CS 213, Winter 2025
Pack Lab: Unpacking, Decompressing, and Decrypting Data

Contents

1 Introduction

The purpose of this assignment is to become more familiar with bit-level representations of integers and floating point numbers, as well as to learn how to manipulate data at the level of bits, bytes, words, etc, the fundamental units on which computing is based.

You will do this by writing tools to interpret ("unpack") data files. Except for the most basic file types, a file contains metadata ("headers") that describe how the bytes of the file are to be interpreted. For example, a music file contains information about how to reconstruct a playable audio signal from the data in the file. A video file is similar. A movie file typically is a "container" that audio and video "streams" are packed together within.

In this lab, you will be given a working utility called `pack`, which is capable of "packing" a file similar to other utilities such as 7Zip, GZip, Zip, and others. `pack` supports four operations:

Checksumming (ensuring data integrity)

Encryption (ensuring data is password protected)

Compression (reducing file size in a lossless manner)

Floating Point (the ability to apply the previous operations on floating point data contained in 2–3 streams)

Your goal for this lab is to write the `unpack` utility, which takes a "packed" file, and reverses the operations applied by the `pack` utility to recover the original file.

2 Logistics

You may work in groups of up to two people for this lab. All handins will be done electronically via Gradescope (see Section 2.2 for details). Clarifications and revisions will be posted to Piazza, the course discussion group.

2.1 Handout Instructions

For this lab, you will need to connect to any of the class servers (`http://it.eecs.northwestern.edu/info/2015/11/03/info-labs.html`) via SSH or your tool of choice. Some of the servers that you can use include:

- `moore.wot.eecs.northwestern.edu`
- The Wilkinson Lab
- The Batman machines

You will need the file `packlab-handout.tar`, which you will find in the `~cs213/HANDOUT` directory on the class servers.

Expect limited setup help from course staff if you choose to use tools outside of SSH (such as VSCode).

We highly recommend working on `moore`, as it is available worldwide and has all the tools you will need already installed. **You should test that your code works on `moore` prior to submitting.**

Please note that you will hand in your work on Gradescope, which is also where it will be tested & graded.

You will need a (protected) directory on a Linux machine in which to do your work. You can create a protected[1] directory like this:

```
unix> cd ~/
unix> mkdir mypacklab
unix> chmod 700 mypacklab
unix> cd mypacklab
unix> tar xvf ~cs213/HANDOUT/packlab-handout.tar
```

[1] The `chmod` command will set things so that only the owner of the directory can read, write, or `cd` into it.

This will cause a number of files to be unpacked in the directory. Your unpacked `packlab-handout.tar` file will contain at least the following files:

`pack` We provide you with the same binary that we used to create the example packed files. Note that this binary was compiled on Moore, so it will likely only work there.

`unpack.c` This is the driver code for the `unpack` tool. It opens the specified packed file, calls `parse_header()` (perhaps multiple times), runs the unpack steps specified by the headers, and then writes the unpacked output to the specified file.

`unpack-utilities.h` This is the header for the utility functions you will implement. You do not need to edit anything within this file, but you should read it because it documents what you must do. You should read the `packlab_config_t` struct and its fields.

`unpack-utilities.c` **This is what you must edit!** You will implement the functions in this file to actually do the unpacking.

Please be sure to read the `README` file.

2.2 Handin Instructions

The Gradescope platform will be used to handle submitting and grading. You can submit as many times as you like. To submit, run the following command on `moore` from the directory containing your lab files (you may be prompted to input your Gradescope credentials the first time):

```
unix> make submit
```

After submitting, the Gradescope website will show whether your solution compiled and whether it successfully unpacked an example file. **Gradescope will also run many more test cases which are hidden until after the deadline, so make sure to thoroughly test before submitting.**

You **must** also mark your partnership on Gradescope. This can be done by clicking the button labeled "Group Members" and selecting your partner from the dropdown. Unfortunately, you will have to do this for each submission.

2.3 Evaluation

You will be graded on having a correct implementation for the major functions in `unpack-utilities.c`. The six major functions listed below are each worth 15% each of the lab's overall grade.

1. `parse_header()`
2. `calculate_checksum()`
3. `lfsr_step()`

4. `decrypt_data()`
5. `decompress_data()`
6. `join_float_array()`

The next 5% will come from your implementation working to unpack every file in `example_files/`. The last 5% will come from your implementation passing every test our randomized test generator can generate.

Partial credit will be given for partially correct code.

Your implementation should successfully unpack **any** file that meets the specification, and should handle errors gracefully.

You will not be graded on the tests you write. However, we still recommend you write small unit tests to verify that your implementation is actually working correctly in pieces before trying to do everything at once. You may also find our randomized tester, described later, to be useful.

3 Lab Breakdown

"Packed" files are files generated by running the `pack` utility on an input file. Packed files are comprised of one or more "streams" of data, with each stream having a "header" section followed by a "file data" section.[2] The header describes the following file data section (e.g. size) as well as what options were applied to the input file during packing (e.g. checksum, compression). The file data section contains the (potentially modified) contents of the original input file.

You will have to implement multiple features to unpack every file we provide. All files will contain *at least one* header (Section 3.1). Each header can have a mixture of checksums (Section 5), encryption (Section 6), and lossless compression (Section 7). Additionally, multiple headers may be combined to provide floating-point streams (Section 9). Each of these features will be discussed in-turn.

3.1 The Header

Each of our packed files consists of one or more (header, packed data) pairs, one for each stream. The header describes the stream and how to interpret the packed data.

