CS 137: Algorithms in the Real World — Spring 2022

The purposes of this project are (1) tie in with Ma/CS 6a and CS 13 content, and (2) to understand the algorithms and formats behind an application that you regularly use.

Project Overview

In this project, you will implement `zip`, a real application, *from scratch*. By the end, you will fully understand how the `zip` utility works–there will be no magic left. This project ramps up in difficulty pretty quickly. Task 1 is a "review" task and should be a nice warm-up. Task 2 involves conceptually understanding theory, and then implementing it. Tasks 3 and 4 involve difficult design decisions and wrap the project together.

A Linux machine is recommended but not required. You must use Linux to run the "deflatesplain" and "zipsplain" binaries. If you do not have a Linux machine, you may use your Labradoodle account for the purposes of working on this project.

Restrictions

You *must* implement this project *without starter code* using C++ or Rust.

Deliverables

- `myzip0` and `myunzip0` as specified in Task 1.1 and Task 1.2
- `inflate` as specified in Task 2
- `huffman` as specified in Task 3.1
- `lz77` as specified in Task 3.2
- `myzip` and `myunzip` as specified in Task 4.1 and Task 4.2

Task 1: Understanding the `Zip` Format

In this part, you will implement a bare-bones `myzip0` utility which will be able to take a single file and create a `.zip` archive containing that file. You will also implement a bare-bones `myunzip0` utility which will extract a single file from a `.zip` archive that only contains one file. Note that you will *not* be compressing the file yet. First, we need to get a sense of what the `.zip` format looks like, then, later, we'll implement the compression.

The `Zip` Format

The full `.zip` format is very complicated and featureful, but we will only implement a small subset of it. One of the major learning outcomes of these first few tasks is for you to realize that the "file formats" on your computer are really just magic numbers along with a specification. While it is unnecessary, if you are interested in reading the full specification for the zip format, it can be found at

`https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT`

This section we will provide you with everything you need to duplicate our restricted version of the zip file format. Our zip files will insist that the zip archive only contain one file and not be encrypted. All of our `.zip` archives will contain three sections (in order): (1) local file record, (2) central directory record, (3) end of central directory record.

Local File Record

A "local file record" is made up of the following bytes:

Field Name	Length	Description
local file signature	4 bytes	Use `0x04034b50`
extract version	2 bytes	Use 20
general purpose flag	2 bytes	Use 0
compression method	2 bytes	0 means not compressed; 8 means compressed with DEFLATE
last mod file time	2 bytes	Use 0
last mod file date	2 bytes	Use 0
crc-32	4 bytes	Use `0xdeadbeef`
compressed file size	4 bytes	Interpreted as an unsigned 32-bit integer
uncompressed file size	4 bytes	Interpreted as an unsigned 32-bit integer
file name length	2 bytes	Interpreted as an unsigned 16-bit integer
extra field length	2 bytes	Use 0
file name	variable	The file name, interpreted as an ASCII string (no null-termination)
extra field	variable	Skip this field
file data	variable	If compression method is 0, this is just the file data; if compression method is 8, this is a DEFLATE stream

Note that we will expect several of these to be *incorrect* values (in particular, the date/time stamps and the crc-32). We make this decision because it doesn't actually affect the archive, but it makes writing the code substantially simpler.

Central Directory Record

A "central directory" record is made up the following bytes:

Field Name	Length	Description
central directory signature	4 bytes	Use `0x02014b50`
specification version	1 byte	Use 30
made by	1 byte	Use 65
extract version	2 bytes	Use 20
general purpose bit flag	2 bytes	Use 0
compression method	2 bytes	0 means not compressed; 8 means compressed with DEFLATE
last mod file time	2 bytes	Use 0
last mod file date	2 bytes	Use 0
crc-32	4 bytes	Use `0xdeadbeef`
compressed file size	4 bytes	Interpreted as an unsigned 32-bit integer
uncompressed file size	4 bytes	Interpreted as an unsigned 32-bit integer
file name length	2 bytes	Interpreted as an unsigned 16-bit integer
extra field length	2 bytes	Use 0
file comment length	2 bytes	Use 0
disk number start	2 bytes	Use 0
internal file attributes	2 bytes	Use 1
external file attributes	4 bytes	Use 1
offset of local header	4 bytes	Use 0
file name	variable	The file name as an ASCII string (no null-termination)
extra field	variable	Skip this field
file comment	variable	Skip this field

