Pack Lab Overview

Revision 4

CS213 – Intro to Computer Systems

Branden Ghena – Fall 2023

High-level overview

- You will be given a utility that can “Pack” a file
  - It already works!

- Supports three operations:
  - Checksums - ensure data integrity
  - Encrypts - file is only readable with password
  - Compresses - reduces file size losslessly

- Your goal: write the “unpack” utility
  - Unpacks a file and writes data to a new output file
  - Unpacking gets you the same original file from before it was packed

Getting the lab files

- A tar of the lab file is available in the ~cs213/HANDOUT directory
  - Must be on the class server: moore.wot.eecs.northwestern.edu

- Steps:
  1. SSH into moore
  2. Make a directory to hold the lab files in
  3. Run the following command

```
tar xvf ~cs213/HANDOUT/packlab-starter.tar
```

- That will get you all the necessary lab files

Code files

- `unpack.c`
  - Application for unpacking files
  - Already written!
- `unpack-utilities.c`
  - Utilities used by the application to perform operations
  - **You need to write this**
- `unpack-utilities.h`
  - Header file for unpacking utilities
  - Includes comments about the purpose of each function
- `test-utilities.c`
  - Test code for unpacking utilities
  - **You will add to this to test your code**

Getting started suggestions

1. Understand what the existing code is doing
2. Implement `parse_header()`
3. Implement `calculate_checksum()`
4. Implement `lfsr_step()`
5. Implement `decrypt_data()`
6. Implement `decompress_data()`

Test as you go! Each of these functions can be tested independently

Submitting the lab files

- Gradescope will be used for grading your code
  - You can submit any number of times
  - Feedback: 1) does it compile and 2) does it run correctly on a test file
    - Many more tests are run, which are hidden until after the deadline

- To submit your code, on Moore run:

```
~cs213/HANDOUT/submit213 submit --hw packlab unpack-utilities.c
```

  - The first time you run the tool, it will ask you to log in with your Gradescope credentials

- You MUST also mark your partnership on Gradescope. Click the button labeled "Group Members" and select your partner from the dropdown
  - Unfortunately, you have to do this each time you submit code

Grading

- 19% each for correct implementations of the five major functions in `unpack-utilities.c`
  - `parse_header()`, `calculate_checksum()`, `lfsr_step()`, `decrypt_data()`, `decompress_data()`
- 5% for the entire unpack program working on the `example_files/`
- With some partial credit given for partially working code
- Your code should successfully unpack any file that meets the specification, and should also handle errors correctly
  - Invalid files, for example
- You are not graded on your tests

Changelog

- 4.0: Initial release for Fall 2023

Outline

- **Header Format**
- Checksums
- Encryption
  - Linear-Feedback Shift Register
  - Stream Cipher
- Compression
  - Compression dictionary
  - Escape sequences
- Testing

Format of packed files

- Remember: files are just a collection of bytes
- "Packed" files have two sections of bytes
  - Header then File data
- Header (4–22 bytes)
  - Identification and configuration of the packed file
  - Includes "magic bytes" and version to identify a packed file
  - Includes flags to determine which options are applied
  - Includes configurations for particular options
- File Data (0–2^{64} bytes)
  - Contents of the original file
  - Possibly encrypted and compressed

Minimal header: Compression and Checksum disabled

Byte offset	0	1	2	3
0	Magic: 0x0213		Version: 0x01	Flags

- Magic
  - Identifies this file as a “packed” file. Always 0x0213 (big-endian)
- Version
  - Identifies which version of the “pack” protocol is used. Always 0x01
- Flags
  - Determines which options have been applied to the file
  - 0 - disabled, 1 - enabled

Bit	7	6	5	4	3	2	1	0
Value	Compressed?	Encrypted?	Checksummed?	Unused: all zero				

Compression enabled, Checksum disabled

Byte offset	0	1	2	3
0	Magic: 0x0213		Version: 0x01	Flags
4	Dictionary[0]	Dictionary[1]	Dictionary[2]	Dictionary[3]
8	Dictionary[4]	Dictionary[5]	Dictionary[6]	Dictionary[7]
12	Dictionary[8]	Dictionary[9]	Dictionary[10]	Dictionary[11]
16	Dictionary[12]	Dictionary[13]	Dictionary[14]	Dictionary[15]

- Compression dictionary
  - 16-byte array used for compression
  - Contains 16 most-used bytes from the original uncompressed file
  - Only present in the header if the file was compressed

