Fabrice Kurmann
fkurmann@ucsc.edu
3 May, 2021

CSE 13S Spring 2021
Assignment 5: Hamming Codes
Design Document

**Prelab questions on page 5

Program Flow:

Encoder: Generate Hamming codes given input.

Accept command line options:

-h - print out program info message
-i - specify an infile other than stdin
-o - specify an outfile for the encoded codes only other than stdout.

Decoder: Decode the Hamming codes made by the encoder.

Print statistics (bytes processed, uncorrected and corrected errors, error rate decimal to 6 digits of precision) to stderr.

Accept command line options:

-h - print out program info message
-i - specify an infile other than stdin
-o - specify an outfile for the decoded codes other than stdout.
-v - enables printing statistics to stderr

Matrices to store:

Generator matrix (G)
Transpose of parity checker matrix, (H^T)

Pseudocode for creating Generator matrix: (Encoder.c)

```
For i in range 0, 4
    set bit matrix i, i // make the diagonal line of 1s
For i in range 4, 8
    for i in range 0, 4
        if i!= j+4
            set bit matrix j, i // set all second half items to one except for the center diagonal
```

Pseudocode for creating Transpose Matrix: (Decoder.c)

```
For i in range 0, 4
    set bit matrix (i+4, i) // make the diagonal line of 1s
For i in range 0, 4
    for j in range 0, 4
        if j != i
            set bit matrix(i, j) // set ones in every place except the diagonal
```

Arrays/vectors to store:

Error reference table
Hamming code vector for input
Decoded vector for output

The Bit Vector ADT:

Represents a one dimensional array of bits.

For n items, use (n/8) + 1 uint 8s

Bitwise operations are required to handle the individual bits.

ADT variables:

Length, store as a uint32_t

Vector, store as an array of uint8_t

Functions to implement:

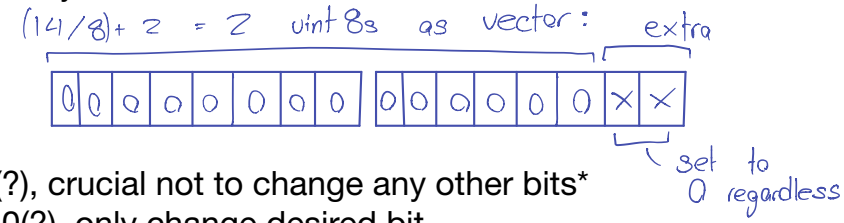
bv_create - return pointer to a bit vector, each bit is initialized to zero.

To initialize vector to zero, it's byte contents must all be zero.

bv_delete - destructor, free memory

bv_length - return length in bits, not bytes

Bit Vector of Length 14 bits:



bv_set_bit - set a specific bit to 1(?), crucial not to change any other bits*

bv_clear_bit - set a specific bit to 0(?), only change desired bit

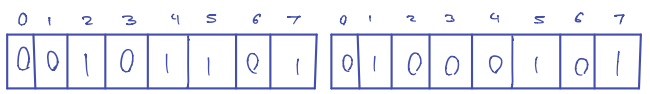
bv_get_bit - get a specific bit's value

bv_xor_bit - XOR comparison of bv element i and given binary value

* changing specific bit: byte = index / 8

place in byte = index % 8

get bit 13: =



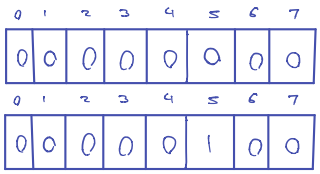
byte 13/8 = 1. (truncate) → byte 1 (0 is first)

bit = 13 % 8 = 5 → points to 14th bit (at index 13)

Modify vector [byte_value] (bit value)

Breaking uint 8 into it's own array:

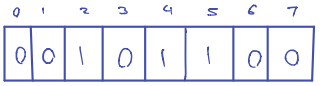
↳ setting a bit done by changing value of UINT.



Change of bv(13) means setting 2nd UINT to original value + 2^5; 32.

---

* accessing specific bit



= 00110100₂ = 4 + 16 + 32 = 52 = uint value

To access bit 3: 52 → USE Logical shifts until 3 is bit 0

- if uint = odd: value = 1

Logical shifts $00110100_2 = 52$

$00011010_2 = 2+8+16 = 26$

$00001101_2 = 1+4+8 = 13$

$00000110_2 = 2+4 = 6$ even, bit is 0

else : value = 0

Logical shifts = using integer division on unit. - 3 times

---

* Using addition to conduct bitwise XOR: XOR 1 with bit 6

$00110100_2 = 52$

$01000000_2 = 2^6 + 52 = (116/2^6) = 1.8 = 1$ so True

- Add UINT to byte in vector (set by bit value

If SUM /2 bit value in byte = 1 return true, else false

bv_print - print the bits, separated by spaces, as they are sorted in vector array.
(Bit vector is read *left to right)* Bit 0 is leftmost, the lsb, unlike standard reading.

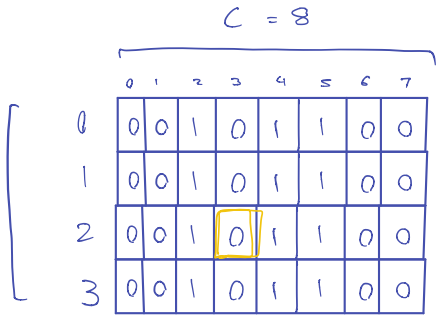
L 7 shifts - check even/odd each time, print results one by one.

