Individual Coursework 3: A Bit-Level LZW Compressor and Decompressor

Due date: 1:05 PM, 21st February 2019

Value: 6% of marks for module

Introduction

Network link speeds increase quickly, as do magnetic disk and flash storage capacities, but so do the sizes of data sets we manipulate. No matter how fast the network link or how big the disk, there are workloads that exceed available capacity. One way to try to squeeze every last bit of capacity out of a link or a disk is to *compress* data—to take an input stream of data, encode it so as to be represented by fewer bits, and send or store the resulting smaller, compressed data stream. A complementary processing pass can then *decompress* the compressed data stream and recover the original data. In a *lossless* compression scheme, decompressing the compressed data recovers the exact same original bits that the compressor took as input. In a *lossy* compression scheme, the result of decompressing the compressed data may be close to the original data, but not identical; JPEG image compression, for example, is a lossy compression scheme.

In this coursework, you will build a complete implementation of the Lempel-Ziv-Welch (LZW) lossless data compression algorithm, including both a compressor (which takes an input file and produces a compressed version of it as output) and a decompressor (which reverses this process, and produces the original data from a compressed file). LZW is a widely used compression algorithm: PDF, TIFF, and GIF files all employ variants of LZW compression. Your LZW implementation will have a unique twist: unlike most others, yours will not operate on input data consisting of 8-bit bytes (as do most file compression utilities), but will rather be tailored for compression and decompression of DNA data. There are only four possible data values in DNA data, which correspond to the four bases that comprise DNA: guanine (G), adenine (A), cytosine (C), and thymine (T).

This coursework will cause you to become proficient in bit-level manipulation of data in C: both the input to your compressor (a stream of DNA bases encoded in a file) and its output, and by symmetry, the input to your decompressor and its output, will consist of data values that are bit strings that will often not begin or end at byte boundaries. And the compressed data you produce with LZW will include bit strings of varying lengths. UNIX's (Linux's) file input and output (I/O) operations, however, operate only at a granularity of whole bytes. A major part of your task in this coursework is thus to bridge between the bit-level granularity of data your compressor and decompressor must manipulate and the byte-level granularity of file I/O. You'll become familiar with basic file I/O operations in C as supported by the UNIX standard I/O (stdio) library along the way.

Tasks

- Implement an LZW compressor that takes a series of DNA bases stored in a packed binary format as input from a file and produces LZW-compressed output to a file.
- Implement an LZW decompressor that takes LZW-compressed input from a file and produces the original, decompressed DNA data encoded as a series of DNA bases in a packed binary format in a file.

Your LZW compressor and decompressor must comply with all of the specification we provide below. It is vital that you understand the specification before you begin coding: an implementation that deviates from the specification will fail the tests, and the tests alone determine your mark.[1] *Read this handout in its entirety carefully before you begin!*

As ever, it is important to get started early. While LZW is not all that complex an algorithm, and a complete solution can fit in on the order of only 300 lines of code, coming to grips with bit-level operations in C will likely take time. You will likely need the two weeks allotted to complete CW3.

Specification

You should implement the LZW algorithms for compression and decompression (one separate C function for each) as described in Welch's 1984 article (available on the UCL CS 0019 web site), with five modifications:

- Because your compressor operates over DNA bases, for which there are only four possible values, each input symbol is two bits long (rather than one byte long, as described in Welch's article). We describe below the exact format of an input file of DNA bases.
- When you initialize your compressor's table, you should populate its first four entries with the four possible input base values (0, 1, 2, 3) in increasing order (as Welch does in his version of the algorithm, but for all 256 possible 8-bit byte values).
- You must initialize your compressor and decompressor to begin with a code length of 3 bits (i.e., such that the first compressed code your compressor outputs and that your decompressor expects as input is 3 bits long).[2]
- If your compressor's or decompressor's table of codes fills, you must dynamically grow it as needed at runtime. (That is, do not implement "clearing" of the code table or a reserved code indicating that the decompressor should clear its table.) See "Tips" later in this handout for more information on how to correctly grow the table of codes.
- Your compressor and decompressor must increase their output or input code length (respectively) when necessary to accommodate growth in the number of entries in the table. See the further explanation in "Tips" later in this handout.