End Of Central Directory Record

An "end of central directory" record is made up the following bytes:

Field Name	Length	Description
end of central dir signature	4 bytes	Use `0x06054b50`
number of this disk	2 bytes	Use `0`
number of the start disk	2 bytes	Use `0`
total number of entries on this disk	2 bytes	Use `1`
total number of entries	2 bytes	Use `1`
size of the central directory record	4 bytes	Interpreted as an unsigned 32-bit integer
offset of start of central directory	4 bytes	Interpreted as an unsigned 32-bit integer
.ZIP file comment length	2 bytes	Use `0`
.ZIP file comment	variable	Skip this field

The `zipsplain` Tool

We have provided you with a program called `zipsplain` which takes a zip archive path as a command-line argument, and explains the contents of the zip by printing out the values for each of the fields. We recommend you try running `zipsplain` on a few zip archives as well as the ones you create with your own `myzip0`.

Task 1.1: Writing `myzip0`

It's finally time for you to do something! Undoubtedly, you skipped some of the above description–that's okay, we all do it. Unfortunately, you will need to pay very close attention to the exact format of each type of record as you are writing your `myzip0` program. `myzip0` should take two command-line arguments: (1) the name of the output zip archive, and (2) the name of the file to put in the archive. You only have to handle actual files with lengths. When run, your program should create a file with the provided name in the zip format with the provided file *using compression method 0* (which means "uncompressed"–we will handle another compression method later).

Task 1.2: Writing `myunzip0`

`myunzip0` is the inverse of `myzip0`. It should take a single command-line argument (the name of the zip file), and create a new file (or overwrite an old file) with the name and content specified in the zip archive. If the zip archive specifies compression method 0, you should copy the file contents to a new file with the name specified inside. If the zip archive specifies compression method 8, you should copy the file contents to a new file with the name specified inside *followed by .deflate* (this is a non-standard format that we will use to temporarily simplify our programs); that is, if the file name specified in the zip archive is "`hullo`", you should create a file called "`hullo.deflate`" (i.e., append `.deflate` to the file name). You do not have to deal with non-conforming, corrupted zip files, zip files spread over multiple disks, or compression methods other than 0 or 8, but you can print out an error message if you like.

`myunzip0` may NOT make the assumptions we've made in `myzip0` about any of the fields. It must work on files generated by normal `zip`.

Task 2: Inflating a File

So far, in `myzip0` and `myunzip0` , we skipped the *compression* and *decompression* which is the fun part! In this task, you will implement *decompression* for mode "8" which tells your program to use compression that conforms to the `DEFLATE` specification. When you're done, you should be able to take files in the `.deflate` format and recover the original file. We ask you implement these algorithms in a separate binary called `inflate`, (which you will combine into `myzip` and `myunzip` later). If your `inflate` binary is given a file `x.deflate`, you should create a file called `x` with the decompressed contents. If the input file does not end in `.deflate`, behavior is undefined.

BEFORE PROCEEDING, MAKE SURE YOU HAVE A SOLID UNDERSTANDING OF HUFFMAN CODES AND LZ77!!

