CompSci 165 Project #3
FILE COMPRESSION

due June 6 (W week 10)

Requirements

Write a program in C/C++ that implements a parameterized variation of the Lempel-Ziv sliding window file compression algorithm. By experimenting with your program, determine parameter values that optimize the compression effectiveness of the algorithm.

Create two executable programs, `LZ` and `EXPAND`. `LZ` outputs a compressed version of the input content; `EXPAND` recovers the original content from the compressed version.

The `LZ` program has one required command-line argument, a filename, and has several parameters. Each parameter has a predefined range and default value. A parameter `P` can be specified as having value `V` by including on the command line the string: `-P=V`

For example, to specify parameter `N` as having value "12", parameter `S` as having value "2", and input from file "grade.xls",
one would invoke the command: `LZ -N=12 -S=2 grade.xls`

Specifying a parameter value outside of its allowable range should result in an error message.

The set of parameters for `LZ` is given in the following table.

parameter	meaning	minimum	maximum	default value
N	number of bits to encode window offset	9	14	11
L	number of bits to encode match length	3	4	4
S	number of bits to encode length of literal string	1	5	3

The `EXPAND` program has one optional command-line argument, an input filename. If no filename is specified then the standard input stream is used.
For example, to specify input filename `book.out`,
one would invoke the command: `EXPAND book.out`

For both programs, the normal output is always directed to the standard output stream. and informative message output is always directed to the stderr output stream.

Your programs should output to stderr the parameter values and the I/O lengths. Your programs should calculate the compression savings (as a percentage of the original file size) and the processing time for encoding/decoding, and also output that information to the stderr output stream.

You will use a "standard suite" of files for benchmark purposes.

The first three bytes of the output created by your `LZ` compression program will contain the parameter values of *N, L,* and *S* that were used in the encoding process so that your `EXPAND` program can properly decompress the compressed file.

The specifications must be strictly adhered to, so the compression effectiveness is completely determined by the algorithm and the parameter values (and not by implementation variations), but there are opportunities for clever implementations that can result in significant improvements in compression/decompression speed.

Parameters for variations of Lempel-Ziv sliding window

N specifies the number of bits used to encode an offset into the window. *N* also determines the window size, *W*, which is 2^N. The default value of *N* is 11.
L specifies the number of bits used to encode the maximum match length. *L* also determines the lookahead buffer size, *F*, which is 2^L. The default value of *L* is 4.
S specifies the number of bits used to encode the maximum length of literal strings. *S* also determines the length of the longest encoded literal string, which is 2^S-1. The default value of *S* is 3.

The window consists of the *W-F* most recent input characters that have been processed and encoded plus the lookahead buffer, that contains the next *F* input characters that are yet to be processed.

You will experiment to determine parameter values that optimize the compression effectiveness of the algorithm for the benchmark files.

To encode

Initialize the window to consist of *W-F* "does-not-exist" characters and the first *F* characters of the input. Output three bytes that specify the parameter values of *N, L, S*. Iterate doing the following steps. Search the window to find the longest match with a prefix of the lookahead buffer. (Part, but not all, of this match may overlap the lookahead buffer.)

After finding a longest match of length 2 or more, output an encoding of the **token double** <*len,offset*> to indicate that the match has length *len* and starts *offset* characters before the beginning of the lookahead buffer. Shift the window forward *len* characters.

len will be represented in *L* bits, where the encoding for value *j* will be the *L*-bit binary string for *j*-1. For example, if *L*=4, then "2" is represented by `0001`, "3" by `0010` ... "16" by `1111`.

Since the offset is less than *W*, it can be represented in $N=\log_2 W$ bits.

However, when there is no match of the first character in the lookahead buffer, or when there is a match of the first character in the lookahead buffer but not the first two characters, provide the value of the next character instead of an offset to its location. In this case, output an encoding of the resulting **token triple** <*code,strlen,chars*>, where *code* is an *L*-bit binary number with value 0, indicating that there is no reference to the window, *strlen* is an *S*-bit binary representation of the number of characters being represented literally (often, but not always, having value 1), and *chars* is a sequence of *strlen* characters represented literally. Then shift the window forward *strlen* characters.

