Assignment 5
Hamming Codes

Prof. Darrell Long
CSE 13S – Spring 2021

First `DESIGN.pdf` draft due: May 6th at 11:59 pm PST
Assignment due: May 9th at 11:59 pm PST

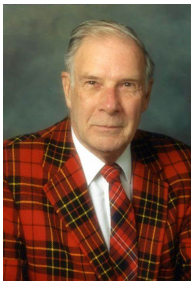
1 Introduction

As we know, the world is far from perfect. In the communications domain, this imperfection is called *noise*. Noise (unwanted random disturbances) makes it difficult to have a reliable signal. Thus, transferring data through a noisy communication channel is prone to errors. Noisy channels are omnipresent. They are present in mobile phone networks and even the wires in a circuit. To counteract noisy interference, we add extra information to our data. This extra information allows us to perform error checking, and request that the sender retransmit any data that was incorrect. We can also add extra information to not only detect errors but also correct them. This technique is called *forward error correction* (FEC). CDs, DVDs, and even hard drives use FEC to account for scratches or bad sectors. In fact, most of our digital world such as Netflix would not be possible without FEC.

2 Hamming Codes

> Hamming: *If you come to the Navy School, I will teach you how to be a great scientist.*
> Long: *And if I go to Santa Cruz, will you still teach me?*
> Hamming: *No.*
>
> Lunch with Richard Hamming

Richard Wesley Hamming was an American applied mathematician whose work had profound impact on computer engineering and telecommunications. His contributions include the Hamming code (which makes use of a Hamming matrix), the Hamming window, Hamming numbers, sphere-packing (or Hamming bound), and the Hamming distance. Hamming served as president of the Association for Computing Machinery from 1958 to 1960.


Richard Hamming

He used to joke that he was the *anti-Huffman*: David Huffman, the inventor of Huffman codes and a professor here at Santa Cruz, was a friend. Hamming added redundancy for reliability, while Huffman focused on removing it for efficiency.

In later life, Hamming became interested in teaching. From 1960 and 1976, before he left Bell Laboratories, he held visiting positions at Stanford University, Stevens Institute of Technology, the City College of New York, the University of California at Irvine and Princeton

University. After retiring from the Bell Laboratories in 1976, Hamming took a position at the Naval Postgraduate School in Monterey, California, where he worked as an adjunct professor and senior lecturer in computer science, and devoted himself to teaching and writing books.

He spent significant effort in trying to recruit a young Dr. Long, telling him that "If you come to the Navy School, I will teach you how to be a great scientist." Dr. Long replied, "If I go to Santa Cruz, will you still teach me?" To which Hamming replied, simply, "No." Sadly, it was an opportunity lost.

2.1 Bits, Nibbles, and Bytes

Around 1948, John Wilder Turkey, an American mathematician, coined the word *bit* to replace the mouthful that was *binary digit*. A bit is the basic unit of information for digital systems. It represents a logical state with only two possible values, either a 1 or 0, on or off, true or false. But one bit is rather limiting. As a result, multiple bits were packed together to make up a *nibble* (4 bits with 2^4 states) or a *byte* (8 bits with 2^8 states).

2.2 Overview

Hamming codes are a linear error-correcting code invented by Richard W. Hamming [1] to correct errors caused by punched card readers. But, we will be using them to correct errors caused by random bit-flips (noise). He introduced the Hamming(7,4) code that encodes 4 bits of data D in 7 bits by adding 3 redundant or parity P bits. An explanation of parity bits will be given through an example. This code can detect and correct *one* error. With 7 possible errors, 3 parity bits can identify which bit is incorrect ($2^3 - 1 = 7$).

The Hamming(7,4) code is shown in table 1 where the Hamming code's *least significant bit* (LSB) is at index 001_2 and the *most significant bit* (MSB) is in position 111_2. While normally in practice we begin counting at 0, the importance of starting the count at 1 and in binary will be evident in the next paragraphs.

Index	111_2	110_2	101_2	100_2	011_2	010_2	001_2
Hamming code	D_3	D_2	D_1	P_2	D_0	P_1	P_0

Table 1: Hamming(7,4) code with data bits D and parity bits P.

