CS-3411 Program III : Compressing Files

Fall 2023

In this project you will learn more about bit-level operations and layered abstractions to make programming easier. For this purpose, we will develop a program called *czy* which compresses the data stream directed at its standard input and writes the compressed stream to its standard output and another program called *dzy* which decompresses a compressed stream directed at its standard input and writes decompressed data to its standard output.

The compression algorithm processes the input one symbol (i.e., byte) at a time and compares it with each of the 8 bytes previously seen. If the byte has been previously seen, it outputs the position of the previous byte relative to the current position using the format described below. Otherwise, the symbol is output as is by prefixing it with a binary one.

There are two data formats used for encoding/decoding:

1. Infrequent : The current symbol is different than any of the previous 8. This is encoded as `<1> <symbol>`. This results in a 9 bit encoding.
2. Recent : The current symbol matches to one of previous 8 symbols. This will be encoded as `<0> <psn>`, where the `psn` is the position of the matching symbol relative to the current symbol. Specifically, if the symbol matches the immediately preceeding symbol then the psn will be zero, if it matches the symbol before the previous one, it will be one, and so on. `psn` is therefore encoded using 3 bits. The complete encoding is 4 bits.

The program should read from its standard input and write to its standard output by using kernel calls read and write using a block size of 1024. No standard i/o calls are permitted. Since we are reading our standard input which may point to anything, including /dev/tty, we cannot assume our input is seekable. In other words, once a character is read, we cannot go back and reposition the file to read it again. The algorithm is summarized below:

1. Read the next byte and compare it with previous 8 bytes in the input stream. Make sure that you properly deal with the initial boundary conditions. When you read the very first character, there are no previous characters to compare against. When you read the next character, you can only check for a match with the previous one, and so on. If there is no match, output a bit that is one. Otherwise, output a zero bit.
2. If the current character is a frequent character (i.e., it matches one of the previously seen), emit a zero bit and then the position of the previously output character that matches the current one. For example, if we are processing the 3rd character and the input was axab, the output will be 0 001, which means the current character was seen at position -1 (position zero is the previous character). We have successfully encoded eight bits using only four.
3. If the current character is not a frequent character, form an infrequent character encoding by prefixing the current character with binary one and output 9 bits of encoded data.
4. repeat the above steps until the end of file.
5. At the end of file, if the total number of bits written to the output is not divisible by eight, output as many one bits as necessary to pad it to a byte boundary. This will cause the decoder to start decoding an infrequent character and since it will encounter an end-of-file while attempting to read the remaining bits, it will realize that this is padding.

The decoding algorithm is outlined below:

1. Read one bit from the input.
2. If the bit is one, output the next 8 bits from the input as the decoded symbol.
3. If the bit is a zero, read the next 3-bits and output the corresponding previously output character again.

In the following example, the diff should not report any differences:

```
czy < input-file  | dzy > input-file.new
diff input-file input-file.new
```

Once you make sure your program is working correctly, use your program to compress its own source and its own object, i.e., czy.c and czy. Compress the same programs with gzip as well. Write a README file and indicate the compression ratio's you obtained in this README file.

Pragmatics

Source Template

You are provided a source template that includes:

- Makefile with the following commands:
  - all - compile project into *czy, dzy.*
  - czy - compile czy only
  - dzy - compile dzy only
  - clean - removes executable and object files.
  - submission - generates prog3.zip with all source files needed for submission. Upload this file to Canvas.
- czy.c - C source for implementation of your encoder
- dzy.c - C source for implementation of your decoder
- bitsy.h - C header with prototypes of proposed bit abstractions
- bitsy.c - C source where you will implement bit abstractions
- README - text file that should contain your comression results and commentary

You are not required to use the abstractions presented in bitsy.c. These are there for your convenience. So long as you implement in czy.c and dzy.c, you are free to implement the algorithm how you choose. You may also add additional source files so long as you update the makefile accordingly. When grading, the program will be compiled with the makefile that you submit on Colossus.

Submission Requirements

Your submission must be written in C.

Use Canvas to submit a zip file named `prog3.zip` generated by running make submission.