Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
Department of Mechanical Engineering

6.050J/2.110J **Information and Entropy** **Spring 2003**

Problem Set 3 Solutions

Solution to Problem 1: My Dog Ate My Codebook

Solution to Problem 1, part a.

There are many possible codebooks, and a few of them are enumerated in Table 3–4.

Codebook 1					Codebook 2				
0	0	1	0	0	0	0	1	1	1
0	1	1	1	1	0	1	1	0	0
1	0	0	1	0	1	0	0	1	0
1	1	0	0	1	1	0	0	0	1

Codebook 3					Codebook 4				
0	0	1	0	1	0	0	1	0	1
0	1	1	1	0	0	1	1	1	0
1	0	0	1	0	1	0	0	1	1
1	1	0	0	1	1	0	0	0	0

Codebook 5					Codebook 6				
0	0	1	1	0	0	0	1	0	0
0	1	1	0	1	0	1	1	1	1
1	0	0	1	1	1	0	0	1	1
1	1	0	0	0	1	0	0	0	0

Table 3–4: Possible Codebook Implementations

Solution to Problem 1, part b.

There are 32 (2^5) possible bit strings. However, only four of those are allowed at any one time, one for each input.

Solution to Problem 1, part c.

For each code if one bit is in error, that produces up to five different codes. Since we have four legal codes, this gives us twenty codes that we detect as errors and can correct to their correct codes.

Solution to Problem 1, part d.

The number of uncorrectable codes is the total number of codes minus the legal codes minus the correctable codes, or 32 minus 4 minus 20, or eight.

Solution to Problem 2: A Different Hamming Code

Solution to Problem 2, part a.

```
>> G = [1 0 0 0 1 0 1; 0 1 0 0 1 1 0; 0 0 1 0 1 1 1; 0 0 0 1 0 1 1];
>> H = [ 1 1 1 0 1 0 0; 0 1 1 1 0 1 0; 1 0 1 1 0 0 1];

>> PATTERN = [0 0 0 0; 0 0 0 1; 0 0 1 0; 0 0 1 1; 0 1 0 0; 0 1 0 1; 0 1 1 0; 0 1 1 1;
              1 0 0 0; 1 0 0 1; 1 0 1 0; 1 0 1 1; 1 1 0 0; 1 1 0 1; 1 1 1 0; 1 1 1 1];
```

Solution to Problem 2, part b.

```
>> CODEBOOK = mod(PATTERN*G, 2)
 0   0   0   0   0   0   0
 0   0   0   1   0   1   1
 0   0   1   0   1   1   1
 0   0   1   1   1   0   0
 0   1   0   0   1   1   0
 0   1   0   1   1   0   1
 0   1   1   0   0   0   1
 0   1   1   1   0   1   0
 1   0   0   0   1   0   1
 1   0   0   1   1   1   0
 1   0   1   0   0   1   0
 1   0   1   1   0   0   1
 1   1   0   0   0   1   1
 1   1   0   1   0   0   0
 1   1   1   0   1   0   0
 1   1   1   1   1   1   1
```

Solution to Problem 2, part c.

This code can detect and correct one error per codeword, or detect (only) two errors per codeword (the decoder can be designed to do either the former or the latter but not both). This is a property of all block codes with minimum Hamming distance of three.

Solution to Problem 2, part d.

Since no errors are introduced, the three CHECK variables should be all zero.

1.
```
>> INPUT1 = [0 1 0 0];
>> CODE1 = mod(INPUT1*G, 2)
CODE1 =
0 1 0 0 1 1 0

>> % No error introduced
>> CHECK1 = mod(CODE1*H', 2)
CHECK1 =
0 0 0
```

```
>> OUTPUT1 = CODE1(1:4)
OUTPUT1 =
0 1 0 0
```

2.
```
>> INPUT2 = [1 1 0 0];
>> CODE2 = mod(INPUT2*G, 2)
CODE2 =
1 1 0 0 0 1 1

>> % No error introduced
>> CHECK2 = mod(CODE2*H', 2)
CHECK2 =
0 0 0

>> OUTPUT2 = CODE2(1:4)
OUTPUT2 =
1 1 0 0
```

3.
```
>> INPUT3 = [1 0 0 1];
>> CODE3 = mod(INPUT3*G, 2)
CODE3 =
1 0 0 1 1 1 0

>> % No error introduced
>> CHECK3 = mod(CODE3*H', 2)
CHECK3 =
0 0 0

>> OUTPUT3 = CODE3(1:4)
OUTPUT3 =
1 0 0 1
```

Solution to Problem 2, part e.