You might notice that each parity bit's index has only one bit set (a power of 2). P_0's index (001_2) has the 0^{th} bit set, and you might notice that the index for D_0, D_1, and D_3 also have the 0^{th} bit set. Thus, P_0 is calculated over D_0, D_1, and D_3, *i.e.*, P_0 is set if the data bits have an odd number of 1s. This is an *even* parity scheme. With an odd parity scheme, the parity bit is set if by setting it there are an odd number of 1s. We could use an odd parity scheme, but for this assignment, we will be using an even parity.

We can calculate P_1 and P_2 in the same way where P_i is calculated over the data bits whose index has the i^{th} bit set. The formulas for the three parity bits P_i are shown below.

$$P_0 = D_0 \oplus D_1 \oplus D_3$$
$$P_1 = D_0 \oplus D_2 \oplus D_3$$
$$P_2 = D_1 \oplus D_2 \oplus D_3$$

[1] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, April 1950.

As a result, each data bit has at least two parity bits that will help recover its value should an error occur. The overlap of parity bits also keeps them in check (parity bits can be erroneously flipped too). For example, the Hamming code for 0001_2 is shown in table 2. From here on, the index will follow convention and start at 0 and in decimal.

Index	6	5	4	3	2	1	0
Label	D_3	D_2	D_1	P_2	D_0	P_1	P_0
Hamming code	0	0	0	0	1	1	1

Table 2: Hamming(7,4) code for 0001_2.

The values of parity bits P_0, P_1, and P_2 are calculated as follows:

$$P_0 = D_0 \oplus D_1 \oplus D_3 = 1 \oplus 0 \oplus 0 = 1$$
$$P_1 = D_0 \oplus D_2 \oplus D_3 = 1 \oplus 0 \oplus 0 = 1$$
$$P_2 = D_1 \oplus D_2 \oplus D_3 = 0 \oplus 0 \oplus 0 = 0$$

We currently have a *non-systematic* code, a code where the parity bits are interspersed throughout the code. While this is fine for a hardware implementation, it is tedious to do in software. Instead, we will be using a *systematic* code, a code where the parity bits are placed after the data bits. We will also be extending the Hamming(7,4) code by adding one more parity bit to make this a Hamming(8,4) code. The extra parity bit, P_3, is calculated over P_0–P_2 and D_0–D_3. If P_3 for a received code is incorrect then we know an error has occurred. Either P_3 is incorrect or one of the the bits in the code is incorrect. This extension has two additional benefits. First, each code is a byte rather than seven bits. Second, we can now detect two errors but only correct one. The new Hamming code is shown below.

Index	7	6	5	4	3	2	1	0
Hamming code	P_3	P_2	P_1	P_0	D_3	D_2	D_1	D_0

Table 3: Hamming(8,4) systematic code, an extension of the Hamming(7,4) code.

where

$$P_0 = D_0 \oplus D_1 \oplus D_3$$
$$P_1 = D_0 \oplus D_2 \oplus D_3$$
$$P_2 = D_1 \oplus D_2 \oplus D_3$$
$$P_3 = D_0 \oplus D_1 \oplus D_2 \oplus D_3 \oplus P_0 \oplus P_1 \oplus P_2$$

2.3 Encoding

One approach to generating a Hamming code for a message is to calculate the parity bits one-by-one and appending them to the end of the message. Instead, we can use a generator matrix, $\boldsymbol{G}$. Given a message, $\vec{m}$, of four bits (a nibble) we can generate its hamming code, $\vec{c}$, by vector-matrix multiplication $\vec{c} = \vec{m}\boldsymbol{G}$ where the resulting code is eight bits in size (a byte). $\boldsymbol{G}$ is defined as:

$$\boldsymbol{G} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}.$$

Vector-matrix multiplication for a vector $\vec{y}$ of size n and matrix $\boldsymbol{A}$ of size $n \times n$ where $\vec{y} = \vec{x}\boldsymbol{A}$ is defined as:

$$y_i = \sum_{k=1}^{n} x_k \cdot A_{i,k}.$$

Those of you with exposure to linear algebra (helpful but not required) will notice the left half of $\boldsymbol{G}$ is the identity matrix I (1s along the diagonal). This ensures the first four bits of the Hamming code are the data bits while the right half is used to calculate the parity of message (notice the 0s along the diagonal). Generating the Hamming Code for $\vec{m} = \begin{pmatrix} 0 & 0 & 1 & 1 \end{pmatrix}$ (1100_2 in binary) is shown below. Note: binary is read from right to left with the LSB in the rightmost position and the MSB in the leftmost position. Vectors (arrays) are read from left to right.

$$\begin{aligned} \vec{c} &= \begin{matrix} 0 & 1 & 2 & 3 \end{matrix} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} \pmod 2 \\ &= \begin{pmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 1 & 1 \end{pmatrix} \pmod 2 \\ &= \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{pmatrix} \end{aligned}$$

In binary, $\vec{c}$ is equivalent to $1100\ 1100_2$.