Compression disabled, Checksum enabled

Byte offset	0	1	2	3
0	Magic: 0x0213		Version: 0x01	Flags
4	Checksum			

- Checksum
  - 16-bit unsigned checksum value (big-endian)
  - Was computed on the data after compression and encryption
  - Only present in the header if the file was checksummed

- Note: you don't need to calculate a checksum here, this is the value that was already computed
  - `unpack.c` compares this to a new checksum value to check for validity

Full header: Compression and Checksum enabled

Byte offset	0	1	2	3
0	Magic: 0x0213		Version: 0x01	Flags
4	Dictionary[0]	Dictionary[1]	Dictionary[2]	Dictionary[3]
8	Dictionary[4]	Dictionary[5]	Dictionary[6]	Dictionary[7]
12	Dictionary[8]	Dictionary[9]	Dictionary[10]	Dictionary[11]
16	Dictionary[12]	Dictionary[13]	Dictionary[14]	Dictionary[15]
20	Checksum			

- Compression dictionary, if used, always comes before Checksum
- Note: encryption does not add any fields to the header

Steps to decoding a header

Steps:

1. Verify that the magic bytes and version byte are correct.

2. Check which options are set in “flags”.
   That will determine the remaining bytes in the header, if any.

3. Pull out compression dictionary data (if enabled).

4. Pull out checksum value (if enabled).

Accessing individual bits

- There's no native way in C to access individual bits of a byte

- Instead, you'll need to use operations on the byte to pull out only the bit(s) you need
  - >>, <<, |, &, etc.

- See "Bit Masking" section of "Data Operations" lecture
  - Slides 56-59: https://drive.google.com/file/d/1fGbCmLVBotqq7afJY2xkaJN5VFVdF5eV/view

You write this - Function: `parse_header()`

- Parses the header from the input data
  - `input_data` is an array of bytes. You can access it with [ ]
  - `input_len` is the number of bytes in `input_data`

- Write header configuration values into the config struct
  - Depending on the flags in the header, some fields in the config struct may not be used at all
  - Look at the header: `unpack-utilities.h` to see the config struct fields

- If the header is invalid for some reason, set `is_valid` to `false`
  - Otherwise, `is_valid` must be set to `true`
  - Be careful to check that the `input_len` is long enough to hold the expected header data!

Outline

- Header Format
- **Checksums**
- Encryption
  - Linear-Feedback Shift Register
  - Stream Cipher
- Compression
  - Compression dictionary
  - Escape sequences
- Testing

What is a checksum?

- Allows verification of file data integrity
  - If a byte in the file has changed, we can detect it!

- Concept
  - Some kind of hash of file data into a much smaller number
  - Process is repeatable and deterministic

- Integrity check: generate checksum twice
  - Once when "packing" the file. Record result in file header
  - Once when "unpacking" the file. Check against saved value in header
  - If they don't match, file contents have changed!

Checksum implementation

- Unsigned 16-bit integer, initialized to zero
- Add every byte of the file data to it, one-by-one
  - Modular arithmetic automatically occurs upon overflow

- Example
  - File data: [0x01, 0x03, 0x04]
  - Checksum: 0x08

- If the checksum doesn’t match when unpacking, the unpack tool should error and exit
  - This code is already written for you in `unpack.c`

You write this - Function: `calculate_checksum()`

- Calculates a 16-bit unsigned checksum value from the input
  - `input_data` is an array of bytes, you can access it with [ ]
  - `input_data` only contains data to calculate the checksum over, it does not contain header bytes
  - `input_len` is the number of bytes in `input_data`
- Must not modify the input data
- Return the calculated checksum value

Outline

- Header Format
- Checksums
- **Encryption**
  - **Linear-Feedback Shift Register**
  - **Stream Cipher**
- Compression
  - Compression dictionary
  - Escape sequences
- Testing

Basic stream cipher encryption

- Process: combine each individual byte of data with a random byte to encrypt it

1. Need some method for creating a series of random bytes
   - Must be deterministic based on some initial state (a password)

2. Need some operation for combining random bytes with data
   - XOR operation works well for this
   - To decrypt, just XOR against the random byte a second time