Things can get confusing, but, remember, files are just a collection of bytes!

The header in this assignment is a variable-length header, with a minimum length of 20 bytes and a maximum of 38 bytes; because some configurations only exist in the header if certain options were applied during packing. The header identifies and configures the packed file, including:

[2] If you look carefully, you will also notice "padding" (blank space) between headers and data. Padding is used to align each header and data chunk to a page boundary to facilitate memory mapping of the file. You'll learn what these concepts mean later in the class, and don't need to understand them (or even observe them) to do this lab.

- “Magic bytes” and version number to identify packed files.
- Flags to determine which options were applied to the file when it was packed.
- Configuration for each of the particular options mentioned in the flags.
  Encrypting the file does **not** add any fields to the header. The fields and their expected values are discussed below.
- The length of the data following the header.
- The length of the original data.

A header always begins at an offset into the file that is an integer multiple of 4096 bytes. Similarly, the data following a header also begins at an offset into the file that is an integer multiple of 4096 bytes. (Note that 0 is a valid integer multiple of 4096!)

3.1.1 The Minimal Header

The minimal header happens when both compression and checksumming are disabled for that data stream[3]. Table 1 shows the header's format.

Byte Offset	0	1	2	3
0	Magic: `0x0213`		Version: `0x03`	Flags
4	Original Length in bytes (8 bytes)			
8				
12	Length in bytes (8 bytes)			
16				

Table 1: Header with Minimal Flags

Magic Identifies this file as a packed file. This will always be `0x0213`, in **big-endian** format.

Version Identifies which version of the pack protocol is used. This will always be `0x03`.

Flags Determines which options were applied when packing the file. `0` means the option was disabled, `1` means the was enabled. Table 2 shows this.

Original Length How long the stream was originally (before packing) **in bytes**. This is an 8-byte unsigned integer, in **little-endian** format. (See explanation below.)

Length How long the stream following the header is **in bytes**. This is an 8-byte unsigned integer, in **little-endian** format.

[3]Remember, we allow multiple streams in a packed file, and those files will have multiple headers.

As mentioned previously, headers and data are aligned to start at file offsets that are an integer multiple of 4096 bytes, and padding is added to meet those invariants. For example, the first header in packed file starts at byte-offset `0x0`, and the corresponding data starts at `0x1000` (decimal 4096). Streams are allowed to be as long as they need to be (so long as their length fits into the `Length` field of the header). You do not need to worry about this since it is handled for you in `unpack.c`.

The Flags Field

Bit	7	6	5	4	3	2	1	0
Value	Compressed?	Encrypted?	Checksummed?	Continuation?	Floats?	Float3?	Unused: All 0	

Table 2: Pack Flags

Table 2 describes the `Flags` field of the header at the bit level—that is, each element is 1 bit wide. The combination of flags that are set (one) defines the format of the header and of the data, and is described next. If all the flags are zero, then the data stored is simply the original data. The `Continuation?`, `Floats?`, and `Float3` flags relate to multiple streams per file. It's easiest to start by considering a file with only one stream, and consider particular combinations of flags.

For a stream, all combinations of `Compressed?`, `Encrypted?`, `Checksummed?` are allowed. Several examples of this, and the affect it has on header layout, are demonstrated below.

3.1.2 Example: Compression Enabled, Checksum Disabled

If the `Compressed?` flag is set, then an additional "compression dictionary" will be included in the header. The compression dictionary is a 16-byte array of the 16-most-used bytes from the original uncompressed file. Table 3 shows the layout. This is only present in the header if *that* stream was compressed!

Byte Offset	0	1	2	3
0	Magic: `0x0213`		Version: `0x03`	Flags
4 8	Original Length in bytes (8 bytes)			
12 16	Length in bytes (8 bytes)			
20	`Dictionary[0]`	`Dictionary[1]`	`Dictionary[2]`	`Dictionary[3]`
24	`Dictionary[4]`	`Dictionary[5]`	`Dictionary[6]`	`Dictionary[7]`
28	`Dictionary[8]`	`Dictionary[9]`	`Dictionary[10]`	`Dictionary[11]`
32	`Dictionary[12]`	`Dictionary[13]`	`Dictionary[14]`	`Dictionary[15]`

Table 3: Header: Compression Enabled and Checksum Disabled

3.1.3 Example: Compression Disabled, Checksum Enabled

If the `Checksummed?` flag is set, then an additional "checksum" value is included in the header. The checksum is a 16-bit unsigned **big-endian** value. It was computed on the file's data **after** compression and encryption. It will only be present in the header if that stream has a checksum. Table 4 shows the layout.

Byte Offset	0	1	2	3
0	Magic: `0x0213`		Version: `0x03`	Flags
4 8	Original Length in bytes (8 bytes)			
12 16	Length in bytes (8 bytes)			
20	Checksum			

Table 4: Header: Compression Disabled and Checksum Enabled

3.1.4 Example: Compression Enabled, Checksum Enabled

If both `Compression?` and `Checksummed?` are set, then the header will contain both a compression dictionary and a checksum value. The header will be laid out like Table 5. The compression dictionary will **always** come before the checksum value.