Note: Addition and multiplication is mod 2 in binary. You might remember from CSE 12 that $\oplus$ (exclusive-or) is the summing function and can be used in lieu of addition, and $\wedge$ (logical and) can substitute multiplication. These operations have the added benefit of operating in mod 2.

$$\begin{aligned} a \oplus b &= a + b \pmod 2 \\ a \wedge b &= a \times b \pmod 2 \end{aligned}$$

2.4 Decoding

The process for decoding a Hamming code is similar to encoding a message as it uses a parity-checker matrix, $\boldsymbol{H}$, to identify any errors and recover the original message if an error occurred. To decode a message, the code is multiplied by the transpose of the parity-checker matrix, $\vec{e} = \vec{c}\boldsymbol{H}^\intercal$ where $\vec{e}$ is the *error syndrome*, $\vec{c}$ is the Hamming code, and $\boldsymbol{H}$ is defined as

$$\boldsymbol{H} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

However, $\vec{e} \neq \vec{m}$. Instead, $\vec{e}$, the *error syndrome*, is a pattern of bits that identifies the error if there is one. For example, if $\vec{e} = [0, 1, 1, 1]$, matching $\boldsymbol{H}^\intercal$'s first row, then we know the error lies in the 0^{th} bit and

flipping the 0^{th} bit in $\vec{c}$ will correct the error (remember the first four bits in the Hamming code are the data bits). If $\vec{e} = \mathbf{0}$ then our message does not contain an error. But, if the error syndrome is non-zero and the vector is not one of $\boldsymbol{H}^\top$'s rows then we cannot correct the error since more than one bit has been flipped.

For example, to decode $\vec{c}$ calculated earlier, we can do the following:

$$\vec{e} = \vec{c}\boldsymbol{H}^\top = \begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{matrix} \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \pmod 2$$

$$= \begin{pmatrix} 0 & 0 & 0 & 0 \end{pmatrix}.$$

Since $\vec{e} = \begin{pmatrix} 0 & 0 & 0 & 0 \end{pmatrix}$, we know our message arrived with no errors. Since this is a systematic code, $\vec{m} = \begin{pmatrix} 0 & 0 & 1 & 1 \end{pmatrix}$ (the first four elements in $\vec{c}$).

If the second bit had been flipped due to noise and we received $\vec{c} = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \end{pmatrix}$ instead, we can calculate $\vec{e}$ to identify the flipped bit.

$$\vec{e} = \vec{c}\boldsymbol{H}^\top = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \pmod 2$$

$$= \begin{pmatrix} 1 & 0 & 1 & 1 \end{pmatrix}.$$

Since $\vec{e} = \begin{pmatrix} 1 & 0 & 1 & 1 \end{pmatrix}$, which is $\boldsymbol{H}^\top$'s second row, we know the second element in $\vec{c}$ was erroneously flipped. Flipping the value of the second element gives us $\vec{c} = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{pmatrix}$. As a result, $\vec{m} = \begin{pmatrix} 0 & 0 & 1 & 1 \end{pmatrix}$ or 1100_2, the original message. One possible method to determine which bit to flip is to compare the error syndrome with each row in $\boldsymbol{H}^\top$, but section 5.3 will go over an optimal approach involving lookup tables.

3 Bit Vector

A ***bit vector*** is an ADT that represents a one dimensional array of bits, the bits in which are used to denote if something is true or false (1 or 0). This is an efficient ADT since, in order to represent the truth or falsity of a bit vector of n items, we can use $\lfloor n/8 \rfloor + 1$ `uint8_t`s instead of n, and being able to access 8 indices with a single integer access is extremely cost efficient. Since we cannot directly access a bit, we must use bitwise operations to get, set, and clear a bit within a byte.

```
1 struct BitVector {
2     uint32_t length; // Length in bits.
3     uint8_t *vector; // Array of bytes.
4 };
```

3.1 `BitVector *bv_create(uint32_t length)`

The constructor for a bit vector. In the event that sufficient memory cannot be allocated, the function must return `NULL`. Else, it must return a `(BitVector *)`, or a pointer to an allocated `BitVector`. Each bit of the bit vector should be initialized to 0.