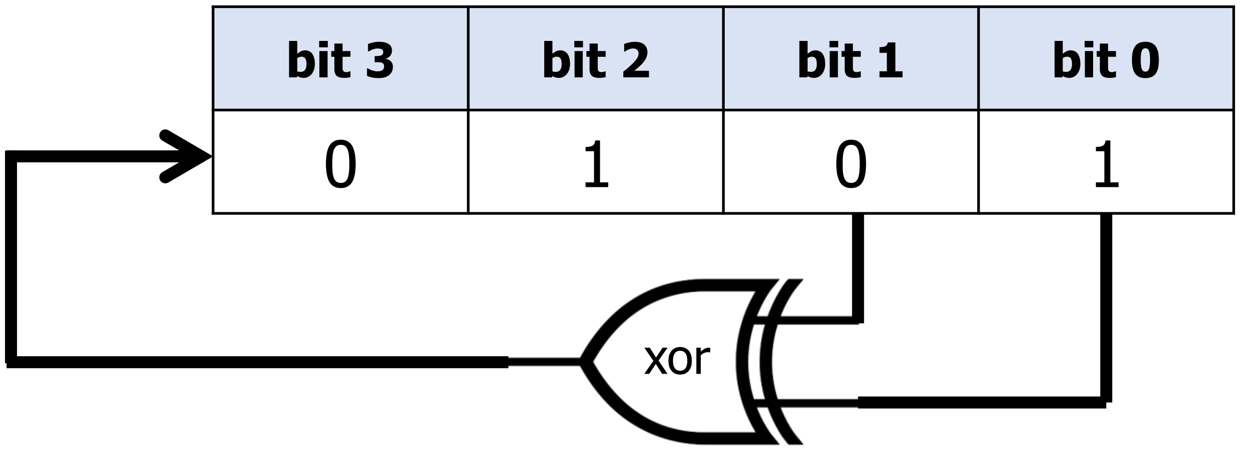
- Note: the method we're using is insufficient to provide good security
  - Only 65535 possible starting states
  - Could be brute-forced to decrypt the file

Method for creating a pseudorandom byte stream

- Linear-Feedback Shift Register (LFSR)
  - Pattern of bit manipulations that is simple to implement in hardware/software
  - Creates pseudorandom sequences of bits that do not repeat for a very long time

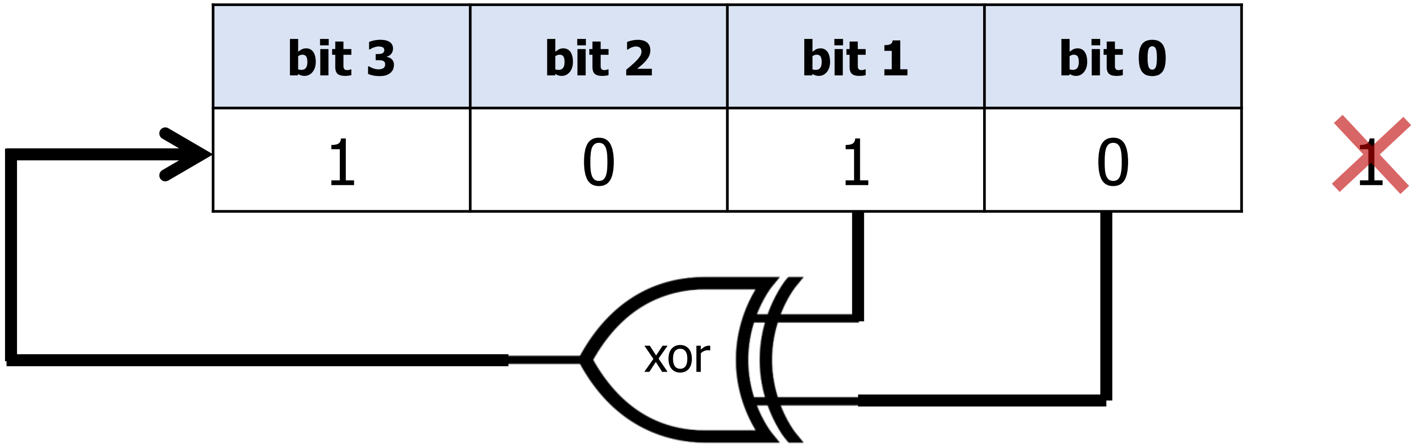
- LFSR takes in an input state and creates an output state
  - xors several bits together to create a new most-significant bit
  - Shifts all bits in state one to the right