Byte Offset	0	1	2	3
0	Magic: `0x0213`		Version: `0x03`	Flags
4 8	Original Length in bytes (8 bytes)			
12 16	Length in bytes (8 bytes)			
20	`Dictionary[0]`	`Dictionary[1]`	`Dictionary[2]`	`Dictionary[3]`
24	`Dictionary[4]`	`Dictionary[5]`	`Dictionary[6]`	`Dictionary[7]`
28	`Dictionary[8]`	`Dictionary[9]`	`Dictionary[10]`	`Dictionary[11]`
32	`Dictionary[12]`	`Dictionary[13]`	`Dictionary[14]`	`Dictionary[15]`
36	Checksum			

Table 5: Header: Compression Enabled and Checksum Enabled

3.2 Multiple Streams

If the `Continuation?` flag is set, then there are more headers **after** this one. The rest of the flags (`Compressed?`, `Encrypted?`, `Checksummed?`, and `Float?`) apply to only **this** data stream. Each header in the file will use the same format as described in this section. If the `Continuation?` bit is cleared to `0` that is the last stream in the packed file. Tables 6 & 7 show example file layouts for some simple cases with multiple streams.

Byte Offset	0	1	2	3
0	Magic: `0x0213`		Version: `0x03`	`00010000`
4 8	Original length in bytes (8 bytes)			
12 16	Length in bytes (8 bytes)			
`0x1000`	Data (Assuming data length $\leq$ 4096 bytes)			
`0x2000` + 0	Magic: `0x0213`		Version: `0x03`	`00000000`
`0x2000` + 4 `0x2000` + 8	Original length in bytes (8 bytes)			
`0x2000` + 12 `0x2000` + 16	Length in bytes (8 bytes)			
`0x3000`	Data			

Table 6: Header: Multiple Streams, Minimal Headers, Short Data Length

Byte Offset	0	1	2	3
0	Magic: `0x0213`		Version: `0x03`	`xxx1x000`
4 8	Original length in bytes (8 bytes)			
12 16	Length in bytes (8 bytes)			
`0x1000`	Data (Assuming 4096 $\leq$ data length $\leq$ 8192 bytes)			
`0x3000` + 0	Magic: `0x0213`		Version: `0x03`	`xxx0x000`
`0x3000` + 4 `0x3000` + 8	Original length in bytes (8 bytes)			
`0x3000` + 12 `0x3000` + 16	Length in bytes (8 bytes)			
`0x4000`	Data			

Table 7: Header: Multiple Streams, Minimal Headers, Long Data Length

3.3 Floating-Point Streams

In this lab, we use multiple streams to encode floating point numbers. The `Floats?` flag indicates that the stream is part of a two stream group that, when combined, will result in the original floating point file contents. The two streams consist of the sign & mantissa/fraction stream and the exponent stream, **in that order**. Table 8 gives a simple example[4] of a packed file containing floating-point data.

Floating-point streams **can** have encryption, compression, and checksumming applied to them, just like any other stream. **You must account for this.**[5]

Byte Offset	0	1	2	3
0	Magic: `0x0213`		Version: `0x03`	`11111000`
4 8	Original length in bytes (8 bytes)			
12 16	Length in bytes (8 bytes)			
20	`Dictionary[0]`	`Dictionary[1]`	`Dictionary[2]`	`Dictionary[3]`
24	`Dictionary[4]`	`Dictionary[5]`	`Dictionary[6]`	`Dictionary[7]`
28	`Dictionary[8]`	`Dictionary[9]`	`Dictionary[10]`	`Dictionary[11]`
32	`Dictionary[12]`	`Dictionary[13]`	`Dictionary[14]`	`Dictionary[15]`
36	Checksum			
`0x1000`	Sign & Mantissa Data (Assuming Length $\leq (1365 * 3 = 4095)$)			
`0x2000` + 0	Magic: `0x0213`		Version: `0x03`	`11101000`
`0x2000` + 4 `0x2000` + 8	Original length in bytes (8 bytes)			
`0x2000` + 12 `0x2000` + 16	Length in bytes (8 bytes)			
`0x2000` + 20	`Dictionary[0]`	`Dictionary[1]`	`Dictionary[2]`	`Dictionary[3]`
`0x2000` + 24	`Dictionary[4]`	`Dictionary[5]`	`Dictionary[6]`	`Dictionary[7]`
`0x2000` + 28	`Dictionary[8]`	`Dictionary[9]`	`Dictionary[10]`	`Dictionary[11]`
`0x2000` + 32	`Dictionary[12]`	`Dictionary[13]`	`Dictionary[14]`	`Dictionary[15]`
`0x2000` + 36	Checksum			
`0x3000`	Exponent Data (Assuming Length ≤ 1365)			

Table 8: Header: Floating-Point Streams

[4]The length of 1365 for the exponent data was chosen for this example because it allows the sign & mantissa to fit into 4096 bytes.

[5]You may notice the `Float3?` flag. This is for extra credit, and is described at the end.

4 Step 1: Implement Header Parsing

You will need implement the function `parse_header()` to decode the header's data. The inputs to `parse_header()` are as follows:

`input_data` An array of bytes from the packed file

`input_len` The length of `input_data`

`config` A pointer to the struct to write header data to Check `unpack_utilities.h` for the definition of the struct.

The general steps to implementing `parse_header()` are as follows:

1. Verify that the magic and version are correct.
2. Check which options are set in `Flags`, set the appropriate fields in the struct, and determine how many more bytes need to be read from the header.
3. Get the length of this stream and the length of the original data.
4. Pull out the compression dictionary for this stream if `Compression?` is enabled.
5. Pull out the checksum value for this stream if `Checksummed?` is enabled.

