

Issued: [February 14, 2005](#)

Problem Set 4 Solutions

Due: February 18, 2005

Solution to **Problem 1: Vegetables**

Solution to Problem 1, part a.

$$p0 = d1 \oplus d2$$

$$p1 = d2 \oplus d0$$

$$p2 = d0 \oplus d1$$

Solution to Problem 1, part b.

Table 4-1 shows the codebook corresponding to the triangular code.

d0	d1	d2	p0	p1	p2	Vegetable
0	0	0	0	0	0	Asparagus
0	0	1	1	1	0	Broccoli
0	1	0	1	0	1	Corn
0	1	1	0	1	1	Lima Beans
1	0	0	0	1	1	Peas
1	0	1	1	0	1	Spinach
1	1	0	1	1	0	Tomatoes
1	1	1	0	0	0	Zucchini

Table 4-1: Vegetable codeword table

Solution to Problem 1, part c.

In this case, the minimum Hamming distance between any 2 codes is 3. And so is the minimum Hamming distance required for single-error correction.

Solution to Problem 1, part d.

The tests to be performed are:

$$t_0 = d1 \oplus p0 \oplus d2 = 0$$

$$t_1 = d2 \oplus p1 \oplus d0 = 0$$

$$t_2 = d0 \oplus p2 \oplus d1 = 0$$

If any single bit from the message was flipped, two parity tests will fail (the two that contain that bit). Instead if a parity bit was flipped, only one test will fail. It suffices then to check which have failed following the procedure outlined in Figure 4-1.

```

L1 if   t0 · t1 ⇒ d2 =~ d
L2 elseif t1 · t2 ⇒ d0 =~ d0
L3 elseif t2 · t0 ⇒ d1 =~ d1
L4 elseif t0 ⇒ p0 =~ p0
L5 elseif t1 ⇒ p1 =~ p1
L6 elseif t2 ⇒ p2 =~ p2
L7 else   bitstring correct

```

Figure 4-1: An algorithm that *corrects single-errors* for the trangular code

Solution to Problem 1, part e.

1. **no error:** consider an order for **peas**, with code [100011] according to Table 4-1. The procedure outlined above will first find the value of t_0 , t_1 and t_2 :

$$\begin{aligned}
 t_0 &= 0 \oplus 0 \oplus 0 \rightarrow \mathbf{0} \\
 t_1 &= 0 \oplus 1 \oplus 1 \rightarrow \mathbf{0} \\
 t_2 &= 1 \oplus 1 \oplus 0 \rightarrow \mathbf{0},
 \end{aligned}$$

then it will follow the procedure from Figure 4-1, but all of the conditions yield a 0, so the code will land in the “catch-all” else clause at line 7 and will conclude **bitstring correct**.

2. **error in data bit:** consider an order for **peas**, with code [100011]. Let us introduce an error in the 3rd bit

$$[100011] \xrightarrow{\text{error}} [101011].$$

Without the parity bits, the system would have ordered **spinach** instead of **peas**. Upon computing t_0 , t_1 and t_2 :

$$\begin{aligned}
 t_0 &= 0 \oplus 0 \oplus 1 \rightarrow \mathbf{1} \\
 t_1 &= 1 \oplus 1 \oplus 1 \rightarrow \mathbf{1} \\
 t_2 &= 1 \oplus 1 \oplus 0 \rightarrow \mathbf{0},
 \end{aligned}$$

our algorithm will find the condition in line 1 to be true, and as a consequence, will flip the value of bit d₂.

$$[101011] \xrightarrow{L1} [100011].$$

The final code will look like [100011]. That is: **peas**.

3. **error in parity bit:** consider again an order for **peas**, with code [100011]. Let us introduce an error in the 6th bit

$$[100011] \xrightarrow{\text{error}} [100010].$$

Without the parity bits, the system would still have ordered **peas**. However, upon computing t_0 , t_1 and t_2 :

$$\begin{aligned}
 t_0 &= 0 \oplus 0 \oplus 0 \rightarrow \mathbf{0} \\
 t_1 &= 0 \oplus 1 \oplus 1 \rightarrow \mathbf{0} \\
 t_2 &= 1 \oplus 0 \oplus 0 \rightarrow \mathbf{1},
 \end{aligned}$$

the system will find an inconsistency in t_2 , and our algorithm better be able to resolve this without correcting any of the data bits. In fact, it will, the conditions in lines 1, 2, 3 of the algorithm in Figure 4-1 will not be met, which means that no data bit will be modified. In detailing the procedure we

made the choice (for tidiness) of correcting also the parity bits, by adding lines 4, 5, and 6. In this case the condition in line 6 will be met and the value of the parity bit p2 will be flipped.

$$[100010] \xrightarrow{L^6} [100011].$$

The final code will look like [100011]. That is **peas**, with the correct parities.