3.2 `void bv_delete(BitVector **v)`

The destructor for a bit vector. Remember to set the pointer to `NULL` after the memory associated with the bit vector is freed.

3.3 `uint32_t bv_length(BitVector *v)`

Returns the length of a bit vector.

3.4 `void bv_set_bit(BitVector *v, uint32_t i)`

Sets the i^{th} bit in a bit vector. To set a bit in a bit vector, it is necessary to first locate the byte in which the i^{th} resides. The location of the byte is calculated as `i / 8`. The position of the bit in that byte is calculated as `i % 8`. Setting a bit in a bit vector should not modify any of the other bits.

3.5 `void bv_clr_bit(BitVector *v, uint32_t i)`

Clears the i^{th} bit in the bit vector.

3.6 `uint8_t bv_get_bit(BitVector *v, uint32_t i)`

Returns the i^{th} bit in the bit vector.

3.7 `void bv_xor_bit(BitVector *v, uint32_t i, uint8_t bit)`

`XOR`s the i^{th} bit in the bit vector with the value of the specified `bit`.

3.8 `void bv_print(BitVector *v)`

A debug function to print a bit vector. You will want to do this first in order to verify the correctness of your bit vector implementation.

4 Bit Matrix

In this assignment, we will define a bit matrix ADT, which will be an abstraction over a bit vector. That is, given an $m \times n$ bit matrix M, $0 \leq r < m$, and $0 \leq c < n$, then $M_{r,c}$ corresponds with index `r * n + c` in the underlying bit vector. Using this bit matrix abstraction will help when performing matrix multiplication mod 2 when encoding and decoding Hamming codes.

```
1 struct BitMatrix {
2     uint32_t rows;
3     uint32_t cols;
4     BitVector *vector;
5 };
```

4.1 `BitMatrix *bm_create(uint32_t rows, uint32_t cols)`

The constructor for a bit matrix. In the event that sufficient memory cannot be allocated, the function must return `NULL`. Else, it must return a `(BitMatrix *)`, or a pointer to an allocated `BitMatrix`. The number of bits in the bit matrix is calculated as `row` × `cols`. Each bit should be initialized to 0.

4.2 `void bm_delete(BitMatrix **m)`

The destructor for a bit matrix. Remember to set the pointer to `NULL` after the memory associated with the bit matrix is freed.

4.3 `uint32_t bm_rows(BitMatrix *m)`

Returns the number of rows in the bit matrix.

4.4 `uint32_t bm_cols(BitMatrix *m)`

Returns the number of columns in the bit matrix.

4.5 `void bm_set_bit(BitMatrix *m, uint32_t r, uint32_t c)`

Sets the bit at row `r` and column `c` in the bit matrix. Remember, given an $m \times n$ bit matrix M, $0 \leq r < m$, and $0 \leq c < n$, then $M_{r,c}$ corresponds with index `r * n + c` in the underlying bit vector. Setting a bit should not modify any of the other bits.

4.6 `void bm_clr_bit(BitMatrix *m, uint32_t r, uint32_t c)`

Clears the bit at row `r` and column `c`. Clearing a bit should not modify any of the other bits.

4.7 `uint8_t bm_get_bit(BitMatrix *m, uint32_t r, uint32_t c)`

Gets the bit at row `r` and column `c`. Getting a bit should not modify any of the other bits.

4.8 `BitMatrix *bm_from_data(uint8_t byte, uint32_t length)`

We will occasionally need to transform the first `length` number of bits in a byte into a bit matrix. The returned bit matrix should have the dimensions $1 \times$ `length`. The value of `length` should never be greater than 8, since we will only need to transform codes (8 bits) or nibbles of data (4 bits) into bit matrices.

4.9 `uint8_t bm_to_data(BitMatrix *m)`

Extracts the first 8 bits of a bit matrix, returning those bits as a `uint8_t`. Hint: since there are 8 bits in a byte, what would be the most efficient way of getting these bits?

