = root;
        update_length();
        lzw_next_char(c);
        if (i > 1) {
          break;
        }
      }
    }
    lzw_bytes_read += i;
    return i;
  }

  void lzw_write_clear_code(void) {
    DTRACE(DB_STATE, "CLEAR_CODE\t%zu\t%d\n", lzw_bytes_written, input_eof());
    if (input_eof()) {
      return; // don't bother
    }
    if (curr != root) {
      write_key(curr->key, lzw_length);
      curr = root;
      // When we read in a code, we always assume that it's a new key
      // (unless if we're at the max). So our reader preemptively
      // updates the length at the key boundaries---we need to do that
      // here too, even though this key isn't new.
      if (key_requires_bigger_length(lzw_next_key + 1)) {
        lzw_len_update();
      }
    }
    write_key(lzw_clear_code, lzw_length);
    lzw_encode_end();
  }

  void lzw_encode_end(void) {
    // if we haven't done anything yet, make that more explicit
    DTRACE(DB_STATE, "ENCODE_END\t%u\t%zu\t%d\n", bitwrite_buffer_size,
           lzw_bytes_written, curr == root);
    if (bitwrite_buffer_size == 0 && lzw_bytes_written == 0 && curr == root) {
      return;
    }
    if (curr != root) {
      write_key(curr->key, lzw_length);
      curr = root;
    }
    if (bitwrite_buffer_size != 0) {
      // We want to finish emitting our last key.
```

```c
    // cap off our buffer: there are (say) 3 valid bits left,
    // we just need to pad it so we can emit those 3 bits as part
    // of a larger byte.
    uint8_t bits_to_add = 8 - (bitwrite_buffer_size % 8);
    write_key(0, bits_to_add);
    ASSERT(bitwrite_buffer_size == 0);
  }
  write_buffer_flush();
}

void bitread_buffer_push_byte(uint8_t c) {
  ASSERT(bitread_buffer_size + 8 < BITREAD_BUFFER_MAX_SIZE);
  bitread_buffer <<= 8;
  bitread_buffer |= c;
  bitread_buffer_size += 8;
}

uint32_t bitread_buffer_pop_bits(uint32_t bitcount) {
  ASSERT(bitread_buffer_size >= bitcount);
  uint64_t bitread_buffer_copy = bitread_buffer;
  // slide down the "oldest" bits
  bitread_buffer_copy >>= (bitread_buffer_size - bitcount);
  bitread_buffer_copy &= (1 << bitcount) - 1;
  bitread_buffer_size -= bitcount;
  return bitread_buffer_copy;
}

// This will read the next bits up to our buffer.
bool read_bits(uint32_t *v) {
  while (bitread_buffer_size < lzw_length) {
    uint32_t c = lzw_read_byte();
    if (c == EOF) {
      DTRACE(DB_BYTE_STREAM, "READBYTE(decode):\t%d\n", c);
      return false;
    }
    DTRACE(DB_BYTE_STREAM, "READBYTE(decode):\t%#x\t%s\n", c, asbits(c, 8));
    lzw_bytes_read++;
    bitread_buffer_push_byte(c);
  }
  *v = bitread_buffer_pop_bits(lzw_length);
  return true;
}

bool lzw_valid_key(uint32_t k) {
  ASSERT(k < (1 << (lzw_length)));
  return lzw_data[k].data != NULL;
}

size_t lzw_decode(size_t limit) {
```

```
    uint32_t curr_key;
    size_t read = 0;
    while (read < limit && read_bits(&curr_key)) {
      DTRACE(DB_KEY_STREAM, "READKEY(%d):\t\t%d\t%s\n", lzw_length, curr_key,
             asbits(curr_key, lzw_length));
      if (curr_key == lzw_clear_code) {
        DTRACE(DB_STATE, "DECODE\tCLEAR_CODE\n");
        lzw_destroy_state();
        // Curious thing: we can return a value greater than lzw_bytes_read,
        // as lzw_init() set that back to 0. We continue because we also
        // promise to always emit something when we're called.
        lzw_init();
        continue;
      }
      ASSERT(lzw_valid_key(curr_key));

