Assignment 1

1 Error-Correcting Codes

This assignment requires you to write a program that encodes a binary file as a Hadamard error-correcting code or decodes a Hadamard coded file to obtain the original binary file.

2 Background

2.1 Images from Mars

The Mariner 9 space probe was launched toward Mars on 30 May, 1971 and reached the planet on 14 November, of the same year, becoming the first spacecraft to orbit another planet—only narrowly beating the Soviet probes, Mars 2 and Mars 3, which both arrived within a month. The probe's mission was to compile a global mosaic of images of the planet's surface and transmit them back to Earth.

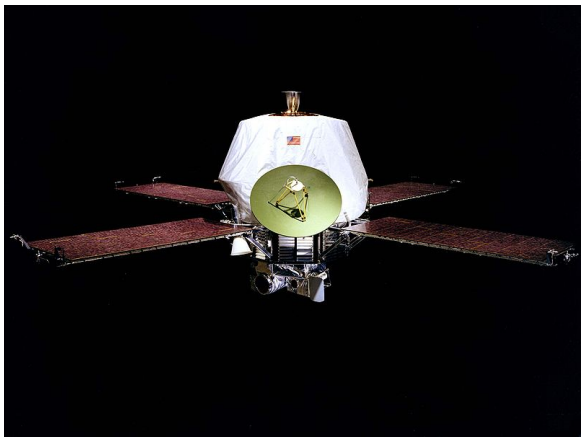

Figure 1: *Mariner 9 space probe.* `Ref:http://en.wikipedia.org/wiki/File:Mariner09.jpg`

When Mariner 9 arrived at Mars, it observed that a great dust storm was obscuring the entire globe of the planet. This unexpected situation made a strong case for studying a planet from orbit rather than merely flying past. Ground controllers sent commands to the probe to wait until the storm had abated. The storm persisted for a month but, after the dust had settled, Mariner 9 was able to obtain high-quality images of the Martian surface. It proceeded to reveal a very different planet than expected—one that boasted gigantic volcanoes and a grand canyon stretching 4,800 kilometers across its surface. More surprisingly, the relics of ancient riverbeds were carved in the landscape of this seemingly dry and dusty planet. After 349 days in orbit, Mariner 9 had transmitted 7,329 images, covering over 80% of Mars' surface and also providing the first closeup pictures of the two small, irregular Martian moons, Phobos and Deimos.

The pictures were obtained by a black and white camera and the intensity of each pixel in an image was given a value in the range 0 to 63. These numbers, expressed in binary, were transmitted to Earth, to the Jet Propulsion Laboratory at the California Institute of Technology in Pasadena. On arrival, the signal was very weak, requiring amplification, and background radiation from space and the signal and thermal noise of the amplifier had the effect that occasionally a bit transmitted as a 1 was interpreted by the receiver as a 0 and vice versa. On average, about five percent of bits were flipped. So, to permit recovery of the original image in the presence of corrupted data, Mariner 9 encoded the bit-stream as a Hadamard error-correcting code prior to transmission.

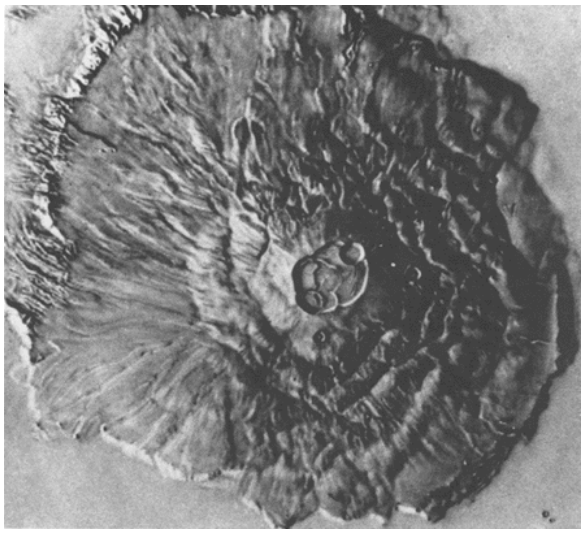

Figure 2: *Typical image of Mars taken from Mariner 9.* `Ref:http://www.thelivingmoon.com/43ancients/04images/Mars5/Mariner/fig38.gif`

2.2 The Hadamard Error-Correcting Code

The essence of an error-correcting code is to incorporate redundancy into the transmitted data stream so that a certain proportion of bit errors can be detected and corrected. For the Mariner 9 images, the set of numbers from 0 to 63 (000000 to 111111 binary) can be encoded in six bits. If the spacecraft observed the gray-scale level of a pixel as 13, for example, it could send the binary signal,

```
001101
```

However, to provide error-correction, it sends a 32-bit Hadamard encoding,

```
10100101010110101010010101011010
```

When the pattern arrives at the telemetry receiving station on Earth, the bit pattern might have been mangled to

```
10000101010101101010000001011010
  *         **        * *
```

where the asterisks denote errors that have occurred during transmission. Because each gray scale value is encoded with 32 bits instead of 6, the original message can be recovered intact. In fact, the 32-bit Hadamard code is 7-bit error-correcting; it can recover from all possible 7-bit errors in a 32-bit packet.[1]

The $N \times N$ Hadamard matrix is an $N \times N$ table of 0's and 1's with a very special property: any two rows differ in exactly $N/2$ places. The Hadamard matrices for $N = 1, 2, 4, 8, 16$, and $N = 32$ are shown in Figure 3, where white represents 0 and black represents 1. In general, the $2N \times 2N$ Hadamard matrix is constructed by aligning 4 copies of the $N \times N$ Hadamard matrix in the form of a large square, and then inverting all of the entries in the lower right $N \times N$ copy.

