CSCI 2122 Assignment 2

Due date: 11:59pm, Friday, February 16, 2024, submitted via git

Objectives

The purpose of this assignment is to practice your coding in C, focusing on low-level bit manipulation with continued use basic constructs such as functions, loops, arrays, and functions from the C standard library.

This assignment is divided into two problems, where the second problem builds on the first. As the second problem subsumes the first. If you complete Problem 2, you have also completed Problem 1.

Preparation:

1. Complete Assignment 0 or ensure that the tools you would need to complete it are installed.
2. Clone your assignment repository:
   `https://git.cs.dal.ca/courses/2024-winter/csci-2122/assignment-2/????.git`
   where ???? is your CSID. Please see instructions in Assignment 0 and the tutorials on Brightspace if you are not sure how.

Inside the repository there is an **`unzipper`** directory, in which the code is to be written. Inside this directory there is a **`tests`** directory that contains tests that will be executed each time you submit your code. Please do not modify the **`tests`** directory or the **`.gitlab-ci.yml`** file that is found in the root directory. Modifying these files may break the tests. These files will be replaced with originals when the assignments are graded. You are provided with a sample **`Makefile`** that can be used to build your program. If you are using CLion, a **`Makefile`** will be generated from the **`CMakeLists.txt`** file generated by CLion.

Background

As we have discussed in class, a piece of text is encoded in a sequence of bytes where each byte represents a character in the text. The number of bits used to encode each character is the same, regardless of how frequently the character is used. I.e., both the letter 'e' and 'x' take 8 bits to encode, even though 'e' occurs much more frequently than 'x'. Compression algorithms work by encoding more frequently used characters with fewer bits while using more bits to encode less frequently used characters. This results in a fewer number of bits being used to encode a piece of text. There is another savings because unused characters do not need to be encoded. For example, if a text consisted of just Xs and Ys, you could use 1 bit per character to encode the entire text: A 0-bit to encode each X and a 1-bit to encode each Y. Your company has developed a new secret compression algorithm, and your boss has asked you to implement the decompression program.

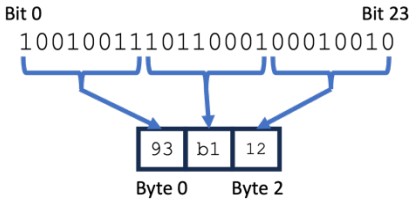
You will develop the program in two parts: In Problem 1, your program will read in (and output) the decoding table, containing the bit sequences used to represent each of the characters in the compressed file. In Problem 2, your program will use the table to decompress a compressed piece of text and output the result.

Problem 1: The Decoding Table

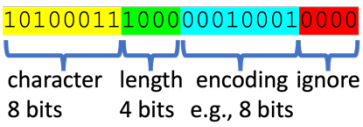
Write a C program in **`main.c`** in directory **`unzipper`** that will be compiled into an executable **`unzipper`**. Your program will read in a series of bytes in hexadecimal notation, representing the decoding table needed to decompress a piece of compressed text.

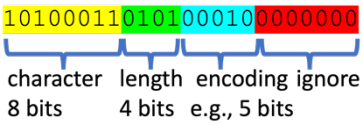
Input

Your program will read its input from **stdin**. The first line contains a single byte denoting the number of entries ***T*** in the decoding table. This byte is to be interpreted as an unsigned integer. If ***T*** is 0, then it is to be interpreted as 256. This is followed by ***T*** lines, each encoding a table entry and consisting of three items: the character, ***C***, that is encoded; the length, ***L***, (in bits) of the encoding; and the encoding, ***E***, which is bit sequence of ***L*** bits. After the table entries there are four bytes encoding the length of the compressed bitstream to be decoded. For Problem 1, these four bytes are *0*, indicating there is no compressed data to follow. In Problem 2, these four bytes will not be all 0s.