C does not have any built-in tools to access individual bits of a byte. You will need to use bitwise operations to pull the single bits out that you need. These include: `>> << | & ^`

Check out the textbook (Sections 2.1.7–2.1.9) and/or `~cs213/HANDOUT/play` (particularly the `data` subdirectory) for a refresher on bitwise operations.[6]

A few notes/hints:

- Make sure to check `input_len` before accessing `input_data` to prevent any out-of-bound accesses
- Some fields of `config` may not be used depending on the flags in the header
- If for any reason the header is not valid, the `is_valid` field of `config` should be set to `false` (and `true` otherwise). If the `is_valid` field is `false`, the values of the other fields do not matter.

Working with memory in C can be dangerous, which is why you must practice **defensive programming**. The most basic step of defensive programming is sanity-checking the values of lengths for buffers. Be careful to check that `input_len` is the right length to hold the expected header data!

[6]Strictly speaking, we are ignoring C bitfields and struct packing here, which are advanced topics. If you are interested, take a look after you finish.

Defensive programming practices are the onus of the programmer, which means memory-safety bugs can appear in seemingly valid programs. This is why memory-safe systems languages, like Rust and Go have been created.

We use C and C++ in this class because most of the world's software systems are still written in these languages. See `https://www.tiobe.com/tiobe-index/` if you are curious about the relative popularity of programming languages in job descriptions.

5 Step 2: Implement Checksumming

When a file is downloaded, it is often accompanied by a **checksum**. A file's checksum is a (usually cryptographic) hash of the *contents* of the file, computed by the uploader/distributor. Checksums allow downloaders to verify that the downloaded file matches the content the distributor uploaded.

To do so, users can recompute the checksum value of the downloaded file and compare it against the checksum provided by the uploader. If **any** bit in the file has changed, the computed and provided checksum values will differ.

In this lab, each stream in a packed file can contain a checksum if `Checksummed?` is enabled. The checksum is represented as an unsigned 16-bit **big-endian** integer in the stream's header. Our checksum implementation is just a simple **big-endian** unsigned 16-bit integer, initialized to zero (`0`). The checksum is computed by adding the value of every byte in the file to this counter, and allowing for overflows.

Take a 3-byte stream for example containing the values `[0x01, 0x03, 0x04]`. The checksum would be $\texttt{0x01} + \texttt{0x03} + \texttt{0x04} = \texttt{0x08}$. If the stream is too long, the resulting summation will *overflow*, but that is intended in a checksum implementation, and is not an error. If your calculated checksum does not match what the `pack` tool calculated, the `unpack` tool must exit with an error. We have already provided this double-checking and error-exiting code for you in `unpack.c`.

You must implement `calculate_checksum()`, which is the same function used by the `pack` utility to calculate a stream's checksum. `unpack.c` uses `calculate_checksum()` to recompute a stream's checksum and compares it against the checksum provided in the header in order to check that the stream's content has not changed. The inputs to `calculate_checksum()` are:

`input_data` An array of bytes to calculate the checksum over. This array does not contain header bytes, and it must **not** be modified.

`input_len` The length of `input_data`.

The general steps to implementing `calculate_checksum()` are as follows:

1. Initialize the checksum to `0`
2. Add each byte in `input_data` to the checksum, one-by-one
3. Return the checksum

6 Step 3: Implement Decryption

Encryption is a process which takes some data and *reversibly* turns it into something that looks like random garbage. This is a key feature underpinning everything you do online, including banking, chatting with ChatGPT, looking at cat videos, and even figuring out what `youtube.com` actually means. You probably even use encryption on the device you are reading this on! Most students using MacOS or Windows laptops will likely have disk-encryption turned on, ensuring data stored on your computer is protected, even if someone were to rip it out of your laptop.

6.1 Linear Feedback Shift Registers

One key step in encryption is to be able to generate a repeatable stream of seemingly random numbers. A linear feedback shift register (LFSR) is a class of pseudorandom number generators built with bit manipulations that are simple to implement in both hardware and software. These create pseudorandom sequences of bits that do not repeat for a very long time. In our implementation, an LFSR takes in an input state and creates a new output state by XOR-ing several bits together to create the new most-significant bit and shifting all bits in the input state to the right. Figure 1 gives you an idea of the operation.

6.2 A 4-bit LFSR Example

This section goes over how an example 4-bit LFSR would work. **However, `pack` uses the 16-bit LFSR described in Section 6.3.** This section is solely to aid your understanding of LFSRs[7].

Figure 1 shows a 4-bit LFSR with an initial state of `0b0101`. To generate the next state, perform the following steps:

1. XOR `bit[0]` with `bit[1]` to get `1`
2. Right shift the state by one bit (`0b0101` becomes `0b0010`)
3. Set the most significant bit to the result from step 1 (`0b0010` becomes `0b1010`)

The new state is shown in Figure 2.

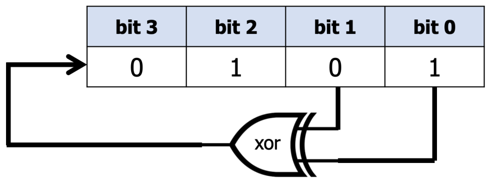

Figure 1: 4-bit LFSR Example, Step 1

[7]Check out this YouTube video if this example isn't clear: https://www.youtube.com/watch?v=1UCaZjdRC_c

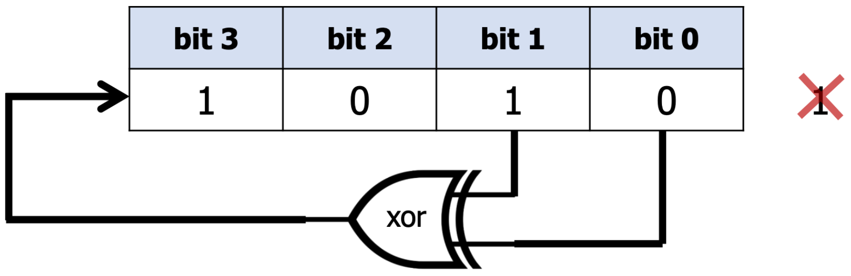

Figure 2: 4-bit LFSR Example, Step 2

To get the next state from here, repeat this process.