The `DEFLATE` Format

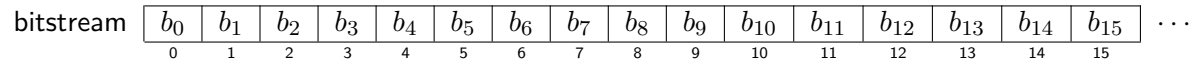
The `DEFLATE` Compressed Data Format is the compression format behind `zip`, `gzip`, `png`, and many more file formats. We will implement a generic program that takes a *raw* `DEFLATE` stream and decompresses it into the original file. To do this, we'll need to get into the nitty-gritty details of bits in `DEFLATE` files. The RFC ("request for comments"–a description of an internet-related format or idea) for `DEFLATE` can be found here:

`https://www.ietf.org/rfc/rfc1951.txt`

That said, we strongly recommend **NOT attempting to read the RFC before this spec**. It is not meant as a way to learn about how to implement the format; rather, it serves as a "standards" document. We have broken down and synthesized various parts of this document for you in the following sections, with sufficient information to complete the project. If something is not clear, we recommend asking the course staff, but you may supplement from the RFC if you like.

Reading Bits

Imagine the bitstream looks like this:



In memory, this looks like one byte: $b_7b_6b_5b_4b_3b_2b_1b_0$, followed by another byte: $b_{15}b_{14}b_{13}b_{12}b_{11}b_{10}b_9b_8$, etc. That is, bytes should be read from least-significant-bit to most-significant bit. (We call this "reverse-stream order.") We recommend writing a "`get_next_bit`" function as a primitive to use in all your other functions. This is not an optimized way to work with the stream, but it is logically the easiest place to start.

The `deflatesplain` Tool

Just like in the previous part, we provide you with a tool that reads and explains the file format. This one is called `deflatesplain`. We recommend running it on various raw `DEFLATE` streams as you try to convert them manually. To generate a raw `DEFLATE` stream, you should use the standard `zip` utility to generate a `.zip` file, and then use your `myunzip` to generate a `.deflate` file.

Some numbers in a `DEFLATE` stream should be read **in reverse stream order** which means "10" indicates "1", and others should be read in **stream order**. We try to specify this inline everywhere.

Blocks

`DEFLATE` streams are made up of "blocks" (pieces of the decoded output). Every block begins with a single bit (BFINAL) that indicates whether it is the final block (1) or is not the final block (0). This bit is followed by two reverse-stream order bits (BTYPE) that indicate which type of compression was used for the current block: no compression (0), fixed Huffman codes (1), and dynamic Huffman codes (2). Your code does not need to handle the no compression case.

Alphabets

`DEFLATE` streams have already been compressed with huffman coding and LZ77 which means we read the stream in a special way. In particular, each "part" of the bitstream is a prefix-free Huffman coded value. That value can be (1) a literal, (2) a number of characters to repeat (a length), (3) an end-of-block character, or (4) a number of characters to go back in the buffer (a distance).

`DEFLATE` utilizes two conceptually distinct alphabets: literal-length and distance. The literal-length alphabet encodes literals (symbols 0-255), the "end of block code" (symbol 256), and lengths of repetitions (symbols 257-285). The distance alphabet (with symbols 0-29) encodes the distances to go back when we hit a repeat. With fixed Huffman codes, the Huffman codes are given by the specification; for dynamic Huffman codes, the Huffman codes are specified as part of the bitstream. We begin with inflating fixed Huffman codes, because they are conceptually simpler.

The look-behind buffer should *not* be reset between blocks.

Task 2.1: Inflating Fixed Huffman Codes

Consider the following `DEFLATE` stream:

1	10	10010001	10010001	0000010:00000	0000000	00
BFINAL	BTYPE	a	a	len 4 : go back 1	EOB	EOS-FILL

Notice that this stream is a single block (because BFINAL is set) and it is a fixed Huffman code block (because BTYPE=1). "10010001" decodes to "a", "0000010" indicates a repeat of length four, "00000" represents "move the buffer pointer one character back", "0000000" is the end-of-block character, and "00" is a necessary filler (**at the end of the entire stream**) to make the number of bits a multiple of 8. What remains is to discuss how to figure out all of the Huffman codes.