Solution to Problem 1, part f.

Comparing [111111] with Table 4-1 we see that it is not a valid code, and it isn't either at a Hamming distance of 1 from any of the codewords. The closest vegetables in the codebook are Lima beans, Spinach and Tomatoes all at a Hamming distance of 2. The procedure outlined above (Figure 4-1) will result in the flipping of the third bit (because t_0 and t_1 will both yield 1) but will still not produce a valid codeword. At this point the system should either crash (following the spirit of old IBM machines), or inform that a malfunction has occurred, or, for more advanced systems, introduce feedback to the user (which is something our model of communication does not cover, **yet**).

Solution to Problem 2: Too Much Caffeine

Solution to Problem 2, part a.

There are many possible codebooks, and two of them are enumerated in Table 4-2.

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

Table 4-2: Possible Codebook Implementations

Solution to Problem 2, 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 2, 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 2, 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 3: A Different Hamming Code

The following code is a transcript of a matlab session solving parts a through e:

```
>>%define the generative matrix g
>>g=[1 0 0 0 1 1 1;0 1 0 0 1 1 0; 0 0 1 0 1 0 1;0 0 0 1 0 1 1]
g =
     1     0     0     0     1     1     1
     0     1     0     0     1     1     0
     0     0     1     0     1     0     1
     0     0     0     1     0     1     1
>>%define the Test matrix H
>>H=[ 1 1 1 0 1 0 0; 1 1 0 1 0 1 0; 1 0 1 1 0 0 1 ]
H =
     1     1     1     0     1     0     0
     1     1     0     1     0     1     0
     1     0     1     1     0     0     1
>>%generate all the possible 4-bit messages (here we use some matlab
>>%features, dec2bin converts frpom decimal to binary)
>>a=(dec2bin(0:15));
>>messages=reshape(str2num(a(:)),16,4)

messages =
     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
>>%generate the codebook by multiplyingthe messages by the matrix g.
>>codebook=mod(messages*g,2)

codebook =
```

```

0 0 0 0 0 0 0
0 0 0 1 0 1 1
0 0 1 0 1 0 1
0 0 1 1 1 1 0
0 1 0 0 1 1 0
0 1 0 1 1 0 1
0 1 1 0 0 1 1
0 1 1 1 0 0 0
1 0 0 0 1 1 1
1 0 0 1 1 0 0
1 0 1 0 0 1 0
1 0 1 1 0 0 1
1 1 0 0 0 0 1
1 1 0 1 0 1 0
1 1 1 0 1 0 0
1 1 1 1 1 1 1
>>%Compute the Hamming distance between any 2 codewords.
>>for j=1:16
for i=1:16
dd(i,j)=sum(xor(codebook(i,:),codebook(j,:)));
end
end
>>dd

dd =
0 3 3 4 3 4 4 3 4 3 3 4 3 4 4 7
3 0 4 3 4 3 3 4 3 4 4 3 4 3 7 4
3 4 0 3 4 3 3 4 3 4 4 3 4 7 3 4
4 3 3 0 3 4 4 3 4 3 3 4 7 4 4 3
3 4 4 3 0 3 3 4 3 4 4 7 4 3 3 4
4 3 3 4 3 0 4 3 4 3 7 4 3 4 4 3
4 3 3 4 3 4 0 3 4 7 3 4 3 4 4 3
3 4 4 3 4 3 3 0 7 4 4 3 4 3 3 4
4 3 3 4 3 4 4 7 0 3 3 4 3 4 4 3
3 4 4 3 4 3 7 4 3 0 4 3 4 3 3 4
3 4 4 3 4 7 3 4 3 4 0 3 4 3 3 4
4 3 3 4 7 4 4 3 4 3 3 0 3 4 4 3
3 4 4 7 4 3 3 4 3 4 4 3 0 3 3 4
4 3 7 4 3 4 4 3 4 3 3 4 3 0 4 3
4 7 3 4 3 4 4 3 4 3 3 4 3 4 0 3
7 4 4 3 4 3 3 4 3 4 4 3 4 3 3 0
>>%the content of dd tells us that this is a  $d=3$  code, therefore, it
>>%can correct for single-bit errors or detect up to two-bit errors.

>>%consider the following three messages (each message in one row):
>>mm=[1 0 0 0; 0 0 0 0; 1 1 1 1]
mm =
1 0 0 0
0 0 0 0
1 1 1 1

>>%obtain their codewords