Bit Matrix ADT:

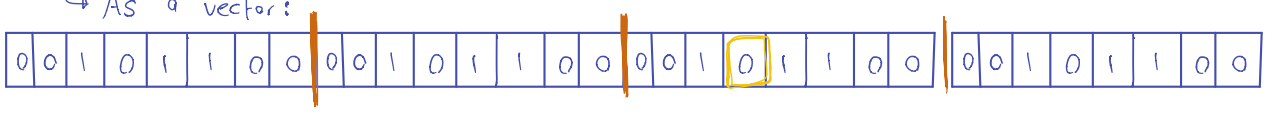
Represents a two dimensional array of bits - or an array of bit vectors.*

M*N matrix where M(r,c) = r*N + C

ADT variables:
Rows, store as a uint32_t
Columns, store as a uint32_t
vector, store as a bitVector.



As a vector:



$M_{2,3} = 2 \cdot 8 + 3 = 16 + 3 = 19$

Functions to implement:

bm_create - return pointer to a bit matrix, number of bits = rows*cols
To initialize vector to zero, it's byte contents must all be zero.
bm_delete - destructor, free memory
bm_rows - return num rows
bm_cols - return num cols

bm_set_bit - set a specific bit to 1(?), crucial not to change any other bits*
bm_clear_bit - set a specific bit to 0(?), only change desired bit
bm_get_bit - get a specific bit's value
These 3 functions will simply call the bit vector functions for the correct bit, showcased

above.

*bm_from_data - make a 1 row bit matrix out of the first LENGTH number of bits in a matrix
bm_to_data - return the first 8 bits of a bit matrix as a uint8_t
Simply return the uint8_t in position 0 of the bit vector.

*bm_multiply - return multiply two bit matrices using matrix multiplication, return a new BM with the result.
NOTE, you multiply matrix A by matrix B MOD2!.

bm_print - print the bits, separated by spaces, as they are sorted in vector array.

The Hamming Code Module:

HAM_STATUS contains three outcomes:
HAM_OK = -3 (no errors)
HAM_ERR = -2 (Error not fixable)
HAM_CORRECT = -1 (Error fixed)

Ham functions:
ham_encode - Given a nibble from a bitMatrix, adds the Hamming bits to return a fill byte (uint8_t) of Hamming code.

ham_decode - Return a nibble (decoded from a coded byte of Ham Code) to the beginning of a specified bitMatrix. Then also return the HAM STATUS from the decoding process.

Systemic Hamming Code : (8, 4)

Note, in the bit vector these bits are reflected.

$P_0 = \text{xor}(D_0\ D_1\ D_3)$
$P_1 = \text{xor}(D_0\ D_2\ D_3)$
$P_2 = \text{xor}(D_1\ D_2\ D_3)$
$P_3 = \text{xor}(\text{all})$

Index & Hamming code

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

Example for 6:

0	0	1	1	0	1	1	0

According to rules → Nibble = 6, 1 →

Example for 1

0	0	1	1	0	0	0	1

Prelab Questions:

1) Lookup table:

Index (decimal of error syndrome	Bit that must be flipped
(0000_2) 0	No Error found
(0001_2) 1	4
(0010_2) 2	5

(0011_2)	3	Ham Error
(0100_2)	4	6
(0101_2)	5	Ham Error
(0110_2)	6	Ham Error
(0111_2)	7	3
(1000_2)	8	7
(1001_2)	9	Ham Error
(1010_2)	10	Ham Error
(1011_2)	11	2
(1100_2)	12	Ham Error
(1101_2)	13	1
(1110_2)	14	0
(1111_2)	15	Ham Error

2) Error decoding:

1) $(11000111) \cdot H^T = [1\ 2\ 3\ 3] \% 2 = [1\ 0\ 1\ 1]$

$1101_2 = 13_{10}$ = Swap bit at 1

output = 1100 → [1011]

2) $(00011011) \cdot H^T = [2\ 1\ 2\ 1] \% 2 = [0\ 1\ 0\ 1]$

$1010_2 = 10_{10}$ = HAM ERROR

output = unknown.

Pseudocode for Bit Matrix functions:

Bm from data:

```
bm create output (1, length)
for i in range 0, length
    if input % 2 == 1
        output.set bit (i)
    srl byte // Same method as get bit method uses, shift and check ones place for 1 or 0 bit
```

Bm to data:

```
int output = 0
for i in range 0, 8
    output += 2^i * (get bit (input, i)
```

```
Bm multiply (matrix A, matrix B)
    for columns in matrix B
        for rows in matrix A
            for item in row(A) or col(B)
                sum += row(a, i) *(col b, i)
            sum = sum % 2

            output matrix row, col = sum
```

Pseudocode for hamming functions:

```
Ham encode:
    return bm_to_data(bm_multiply (bm_from_data(message nibble, 4), generator))
    // Nibble to a bit matrix, multiply with generator, turn into uint and return.

Ham decode:
    error = bm_to_data(bm_multiply(bm_from_data(input code byte, 8), Transpose))
    // Byte to matrix, multiply with transpose, turn into uint

    // Error Handling
    if error == lookup table (ham ok)
        return lower nibble (input code)
    if error == lookup table (ham err)
        errors ++
        return lower nibble (input code)
    if error == lookup table (ham correct)
        swap bit (lower_nibble(input code), lookup_table(error))
        return lower nibble)
```