To decode symbols from the literal-length alphabet, we first attempt to match with 7 characters, then 8, then 9, as necessary, using the following conversions. In this case, we can treat all the codes as numbers, because they have been designed to allow that simplification. Note that these conversions should be read in stream order.

$$\begin{array}{lcllcl} 0000000 & - & 0010111 & \longleftrightarrow & 256 & - & 279 \\ 00110000 & - & 10111111 & \longleftrightarrow & 0 & - & 143 \\ 11000000 & - & 11000111 & \longleftrightarrow & 280 & - & 287 \\ 110010000 & - & 111111111 & \longleftrightarrow & 144 & - & 255 \end{array}$$

Looking back to our above example stream and recalling that "a" is 97 in ASCII, we see that the chart marks $(00110000)_2 + (97)_{10} = (145)_{10} = (10010001)_2$ as required!

Following this algorithm on the next bits in the stream, we find a match on $(0000010)_2 + (256)_{10} = (258)_{10}$. Since 258 is larger than 256, it must be a *length*. The conversion for lengths is:

257 (+0 bits) $= 3$	265 (+1 bit) $= 11$ to 12	273 (+3 bits) $= 35$ to 42	281 (+5 bits) $= 131$ to 162
258 (+0 bits) $= 4$	266 (+1 bit) $= 13$ to 14	274 (+3 bits) $= 43$ to 50	282 (+5 bits) $= 163$ to 194
259 (+0 bits) $= 5$	267 (+1 bit) $= 15$ to 16	275 (+3 bits) $= 51$ to 58	283 (+5 bits) $= 195$ to 226
260 (+0 bits) $= 6$	268 (+1 bit) $= 17$ to 18	276 (+3 bits) $= 59$ to 66	284 (+5 bits) $= 227$ to 257
261 (+0 bits) $= 7$	269 (+2 bits) $= 19$ to 22	277 (+4 bits) $= 67$ to 82	285 (+0 bits) $= 258$
262 (+0 bits) $= 8$	270 (+2 bits) $= 23$ to 26	278 (+4 bits) $= 83$ to 98	
263 (+0 bits) $= 9$	271 (+2 bits) $= 27$ to 30	279 (+4 bits) $= 99$ to 114	
264 (+0 bits) $= 10$	272 (+2 bits) $= 31$ to 34	280 (+4 bits) $= 115$ to 130	

258 does indeed represent length 4. Notice that some of these have a "$+n$ bits" designation. This means that after reading the length code, you should read n more bits, chunk them into a number and index into the range (on the right side of the equals sign) using that number (these bits need to be interpreted in reverse-stream order). Immediately after a length code, we will always see a "distance code". (Recall that LZ77 needs a length and a distance to index into the look-behind buffer.)

286 and 287 will never appear in a valid `DEFLATE` file!

Distance codes are always 5-bit numbers (in stream order) potentially followed by extra digits interpreted in the same way as with the length codes. To interpret 5-bit distance codes, we use a similar chart as above:

0 (+0 bits) = 1	8 (+3 bits) = 17 to 24	16 (+7 bits) = 257 to 384	24 (+11 bits) = 4097 to 6144
1 (+0 bits) = 2	9 (+3 bits) = 25 to 32	17 (+7 bits) = 385 to 512	25 (+11 bits) = 6145 to 8192
2 (+0 bits) = 3	10 (+4 bits) = 33 to 48	18 (+8 bits) = 513 to 768	26 (+12 bits) = 8193 to 12288
3 (+0 bits) = 4	11 (+4 bits) = 49 to 64	19 (+8 bits) = 769 to 1024	27 (+12 bits) = 12289 to 16384
4 (+1 bits) = 5 to 6	12 (+5 bits) = 65 to 96	20 (+9 bits) = 1025 to 1536	28 (+13 bits) = 16385 to 24576
5 (+1 bits) = 7 to 8	13 (+5 bits) = 97 to 128	21 (+9 bits) = 1537 to 2048	29 (+13 bits) = 24577 to 32768
6 (+2 bits) = 9 to 12	14 (+6 bits) = 129 to 192	22 (+10 bits) = 2049 to 3072	
7 (+2 bits) = 13 to 16	15 (+6 bits) = 193 to 256	23 (+10 bits) = 3073 to 4096	

Finally, every block ends with an end-of-block code which is always the code word for 256. In the case of fixed Huffman codes, this is "0000000". Note that these bit-streams must be byte-aligned; so, there may be extra garbage bits to throw away at the very end of the stream.