Each entry is encoded as a three byte (24-bit) bitstream, where the bits in each byte are ordered most-significant (first bit) to least significant (last bit). The first 8 bits encode the character (***C***). The next 4 bits encode length, ***L*** (an unsigned integer), and the first ***L*** bits of the remaining 12 bits are the compressed encoding ***E***, of character ***C***. The three bytes are encoded in the input in hexadecimal. (**Hint**: Use `scanf` to read the bytes). For the hexadecimal, use the format specifier with `"%x"`. On the right is an example of a table for encoding the string "`Hello World!\n`", Observe that in this case the encodings for the characters are between 2 and 4 bits long instead of the full 8 bits per byte.






```
0x0a
0x21 0x30 0x00
0x57 0x42 0x00
0x48 0x43 0x00
0x6c 0x24 0x00
0x6f 0x38 0x00
0x0a 0x3a 0x00
0x65 0x4c 0x00
0x64 0x4d 0x00
0x20 0x4e 0x00
0x72 0x4f 0x00
0x00 0x00 0x00 0x00
```

Processing

Your program will need to read in the table and output it in a human-readable form.

- The decoding table will have at most 256 entries. (Use an array of structs to store the table, as it will be needed for Problem 2.)
- For each entry, read in the 3 bytes and interpret the 24-bit bitstream as described above.
- The encoding of each character will be at most 12 bits. Any extra bits can be ignored.

Output

All output should be performed to **stdout**. The output must be exactly as specified. The output consists of ***T*** lines, one line per entry in the table. Each line should be terminated by a new-line character. The entries should be displayed in the same order as the input.

Each entry has the following format:

***C* : *L* E_2**

where

- ***C*** is the character in hexadecimal using the format "`0x%2.2x`"
- ***L*** is a decimal integer, using the format "`%d`"
- E_2 is the binary representation of the encoding, it should be exactly ***L*** characters (0s and 1s) long. (See example)

Examples

Input (Ex 1)	Output (Ex 1)	Input (Ex 2)	Output (Ex 2)
0x0a 0x21 0x30 0x00 0x57 0x42 0x00 0x48 0x43 0x00 0x6c 0x24 0x00 0x6f 0x38 0x00 0x0a 0x3a 0x00 0x65 0x4c 0x00 0x64 0x4d 0x00 0x20 0x4e 0x00 0x72 0x4f 0x00 0x00 0x00 0x00 0x00	0x21 : 3 000 0x57 : 4 0010 0x48 : 4 0011 0x6c : 2 01 0x6f : 3 100 0x0a : 3 101 0x65 : 4 1100 0x64 : 4 1101 0x20 : 4 1110 0x72 : 4 1111	0x0d 0x6b 0x10 0x00 0x6a 0x28 0x00 0x69 0x3c 0x00 0x68 0x4e 0x00 0x67 0x5f 0x00 0x0a 0x6f 0x80 0x78 0xcf 0xc0 0x61 0xcf 0xc1 0x62 0xbf 0xc2 0x63 0xaf 0xc4 0x64 0x9f 0xc8 0x65 0x8f 0xd0 0x66 0x7f 0xe0 0x00 0x00 0x00 0x00	0x6b : 1 0 0x6a : 2 10 0x69 : 3 110 0x68 : 4 1110 0x67 : 5 11110 0x0a : 6 111110 0x78 : 12 111111000000 0x61 : 12 111111000001 0x62 : 11 11111100001 0x63 : 10 1111110001 0x64 : 9 111111001 0x65 : 8 11111101 0x66 : 7 1111111

Problem 2: The Full Decompress

Extend your program from Problem 1 to use the decoding table to decompress the bitstream that follows it.

Input

The input format is the same as in Problem 1, except that the four bytes after the table encode an integer, ***N***, in big endian byte order, denoting the number of compressed characters encoded in the bitstream that follows. The rest of the input consists of zero or more bytes (in hexadecimal representation) encoding a bitstream, using the same bit order as the encoding of the table. For convenience, no line will be longer than 60 characters, but it is recommended to simply read in one byte at a time as the bitstream is being decoded. (See example.)

Processing

Same as problem 1, except that

- If ***N*** is not 0, do **not** output the decoding table as described in Problem 1, and output the text from the decompressed bitstream instead.
- To decompress a character from the bitstream:
  - Start with an encoding ***D*** of length 0 (0 bits)
  - In a loop (Note: ***D*** can be at most 12 bits long)
    - Add the next bit from the bitstream to ***D***
    - Search the table for encoding ***D***. Both bit pattern and length must match
    - If found, break
  - Output corresponding character ***C***.
- Once the expected number of characters is decoded, stop.
- All encodings are prefix free, meaning that no encoding is a prefix of another encoding.