Background: 4-bit LFSR example



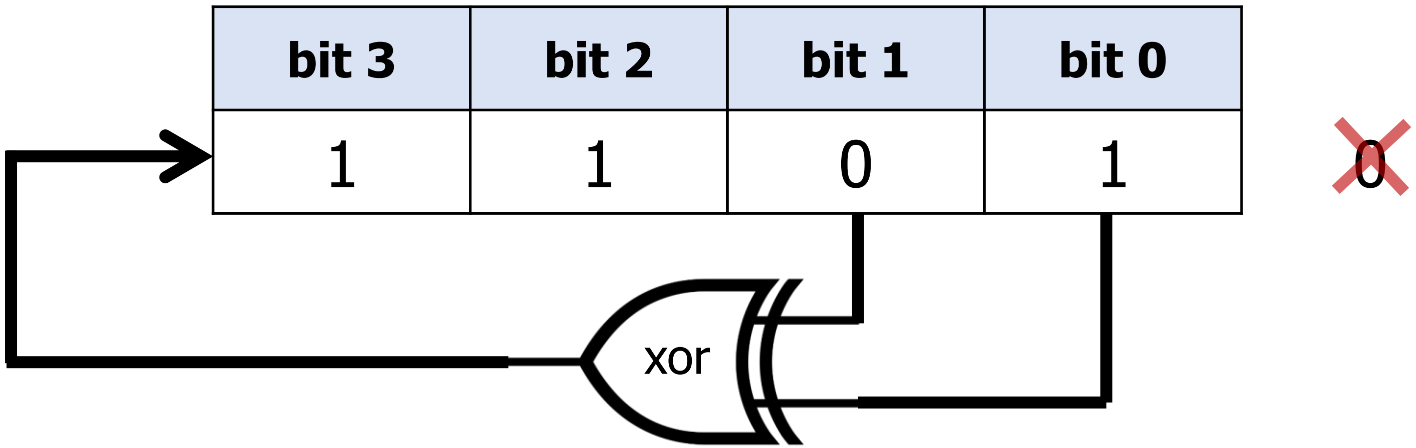
- Initial state: 0b0101

Background: 4-bit LFSR example, step 1



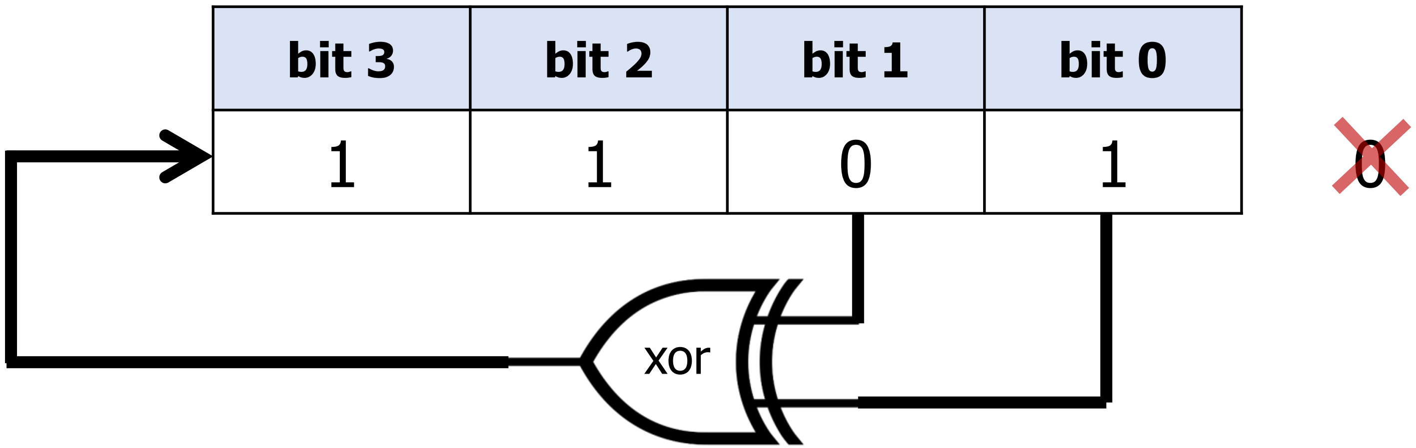
- Initial state: 0b0101
  - XOR of bits 0 and 1 = 1
  - Shift all bits right once, 0101 becomes 010
    - The former least-significant bit (1) is deleted
  - Set most-significant bit to xor result