Here is another example of a fixed code stream. We recommend you work through it by hand.

1	10	10010001	10010010	10010011	10010100	10010001	0000101:00011	10010101	00111010	0000000	00
BFINAL	BTYPE	a	b	c	d	a	len 7 : go back 4	e	\n	EOB	EOS-FILL

Task 2.2: Inflating Dynamic Huffman Codes

There are only a few differences between dynamic and fixed Huffman codes, but those differences are non-trivial. Between the BTYPE and the actual content bits, there are several pieces which, together, make up the Huffman trees for both alphabets. For "even greater compactness", the Huffman trees themselves are compressed. . . with another hardcoded Huffman code. (Yay, fun.)

Remember that a single "Huffman Code" consists of two parts: a set of abstract *symbols* called the *alphabet*, and a bijection from each symbol to a set of bitstrings called *codes*. The codes, as a set, are prefix-free. Confusingly, each bitstring is sometimes also called a single "code" or "huffman code".

Canonical Huffman Codes

To specify the various different Huffman codes, the `DEFLATE` specification uses *canonical Huffman codes* which allows the stream to only include the code lengths of each symbol. Canonical Huffman codes add two additional rules to standard Huffman coding: (1) all codes of a particular bit-length have lexicographically consecutive values, (2) shorter codes are lexicographically before longer codes. To read a canonical Huffman code, use the following algorithm:

(1) Count the number of codes for each code length by populating an array `bl_count`, where `bl_count[N]` is the number of codes of length `N`. Make sure to set `bl_count[0] = 0`.

(2) Find the numerical value of the smallest code for each code length with the following algorithm:

```
code = 0
for (i = 1; i <= max_length; i++) {
   code = (code + bl_count[i-1]) << 1
   next_code[i] = code
}
```

(3) Loop through the alphabet *in lexicographical order* and assign consecutive values starting length `i` at `next_code[i]`

For example, imagine that we read in the following lengths: 3, 3, 3, 3, 3, 2, 4, 4, and the (known) alphabet order is A, B, C, D, E, F, G, H. Then, we can make the following association:

A	3
B	3
C	3
D	3
E	3
F	2
G	4
H	4

Going through the algorithm with these code length associations:

(1) `bl_count[0] = 0`
`bl_count[1] = 0`
`bl_count[2] = 1`
`bl_count[3] = 5`
`bl_count[4] = 2`

(2) `next_code[1] = 0`
`next_code[2] = 0`
`next_code[3] = 2`
`next_code[4] = 14`

(3) A = "2" at length 3 = 010
B = "3" at length 3 = 011
C = "4" at length 3 = 100
D = "5" at length 3 = 101
E = "6" at length 3 = 110
F = "0" at length 2 = 00
G = "14" at length 4 = 1110
H = "15" at length 4 = 1111

Dynamic Code Block Header

Just like fixed blocks, dynamic blocks begin with BFINAL (1 bit) followed by BTYPE (2 bits). After these two fields, there are three numbers (in reverse stream order): HLIT (5 bits), HDIST (5 bits), HCLEN (4 bits). HLIT + 257 is the number of Huffman codes representing the length-literal codes, HDIST + 1 is the number of Huffman codes representing distance codes, and HCLEN + 4 is the number of "code length" codes. The "code length" codes are used to decode the literal/length and distance codes. So, our first task is to re-construct the canonical Huffman code for the *code length alphabet*.

