Lab 1: Error Correction

Task 0:

If you haven't already, complete labsheet 0. The skills required for that lab are directly applied here.
You will have the opportunity to re-use some of your old code.

Task 1: (Q1)

The file *Q1.datafile* contains the output of the following code

```
#include <fcntl.h>
#include <stdint.h>
#include <unistd.h>
.....
 int  fd = open("Q1.datafile", O_CREAT|O_WRONLY, 0600);
    if(fd >= 0) {        //  IFF FILE OPENED SUCCESSFULLY
        for(int32_t i = -50 ; i<50 ; i++) {
           write(fd, &i, sizeof(i));
        }
        close(fd);
}
```

when compiled and executed on a Sun SPARC computer (a CPU architecture that is no longer common).
Note that the integers are *raw binary values*, not a textual representation produced by printf()
Write a short program to read in the *100* integers and display them

- Try to interpret what you see and why?

- Does the UNIX 'od' program help (run 'man od')
- Think about why this could be an issue when networking computers

Task 2: (Q2)

Peer into the source code of three standard cyclic-redundancy check (CRC) algorithms

- checksum_ccitt()
- checksum_crc16()
- checksum_internet()

Each accept two parameters, an array of characters or bytes and an integer indicating how many are provided.
Each function produced the checksum value of all bytes in the array. Each of the three algorithms are different
but 'combine together' the value of all bits from all bytes provided. You have been lightly introduced to
checksum_ccitt() in labsheet 0. (You do not need to understand the implementation of each fully, you may treat them as black boxes)

Write a piece of code that runs a string through each of the algorithms and prints the result.

Task 3: (Q3-6)

We can generate frames of random data with code such as:

```
#define FRAMESIZE  200
typedef unsigned char    byte;
```

```
byte    frame[FRAMESIZE];
//  SEED THE RANDOM NUMBER GENERATOR (TO PRODUCE DIFFERENT RESULT, EACH
TIME)
    srand( getpid() );
//  POPULATE THE FRAME WITH RANDOM BYTES
    for(int i=0 ; i < FRAMESIZE ; ++i) {
        frame[i] = rand() % 256;
    }
```

Using the simple function 'corrupt_frame()' and some of its suggested modifications we will investigate how robust these CRC algorithms are.
Run a large number (10000) of randomly generated frames and note the proportion of errors which are not detected by the CRC algorithms.

Task 4: (Q7)

Now implement a trivial checksum function which simply adds the value of each character in a frame. Think about what type of errors this checksum function would fail to detect.

Task 5: (Q8)

The unix system call `time()` can be used to measure the execution time of sections of code. Using this function, is any of the three CRC algorithms significantly faster than the others? A simple timing program is available in timing.c for your reference.