**Name:**

**Date:**

**Period:**

**Hamming Codes Tasksheet**

**Part 1:**

**Binary (Base 2) Conversion Practice**

*Convert the following Base 10 values into Binary (Base 2).*

a. 9 b. 45 c. 20

d. 12 e. 4 f. 15

*Choose a time of day. (For example I will choose 5:30pm)*

*Your chosen time of day:*  \_\_\_ **:** \_\_\_\_\_\_ \_\_\_

Let’s convert your chosen time of day into a binary code. There will be eleven binary digits (called “bits”) in your code.

Let the very first bit represent am or pm. Let it be zero for am, and let it be one for pm.

Let the next four digits represent the hour in binary.

Let the next six digits represent the minute in binary.

Example (5:30pm):

1 0101 011110

pm 5 30

*Your time of day in binary code:*

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

*We’ll pretend you are sending this time of day binary string to a friend. To make sure the string is not corrupted, we’ll learn how to protect this binary code you’ve made with a Hamming Code.*

**Part 2:**

**Parity, Parity Bits, and Parity Checks**

“Parity” comes from a latin root “par” which means “equal”. You have probably heard of *income disparity*, which is the quality of two income classes being *unequal*. Maybe you have heard of *par* in golfing, which means taking a number of swings equal to the expected number of swings. The latin “par” also is a root of the word “pair”, as in *a group of two*.

What other words can you think of that have the root “par”?

*pareja* (spanish) -- a (romantic) couple

In mathematics, “parity” is the quality of a number of being either even or odd.

*Examples:*

The number 3 has an odd parity.

The number 12 has an even parity.

The number 5 and 9 have the same parity.

Now consider a binary string, like the one below. A binary string’s parity is even if the string has an even number of “1”s. A binary string’s parity is odd if the string has an odd number of “1”s.

10101011110

What is the parity of the above binary string? Well, how many ones are there? There are seven ones, so the parity of the string is odd.

What are the parities of the following strings?

a. 01001111010 b. 10011110011

Notice that if we change just one of the bits, then the parity will change from odd to even or from even to odd. Maybe we can use this fact to detect whether or not there was an error in the message transmission. But how?

Notice that each binary string so far is exactly eleven bits long. Let’s add one more bit, on the end, and we’ll call it the *Parity Bit*. To tell the parity bit apart from the message bits, we will color the bit red. We will choose the value of this bit to make sure the overall parity is always even.

*Example:*

10101011110 10101011110**1**

No parity bit -- Odd Parity With Parity Bit -- Even Parity

Attach a parity bit to the following binary strings. Ensure that the resulting parity is always even.

a. 01001111010 b. 10011110011

Copy down your chosen time of day binary code from part (a). Attach to it an appropriate parity bit to ensure the overall parity is even.

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

Observe the strings below. Each has a parity bit already attached. Check whether the overall parity of each string is even or not. If the parity is not even, then you can know that the string was corrupted in transmission. This is called performing a “Parity Check”.

a. 101110010110 b. 000010110001

Parity: Parity:

Was there an error? Was there an error?

Because this new parity bit ensures that the parity of any message should be even, we can know that any message that arrives with an odd parity is guaranteed to have an error somewhere.

This parity has a special job. If we add more bits to our message, we will have to revise the value of our parity bit again to make sure that the parity is even. We will be adding four more bits for data protection purposes, so don’t forget to revise this bit at the very end.

**Part 3:**

**The Hamming Protection Grid**

So far, we have been able to detect whether or not there is an error in the message using parity bits. However, we have *not* been able to tell where in the string the error occurred. In order to tell where an error occurred, we will need to add four more bits to our message. Each of these bits will be a parity bit on a specific subset of the data bits.

These four new bits will be colored blue to help us remember their purpose.

To help us more visually understand these four more bits, let’s start putting our binary strings into 4x4 grids. Bear with me as I explain the grid.

Let each data digit in an 11 digit binary code be denoted by D*i*, where *i* denotes the *i*th place in the data sequence. Remember that places count from right to left, and that the right-most place is the 0th place. For example, in the string “00000000100”, the digit D2 is 1.

We’ll have a total of 5 parity bits when we’re finished, so let each parity bit be denoted by P*i*, where *i* denotes the *i*th place in the parity sequence. Remember that places count from right to left, and that the right-most place is the 0th place. For example, in the string “**01000**”, the digit P3 is 1.