Output

Same output format as for Problem 1 if the ***N*** is 0, otherwise, output the decompressed text. (See example.)

Example

Input	Output
0x0a 0x21 0x30 0x00 0x57 0x42 0x00 0x48 0x43 0x00 0x6c 0x24 0x00 0x6f 0x38 0x00 0x0a 0x3a 0x00 0x65 0x4c 0x00 0x64 0x4d 0x00 0x20 0x4e 0x00 0x72 0x4f 0x00 0x00 0x00 0x00 0x0e 0x3c 0x59 0xc5 0x3d 0xd1 0x68	Hello World!

Grading

If your program does not compile, it is considered non-functional and of extremely poor quality, meaning you will receive 0 for the solution.

The assignment will be graded based on three criteria:

Functionality: "Does it work according to specifications?". This is determined in an automated fashion by running your program on several inputs and ensuring that the outputs match the expected outputs. The score is determined based on the number of tests that your program passes. So, if your program passes t/T tests, you will receive that proportion of the marks.

Quality of Solution: "Is it a good solution?" This considers whether the approach and algorithm in your solution is correct. This is determined by visual inspection of the code. It is possible to get a good grade on this part even if you have bugs that cause your code to fail some of the tests.

Code Clarity: "Is it well written?" This considers whether the solution is properly formatted, well documented, and follows coding style guidelines. A single overall mark will be assigned for clarity. Please see the Style Guide in the Assignment section of the course in Brightspace.

The following grading scheme will be used:

Task	100%	80%	60%	40%	20%	0%
Functionality (20 marks)	Equal to the number of tests passed.					
Problem 1 Solution Quality (10 marks)	Appropriate approach used. No apparent bugs. Code is robust.	Appropriate approach used. Minor bugs in implementation. Code is robust.	Appropriate approach used. Major bugs present.	Appropriate approach used.	Reasonable attempt has been made.	No code submitted or code does not compile
Problem 2 Solution Quality (10 marks)	Appropriate approach used. No apparent bugs. Code is robust.	Appropriate approach used. Minor bugs in implementation. Code is robust.	Appropriate approach used. Major bugs present.	Appropriate approach used.	Reasonable attempt has been made.	
Code Clarity (10 marks) Indentation, formatting, naming, comments	Code looks professional and follows all style guidelines	Code looks good and mostly follows style guidelines.	Code is mostly readable and mostly follows some of the style guidelines	Code is hard to read and follows few of the style guidelines	Code is illegible	

Hints and Suggestions

- Lab 3 will be very helpful for doing Assignment 2.
- Start early. The sample solution is under 100 lines of code, but bit manipulations can be tricky.
- The Standard Library functions I found most useful are: `scanf()` and `printf()`.
- I found it helpful to create a couple helper functions. I recommend keeping all functions together in your `main.c` but having a couple helper functions will make your `main()` function simpler.

Assignment Submission

Submission and testing are done using Git, Gitlab, and Gitlab CI/CD. You can submit as many times as you wish, up to the deadline. Every time a submission occurs, functional tests are executed, and you can view the results of the tests. To submit use the same procedure as Assignment 0.

Assignment Testing without Submission

Testing via submission can take some time, especially if the server is loaded. You can run the tests without submitting your code by using the provided `runtests.sh` script. Running the script with no arguments will run all the tests. Running the script with the test number, i.e., 00, 01, 02, 03, … 09, will run that specific test. Please see below for how run the script.

Get your program ready to run

If you are developing directly on the unix server,

1. SSH into the remote server and be sure you are in the **marsdec1** or **marsdecx** directory.
2. Be sure the program is compiled by running **make**.

If you are using CLion

1. Run your program on the remote server as described in the CLion tutorials.
2. Open a remote host terminal via **Tools → Open Remote Host Terminal**

If you are using VSCode

1. Run your program on the remote server as described in VSCode tutorials.
2. Click on the Terminal pane in the bottom half of the window or via **Terminal → New Terminal**

Run the script

3. Run the script in the terminal by using the command:

```
./runtest.sh
```

to run all the tests, or specify the test number to run a specific test, e.g. :

```
./runtest.sh 07
```

You will see the test run in the terminal window.