4.10 `BitMatrix *bm_multiply(BitMatrix *A, BitMatrix *B)`

Performs a true matrix multiply, multiplying bit matrix `A` and bit matrix `B` mod 2. Returns a new bit matrix that holds the result of the multiplication. Remember, given an $m \times n$ matrix A and an $n \times p$ matrix B, the $m \times p$ matrix product C is defined as follows for $1 \leq i \leq m$ and $1 \leq j \leq p$:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \times B_{k,j}.$$

4.11 `void bm_print(BitMatrix *m)`

A debug function to print a bit matrix. You will want to do this first in order to verify the correctness of your bit matrix implementation.

5 Hamming Code Module

This module will implement the Hamming(8,4) code described in §2.2. The API is defined in `hamming.h`. As with all provided header files, you may not modify `hamming.h`. The implementation of the module should be defined in `hamming.c`.

5.1 `HAM_STATUS`

The module will contain enumerated status codes that reflect the different possible cases that can occur when decoding. `HAM_OK` will indicate that no corrections were needed when decoding. `HAM_CORRECT` will indicate that an error was detected and corrected when decoding. Lastly, `HAM_ERR` will indicate that more than error was detected and a code could not be properly decoded. These status codes are provided in `hamming.h`.

```
typedef enum HAM_STATUS {
    HAM_OK      = -3,  // No error detected.
    HAM_ERR     = -2,  // Uncorrectable.
    HAM_CORRECT = -1   // Detected error and corrected.
} HAM_STATUS;
```

5.2 `uint8_t ham_encode(BitMatrix *G, uint8_t msg)`

Generates a Hamming code given a nibble of data stored in the lower of nibble of `msg` and the generator matrix $\boldsymbol{G}$. Returns the generated Hamming code, which is stored as a byte, or a `uint8_t`.

5.3 `HAM_STATUS ham_decode(BitMatrix *Ht, uint8_t code, uint8_t *msg)`

Decodes the Hamming code using the transpose of the parity-checker matrix, $\boldsymbol{H}^\top$, and passes back the decoded data through the pointer `msg`. The decoded data bits should constitute the lower nibble. If no correction is needed when decoding `code`, return `HAM_OK`. If one error is detected and corrected, return `HAM_CORRECT`. In the event that more than one error is detected, return `HAM_ERR` to indicate an uncorrectable error without modifying `msg`.

To avoiding comparing the error syndrome with each row in $\boldsymbol{H}^\top$ until a match is found (or no match is found and the error cannot be corrected), we can refer to a lookup table. A lookup table is an array that contains precomputed information that is referred to often. By constructing a lookup table, we can avoid performing the same computation many times at the expense of storing the table in memory (in this case, storing 2^4 or 16 bytes is *negligible*). The index to the table will be the error syndrome and the value is the bit(start counting at 0) that needs to be flipped if the error can be corrected. Thus, if $\vec{e} = \begin{pmatrix}1 & 0 & 1 & 1\end{pmatrix}$ or 1101_2 (13_{10}), then `table[13]` = 1. If the error cannot be corrected then `table[`$\vec{e}$`] = HAM_ERR`.

Pre-lab Questions

1. Complete the rest of the look-up table shown below.

0	0
1	4
…	…
15	HAM_ERR

2. Decode the following codes. If it contains an error, show and explain how to correct it. Remember, it is possible for a code to be uncorrectable.
   (a) $1110\ 0011_2$
   (b) $1101\ 1000_2$

6 Your Task

You will be creating two programs for this assignment: an *encoder* and a *decoder*. The encoder will generate Hamming codes given input data and the decoder will decode the generated Hamming codes. Both will operate with the Hamming(8, 4) systematic code. The generator matrix $\boldsymbol{G}$ and the transpose of the parity-checker matrix, $\boldsymbol{H}^\top$, will be represented with bit matrices.

They must read from `stdin` by default or a file and write to `stdout` by default *or a file*. Thus, they should support command-line arguments, `-i` and `-o`, to specify input and/or output files. The decoder will also print statistics such as total bytes processed, uncorrected errors, corrected errors, and the error rate to `stderr`.

6.1 Encoder Command-line Options

Your encoder program, named `encode`, must support any combination of the following command-line options.

- `-h`: Prints out a help message describing the purpose of the program and the command-line options it accepts, exiting the program afterwards. Refer to the reference program in the resources repo for an idea of what to print.
- `-i infile`: Specify the input file path containing data to encode into Hamming codes. The default input should be set as `stdin`.
- `-o outfile`: Specify the output file path to write the encoded data (the Hamming codes) to. If not specified, the default output should be set as `stdout`.