The format of files containing uncompressed DNA data is as follows:

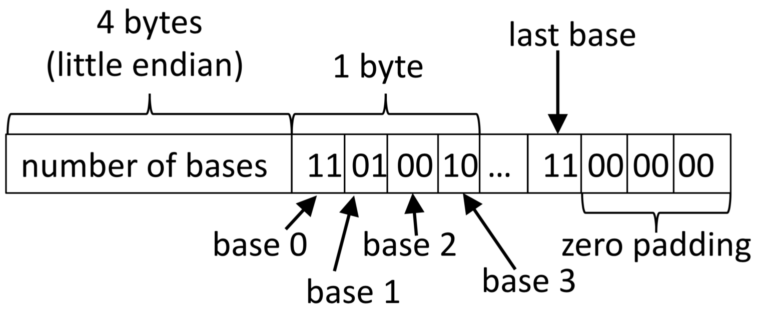
- the first four bytes of a file of DNA data contain an unsigned, little-endian integer that specifies the number of bases that follow in the remainder of the file;
- each base is encoded in two bits; each of the four possible two-bit values represents one of the four bases;

[1]There is one exception: if a submission clearly produces output using code that is not an LZW implementation, that submission will receive zero marks, regardless of test results.

[2]We make this requirement because if you begin running your compressor and decompressor with a table of codes that is already fully populated, there is a corner case that arises at the decompressor that Welch doesn't discuss (because he uses a 4K table that has only 256 entries at the start). It's readily solvable, but we didn't feel the need to include it in the coursework.

- within an eight-bit byte, the order of the bases stored within that byte is as follows: the first base is in the two most significant bits, the second base in the next two less significant bits, the third base in the next two less significant bits, and the fourth base in the two least significant bits;
- each successive byte contains bases that are ordered *after* the bases contained in the prior byte
- in the final byte of a DNA file, any trailing bit positions that do not contain base data *must* be zero-filled (i.e., must contain only zero bits)

In schematic form, the above format looks like this:



The format of files containing LZW-compressed DNA data is as follows:

- the first four bytes of a file of LZW-compressed DNA data contain an unsigned, little-endian integer that specifies the number of bases represented by the compressed stream in the remainder of the file;
- *unlike* in the figure above, which shows fixed-width, two-bit bases, LZW codes increase in width (in bits) over the course of a run of LZW, and a single code may span byte boundaries; the compressor and decompressor track how long in bits the codes they write and read (respectively) should be;
- a series of codes fills in the bits of an eight-bit byte from most significant to least significant; should a code span two bytes, it continues in the second byte's most significant bit;[3]
- in the final byte of an LZW-compressed DNA file, any trailing bit positions that do not contain codes *must* be zero-filled (i.e., must contain only zero bits)

Further Requirements

Implement your compressor in C source files named `comp0019.{c,h}` (pun intended), and your decompressor in C source files named `decomp0019.{c,h}`. If you wish, you may implement library routines that you use in both the compressor and decompressor in C source files named `comp0019lib.{c,h}`. We provide vestigial, nearly empty versions of these files for you in the code we hand out. *Don't delete any of these files from your repo, even if you don't wind up using them. The grading server requires that these files all exist, and its compilation of your code will fail, yielding a failure of all tests, if any of them is missing.*

Your compressor and decompressor must be invoked with the following function prototypes:

```
void Encode(FILE *in_file, FILE *out_file); // compressor
void Decode(FILE *in_file, FILE *out_file); // decompressor
```

[3]As it happens, this is how PDF and TIFF files pack bit strings shorter than one byte in length into bytes; GIF files do so in the opposite direction.

Our tests provide a `main()` that invokes your compressor and decompressor by the above API.