1. XOR `bit[0]` with `bit[1]` to get `1`
2. Right shift the state by one bit (`0b1010` becomes `0b0101`)
3. Set the most significant bit to the result from step 1 (`0b0101` becomes `0b1101`)

The new state generated from this process is `0b1101`.

As you can see, this process can be repeatedly applied to generate a sequence of numbers. If you continue to work out this process, you should find that the next few states are `0b1110`, `0b1111`, and `0b0111` (give it a shot with pencil and paper!). After repeating this process 15 total times, you should also see that the states begin to repeat.

6.3 `pack`'s LFSR

In this lab, we use a 16-bit LFSR, where bits 0, 6, 9, and 13 are used to form the next state's bit. Figure 3 shows these connections visually.

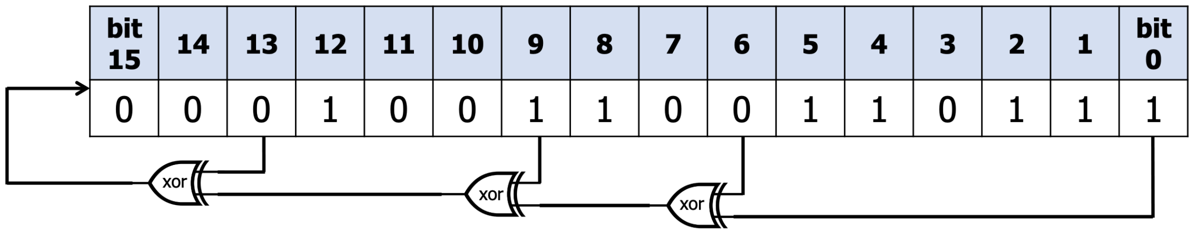

Figure 3: `pack`'s LFSR

You should find that the current state in Figure 3 is `0x1337` and the next state is `0x099b`.

`lfsr_step()`

You must implement `lfsr_step()` takes in an LFSR state and returns the next LFSR state. You will use this to implement `decrypt_data()` later.

To generate the next state, perform the following steps:

1. XOR `bit[0]`, `bit[6]`, `bit[9]`, and `bit[13]`
2. Right shift the state by one bit
3. Set the most significant bit to the result from step 1

Testing your LFSR

We have provided some code for you that can test your LFSR implementation, inside `test-utilities.c`. This tests two things:

1. Your LFSR must iterate in the known pattern expected by the LFSR we described.
2. Your LFSR must iterate over all non-zero 16-bit integers.

We recommend ensuring that `lfsr_step()` passes these tests before attempting `decrypt_data()`, otherwise the process can be very frustrating. If your `lfsr_step()` is not working, try working out the bit patterns you expect as both inputs and outputs on paper and compare them against your code's results.

Your implementation must **not** save any state internally. To iterate through multiple LFSR states, call `lfsr_state` with the previous input multiple times:

```
new_state1 = lfsr_step(init_state)
new_state2 = lfsr_step(new_state1)
new_state3 = lfsr_step(new_state2)
```

6.4 Stream Encryption

Stream cipher encryption is a basic form of encryption where every individual chunk of data in a stream is combined with random data to encrypt it. However, since encryption must be reversible, the "random" bytes must actually be pseudorandom, meaning that they appear random but are actually deterministic based on some initial state (such as a password or key).

6.5 Decrypting Data

The `pack` utility encrypts data by XOR-ing bytes of data with the 16-bit pseudorandom numbers generated using `lfsr_step()`. To decrypt the data, we can simply XOR the encrypted data with the same 16-bit pseudorandom numbers used for encryption. However, this requires initializing the LFSR with the same initial state used during encryption. In this lab, the initial state is a 16-bit unsigned encryption key, which is generated by running `calculate_checksum()` on a user supplied password.

> **For this lab, all encrypted files in `example_files/` are encrypted using the password** `cs213`

While this implementation is simple and sufficient for this lab, a few notes about why this encryption method is insecure:

- Computing encryption keys with `calculate_checksum` results in many passwords mapping to the same key (e.g. `calculate_checksum("ba") == calculate_checksum("ab")`)
- Since the initial state is a 16-bit number, there are only $2^{16} - 1$ possible initial states, making it possible to decrypt the file through brute force

6.6 Implementing `decrypt_data()`

For this lab, you will need to implement `decrypt_data()`, which takes some data encrypted by the `pack` utility and reverses the encryption. The inputs to `decrypt_data()` are:

`input_data` An array of bytes to decrypt. It does not contain any header bytes and must **not** be modified

`input_len` The length of `input_data`

`output_data` The array to write the decrypted data to

`output_len` The maximum number of bytes which can be written to `output_data`

`encryption_key` The initial state for the LFSR generated from the user password

The general steps to implementing `decrypt_data()` are as follows:

1. Generate a new LFSR state based on the previous state using `lfsr_step()` (remember that the initial state should be the encryption key discussed in Section 6.5).
2. Grab the next two bytes of data and XOR them with the LFSR state in **little-endian order**.
3. Write the results out.
4. Repeat for every two bytes of input data.
5. If there is one remaining byte (e.g. there were an odd number of bytes in the input data), then the remaining byte should be XOR-ed with the **least-significant byte** of the LFSR state.