Background: 4-bit LFSR example, step 2



- Initial state: 0b1010
  - XOR of bits 1 and 0 = 1
  - Shift all bits right once, 1010 becomes 101
    - The former least-significant bit (0) is deleted
  - Set most-significant bit to xor result

Background: 4-bit LFSR example, continued steps



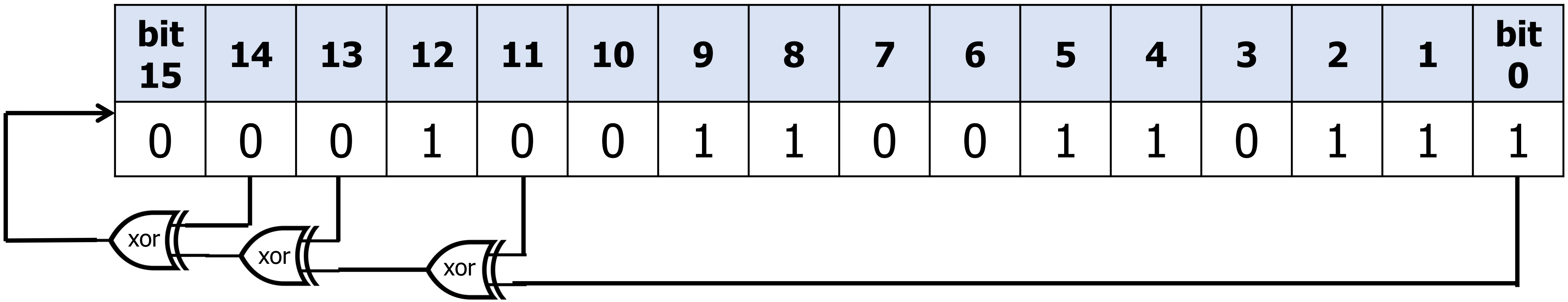
- Next states:
  - 0b1110, 0b1111, 0b0111, 0b0011, 0b0001, 0b1000, 0b0100, 0b0010, 0b1001, 0b1100, 0b0110, 0b1011, 0b0101 (←that's the initial state again)
  - Iterates through 15 total states before repeating
    - Never hits 0b0000 (it would stick there permanently)

Background: 4-bit LFSR example

- Still feel like LFSRs don’t make sense?
- Sometimes videos and animations can help!
- Here’s a great Youtube video explaining LFSRs: https://www.youtube.com/watch?v=1UCaZjdRC_c

Pack Lab LFSR design

- 16-bit LFSR
  - Accesses bits 0, 11, 13, and 14



- Example initial state: 0x1337
  - XOR of bits: 1
  - Next state: 0b1000100110011011 -> 0x899B

Testing your LFSR

- We’ve provided some code for you that can test your LFSR implementation
  - Within `test-utilities.c`

- Tests two things:
  1. Your LFSR must iterate in a known pattern
  2. Your LFSR must iterate over all 16-bit integers (except zero)

- If your implementation is not working, it can be annoying to debug
  - Suggestion: use paper to work out the bit pattern for input and output and compare to your code’s results

You write this - Function: `lfsr_step()`

- Determines the next LFSR state given an initial state input
- Return the new LFSR state

- Must not save state internally. To iterate through multiple LFSR states, call the function with the prior state
  - `new_state = lfsr_step(old_state);`

Decrypting data

- Once you've implemented the LFSR, you can use it to generate 16-bit pseudorandom numbers
  - Each newstate returned is used as the pseudorandom number
  - Never use the encryption key as a pseudorandom number, always LFSR step first

- To decrypt data:
  - Generate a new LFSR state based on the previous state
  - XOR the LFSR state against the next two bytes of data in **little-endian** order
  - Repeat this process for every two bytes in the `input_data`

- Example: `input_data`=[0x60, 0x5A, 0xFF, 0xB7]
  - First LFSR output is 0x899B and second LFSR output is 0x44CD
  - 0x9B ^ 0x60 = 0xFB and 0x89 ^ 0x5A = 0xD3
  - 0xCD ^ 0xFF = 0x32 and 0x44 ^ 0xB7 = 0xF3
  - `output_data` = [0xFB, 0xD3, 0x32, 0xF3]

Initializing the LFSR

- The initial state for the LFSR is the encryption key
  - Then each iteration after that, the state is the previous output

- The encryption key is a 16-bit unsigned integer formed by running the checksum operation on the user’s entered password

- Note: this is not very secure
  - There are many collisions where multiple passwords have the same value
  - Password “ab” checksums to the same value as password “ba”

Decryption edge case

- When decrypting, there may be an odd number of bytes in the input data!