If your compressor or decompressor function in the above API encounters input that is invalid (i.e., that deviates from the specification for the input format for a file containing a sequence of bases or the input format for a file containing an LZW-compressed sequence of bases, respectively), it should print an error message `Invalid {encoder,decoder} input: aborting...` (depending on whether the compressor or decompressor is running) and return cleanly, without crashing. If you encounter this error case in your decompressor, the code should also write out any buffered valid decompressed bits to the output file and terminate the output file correctly, without writing any invalid decompressed bits to the output file.

Note that an LZW compressor or decompressor will yield very different results depending upon the width in bits with which it considers its input and renders its output, respectively: a byte-width-symbol LZW compressor or decompressor will yield different results than a two-bit-width-symbol LZW compressor or decompressor. Our tests (which determine your mark on this coursework) expect a compressor that treats its input as two-bit symbols and a decompressor that renders its output as two-bit symbols.

You should use the C stdio file I/O library routines for file I/O; these include `fopen()`, `fread()`, `fwrite()`, and `fclose()`, among others. Do not use the UNIX I/O system calls in this coursework (e.g., `open()`, `read()`, `write()`, `close()`, etc.). (Code that writes very small numbers of bytes at a time to a file, such as the code you are likely to write in this coursework, benefits in performance from the buffering that the stdio file I/O library routines provide.)

Tips

One detail of implementing LZW omitted from Welch's classic paper is what to do when the table of codes fills—i.e., when the compressor or decompressor generates a table index that falls outside the currently allocated number of entries in the table. For this coursework, you are required to *grow* your table (both in the compressor and decompressor) in this situation. Broadly speaking, you should do so by allocating more memory for further table entries (so that you can continue by filling in entries in the now-larger table). Your implementation should be capable of growing its table without bound (until the system reports that no more memory is available).

Another implementation detail omitted by Welch is how to dynamically increase the size of a code in bits (i.e., the number of bits in a single compressed output value; notionally a "table index" in Welch's description of the compressor). That is, while the specification in this coursework requires you to begin by emitting codes that are three bits long, as your table grows, you will find that you need to emit codes that cannot be represented in three bits, because the table index to emit is too large to fit in three bits. You will thus need to maintain a notion of the current code size in bits, and dynamically increase it when necessary as your table grows. Your compressor and decompressor must note when they need to emit or read (respectively) a code longer than the current code size, increment the current code size at that moment, and thereafter emit (read) only codes of that length in bits (until the next code length increase, if further ones are needed). N.B. that once you increase the code size, subsequent emissions/reads of *all* codes must use the new, increased code size. That is, even if some code had previously been emitted/read using a smaller code size, from the moment your implementation increases the code size onward, that code should be emitted/read using the increased code size.

For the compressor, you will need to implement a data structure that lets you search for the table entry that contains a string. Various data structures, including some variant of trie or a hash table, allow efficient lookups of this sort. We do not require you to implement any particularly efficient data structure for this part of the compressor. Even a linear search, while asymptotically inefficient, is allowed. You are of course welcome to implement a more efficient data structure if you so choose, but the marking scheme doesn't reward doing so (the glory of knowing you've implemented a clever, efficient data structure is its own reward). Strategically speaking, it's in your interest not to optimize too early: better to focus first on producing a simple implementation with an inefficient string search that passes all the tests and earns full marks, and then to circle back to implement a more ambitious data structure for this search if that's your bag, than to risk running out of time before the deadline because the implementation of the more ambitious data structure winds up taking more time than expected. (This really happens to people; don't let it happen to you!)

In his pseudocode in the assigned LZW reading, Welch (writing in 1984) is very concerned about memory efficiency. For the decompressor, consider a single code *c* read from the compressed input that corresponds to some string *B* of multiple bases. Being memory-efficient in the decoder means storing in code *c*'s table entry only the *last* base in *B* represented by code *c*, along with a "pointer" (in the guise of another code *p* that represents *all prior* bases in *B*) to code *p*'s table entry (and so on, until reaching a "terminal" code for the first base represented by *c*, whose table entry contains only that first base, and no pointer to a further code). This approach is asymptotically memory-efficient as the size of the table increases, as it stores a constant number of bits per table entry. We don't require your implementation to be this memory-efficient: in 2019, it's fine for you to keep the *entire* sequence of bases represented by a code in that code's table entry, which will make your implementation simpler (not least because the memory-efficient one above must also reverse the order of bases extracted from the table before outputting them—the job of the stack in the Welch article). Again, while the marking scheme doesn't reward it, if you're looking for more of a challenge *after* you've passed all the tests, you're welcome to circle back and for your own edification attempt the memory-scrimping design in Welch's pseudocode.