6.2 Decoder Command-line Options

Your decoder program, named `decode`, will need to support the same command-line options that your encoder does, plus an option to trigger the verbose printing of statistics.

- `-h`: Prints out a help message describing the purpose of the program and the command-line options it accepts, exiting the program afterwards. Refer to the reference program in the resources repo for an idea of what to print.
- `-i infile`: Specify the input file path containing Hamming codes to decode. The default input should be set as `stdin`.
- `-o outfile`: Specify the output file path to write the decoded Hamming codes to. If not specified, the default output should be set as `stdout`.
- `-v`: Prints statistics of the decoding process to `stderr`. The statistics to print are the total bytes processed, uncorrected errors, corrected errors, and the error rate. The error rate is defined as (uncorrected errors/total bytes processed), the ratio of uncorrected errors to total bytes processed.

6.3 Example Program Output

```
$ ./encode -i frankenstein.txt | ./decode | diff - frankenstein.txt
Total bytes processed: 902092
Uncorrected errors: 0
Corrected errors: 0
Error rate: 0.000000
```

Figure 1: Example usage and decoder statistics. Here `diff` uses the dash to represent `stdin`.

We will be providing source code for a program (`error.c`) that will inject errors (noise) into your Hamming codes. Note that not all errors will be correctable. The rate at which the program injects errors is a command-line argument and is specified by `-e rate` (default is 0.01 or 1%) and must be between [0.0, 1.0]. The seed is specified with `-s seed` and it must be a positive integer.

```
$ ./encode < frankenstein.txt | ./error -e 0.002 -s 2021 | ./decode > /
   dev/null
Total bytes processed: 902092
Uncorrected errors: 89
Corrected errors: 14267
Error rate: 0.000099
```

Figure 2: Example usage with added noise.

6.4 `FILE *`

There is a lot of complicated logic at the hardware level to read and write actual bits of information. This is out of scope for a person wanting to write a software program. To make things simpler, the `C` standard library provides functions that abstracts much of this complicated logic. All of these functions are defined in `stdio.h`, and utilize the `FILE` object. The `FILE` object contains all the information about the file such as its name, file position indicator, file access mode (read/write), and other details necessary for I/O operations. Some functions of note for this assignment are `fopen()`, `fclose()`, `fgetc()`, and `fputc()`. Read the `man` pages for these functions to see what they do.

6.5 File I/O and Permissions

In the last assignment, graphs were read from either `stdin` or a file, and the final paths was printed to either `stdout` or a file. Thus, you should be familiar with file I/O. However, we must pay special attention to the input and output file's permissions when generating or decoding Hamming codes. For example, imagine you have a sensitive file on your UNIX machine that only you can read or write. Everyone else will receive an error if they attempt to read or write to it. The file containing the generated Hamming codes should also only be read or written by you, *i.e.*, it should inherit the file permissions of the file for which it is generating Hamming codes. This will prevent another user with prying eyes to decode the file with Hamming codes and learn the contents of your sensitive file (if it is *that* sensitive, it should be encrypted in the first place).

6.5.1 UNIX File Permissions

Every file in UNIX has a set of access modes for the owner, group, and everyone else. The `ls -l` command will lists all the files in a directory and also displays its permissions. In figure 3, the owner of the file, `bat-notes`, is batman, and the group is `dc`. Users in UNIX can be members of a group such as the `sudo` group. The `groups` command lists the groups the current user is in. Groups allow users to share files within their groups while controlling the extent of their access.

```
$ ls -l
  -rw-rw-r-- 1 batman dc 102 Jan 21 12:33 bat-notes
```

Figure 3: Output of `ls -l`

The file's permissions are presented in the same order from left to right after the first dash: owner, group, and others. Read permission is represented by `r`, write permission, the ability to modify, create, or delete a file, is represented by `w`, and execute permission for programs and shell scripts are represented by `x`. Thus, in figure 3, batman has the following permissions on his file, bat-notes: `rw-rw-r-`. The owner, `batman`, has read and write permissions but cannot execute the file. The group, `dc`, has read and write permissions but cannot execute the file. Everyone else can only read the file.