Dynamic Code Block Alphabets

The *code length alphabet* is made up of the numbers 0 - 18. This alphabet is used to specify the lengths of the Huffman codes for the remaining two alphabets. In particular, the numbers should be interpreted as follows:

- $0 - 15$: These numbers indicate that the next code is of that literal length
- 16 ($+2$ bits): 16 indicates that the previous code should be copied; the two bits specify how many times (between 3 and 6)
- 17 ($+3$ bits): 17 indicates that the next chunk of codes are all 0; the three bits specify how many times (between 3 and 10)
- 18 ($+7$ bits): 18 indicates that the next chunk of codes are all 0; the seven bits specify how many times (between 11 and 138)

The remaining two alphabets, you are already familiar with: the literal-length alphabet and the distance alphabet. The Huffman codes for each of these alphabets will be specified by their lengths (because they are canonical Huffman codes)–which will be specified using the code length alphabet.

Remember to use **lexicographical order** in the canonical Huffman codes algorithm–not the order they are specified.

Dynamic Code Block Body

After the header fields, there are three "groups" of bits which specify the three alphabets:

- (HCLEN + 4) × 3 bits: the code lengths for the "code length alphabet" which is made up of the numbers 0 through 18. Note that the number of code lengths specified here is variable; the stream may omit code lengths at the end (and we will assume those are unused). The *order* of the code lengths is 16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15. Each code is specified as a 3-bit (reverse stream) integer. A code length of 0 means that that symbol is unused in the alphabet specifications. When specifying codes, each of these numbers has the meaning described above. After using the canonical Huffman code algorithm to recover the code length alphabet, we are ready to decode the remaining two alphabets using the above translation. Note that this stream should be treated as one giant list of "codes" that eventually get separated into two alphabets.

- HLIT + 257 Huffman codes (for the code length alphabet) giving the code lengths for the literal/length alphabet, which is made up of symbols 0 - 285.

There may be fewer codes than the number of symbols, because a single code in this stream can represent more than one length.

- HDIST + 1 Huffman codes (for the code length alphabet) giving code lengths for the distance alphabet, which is made up of the symbols 0 - 29.

After the codebooks are decoded, we are finally ready to decompress the actual data. The remaining format of the block is identical to the fixed codes (including the end-of-block character at the end) except the actual Huffman codes used. Unfortunately, unlike the fixed codes, it is possible that the same numerical number will be used as two different length codes. For example, it's possible that 0001 and 1 are both valid codes in an alphabet. We recommend you store codes as *numerical values BY LENGTH*. (Think of this as a 2-dimensional-array-like structure). We have deliberately not described the full algorithm or data structure here as we want you to figure it out on your own.

You should implement these algorithms in a binary called `inflate`. `inflate` should take the input file name as a single argument and output to a file with the same name without the suffix `.deflate`. (So, an input file `x.deflate` would output to a file called `x`.)

Task 3: Deflating a File with Fixed Huffman Codes

Deflation requires more design decisions than the previous parts. There are many algorithms that can be used to effectively compress files into the DEFLATE format, and we only specify a single one here. Keep in mind that you can (and should!) diverge from our algorithm if you think you have something better or more effective. We consider our algorithm to be a "baseline", and it can be significantly improved with a little effort. Note that we will only use fixed Huffman codes for compression, but you could implement a switch to dynamic Huffman codes as above and beyond. The tests will expect fixed Huffman codes, however.

Task 3.1: Huffman Coding

As a first pass, you should only do the Huffman coding and treat the entire file as a single block. You will need to pay close attention to bit order, and the codes will need to be "reversed" (remember, we have to respect the stream ordering). To output the codes, you will need to segment the bits into bytes inside an integer buffer. Make sure to reference the fixed Huffman codes we previously used to decode symbols; you will be writing the inverted transformation this time! Ultimately, this means that you will output the header, a bunch of Huffman codes, and then the end-of-block character. You should write this code separately in a binary called `huffman`. `huffman` should take the input file name as a single argument and

output to a file with the same name and the suffix `.deflate`. (So, an input file `x` would output to a file called `x.deflate`.) Later, we will add LZ77 to `huffman`.