- In that case, for the last byte of `input_data`, use the least-significant byte of the LFSR result, but not the most-significant byte

- Example: `input_data`=[0x21] and LFSR output 0x899B
  - 0x9B ^ 0x21 = 0xBA
  - `output_data` = [0xBA]
  - (do nothing with the most-significant byte from the LFSR)

You write this - Function: `decrypt_data()`

- Decrypts the `input_data` and writes result into `output_data`
  - `input_data` is an array of bytes, you can access it with [ ]
  - `input_data` only contains data to decrypt, it does not contain header bytes
  - `input_len` is the number of bytes in `input_data`
  - `output_data` is where you write decrypted bytes to
  - `output_len` is the maximum number of bytes you can write to `output_data`

- Use `lfsr_step()` to generate pseudorandom numbers for decryption
  - You will need a loop to iterate over bytes from `input_data`
  - The initial state for the LFSR should be the `encryption_key`
  - The output state from the LFSR is used as both a random number for decryption and as the input state for the LFSR in the next iteration

Outline

- Header Format
- Checksums
- Encryption
    - Linear-Feedback Shift Register
    - Stream Cipher
- **Compression**
    - **Compression dictionary**
    - **Escape sequences**
- Testing

How do you make a file smaller?

- Compression is the act of making a file smaller
  - Files can get really large, so it would be nice to make them smaller
  - Actually, all of your pictures, music, and videos are compressed already
    - Otherwise the files would be even larger!

- Lossless compression means the process can be undone (decompression) and the output will exactly match the original input
  - Lossy compression is the other option, which is sometimes done for media
  - For example: delete the parts of the audio file that humans can't hear (MP3)

- We're going to use **lossless** compression
  - So the unpacked file should *exactly* match the original input file

Lossless compression algorithms

- There has been a lot of engineering put into compression algorithms
- One really good algorithm comes up with new bit encodings for each byte based on usage: Huffman Encoding
  - It’s a little complex to implement though
- We will use a simpler algorithm: Run-length encoding

Run-length encoding concept

- Run-length encoding looks for repeated bytes and replaces them with an indication of how many times the byte repeated

- Conceptually: “aaaaabb” could turn into “five a’s and two b’s”
    - If there are enough repeated characters, this can save a lot of space!

- This kind of algorithm works really well on text files and raw image files

Pack Lab compression implementation

- We will use a version of run-length encoding where repeated bytes get replaced by a two-byte sequence
  - Specifies which byte and how many repeats
- However, not all repeated bytes get reduced, we only reduce the 16 most-popular bytes in the file
  - The header contains a dictionary with the 16 most-popular bytes

Compression dictionary

- 16-byte array of `uint8_t` (unsigned bytes)

- Bytes are arranged in index order and are zero-indexed (0–15)

Byte offset	**0**	**1**	**2**	**3**
0	Magic: 0x0213		Version: 0x01	Flags
4	Dictionary[0]	Dictionary[1]	Dictionary[2]	Dictionary[3]
8	Dictionary[4]	Dictionary[5]	Dictionary[6]	Dictionary[7]
12	Dictionary[8]	Dictionary[9]	Dictionary[10]	Dictionary[11]
16	Dictionary[12]	Dictionary[13]	Dictionary[14]	Dictionary[15]

Special byte sequence

- When packing a file, if one of the 16 bytes in the dictionary appears twice or more in-a-row, instead we replace up to 15 repetitions with a special two-byte sequence

- First byte: "escape byte"
  - Signifies that this is a special sequence, not normal data
  - Always 0x07 which is unlikely to be used in text files

- Second byte:
  - Information about which dictionary character and how many repetitions

Normal case: repeated character

- First byte signals that something special is happening
- Second byte contains a 4-bit unsigned "repeat count"
- Followed by a 4-bit unsigned "dictionary index"

- Example: 0x0737
  - Three repetitions of dictionary index 7

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

Special case: literal escape byte

- What if the file actually uses the byte value 0x07?