Same input values but now errors are introduced.

1.
```
>> % Error in position 3 in CODE1
>> CODE4 = CODE1
CODE4 =
0 1 0 0 1 1 0
>> CODE4(3) = ~CODE4(3)

CODE4 =
0 1 1 0 1 1 0

>> CHECK4 = mod(CODE4*H', 2)

CHECK4 =
1 1 1
```

```
>> >> CODE4(3) = ~CODE4(3)

CODE4 =
0 1 0 0 1 1 0

>> OUTPUT4 = CODE4(1:4)

OUTPUT4 =
0 1 0 0
```

2.
```
>> % Error in position 4 in CODE2
>> CODE5 = CODE2
CODE5 =
1 1 0 0 0 1 1

>> CODE5(4) = ~CODE5(4)

CODE5 =
1 1 0 1 0 1 1

>> CHECK5 = mod(CODE5*H', 2)

CHECK5 =
0 1 1

>> % Repair damage by flipping bit
>> CODE5(4) = ~CODE5(4)

CODE5 =
1 1 0 0 0 1 1

>> OUTPUT5 = CODE5(1:4)

OUTPUT5 =
1 1 0 0
```

3.
```
>> CODE3
>> % Error in position 5 (a parity bit)
CODE6 =
1 0 0 1 1 1 0

>> CODE6(5) = ~CODE6(5)

CODE6 =
1 0 0 1 0 1 0

>> CHECK6 = mod(CODE6*H', 2)

CHECK6 =
1 0 0
```

```
>> % Correction is optional since data bits are all OK >> CODE6(5) = ~CODE6(5)

CODE6 =
1 0 0 1 0 0 1

>> OUTPUT6 = CODE6(1:4)

OUTPUT6 =
1 0 0 1
```

Solution to Problem 2, part f.

This code cannot correct or even detect double errors. Instead, it interprets the symptoms as a single error and changes some bit that is probably OK. (If the only tool you have is a hammer, everything tends to look like a nail.)

```
>> INPUT7 = [0 1 0 0];
>> CODE7 = mod(INPUT7*G, 2)
CODE7 =
0 1 0 0 1 1 0

>> % Two errors, in positions 3 and 7
>> CODE7(3) = ~CODE7(3);
>> CODE7(7) = ~CODE7(7);
>> CODE7
CODE7 =
0 1 1 0 1 1 1

>> CHECK7 = mod(CODE7*H', 2)
CHECK7 =
1 1 0

>> % Incorrectly concludes there is an error in the second bit
>> CODE7(2) = ~CODE7(2)
CODE7 =
0 0 1 0 1 1 1

>> OUTPUT7 = CODE7(1:4)
OUTPUT7 =
0 0 1 0
```

Solution to Problem 3: Ben's Best Bet

Solution to Problem 3, part a.

Ben's efficiency is indeed higher. His code rate is $8/16 = 0.5$. Your code rate is $8/18 = 0.44$.

Error Bit	Failed Checks
D0	PC0, P0
D1	PC1, P0
D2	PC0, PR1
D3	PC1, PR1
PR1	PR1, P0
PC0	PC0
PC1	PC1
P0	P0

Table 3–5: Failed checks on a single error in one block

Parity Bit	Parity Checks
PR1	D2 $\oplus$ D3
PC0	D0 $\oplus$ D2
PC1	D1 $\oplus$ D3
P0	PR1 $\oplus$ D0 $\oplus$ D1

Table 3–6: Definition of Parity Checks

Solution to Problem 3, part b.

Ben's design can correct all single errors. To see this, consider Table 3–5, which enumerates the possible places a single error can occur. (Note that it is sufficient to consider only a single block.) In this table, we assume that we calculate a 'dummy' row parity bit from the D0 and D1, and use this dummy row parity bit to check the total parity bit. We can see here that we can distinguish all errors from each other. This can also be deduced by looking at a definition of the parity bits as shown in Table 3–6.

Solution to Problem 3, part c.

Yes, Ben's code can correct some double errors - consider for instance single errors in each block. This is a double error and can be corrected.

Solution to Problem 3, part d.

Ben's design does not detect all double errors and identify them as such. Consider the case where PC0 and P0 receive errors. From the parity checks that the decoder can perform, this case cannot be distinguished from a single error occurring in D0. In this case Ben's design not only fails to detect a double error, but it makes matters even worse by introducing a new error.