A generic binary string will be represented as:

D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0 P4 P3 P2 P1 P0

The grid is filled as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| P4 | P0 | P1 | D0 |
| P2 | D1 | D2 | D3 |
| P3 | D4 | D5 | D6 |
| D7 | D8 | D9 | D10 |

This grid system is a lot to take in, so take a look at our class example on the next page.

Here’s our class example in a 4x4 grid. The red parity bit will be revised at the end, so I’ll make that bit a question mark. We haven’t attached the four remaining parity bits either, so I will denote them with question marks also.

10101011110?????

|  |  |  |  |
| --- | --- | --- | --- |
| ? | ? | ? | 0 |
| ? | 1 | 1 | 1 |
| ? | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 |

Whoo, that was quite a bit of work, just putting numbers in the grid correctly. Remember, this grid will help us understand the four new parity bits. These parity bits will enable us to identify *where* in the data string an error has occurred, not just *that* and error has occurred.

Okay, now put your chosen time of day binary code in the blank grid below. For now, leave blank spaces in the five cells devoted for the parity bits. You’ll come back to this grid to fill in the parity bits when we learn how to give these bits values.

*Your Protection Grid:*

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

|  |  |  |  |
| --- | --- | --- | --- |
| \_ | \_ | \_ | \_ |
| \_ | \_ | \_ | \_ |
| \_ | \_ | \_ | \_ |
| \_ | \_ | \_ | \_ |

**Part 4:**

**Pinpointing the Error.**

The big question of this section is, *Where is the error?*

The four new parity bits will help us. But what do they do?

Remember the purpose of a parity bit is to ensure that the parity of a certain group of bits is even. Previously, this group of bits has been the entire data string. For these new parity bits, the group of bits will be a certain half of the data string.

|  |  |  |  |
| --- | --- | --- | --- |
| ? | P0 | P1 | 0 |
| P2 | 1 | 1 | 1 |
| P3 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 |

The P0 parity bit will establish parity for the second and fourth columns together only.  
In this case, P0 should be 0 to maintain even parity. Now, we can perform a parity check on these columns only to see if there is an error in this half of the data string.

|  |  |  |  |
| --- | --- | --- | --- |
| ? | **0** | P1 | **0** |
| P2 | **1** | 1 | **1** |
| P3 | **1** | 0 | **1** |
| 0 | **1** | 0 | **1** |

The P1 parity bit will establish parity for the third and fourth columns together only.  
In this case, P1 should be 0 to maintain even parity. Now, we can perform a parity check on these columns only to see if there is an error in this half of the data string.

|  |  |  |  |
| --- | --- | --- | --- |
| ? | 0 | **0** | **0** |
| P2 | 1 | **1** | **1** |
| P3 | 1 | **0** | **1** |
| 0 | 1 | **0** | **1** |

The P2 parity bit will establish parity for the second and fourth rows together only.  
In this case, P2 should be 1 to establish even parity. Now, we can perform a parity check on these columns only to see if there is an error in this half of the data string.

|  |  |  |  |
| --- | --- | --- | --- |
| ? | 0 | 0 | 0 |
| **1** | **1** | **1** | **1** |
| P3 | 1 | 0 | 1 |
| **0** | **1** | **0** | **1** |

The P3 parity bit will establish parity for the third and fourth columns together.

In this case, P3 should be 0 to maintain even parity. Now, we can perform a parity check on these columns only to see if there is an error in this half of the data string.

|  |  |  |  |
| --- | --- | --- | --- |
| ? | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| **0** | **1** | **0** | **1** |
| **0** | **1** | **0** | **1** |

And finally, the P4 parity bit will establish overall parity for the entire grid.

In this case, P4 should be 0 to maintain even parity. This bit will enable parity checks on the whole grid to check whether or not an error occurred.

|  |  |  |  |
| --- | --- | --- | --- |
| **0** | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 |

Which is:  1010101111000100

*Calculate the parity bits for your time of day binary code, and complete your Hamming protection grid from before.*

Whoo... okay. We have our parity bits all set up properly. Now what? How do these blue parity bits help us to identify *where* the error occurred? Well, in the above grid *there is no error*. The error occurs only *during data transmission or storage*.

Let’s pretend to send this grid to Robert. Along the way, a solar flare corrupts one of the data bits and he receives the message:

1000101111000100

Robert colors the bits that he knows are the parity bits:

1000101111000100

Robert doesn’t know for sure whether this data has been corrupted or not. To check whether or not there is an error, he checks the overall parity of the string. If there is no error, then the red bit ensures that the parity is even.