You will want to familiarize yourself with the C operators for manipulating individual bits, including bitwise and (`&`), bitwise or (|), shift left and right (<<, >>), and complement (˜). These operators allow you to extract bit fields from within a byte and set bit fields within a byte. You can find helpful examples of the use of these operators in CS:APP/3e §2.1.7 and 2.1.9. The discussion of *masking* in 2.1.7 is particularly relevant.

Introductory programming classes often suggest a "top-down" approach to programming: implementing code for the high-level sequence of operations a program should take first, leaving the details of lower-level functionality initially unspecified, and then iteratively writing routines that fill in progressively lower levels of detail. Experienced programmers who study the definition of the problem they are solving often proceed in the exact opposite fashion: by first building a library of low-level routines that parse input and place it into a representation where the program can operate on it, and that convert this program-internal representation into the needed output format. We suggest you adopt this "bottom-up" approach as you work on this coursework: first implement routines that can do the bit-twiddling necessary to read and write two-bit symbols in the file format specified in this handout, and to handle the generation and reading of variable-length bit strings, as needed by a bit-granularity LZW compressor and decompressor. Once these routines are in place, you can then tackle implementing the LZW compressor and decompressor atop them.

It is likely that you may find implementing the bit-twiddling routines more time consuming than implementing the LZW compressor and decompressor—possibly significantly moreso, depending on how much you've worked with bit fields in C or other languages before. This is intentional: one of our chief goals for this coursework is to give you experience grappling with bit fields in C.

You may find the shell command `xxd -b` useful if you'd like to examine the bit-level content of files while debugging your compressor and decompressor.

Logistics

We will use GitHub for CW3 in much the same manner as for CW2. To obtain a copy of the initial code for CW3, please visit the following URL:

```
https://classroom.github.com/a/6Zq7pVXd
```

If you'd like a refresher on using git with your own local repository and syncing it to GitHub, please refer to the CW2 handout.

Each time you push updated code to your GitHub repository for CW3, our automatic grading server will pull a copy of your code, run our automated tests on your code, and place a grade report in a file `report.md` in your GitHub repository. Your mark on CW3 will be that produced by the automated tests run by our automatic grading server on the latest commit (update) you make to your GitHub repository before the CW3 deadline. More on these tests below.

You must develop your code for CW3 in the 0019 Linux environment. As for CW2, the UCL CS Linux lab machines (remotely accessible via `ssh`) the CS Remote Worker Linux desktop (remotely accessible via the ThinLinc client), and our 0019 Linux VM are all available to you for CW3.

> Please note that your code's behavior on the automated tests when run in the 0019 Linux VM on the grading server will determine your mark on CW3, without exception. The CS 0019 staff cannot "support" development environments other than the CS Linux lab machines, CSRW Linux desktop, and 0019 Linux VM; we cannot diagnose problems you encounter should your code pass the tests in some other environment, but fail them in the 0019 Linux VM.

Building Your Code and Running the Tests

Once you've cloned your repository from GitHub to your working environment, you will find the C source files named above in the `src/` subdirectory of your repository. You should add your implementation of LZW to these files in this directory.

To prepare your working copy for running the tests, you must first issue the following shell commands in the top-level directory of your working copy:

```
mkdir build
cd build
cmake ../
```

(You only need do the above once, when you first clone the repository.)

To build your code and run the tests, go into the `build` directory that you created above, and issue the shell command

```
make test-all
```

Further information on how to interpret the output of the tests and debug follows below.