File permissions are not set in stone and can be modified by `chmod` (both a command and a system call). For example, if batman wants to remove read access to everyone else other than himself and members of the dc group, he can execute the command `chmod o-r bat-notes` to remove read permissions from others. If he changes his mind later, he can use the command `chmod o+r bat-notes` to add read permissions to others.

```
$ chmod o-r,g-w bat-notes
$ ls -l
  -rw-r---- 1 batman dc 102 Jan 21 12:33 bat-notes
```

Figure 4: Removing other's read permission and the group's write permission

As mentioned earlier, `chmod` is also a system call and can be called from a **C** program. For this assignment, all output files should inherit the file permissions of the input file. `fstat()` should be used to retrieve the input file's permissions and `fchmod()` to set the output file's permissions to match the input's. Refer to `man chmod, man fchmod, man fstat` for more information. Note: both of these functions expect a *file descriptor* that is returned by low-level IO (`open()`), so you will need to use `fileno()` to get the file descriptor of an open stream.

Using `fstat()` and `fchmod()`

```c
// Opening files to read from and write to.
FILE *infile = fopen("bat-notes", "rb");
FILE *outfile = fopen("bat-notes-encoded", "wb");

// Getting and setting file permissions
struct stat statbuf;
fstat(fileno(infile), &statbuf);
fchmod(fileno(outfile), statbuf.st_mode);
```

Another way to represent file permissions is with three *octal* digits, one for owner, group, and others. The octable table is shown in table 4. In fact, those with a keen eye will notice that file permissions are in fact a set. Thus, if a file has its permissions set to 755_8 then the owner has read, write, and execute permissions and the group and others only have read, execute permissions.

7 Packing and Unpacking Nibbles

Here are some functions to aid you in getting either the low or high nibble in a byte, as well as packing them together back into a byte.

Ref.	Octal	Binary
--x	1	001
-w-	2	010
-wx	3	011
r--	4	100
r-x	5	101
rw-	6	110
rwx	7	111

Table 4: Octal and binary representation of file permissions.

Helper functions

```
// Returns the lower nibble of val
uint8_t lower_nibble(uint8_t val) {
    return val & 0xF;
}

// Returns the upper nibble of val
uint8_t upper_nibble(uint8_t val) {
    return val >> 4;
}

// Packs two nibbles into a byte
uint8_t pack_byte(uint8_t upper, uint8_t lower) {
    return (upper << 4) | (lower & 0xF);
}
```

8 Encoder Program Specifics

In `main()` do the following:

1. Parse the command-line options with `getopt()` and open any input and/or output files using `fopen()` and correct file permissions.
2. Initialize the generator matrix $\boldsymbol{G}$ using `bm_create()`.
3. Read a byte from the specified file stream or `stdin` with `fgetc()`.
4. Generate the Hamming(8,4) codes for both the upper and lower nibble with `ham_encode()` and write to the specified file or `stdout` with `fputc()`. The Hamming code for the lower nibble should be written first followed by the code for the upper nibble. Note: You'll notice this operation becomes repetitive with larger files. If only there was a lookup table of Hamming codes to refer to.
5. Repeat steps 3 – 4 until all data has been read from the file or `stdout`.

6. Close both the input and output files with `fclose()` and make sure that any memory allocated is freed.

9 Decoder Program Specifics

1. Parse the command-line options with `getopt()` and open any input and/or output files using `fopen()` and correct file permissions.
2. Initialize the transpose of the parity-checker matrix, $\boldsymbol{H}^{\mathsf{T}}$, using `bm_create()`.
3. Read *two* bytes from the specified file stream or `stdin` with `fgetc()`. Note: The first byte read is the Hamming code for the lower nibble, and the second is the upper nibble.
4. For each byte pair read, decode the Hamming(8,4) codes for both with `ham_decode()` to recover the original upper and lower nibbles of the message. Then, reconstruct the original byte. Note: The return code from `ham_decode()` should be used for statistics. Your program should count the number of bytes proccessed, Hamming codes that required correction, and Hamming codes that could not be corrected.
5. Write the reconstructed byte with `fputc()`. Note: You should still write the byte even for Hamming codes that could not be corrected, leaving the data bits in the byte as is.
6. Repeat steps 3 – 5 until all data has been read from the file or `stdout`.
7. Print the following statistics to `stderr` with `fprintf()`:
   - Total bytes processed: The number of bytes read by the decoder.
   - Uncorrected errors: The number of Hamming codes that could not be corrected.
   - Corrected errors: The number of Hamming codes that experienced an error that was recoverable.
   - Error rate: The rate of uncorrected errors for a given input. Refer to §6.2 for the calculation.
8. Close both the input and output files with `fclose()` and make sure that any memory allocated is freed.