      // emit that string:
      uint8_t *s = lzw_data[curr_key].data;
      uint32_t l = lzw_data[curr_key].len;
      ASSERT(l);
      for (uint32_t i = 0; i < l; ++i) {
        DTRACE(DB_BYTE_STREAM, "EMITBYTE(decode):\t%#x\n", s[i]);
        lzw_write_byte(s[i]);
      }
      lzw_bytes_written += l;
      read += l;
      curr = lzw_data[curr_key].parent;

      if (key_requires_bigger_length(lzw_next_key + 1)) {
        lzw_len_update();
      }

      // peek at the next string:
      DTRACE(DB_STATE, "DECODE(peek)\n");
      if (!read_bits(&curr_key)) {
        DTRACE(DB_STATE, "DECODE(break)\n");
        // We're at EOF, so just early-out
        break;
      }
      if (curr_key == lzw_clear_code) {
        // Skip our checks: we don't want to evolve our state.
        bitread_buffer_size += lzw_length;
        continue;
      }
      DTRACE(DB_STATE, "DECODE(continue)\n");
      bitread_buffer_size += lzw_length;

      // Find the next character.
      // If the next key is valid, that means
```

```c
      // the next key is already something we've seen,
      // otherwise it's going to be the key for the new
      // string. Either way, we take the next step
      // on that character, but then manually reset
      // our curr node back to the root in prep for
      // really reading the next string.
      if (lzw_valid_key(curr_key)) {
        s = lzw_data[curr_key].data;
      } else {
        ASSERT(curr_key - 1 == lzw_clear_code || lzw_valid_key(curr_key - 1));
      }
      DEBUG_STMT(int b =)
      lzw_next_char(s[0]);
      ASSERT(b != NEXT_CHAR_CONTINUE);
      curr = root;
    }
    write_buffer_flush();
    return read;
}
```

B. Command Line Implementation

```c
#include "lzw.h"
#include <assert.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int getopt(int, char *const[], const char *);
char *optarg;

bool do_decode = false;
bool do_encode = false;
bool do_ratio = false;
bool trace_ratio = false;
char *ratio_log_filename = NULL;

FILE *user_input;
FILE *user_output;

int verbosity = 0;

size_t page_size = 4096;

```

```
uint64_t total_stream_read = 0;
uint64_t total_stream_written = 0;

uint64_t prev_bytes_written = 0;
uint64_t prev_bytes_read = 0;
uint64_t page_bytes_written = 0;
uint64_t page_bytes_read = 0;
double next_ratio() {
  page_bytes_written = lzw_bytes_written - prev_bytes_written;
  page_bytes_read = lzw_bytes_read - prev_bytes_read;
  prev_bytes_written = lzw_bytes_written;
  prev_bytes_read = lzw_bytes_read;
  // fprintf(stderr, "page_bytes_written: %zu\n", page_bytes_written);
  // fprintf(stderr, "page_bytes_read:    %zu\n", page_bytes_read);
  assert(page_bytes_written);
  double compression_ratio =
      (double)page_bytes_written / (double)page_bytes_read;
  if (do_decode) {
    compression_ratio = (double)page_bytes_read / (double)page_bytes_written;
  }
  return compression_ratio;
}

void reset_written() {
  prev_bytes_written = 0;
  prev_bytes_read = 0;
}

void decode_stream() {
  total_stream_written = 0;
  lzw_init();
  // Decode is guaranteed to make progress (even in presence of clear-codes)
  for (;;) {
    size_t written = lzw_decode(page_size);
    if (!written) {
      break;
    }
    total_stream_written += written;
  }
  lzw_destroy_state();
}

void encode_stream() {
  total_stream_read = 0;
  total_stream_written = 0;
  FILE *ratio_log_file = stderr;
  if (ratio_log_filename) {
    ratio_log_file = fopen(ratio_log_filename, "w");
  }
```