Robert counts seven ‘1’s (make sure you count the ‘1’s among the parity bits too), so the parity is **odd**. Because the parity is odd, Robert knows there was an error in the data somewhere.

Because Robert knows there was an error, he decides to hunt down exactly *where* the error occurred. He puts the message into the Hamming Grid:

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 |

Robert plans to check the groups that correspond to each blue parity bis one at a time. For P0, counts the number of ‘1’s in the second and fourth columns together only. He counts **five** ones, which is **odd parity**. Robert concludes that **there is an error somewhere in these two columns**.

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | **0** | 0 | **0** |
| 1 | **1** | 1 | **1** |
| 0 | **1** | 0 | **1** |
| 0 | **0** | 0 | **1** |

**Odd Parity! Error in these two columns!**

Robert then checks P1, which corresponds to the third and fourth columns together only. He counts the number of ‘1’s he finds in those columns. He finds **four**, which is **even parity**. He concludes that **there is no error in these two columns**.

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 0 | **0** | **0** |
| 1 | 1 | **1** | **1** |
| 0 | 1 | **0** | **1** |
| 0 | 0 | **0** | **1** |

Robert pauses for a moment. He combines his knowledge from the past two parity checks, and concludes that the **error must be in the second column**.

Robert continues his parity checks and moves along to the P2 digit. He counts the number of ‘1’s in the second and fourth rows together only. He counts **five**, which indicates **odd parity**. Robert concludes that **there is an error somewhere in these two rows**.

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 0 |
| **1** | **1** | **1** | **1** |
| 0 | 1 | 0 | 1 |
| **0** | **0** | **0** | **1** |

**Odd Parity! Error in these two rows!**

Robert continues. For the P3 parity digit, Robert counts the number of ‘1’s in the third and fourth rows together only. He finds **three**, which indicates **odd parity**. He concludes that **there must be an error somewhere in these two rows**.

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| **0** | **1** | **0** | **1** |
| **0** | **0** | **0** | **1** |

Robert pauses again. Combining the knowledge of the two parity checks he just performed, Robert deduces that **there must be an error somewhere in the fourth row**.

Finally, Robert knows in which column *and in which row* the error lies:

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 |

Robert knows that the error occurred in the digit in the second column and in the fourth row. This digit corresponds to D8 in the message he received, so Robert changes D8 from zero to one, thereby correcting the error in the message:

Received:  1000101111000100

Corrected:  1010101111000100

Robert then decodes the data portion of the message, and finds that the time of day indicated in the message was 5:30pm.

*Convert your completed Hamming Protection Grid into a binary string of 16 digits. Secretly change one of the bits, and trade corrupted codes with your partner. Correct your partner’s code, then decode the time of day they encoded.*

Partner’s Name:

Partner’s Code:

Received:  \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

Corrected:  \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

|  |  |  |  |
| --- | --- | --- | --- |
| \_ | \_ | \_ | \_ |
| \_ | \_ | \_ | \_ |
| \_ | \_ | \_ | \_ |
| \_ | \_ | \_ | \_ |

Partner’s decoded time of day:

Additional notes:

This activity is meant to help students practice conversions to and from binary. This activity is quite dense, so be sure to budget enough time for the students to finish, be it as homework or over several class periods.

Hamming Codes are relatively simple to understand error protection algorithms. There are more detailed algorithms, but Hamming Codes are probably the best to start with.

It was not a coincidence that the Hamming Protection Grid was 4x4. The most commonly used protection grids are ones with a power of 2 number of cells. More possible cells can be 8x8 grids, 16x16 grids, 32x32 grids, 64x64 grids, and 124x124 grids. In each of these grids, a certain number of cells are reserved for parity bits. For a 2n x 2n grid, the 0th, 20, 21, 22, 23, …, 22n-2, and 22n-1 cells are reserved for parity bits. Larger grids than 124x124 are not typically used in real computing because if an error is prone to occur, then in larger grids it is more likely to have more than one error. Hamming codes are not reliable when trying to correct more than one error.

If Hamming Codes were still used in internet transmissions (more sophisticated algorithms are used now), then the most typical internet packet size would be 124x124/8/1028 = 1.87 kilobytes each. Packets are actually about 1.5 kilobytes each.

If the students are interested, you can show them the binary codes for the alphabet, and they can encode longer messages with 8x8 grids.

An automatic hamming code generator can be found at <http://www.ecs.umass.edu/ece/koren/FaultTolerantSystems/simulator/Hamming/HammingCodes.html>