If two literals are presented contiguously, output one token triple <*0,2,char*$_1$ *char*$_2$> instead of two token triples <*0,1,char*$_1$> <*0,1,char*$_2$>. Similarly, up to 2^S-1 literals in a row can be represented with one token triple.

The special case of the **token double** <*code,strlen*>, where *code* has value 0 and *strlen* has value 0, indicates that no character is being represented literally. This special code, using a total of *L*+*S* bits, represents EOF (end-of-file).

To decode

Read three bytes that specify the paremeter values of *N, L,* and *S*. Initialize the window to consist of all blanks. Iterate doing the following steps until the special code for EOF is encountered.

1. Read *L* bits to get the representation of *len* (or *code*= all zeroes)
2. *len* has value one more than the binary value of those *L* bits
3. If *len* has value 2 or more then
    - read *N* bits to obtain value of *offset*
    - output the *len* characters, one at a time, allowing for overlap with the buffer
    - shift the window forward *len* characters
4. Otherwise the *L* bits were all 0's, indicating a literal string
    - read *S* bits to get the value of *strlen*
    - if *strlen* is zero then exit
    - read and output *strlen* 8-bit characters
    - shift the window forward *strlen* characters

The definition of Lempel-Ziv includes the fact that it makes use of references to previously encountered strings within a window. The set of possible such string references constitutes a dictionary. This set is changing (since the window is changing), and so the dictionary is not static in nature, it is dynamic. Hence Lempel-Ziv makes use of a dynamic dictionary. The contents of the dictionary at any point in time is precisely defined. However, the implementation details, how the set is maintained and searched, are up to you. Different correct implementations should yield exactly the same results, but they will differ in speed.

A straightforward implementation has very rapid dictionary insertion and deletion, but extremely slow search. An efficient implementation uses clever data structures that enable all of insertion, deletion, and search to be executed moderately quickly. For full credit, your implementation must be time-efficient.

Deliverables

- Source code
- Documentation explaining how to compile and run your program system under Linux on the ICS openlab machine
- Documentation explaining major data structures used in your program system
    - description of data structures and why they are useful for your system
    - worst-case and average-case analysis of performing certain basic operations needed by your programs as functions of *N,L,S*
- A table showing, for each file in the "standard suite",
    - the best compression savings obtained by `LZ`
    - which combination of values for *N, L, S* produces that compression savings
    - encoding/decoding processing times using `LZ` and `EXPAND`

Evaluation Process

The grader will perform the following tasks

- Place your source files in his directory that contains the class-supplied files: `kennedy.xls` and `book1`
- View provided documentation
- Compile the programs on Linux following directions given in documentation provided by you
  - This should produce executables `LZ` and `EXPAND`
- View the source code
- Run the following commands (and several others not listed here)
  - `LZ -N=12 -L=3 -S=4 book1 > book1.compressed`
  - `EXPAND book1.compressed > book1.recover`
  - `diff book1 book1.recover`
  - `LZ kennedy.xls | EXPAND > kenn.recover`
  - `diff kennedy.xls kenn.recover`
- If the `LZ` program does not finish execution within three minutes then the grader will abort the execution

Evaluation Rubrics

- 5 – instructions on how to compile on the ICS Linux box
- 5 – compiles on the ICS Linux box with no errors
- 5 – good program structure and internal comments
- 5 – document data structures
- 5 – worst-case and average-case analyses of basic operations
- 50 – correct implementation of `LZ` and original file recovered by `EXPAND`
- 20 – time-efficient execution of `LZ`
  - 1 sec (20), 5 sec (16), 10 sec (12), 30 sec (8), 60 sec (4)
- 5 – determine best `LZ` parameter values for each file in the standard suite