Submitting via GitHub

We will deem the timestamp of your CW3 submission to be the timestamp on GitHub's server of the last commit you make before the submission deadline of 1:05 PM GMT on 21st February 2019. Your mark will be the mark you receive on the automated tests for that version of your code. Coursework lateness penalties are described in detail on the 0019 class web site.

If you wish to submit after the deadline, *you must take the following steps for your coursework to be marked:*

1. When you wish to receive a mark for a version of your code that you push to GitHub after the submission deadline, you must begin your commit log message for that commit with

the exact string `LATESUBMIT`. *Our grading system will not record a mark for your late submission unless you comply with this requirement.* We follow this policy so that if a student accidentally pushes a further commit after the deadline, they aren't penalized for a late submission.

2. If you wish to claim late days, per the 0019 late days policy (details on the 0019 class web site), you must post a private message on Piazza to the Instructors *within one hour after pushing the version of your code you wish to submit late to GitHub* with a subject line `CW3 LATE DAYS` and a body stating the number of late days you would like to claim. *Please follow these instructions exactly: note the all capitals, and please use this exact subject line string.*
3. You may make only one late submission (i.e., one GitHub commit with the initial string `LATESUBMIT` and one Piazza posting with the subject line `CW3 LATE DAYS`). If you make more than one late submission, we will only mark the first one.

Tests and Marking Scheme

There are three groups of tests, as follows:

1. Group 1: Exhaustive tests of compression and decompression of all possible base sequences up to and including length 6.
2. Group 2: Tests of compression and decompression of fixed, randomly pre-generated base sequences up to a length of 1 million bases.
3. Group 3: Tests of corner-case inputs to your compressor and decompressor, and whether your implementations handle these cases correctly.

Within each group of tests, there are multiple individual tests. In groups 1 and 2, each individual test consists of an input file of bases and the corresponding correct compressed output file from a run of a correct, spec-compliant compressor. Each of the tests in groups 1 and 2 runs your compressor on the input file of bases and compares the resulting output with that of the correct compressed output; if they don't match (even in a single bit), you will receive a report about the failure. The test will then progress to run your decompressor on the correct compressed data, and compare your decompressor's output with the original input file of bases; if they don't match (again, even in a single bit), you will receive a report about the failure.

In group 3, each individual test consists of an invalid input file, either for your compressor or decompressor, runs the compressor or decompressor on that file, and validates that the invoked code handles the invalid input in a way that complies with the specification in this handout.

In some cases, a group of tests includes as many as thousands of individual tests. Do not fear the sheer total number of tests; they are intended to be exhaustive and are highly redundant. By redundant, we mean that code that passes one test will almost certainly pass a great many of them (and similarly, fixing one bug that had caused many tests to fail will almost certainly cause many of those tests to pass thereafter).

Because there are so many individual tests, the test code we provide runs them in a *test harness* that hides much of the detail of the individual tests. The test harness outputs results for the tests within a group in "subgroups" of closely related individual tests (which again may be numerous), and summarizes the results for each such subgroup.

Again, because there are so many individual tests, we don't provide you the thousands of input files and output files for all of them, as that would be unwieldy. Rather, the test harness dynamically synthesizes the test input and output files as it runs the tests. When you fail an individual test, we of course realize that you may wish to learn more detail about the individual test that failed, and further to run and debug your code on that test's input, and/or study the content of the input or the correct output. To let you do so, when your code fails a test, the test harness will automatically tell you two shell commands to explore the failed test further: one command for running just that failed test and seeing its detailed output, and a second for generating the input and output files for the test you failed, and reporting their names. You can then run your code on these files.

For example, here is the start of the output from the tests when the implementation of the compressor and decompressor is empty, which is the state of affairs in the code you receive when you first create your GitHub repository:

```
Test group one (all possible sequences with N elements):
testing encode for 4 sequences of size 1: FAILED
        to repeat only this test run:
        ./lzw_test --file ../test_data/all_bases_1 --test_index 0
  --type encode
        to extract input for test run:
        ./lzw_test --file ../test_data/all_bases_1 --test_index 0
  --type encode --dump
```

To see more detail on the individual test you failed in this subgroup of tests, you can issue the first command above, which will produce:

```
[student@host] ./lzw_test --file ../test_data/all_bases_1 --test_index 0
  --type encode
Will run test at index 0
Encoder output does not match model output
```

The above output tells you the stage of this test failed: your encoder's output doesn't match that produced by a spec-compliant one (as it shouldn't; we're running a null implementation here).