Task 3.2: Self-Contained LZ77 Compression

Before we add in LZ77 to our Huffman codes, it is beneficial to write a separate binary that writes out LZ77 compressed text in a readable format. In this task, we will describe a baseline LZ77 algorithm to implement along with the expected output. Recall that LZ77 compression outputs two "types" of symbols: literals and length-distance pairs. You should output literals as themselves and length-distance pairs as `<length,distance>`. So, the file "aaaaaa" might be output as "a<5,1>". You should implement this algorithm in a separate binary called `lz77`. `lz77` should take the input file name as a single argument and output to a file with the same name and the suffix `.lz77`. (So, an input file `x` would output to a file called `x.lz77`.)

There are several restrictions on what we're allowed to do in our compression based on the `DEFLATE` specification:

- Our sliding window cannot be bigger than 32K (32768 bytes)
- Lengths cannot be smaller than 3 or larger than 258.
- Distances cannot be larger than 32768

Beyond that, we are free to replace whatever characters we want with length-distance pairs; on one extreme, we have the situation from the previous section (where we never use length-distance pairs), and on the other extreme we have the "optimal" compression ratio which chooses exactly the right places to use length-distance pairs to minimize the file length. In an ideal world, we would have the latter, but, in practice, it would take far too long to actually compute; so, we settle for "pretty good".

Now, we discuss our algorithm to compress "enough" using LZ77 which we call `lz77_baseline`.

We keep two indices into the window: `idx` (the location of the next byte to output) and `lookahead_idx` (the location of the last byte we've actually seen). In addition to the window, we keep a hash table with *three byte* keys and a list of corresponding positions in the window that match that hash code.

For example, if the stream is "abcabcd", our hash table would conceptually look like:

```
{
    abc: 3 -> 0 -> NULL,
    bca: 1 -> NULL,
    bcd: 4 -> NULL,
    cab: 2 -> NULL
}
```

Since our hash table has three-byte keys, we take as an invariant that we always look ahead two bytes (so that instead of looking for a single byte, we're always looking up *three*). If there are multiple matches (for example, `abc` above), we take the (1) longest and (2) closest match (prioritizing length over closeness); to facilitate this, we always prepend new positions into the hash chains rather than appending them. To simplify implementation, we do not ever delete from the chains (though, we DO have to check that a match is not too old before choosing it).

Finally, we can write pseudocode for `lz77_baseline`:

```
1 lz77_baseline():
2    initialize our current match as non-existent
3    while there are bytes left:
4       read in bytes until we have at least three unprocessed ones and add them to the buffer
5       look up the next three bytes in the hash table
6       if we found a longer match that is no more than max length:
7          update our current match with the one we found in the hash table
```

```
    else:
        if there is a current match:
            output the match
        else:
            output the next byte as a literal
        update the hash table
```

There are (clearly) details we have omitted from our pseudocode (for example, how to actually look for the match), but this is the basic algorithm we recommend you implement first. We recommend you check your implementation on "obviously compressible data" (repeated "a" bytes, a wikipedia page, etc.).

Task 3.3: Adding LZ77 Compression To Your Huffman Compression

Combine your Task 3.1 and Task 3.2 implementations to integrate your LZ77 implementation into the code that outputs a `DEFLATE` stream. In particular, this will involve translating literals and length-distance codes into their fixed Huffman code equivalents. The `huffman` binary should now also perform LZ77 (you are encouraged to share code between binaries).

Task 4: Combining Everything

We now have two conceptually separate pieces of `zip`: (1) programs that generate and deconstruct zip files, and (2) programs that compress and decompress using method 8. In this task, your job is to combine these pieces into one final product.

Task 4.1: Writing `myzip`

Integrate your code for method 8 into your code for `myzip`.

Task 4.2: Writing `myunzip`

Integrate your code for method 8 into your code for `myunzip`.