[1] Compare this to the more obvious *repetition code* whereby each bit is simply repeated five times, converting a 6-bit packet into a 30-bit packet. On receiving a transmission, the recovered bit value (0 or 1) is that which is most common in each 5-bit sequence. However, this is a poor code as it is only 2-bit error-correcting; such that, for a 30-bit packet there are certain combinations of 3-bit errors that cannot be corrected. In practical terms, for a signal with 5% flipped bits, a five-times repetition code permits about 0.7% errors in the decoded data on average; a 32-bit Hadamard code permits about 0.001% errors.

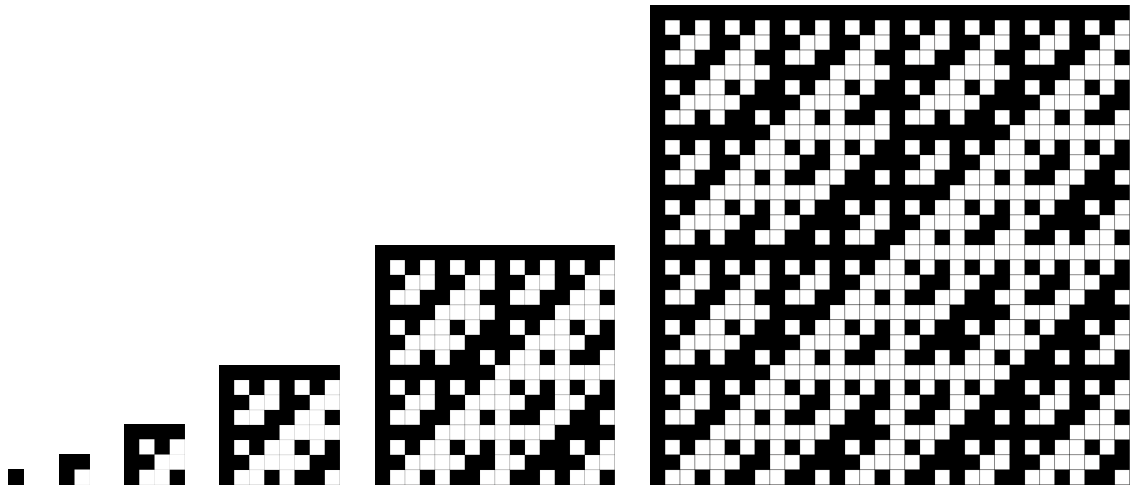

Figure 3: *Hadamard matrices for N = 1, 2, 4, 8, 16, and 32. White represents 0 and black represents 1.*

A value x is represented by row x of the Hadamard matrix. So, for example, the value 13 is encoded by row 13 of the matrix (counting, as always, from 0). For an $N \times N$ Hadamard matrix, it is possible to encode values up to $2N$; the values 0 to $N-1$ are encoded by the matrix rows directly and the values N to $2N-1$ by the rows 0 to $N-1$ with their bits inverted. Thus, for a 32-bit Hadamard code, we can encode the values 0 to 63.

Decoding a Hadamard code involves comparing an N-bit codeword with each of the N rows of the N-bit Hadamard matrix, and also with each of the N rows of the inverted matrix, until a "match" is found. If no errors were made in transmission, then the codeword will perfectly match one (and only one) of the rows. Otherwise, for a corrupted codeword, we simply choose the row that it most closely resembles; the row with the greatest number of matching bits. Why does this work? Because any two rows of an N-bit Hadamard matrix differ in $N/2$ places. Thus, for example, each row in a 32-bit Hadamard matrix differs from every other row in 16 places,[2] so if the codeword has one bit error, it will differ from the correct row in one place and from all other rows by at least 15 places. A similar thing happens if there are two, three, or even seven errors. However, if eight errors occur, then the codeword may be eight places away from an "incorrect" row. Therefore, a 32-bit Hadamard code will correctly decode *all* packets with up to seven bit errors but cannot guarantee correct decoding of packets with eight or more bit errors.

For this assignment we will be working with a 64-bit Hadamard matrix and code.