As an example, take a 4-byte `input_data`, with values `[0x60, 0x5A, 0xFF, 0xB7]` and an LFSR that was initialized with `0x1337`, as seen in Figure 3. As we saw in that example, the LFSR's first output is `0x099B` and the second is `0x84CD`. Using this, we can calculate:

$$\begin{aligned} \texttt{0x9B} \mathbin{\hat{}} \texttt{0x60} &= \texttt{0xFB} \\ \texttt{0x09} \mathbin{\hat{}} \texttt{0x5A} &= \texttt{0x53} \\ \texttt{0xCD} \mathbin{\hat{}} \texttt{0xFF} &= \texttt{0x32} \\ \texttt{0x84} \mathbin{\hat{}} \texttt{0xB7} &= \texttt{0x33} \end{aligned}$$

This makes `output_data` have the values `[0xFB, 0x53, 0x32, 0x33]`.

As another example, if `input_data` is `[0x21]`, and LFSR output is `0x099B`, then:

$$\texttt{0x9B} \mathbin{\hat{}} \texttt{0x21} = \texttt{0xBA}$$

This gives `output_data` the value `[0xBA]`. The LFSR's most-significant byte of output (`0x09`) is unused in this case.

7 Step 4: Implement Decompression

Compression is best exemplified by the following quote made by Kevin from *The Office*:

> Why waste time say lot word when few word do trick?

Compression is a way to reduce the size of data. There are two kinds of compression, lossless and lossy.

Lossless compression is used everywhere in the world. `.zip`, `.wav`, and `.png` files are losslessly compressed file formats. Lossless compression reversibly reduces a file's size; if you know the algorithm that was used to compress the file, you can use the algorithm's inverse to decompress the file to get the original contents.[8]

In contrast, `.mp3`, `.jpeg`, `.mp4` are lossy file formats. Lossy compression is used everywhere, but you often do not see it. Netflix, YouTube, and most other streaming platforms, serve videos in the MP4 format, which uses lossy video compression to remove unneeded elements in a video to make it smaller. Lossy formats achieve greater size reduction by "losing" some of the data during compression. As the name suggests, lossy compression is a one-way street. Once you lossy-compress a file, you cannot perfectly decompress the file to its original version.

In this lab, the `pack` utility utilizes lossless compression, since we want you to recover the files exactly as they were created. Lossless compression works by finding commonly repeated (redundant) patterns, and changing them to share a smaller and non-redundant pattern between all locations. `pack` uses a dictionary-based run-length compression scheme. However, there are better lossless compression algorithms that exist, such as Huffman encoding and LZW.

[8]You may find it interesting that the limit to how small lossless compression can possibly make a sequence is essentially equivalent to the size of the smallest program that can print out that sequence.

7.1 Our Run-Length Compression

We use a variation of run-length encoding where repeated bytes get replaced by a two-byte sequence. Unlike traditional run-length encoding you may have already seen, we are using a dictionary to encodes runs of the 16 most-frequently occurring bytes in the file. These bytes are stored in the dictionary in the file's header (see Table 3).

When `pack`'s run-length compressor finds bytes that appear twice or more in a row, we replace up to (and including) 15 repetitions with a two-byte sequence. The two-byte sequence is broken up into two one-byte components:

1. A special byte (an "escape byte") that denotes that a byte in the file is compressed. The escape byte will always be `0x07`, as it is unlikely to be used in text files.[9]

2. Information about which dictionary entry and how many repetitions should occur.

The second byte is broken down as follows:

Dictionary Index (bits 0-3) A 4-bit unsigned number which is the index in the compression dictionary of the character repeated in the run.

Repeated Count (bits 4-7) A 4-bit unsigned number which is the length of the run.

Table 9 shows the format that these two bytes will have. An example is given at the end of this section.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	Escape Byte: `0x07`								Repeat Count				Dictionary Index			

Table 9: Compressed Bytes Normal Case

But what happens if the stream contains a `0x07` and we need to compress it? We use a literal escape byte, which is an escape byte that has an invalid encoding. You are already familiar with this concept, because many programming languages use it (e.g. `"\\"` is a literal escape byte for the \ character). Table 10 shows how the literal character `0x07` is encoded.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	Escape Byte: `0x07`								Literal Escape Byte `0x00`							

Table 10: Compressed Bytes Escaped Case

7.2 Implementing `decompress_data()`

For this lab, you will need to implement the `decompress_data()` function, which takes some data compressed by the `pack` utility and decompresses it. The inputs to `decompress_data()` are as follows:

[9] `0x07` in ASCII means "ring a bell", which today just means your terminal beeps.

`input_data` An array of bytes to decompress. It does not contain any header bytes and should **not** be modified

`input_len` The length of `input_data`

`output_data` The array to write the decompressed data to

`output_len` The maximum number of bytes which can be written to `output_data`

`dictionary_data` The compression dictionary used when compressing the data (taken from the header)

The general steps to implementing `decompress_data()` are as follows:

1. Iterate through `input_data`
2. If the byte is normal, write it to output
3. If the byte is an escape byte, read the second byte to determine what to do. Either:
   (a) Write a repeated run of bytes to output
   (b) Or, write a single literal escape byte
4. If the very last byte is an escape byte, treat it as a normal byte and write it to output
5. Return the number of bytes written to output

Some hints/notes:

- Be careful not to read past the end of input or write past the end of output.
- `output_data` will always be large enough to hold the decompressed input (assuming you parsed all of the header fields correctly).