```c

  const int ema_delay = 32;

  for (int block_count = 0;; block_count++) {
    reset_written();
    double ema_slow = 0.0;
    double ema_slow_alpha = 0.0001;
    double ema_fast = 0.0;
    double ema_fast_alpha = 0.01;

    lzw_init();

    for (int page_count = 0;; page_count++) {
      // fprintf(stderr, "processing page: %d\n", page_count);
      size_t bytes_processed = lzw_encode(page_size);
      // We've hit EOF.
      if (!bytes_processed) {
        total_stream_read += lzw_bytes_read;
        total_stream_written += lzw_bytes_written;
        lzw_destroy_state();
        return;
      }

      // We've processed a page's worth of data, now
      // evaluate our compression ratio and windows.
      double compression_ratio = next_ratio();
      if (page_count < ema_delay) {
        ema_slow = compression_ratio;
        ema_fast = compression_ratio;
      } else {
        ema_slow += ema_slow_alpha * (compression_ratio - ema_slow);
        ema_fast += ema_fast_alpha * (compression_ratio - ema_fast);
      }
      if (trace_ratio) {
        fprintf(ratio_log_file,
                "%s ratio: %f\tema_slow=%f\tema_fast=%f\tpage_bytes_read: "
                "%zu\tpage_bytes_written: %zu\n",
                do_encode ? "compression  " : "decompression",
                compression_ratio, ema_slow, ema_fast, page_bytes_read,
                page_bytes_written);
      }

      // Now consume our ratio information: should we start a new block?
      if (do_ratio && page_count >= ema_delay &&
          (ema_slow * 1.5 < ema_fast || ema_fast > 0.7)) {
        if (trace_ratio) {
          fprintf(ratio_log_file, "resetting %d\n", page_count);
        }
        lzw_write_clear_code();
```

```c
        break; // this will lead to the destory-state and init on the back-edge
      }
    }
    total_stream_read += lzw_bytes_read;
    total_stream_written += lzw_bytes_written;
    lzw_destroy_state();
  }
}

// process_stream consumes all the globally-set parameters
void process_stream() {
  if (do_decode) {
    decode_stream();
  } else {
    encode_stream();
  }
}

// This is a shared correctness routine/helper
void init_streams(char *inbuffer, size_t insize, char **outbuffer,
                  size_t *outsize) {
  FILE *out = open_memstream(outbuffer, outsize);
  FILE *in = fmemopen(inbuffer, insize, "r");
  lzw_input_file = in;
  lzw_output_file = out;
}
void close_streams(void) {
  fclose(lzw_input_file);
  fclose(lzw_output_file);
}

void round_trip() {
  assert(user_output == stdout);
  user_output = fopen("roundtrip_result.dat", "w");
  assert(user_output);
  do_encode = true;
  do_decode = false;
  FILE *intermediate = tmpfile();
  lzw_output_file = intermediate;
  fprintf(stderr, "Encoding stream\n");
  process_stream();
  fflush(intermediate);
  fseek(intermediate, 0, SEEK_SET);
  lzw_input_file = intermediate;
  lzw_output_file = user_output;

  do_encode = false;
  do_decode = true;
  fprintf(stderr, "Decoding stream\n");
```

```
  process_stream();

  fclose(intermediate);
}

void dumpbytes(const char *d, size_t c) {
  for (size_t i = 0; i < c; i++) {
    if (i > 0 && i % 20 == 0) {
      fprintf(stderr, "\n");
    }
    fprintf(stderr, "%02x ", (unsigned char)d[i]);
  }
  fprintf(stderr, "\n");
}

void round_trip_in_memory(const char *Data, size_t Size) {
  char *encodechunks = NULL;
  size_t encodechunks_size = 0;
  init_streams((char *)Data, Size, &encodechunks, &encodechunks_size);
  do_encode = true;
  do_decode = false;
  process_stream();
  close_streams();
  assert(total_stream_read == Size);
  assert(total_stream_written == encodechunks_size);

  char *decodechunks = NULL;
  size_t decodechunks_size;
  init_streams(encodechunks, encodechunks_size, &decodechunks,
               &decodechunks_size);
  do_encode = false;
  do_decode = true;
  process_stream();
  close_streams();
  // assert(total_stream_read == encodechunks_size); // because of clear-codes, this isn't quite
  assert(total_stream_written == decodechunks_size); // We can compute this by summing decode ret

  assert(Size == decodechunks_size);
  assert(!memcmp(decodechunks, Data, Size));

  free(encodechunks);
  free(decodechunks);
}

#ifdef FUZZ_MODE
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  if (Size < 12) { // just reserve enough bytes.
    return 0;
  }
```