- Special case: if the first byte is 0x07 and the second byte is 0x00
  - Then the output should be a single byte: 0x07

- Any other pattern with a “repeat count” of zero is invalid
  - You don’t have to check for this

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

Decompression example

- Input data: {0x01, 0x07, 0x42}
- Dictionary: {0x30, 0x31, 0x32, 0x33 …} (didn't write the rest due to space)

- Resulting output data: {0x01, 0x32, 0x32, 0x32, 0x32}

- Explanation
  - First byte isn't special and just gets copied over
  - Second byte is the escape byte, which means the third byte holds a repeat count (4) and dictionary index (2)
    - So, the output should be four copies of `dictionary[2]` (0x32)

Implementation guide

- Iterate through bytes in the input
  - Either it's a normal byte
  - Or it's an escape character

- For normal bytes, just copy them over to the output
- For special bytes, read the second byte and determine what to do
  - Either multiple repetitions of a dictionary byte
  - Or a single literal escape byte

- Be careful to not go past the end of the input!
  - Check against lengths as you go
  - Special case: if the very last character is the escape byte, treat it as a normal byte and copy it over to the output

You write this - Function: `decompress_data()`

- Decompresses the `input_data` and writes the result into `output_data`
  - `input_data` is an array of bytes, you can access it with [ ]
  - `input_data` only contains data to decompress, it does not contain header bytes
  - `input_len` is the number of bytes in `input_data`
  - `output_data` is where you write decompressed bytes to
  - `output_len` is the maximum number of bytes you can write to `output_data`
  - `dictionary_data` is the compression dictionary used when compressing the data

- Return the total length of the decompressed data

Outline

- Header Format
- Checksums
- Encryption
  - Linear-Feedback Shift Register
  - Stream Cipher
- Compression
  - Compression dictionary
  - Escape sequences
- **Testing**

Testing overview

1. Write tests in `test-utilities.c`

2. You can pack your own files using the `pack` application

3. We have provided some example packed files, and their original versions, in the `example_files/` directory

4. There are some other useful tools you should know about for looking at files
   - `xxd` and `diff`

Testing your utility function implementations

- Each operation takes in an array of data
- You can craft your own array of unsigned 8-bit data and pass it into the function
- This is much easier than crafting full files files to unpack
- See the `example_test()` provided in `test-utilities.c` as an example for how you might make a test

Using the Pack application

```
./pack [-cek] inputfilename outputfilename
```

- -c: Optionally compresses the file
- -e: Optionally encrypts the (compressed) file with a password
- -k: Optionally checksums the (compressed & encrypted) file

- The three options can be combined in any way
  - -e: Encryption only
  - -ck: Compression and Checksum
  - -cek: Compression, Encryption, and Checksum
  - no flags: Add header, but perform no operations

Example packed files

- Original and packed versions of some files have been provided to you in the `example_files/` directory

- Each fits the pattern of filename.options.pack
  - Where options are
    - c – compress
    - e – encrypt
    - k – checksum

  - The password for any encrypted file is: cs213

Other useful tools – seeing raw hex values inside a file

- `xxd filename`
  - Dumps raw hex values of the file

- Format:
  - On the left are addresses starting at 0x00000000
  - In the middle are the hexadecimal values
  - On the right are the same values interpreted as an ASCII encoding
  - Example:

```
[brghena@ubuntu example_files] [main *] $ xxd long_file.txt
00000000: 6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaa
00000010: 3162 6262 6363 6364 6464 6565 6566 6666  1bbbcccdddeeefff
00000020: 6767 6768 6868 6969 696a 6a6a 6b6b 6b6c  ggghhhiiijjjkkkl
00000030: 6c6c 6d6d 6d6e 6e6e 6f6f 6f70 7070 7171  llmmmnnnooopppqq
00000040: 7172 7272 7373 7374 7474 7575 7576 7676  qrrrssstttuuuvvv
00000050: 7777 7778 7878 7979 797a 7a7a 0a         wwwxxxyyyzzz.
```

Ways to use xxd

- `xxd filename | head –2`
  - Only show the first ten lines of hexadecimal output for a file
  - Useful for looking at the header bytes of a file
- `xxd filename > filename.hex`
  - Convert a normal file into hexadecimal
- `xxd –r filename.hex > filename`
  - Convert hexadecimal back into a normal file
  - Can do this after editing some bytes in the hex to craft your own input

Other useful tools – checking for differences in files

- `diff filename1 filename2`
  - Checks for differences between two files
  - Doesn’t output anything if they match
  - Useful for determining if an unpacked file matches the original file

- If the two files do differ:
  - For text files, it will show you the text that’s different
  - For raw binary files, it will just say that they differ

  - To see the difference for binary files, convert both into .hex files and then diff those!