A Short Example

Take a file with the following bytes: `[0x01, 0x07, 0x42]` and a dictionary that contains `[0x30, 0x31, 0x32, ..., 0x3F]`. The decompressed output should be `[0x01, 0x32, 0x32, 0x32, 0x32]`, because the second byte in the input is an escape byte, and the third byte says that `Dictionary[2]` should be copied 4 times.

8 Step 5: Understand Multiple Streams

Up to this point, you have only seen packed files with a single "stream" of data. Packing this way will only work on singular files. But what if we want to have multiple items in a single pack file? We could figure out a way to make a single header work with multiple pieces of data, but an (arguably) simpler solution is to allow multiple headers in a file, each of which describes their data stream in-turn. In fact, all of your video files work this way! A video file that you can double-click and watch is actually a container that has a video stream and an audio stream, where each is separated in the file, but is locatable by using the headers.

We do something similar in this lab, where a single pack file can have multiple data streams inside of it. The header will precede each data stream. Please see Tables 6 & 7 for what you can expect.

> There is nothing to implement to support this feature, you just need to be aware of it for the next portion. Multiple streams will only be used for floating-point streams (both `Floats?` and `Float3?`).

9 Step 6: Implement Two-Stream Floating Point

If the `Floats?` flag in the header is enabled, then the original file that was packed was a file of IEEE 754 single-precision floating point numbers (a vector, if you will). The `pack` utility splits these floats into two streams:

1. `signfrac`: A stream containing the sign+mantissa (sign+fraction) bits
2. `exp`: A stream containing the exponent bits

This splitting process is shown in Figure 4. Note that a sign is 1 bit, a mantissa/fraction is 23 bits, and an exponent is 8 bits. The two streams need to be joined so that `signfrac[0]` and `exp[0]` together will form `float[0]`.

For example, say we have packed just a single floating-point number, 300.0. This is `0x43960000` in hexadecimal (in little-endian, the byte values will be `0x00`, `0x00`, `0x96`, `0x43` in that order). If we break that into parts, the 24-bit `signfrac[0]` value will be `0x160000` (little-endian bytes `0x00`, `0x00`, and `0x16`), and `exp[0]` will be `0x87`. You must reconstruct the original floating-point value using these two values. Figure 4 shows this same operation visually. You might try writing that example in binary on paper to see how it works.

If you want a refresher on the IEEE 754 single-precision floating-point format, please refer to the textbook. You can also find interactive FP converters online, which can show how a number changes as you change its bits.

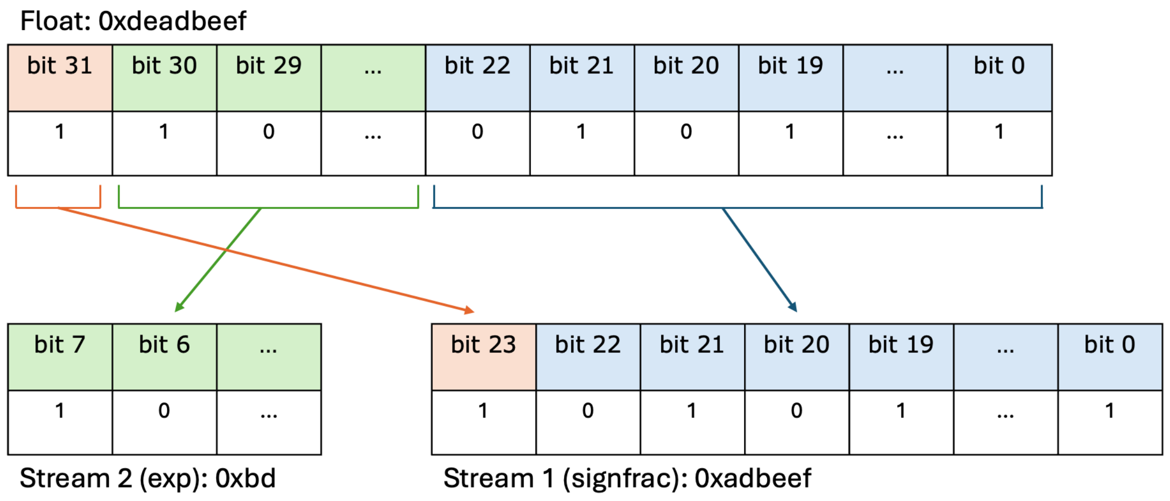

Figure 4: Example Floating-Point Splitting

`join_float_array()`

This function takes in `input_signfrac` and `input_exp` streams of sign+fraction and exponent portions of IEEE 754 floating-point numbers, and combines them to create the `output_data` stream. The header is not present in either of the input streams.

The order of the streams is:

Sign & Fraction Stream The sign & fraction stream is composed of the 1-bit sign and 23-bit mantissa to make 24 bits (3-bytes). The mantissa is in little-endian format, and the sign bit is the most-significant bit.

Exponent Stream The exponent stream is composed of 1-byte values, the 8-bit exponent.

The inputs to `join_float_array()` are:

`input_signfrac` An array of bytes that contains the sign bit and fractional (mantissa) portion of the floating-point number

`input_len_bytes_signfrac` The number of bytes in the sign+fraction array.

`input_exp` An array of bytes that only contains data to decompress.