```

```

>>codes=mod(mm*g,2)

codes =
     1     0     0     0     1     1     1
     0     0     0     0     0     0     0
     1     1     1     1     1     1     1
>>%test the codewords
>>checks=mod(codes*H', 2)
checks =
     0     0     0
     0     0     0
     0     0     0
>>%as expected, the syndromes yield all zeros, i.e. no errors.

>>%prepare to introduce errors in the codes
>>codes_error=codes;
>>%introduce errors in the 5th bit for the first message,
>>codes_error(1,5)= codes_error(1,5);
>>%in the 3rd bit for the second message, and
>>codes_error(2,3)= codes_error(2,3);
>>%in the 1st bit for the third message
>>codes_error(3,1)= codes_error(3,1)

codes_error =

     1     0     0     0     0     1     1
     0     0     1     0     0     0     0
     0     1     1     1     1     1     1
>>%verify tha codes and codes_error are different
>>codes_error =codes
     0     0     0     0     1     0     0
     0     0     1     0     0     0     0
     1     0     0     0     0     0     0
>>%test for errors with H
>>checks_error=mod(codes_error*H', 2)

checks_error =
     1     0     0
     1     0     1
     1     1     1
%compare with H
>>H

H =
     1     1     1     0     1     0     0
     1     1     0     1     0     1     0
     1     0     1     1     0     0     1
>>%first row of checks_error is the same as column 5 of H,
>>%second row of checks_error is the same as column 3 of H, and
>>%third row of checks_error is the same as column 1 of H.

```

Solution to Problem 3, part f.

We can see what happens when introducing two errors by reusing some of the previous matlab code. We flip the value of the 2nd and 3rd bit of the first code in the previous examples:

```
>> codes(1,[2,3])= codes(1,[2,3])
1 1 1 0 1 1 1
>> mod(codes(1,:)*H', 2)
ans =
0 1 1
```

The syndrome seems to tell us that bit 4 is wrong (compare with H), however we know that it is actually two bits (bits 2 and 3) that were flipped. In real life, not knowing the cause of the error, would lead us to erroneously correct for bit 4. Alternatively we could have used the Hamming code to simply detect errors, this removes meaning from the syndromes, but allows us to catch also 2-bit errors. Take this case as an example, we do not know if it was bit 4 or bits 2 and 3 that signalled the error but we can still safely detect there was an error in the transmission.

Solution to Problem 3, part g.

We know from the way we have identified errors before that each syndrome will be identical to one of the columns of H , precisely the column where the error occurred in the original bit. This tells us that the table of syndromes will be just the same than H^T :

bit	Syndrome
1	1 1 1
2	1 1 0
3	1 0 1
4	0 1 1
5	1 0 0
6	0 1 0
7	0 0 1

We note also that the last 3 columns of G are precisely the first four rows of this matrix, that is, the first four columns of H are the transpose of the last 3 columns of G . This observation should hint you on why do Hamming codes work.

Solution to Problem 3, part h.

The element (3,4) of matrix H had a a 0 in the original version. This must have been a typo because:

1. two columns (columns 4 and 6) would have been equal. So, upon seeing a syndrome [010] it would have been impossible to identify the faulty bit. This tells you that there must be a typo but does not tell you were it was. (Error detection in the problem set, but not correction)
2. According to the concluding remarks of part g, the first four columns of H should have been the transpose of the last 3 columns of G . In this case it was not like that. This tells you that either G or H has a typo, but you do not know which. (Again error detection in the Problem set, but not yet correction, of course you had to know the solution to g to be able to spot this.)
3. In this problem, by construction of G and H , parity bits were the last three bits (you can see that because the multiplication AG does not change the first four bits). By construction of the check matrix (H), H recomputes the parities on the data bits but does no operation on the parity bits (in this case, three last columns of H). This means that the three last columns of H should resemble an identity matrix (a matrix of zeros everywhere except for the diagonal, which is composed of ones). In the

matrix with a typo the three last columns were still an identity matrix. (This tells you that the last three columns were right but doesn't say much about the error.)

Combining the first and third explanation gives you error correction. Similarly combining the first and the second gives you error correction.