To obtain the input and output files for the test you failed, so that you can examine them and run your code on them directly, you can issue the second command above, which will produce:

```
./lzw_test --file ../test_data/all_bases_1 --test_index 0 --type encode
  --dump
Encoder input saved to 'encoder_input'
Model encoder output saved to 'model_encoded_stream'
```

You can then run your compressor on this test's base sequence input with:

```
./encode encoder_input outfile
```

where your compressor's output will be written to `outfile`, and your decompressor on a spec-compliant compressed version of the base sequence with

```
./decode model_encoded_stream
```

(You can of course run gdb on these two programs with these inputs, as well.)

The tests in Groups 1, 2, and 3 are worth 35, 35, and 30 marks respectively. Within a group, you earn marks linearly in the number of lines (subgroups) whose tests you fully pass. (For example, if there are 5 separate lines of tests in a group and you fully pass the tests in two lines only, that will be 40% of the marks within that group.)

Once again, we urge you to get started early.

Academic Honesty

This coursework is an *individual coursework*. Every line of code you submit must have been written by you alone, and must not be a reproduction of the work of others—whether from the work of students in this (or any other) class from this year or prior years, from the Internet, or elsewhere (where "elsewhere" includes code written by anyone anywhere).

Students are permitted to discuss with one another the definition of a problem posed in the coursework and the general outline of an approach to a solution, but not the details of or code for a solution. Students are strictly prohibited from showing their solutions to any problem (in code or prose) to a student from this year or in future years. In accordance with academic practice, students must cite all sources used; thus, if you discuss a problem with another student, you must state in your solution that you did so, and what the discussion entailed.

Any use of *any* online question-and-answer forum (other than the CS 0019 Piazza web site) to obtain assistance on this coursework is strictly prohibited, constitutes academic dishonesty, and will be dealt with in the same way as copying of code. The same goes for use of any online material specifically directed toward solving this coursework.

You are free to read other reference materials found on the Internet (and any other reference materials), ***apart from** source code in any language that implements LZW compression or implements storage of variable-length bit strings in files.* You may of course use the code we have given you. *Again, all other code you submit must be written by you alone.*

Copying of code from student to student or from the Internet is a serious infraction; it typically results in awarding of zero marks to all students involved, and is viewed by the UCL administration as cheating under the regulations concerning Plagiarism, Collusion, and/or Falsification. Penalties imposed can include exclusion from all further examinations at UCL. The course staff use extremely accurate plagiarism detection software to compare code submitted by all students (as well as code found on the Internet) and identify instances of copying of code; this software sees through attempted obfuscations such as renaming of variables and reformatting, and compares the actual parse trees of the code. Rest assured that it is far more work to modify someone else's code to evade the plagiarism detector than to write code for the assignment yourself!

Read the Piazza Web Site

You will find it useful to monitor the 0019 Piazza web site during the period between now and the due date for the coursework. Any announcements (*e.g.*, helpful tips on how to work around unexpected problems encountered by others) will be posted there. And you may ask questions there. *Please remember that if you wish to ask a question that reveals the design of your solution, you must mark your post on Piazza as private, so that only the instructors may see it.* Questions about the interpretation of the coursework text, or general questions about C that do not relate to your solution, however, may be asked publicly—and we encourage you to do so, so that the whole class benefits from the discussion.

References

Welch, Terry, A Technique for High-Performance Data Compression, in *IEEE Computer*, 17(6), June 1984.

CS:APP/3e §2.1.7 and 2.1.9

K&R §2.9

Linux/UNIX man pages for `fread(), fwrite(), fputc(), xxd`