`input_len_bytes_exp` The number of bytes in the exponent array.

`output_data` Where you write the decompressed bytes of floating-point values in little-endian order.

`output_len_bytes` The **maximum** number of bytes you can write to `output_data`.

10 Testing Your Code

This lab is implementation heavy, so doing thorough testing is important for ensuring correctness and to help with debugging. **You will not receive any debugging help from course staff if you have not made a good-faith effort to debug through testing.**

The codebase is set up to compile into a Debug version, which supports source-level debugging with `gdb` and also has several sanitizers enabled which will print warnings at runtime if your code does unfortunate things.

10.1 Unit Testing

Unit tests should be written in `test-utilities.c`. Each of the functions you need to implement takes in an array of data, so you can craft your own array of unsigned 8-bit data and give it to the function. This is significantly easier than crafting entire files to unpack. We have already implemented a test for you as an example, creatively named `example_test()`.

10.2 Whole-File Testing

We provide you with a functioning `pack` program that you can use to pack up your own files. You can then run your implementation of `unpack` and compare against the original file to see if it works. We have provided some already-packed files as examples, along with their original versions in `example_files/`.

Using `pack`:

```
usage: ./pack [-cekfg] inputfilename outputfilename

  -c           Enable Compression
  -e           Enable Encryption
  -k           Enable Checksum
  -f           Enable packing of IEEE-754 Single-Precision
                Floating-Point Numbers
  -g           Enable advanced packing of IEEE-754 Single-Precision
                Floating-Point Numbers (extra credit)
```

Your implementation must allow `unpack` to determine which packing options were used and undoes them. If the file was encrypted, a password must be entered. Your output file should be identical to the original file before packing, though this will not work fully until all of your code is done.

10.3 Randomized Testing

Randomized testing involves generating random inputs (both data and flags/header options) for your program (in this case, your `unpack` implementation). The randomized tester is the executable `test_rand_multiple.pl`.

You can run it to find out the arguments, and we will give you some suggested arguments later. This tool will randomly generate data, pack it with some combination of flags, get your `unpack` to unpack it, and then compare your output with the original generated data. If your `unpack` crashes or your output is not the same as the original, it will tell you. It will repeat this process as many times as you would like. Your goal is to never crash, and always produce the identical output.

10.4 Useful Tools

One common debugging need is to see the bytes in a packed (or unpacked) file. To do so, you can use one of several tools for getting the raw byte values from a file. This allows the header bytes to be inspected manually.

xxd Allows you to examine the individual bytes of a file

hexdump Another tool for examining the bytes of a file

For example:

```
$ xxd example_files/short_file.txt.cek.pack

 00000000: 0213 03e0 0400 0000 0000 0000 0300 0000  ................
 00000010: 0000 0000 610a 0001 0203 0405 0608 090b  ....a...........
 00000020: 0c0d 0e0f 01b2 0000 0000 0000 0000 0000  ................
 ...
```

Another common debugging need is to compare the output after unpacking with the original file contents. The unpacked file and original version of the file should match exactly if your code is correct. `diff` is the best tool for this. It compares two files. If they match it won't output anything. If they differ it will either tell you that they differ or show you *how* they differ if they are plain text files.

For example:

```
$ ./unpack example_files/short_file.txt.cek.pack output_short_file.txt
$ diff example_files/short_file.txt output_short_file.txt
```

For debugging your code, use `gdb`. Even understanding code is easier if you can single-step through its execution. We cannot stress enough how important it is to become comfortable with a debugger like gdb.

11 Extra Credit

There is an extra credit opportunity for this lab, worth up to 10 extra points.

Since this is extra credit, expect limited implementation/debugging help from class staff.

We will also comment about a related research problem in this section, one which you might find interesting.

11.1 Extra Credit: Tri-stream Floating-Point

This extra credit opportunity is an extension of your 2-stream floating-point implementation. If `Floats?` is set and `Float3?` is *also set* then this is the 3-stream floating-point packed format.

In the floating point component of the main lab, we made things a bit easier on you by having the elements on the two streams be a multiple of *bytes* wide. Recall that the exponent stream was 1 byte wide, while the the combined mantissa and sign stream was 3 bytes wide.

In this extra credit, you will use the "`-g`" option of `pack`. This will split a 32 bit floating point number into *three* streams:

Mantissa Stream The mantissa stream is a stream of 23-bits for the fractional portion of the floating-point number. The mantissa is in little-endian format. These are **densely-packed** so an individual byte may include some bits from two different mantissas.

Exponent Stream The exponent stream is composed of 1-byte values, the 8-bit exponent.

Sign Stream The sign stream is a **densely-packed** stream of 1-bit sign values. This means that for every byte (8-bits) of the sign stream, there are 8 sign bits present. You must match up the 8 bits in each byte to their corresponding 23-bit mantissas and 8-bit exponents. **The bit-ordering within the byte of the sign stream could be "backwards" from what you might expect.** For example, `0b01100101` means the first sign bit is 1, the second sign bit is 0, and so on. This is because there is a convention that bit offsets start at the low-order bit.

Your job in this extra credit is to write `join_float_array_three_stream()`. Unlike your `join_float_array()` function, this one will need to operate bit-by-bit through the input data. We suggest that you write helper functions that allow you to operate on arbitrary-length arrays of bits. Our own implementation includes, for example, a `read_bits(src,dest,bit_offset,num_bits)` function, which reads `num_bits` bits from the bit array `src`, starting at `bit_offset` and writes them to the bit array `dest` starting at bit offset 0.