Remember: In *both* the encoder and decoder, if an input and output file are specified, the output file should have the same file permissions as the input file. You can use `fstat()` to retrieve an open file's permissions, `fchmod()` to change the permissions of the output file to match the input's. Since both of these functions expect a *file descriptor* that is returned by low-level IO (`open()`), you will need to use `fileno()` to get the file descriptor of an open stream.

10 Deliverables

You will need to turn in:

1. `encode.c`: This file will contain your implementation of the Hamming Code encoder.

2. `decode.c`: This file will contain your implementation of the Hamming Code decoder.

3. `error.c`: This file will be provided in the resources repo, but should be included in your repo as well. You *must not* modify this file.

4. `entropy.c`: This file will be provided in the resources repo, but should be included in your repo as well. You *must not* modify this file.

5. `bv.h`: This fill will contain the bit vector ADT interface. This file will be provided. You *may not* modify this file.

6. `bv.c`: This file will contain your implementation of the bit vector ADT. You *must* define the bit vector `struct` in this file.

7. `bm.h`: This file will contain the bit matrix ADT interface. This file will be provided. You *may not* modify this file.

8. `bm.c`: This file will contain your implementation of the bit matrix ADT. You *must* define the bit matrix `struct` in this file.

9. `hamming.h`: This file will contain the interface of the Hamming Code module. This file will be provided. You *may not* modify this file.

10. `hamming.c`: This file will contain your implementation of the Hamming Code module.

11. `Makefile`: This is a file that will allow the grader to type `make` to compile your programs.

    - `CC=clang` must be specified.
    - `CFLAGS=-Wall -Wextra -Werror -Wpedantic` must be included.
    - `make` should build the encoder, the decoder, the supplied error-injection program, *and* the supplied entropy-measure program, as should `make all`.
    - `make encode` should build *just* the encoder.
    - `make decode` should build *just* the decoder.
    - `make error` should build *just* the supplied error-injection program.
    - `make entropy` should build *just* the supplied entropy-measure program.
    - `make clean` must remove all files that are compiler generated.
    - `make format` should format all your source code, including the header files.

12. Your code must pass `scan-build` *cleanly*. If there are any bugs or errors that are false positives, document them and explain why they are false positives in your `README.md`.

13. `README.md`: This must be in *Markdown*. This must describe how to build and run your program.

14. `DESIGN.pdf`: This *must* be a PDF. The design document should answer the pre-lab questions, describe the purpose of your program, and communicate its overall design with enough detail such that a sufficiently knowledgeable programmer would be able to replicate your implementation. This does not mean copying your entire program in verbatim. You should instead describe how your program works with supporting pseudocode. **C** code is **not** considered pseudocode.

15. `WRITEUP.pdf`: This document *must* be a PDF. You will need to use the `entropy.c` program that will be supplied to you. The program reads data from `stdin` and prints out the amount of entropy in the read data. Entropy, as defined by Claude Shannon, quantifies the amount of information that is produced by some process. This will be discussed further in lecture. Your writeup must include the following:

    - Graphs showing the amount of entropy of data before and after encoding it.
    - Analysis of the graphs you produce.

11 Submission

To submit your assignment through `git`, refer to the steps shown in `asgn0` Remember: *add, commit,* and *push*! Your assignment is turned in *only* after you have pushed and submitted the commit ID to Canvas. Your design document is turned in *only* after you have pushed and submitted the commit ID to Canvas. If you forget to push, you have not turned in your assignment and you will get a *zero*. "I forgot to push" is not a valid excuse. It is *highly* recommended to commit and push your changes *often*.

12 Supplemental Readings

> *The more that you read, the more things you will know. The more that you learn, the more places you'll go.*
>
> —Dr. Seuss

- *The C Programming Language* by Kernighan & Ritchie
    - Chapter 2 §2.9
    - Chapter 5 §5.7
    - Chapter 7

`Wrasznstizs qa I ps xusr mpvuzo n svnwqg n ioauzanc.`