```
  // Peel off of the first chunk of data to choose our settings
  lzw_max_key = *(uint32_t*)(&Data[0]);
  if (lzw_max_key < 257) {
    return 0;
  }
  page_size = *(uint32_t*)(&Data[4]);
  if (page_size == 0) {
    return 0;
  }
  do_ratio = (Data[9] % 2) == 0;
  //fprintf(stderr, "page_size=%zu\tlzw_max_key=%u\tdo_ratio=%d\n", page_size, lzw_max_key, do_ra
  round_trip_in_memory((const char *)Data+10, Size-10);
  return 0;
}
#else
int main(int argc, char *argv[]) {
  char c;
  bool correctness_roundtrip = false;
  bool correctness_roundtrip_memory = false;

  user_input = stdin;
  user_output = stdout;

  while ((c = getopt(argc, argv, "deg:m:p:r:q:l:v:xcCb:i:o:")) != -1) {
    switch (c) {
    case 'd':
      do_decode = true;
      break;
    case 'e':
      do_encode = true;
      break;
    case 'g':
      lzw_set_debug_string(optarg);
      break;
    case 'm':
      lzw_max_key = atoi(optarg);
      break;
    case 'p':
      page_size = atoi(optarg);
      break;
    case 'q':
      trace_ratio = true;
      // if optarg is anything but '-'
      if (strcmp(optarg, "-")) {
        ratio_log_filename = strdup(optarg);
      }
      break;
    case 'v':
      verbosity = atoi(optarg);
```

```c
        break;
      case 'x':
        do_ratio = true;
        break;
      case 'c': // for "correctness"
        correctness_roundtrip = true;
        break;
      case 'C': // for "correctness"
        correctness_roundtrip_memory = true;
        break;
      case 'i':
        user_input = fopen(optarg, "r");
        assert(user_input);
        break;
      case 'o':
        user_output = fopen(optarg, "wx");
        assert(user_output);
        break;
      default:
        break;
    }
  }

  if (correctness_roundtrip && correctness_roundtrip_memory) {
    printf("Error, can't do both in-memory and through-file roundtrip (cC)\n");
    return 2;
  }
  if (lzw_max_key && lzw_max_key < 256) {
    printf("Error, max key too small (need >= 256, got %u)\n", lzw_max_key);
    return 2;
  }

  if (trace_ratio && !do_ratio) {
    fprintf(stderr,
            "Warning, do_ratio=%s, trace_ratio=%s, unexpected behavior\n",
            do_ratio ? "true" : "false", trace_ratio ? "true" : "false");
  }

  if (verbosity) {
    fprintf(stderr, "lzw_max_key: %d\n", lzw_max_key);
    fprintf(stderr, "page_size  : %zu\n", page_size);
  }

  lzw_input_file = user_input;
  lzw_output_file = user_output;

  if (correctness_roundtrip) {
    round_trip();
  } else if (correctness_roundtrip_memory) {
```

```
    char *inputbuffer = NULL;
    size_t inputbuffer_size = 0;
    FILE *copy_file = open_memstream(&inputbuffer, &inputbuffer_size);
    char buffer[1028 * 1028];
    size_t n = 0;
    while ((n = fread(buffer, 1, sizeof(buffer), lzw_input_file)) > 0) {
      fwrite(buffer, 1, n, copy_file);
    }
    fclose(lzw_input_file);
    fclose(copy_file);
    round_trip_in_memory(inputbuffer, inputbuffer_size);
    free(inputbuffer);
  } else {
    if (do_encode == do_decode) {
      printf("Error, must uniquely choose encode or decode\n");
      return 1;
    }
    process_stream();
  }

  return 0;
}
#endif
```