2.3 Further Reading

For further information on Hadamard codes, the following sources were used in the development of this assignment.

```
www.cs.princeton.edu/courses/archive/spring03/cs126/assignments/mariner.html
www.mcs.csuhayward.edu/~malek/TeX/Hadamard.pdf
www.quadibloc.com/crypto/mi0602.htm
```

There are many types of error-correcting codes, more sophisticated and powerful than the Hadamard code. For example, the concatenated Reed-Solomon code, used by CD-players and

[2] A row from the 32-bit Hadamard matrix also differs from each row in the inverted Hadamard matrix in 16 places, except for its exactly inverse row, where it differs in all 32 places.

computer hard-disks, is capable of correcting for bursts of 4000 consecutive bit errors. The theory of error correcting codes is a fascinating and active field of research and part of the broader field of *information theory*, which encompasses communication theory, data compression and data encryption.

2.4 Image Format and Encoded Format

The input binary file to be encoded is in the PGM format (http://en.wikipedia.org/wiki/Netpbm_format).

Image file format description:

- Each file starts with a two-byte number (in ASCII) that explains the type of file it is followed by a 'space'. In our case the image is a PGM binary file format and is represented by 'P5'
- The next two numbers give the width `u` and the height `v` of the image in pixels and are separated by spaces.
- The fourth number represents the maximum value (numbers of grey between black and white). Black is 0 and max value is white. Using a 64x64 Hadamard encoding we can only represent numbers between 0 and 127 so the max value in the provided images is 127.
- The ASCII line ends with a newline character '\n'. This line is known as a **header** as it stores information about the actual data and is read in by viewers to understand how to render the actual data.
- The second line contains the image data (in binary) with the grayscale intensity value of each pixel stored as a single byte (8-bits). Note that if we have max value of 127, we actually only need 7 bits per pixel to represent the greyscale value. However, for standard image viewers to render the image correctly, the format must have 8 bits for each pixel.

Example: The test image test.pgm has in the first line: P5 16 8 127

- This means that it is a binary pgm file, it has `u` and `v` dimensions of 16 and 8 and the max value for white is 127.
- It is recommended that you test all your code on test.pgm as it is small and easy to check values.

PGM images can be rendered using IrfFanView on a Windows machine which can be downloaded from http://www.irfanview.com/ . On a Linux machine, the standard viewer can already read PGM format. Note that test.pgm is quite small so will need to be magnified to see any detail.

Encoded File Format When encoding the image, make sure that you strip the ASCII header as it will not survive the trip from Mars to Earth and will be randomly scrambled. Given that we know the image format, we can insert the ASCII header again once the encoded data has been decoded. Remember to change the header you insert appropriately and according to the image you use.

The encoded file contains the binary encoding of the image with each pixel represented as an `unsigned long long` (64 bits).

2.5 Bit Packing

The nominal-level implementation is limited in that each 8-bit byte read from the input file must hold a value less than 128 to permit simple representation as a 64-bit Hadamard code. That is, we are only using 7 bits in each byte, and can only encode files that abide by this limitation. We can extend the program to deal with a file that packs the 7 bits back to back without padding the most significant bit with zeros. Thus each input byte may hold a value in the range 0 to 255.

- For example, let's take the first eight bytes (from the pixel data) from a standard PGM file with max value 127:

  `01111111 00110011 00001100 00110000 01111111 00110011 00001100 00110000`

- Note that the most significant bit of each byte is zero. We can pack these eight 7 bit values into seven bytes by removing the padded zeros and putting the 7 bit values back to back:

  `11111110 11001100 01100011 00001111 11101100 11000110 00110000`

- So our packed image files (with extension.pak) can be $7/8^{th}$ the size of the original PGM format.

- You are to unpack a packed image file so that it could be then encoded with your implementation. You are also required to pack a PGM file into a packed image file. This involves some bitwise trickery.

- You should copy the PGM header into the packed file however leave the header UNPACKED.

3 The Implementation

- You must have a header file called `hadamard.h` with at least the following function declarations:

```
void create_hadamard_matrix(char** h, int N);

void convert_to_lookup_table(unsigned long long* lookup, char **h, int N);

void pgm_read(char* fin, char** img, int u, int v);

void pgm_write(char* fout, char** img,  int format, int u, int v, int depth);

void enc_write(char* fout, char** img, int u, int v,
        unsigned long long* lookup);

void enc_read(unsigned long long* encin, char* fin, int u, int v);

void dec_img(char** img, unsigned long long* encin,
        unsigned long long* lookup, int u, int v, int N);

void enc_pack(char* fin, char* fout);

void enc_unpack(char* fin, char* fout);

void dec_msg(char* fout, unsigned long long* encin, unsigned long long* lookup,
        int u, int v, int N);
```

Do not change these definitions, a template hadamard.h has been included in the support files.

- You must have a file called `hadamard.c` with at least the above function definitions (i.e. the implementation of those functions).
- You must have a file called `main.c` which calls these functions at some point in your main program.

The following functions must be implemented, marks will be allocated as shown.

1. `void create_hadamard_matrix(char** h, int N)`

   [1 mark] Create a N-bit Hadamard matrix. You should implement this as a N by N multidimensional `char` array. In each array element store a decimal 1 or a 0 value according to the Hadamard format described above. Store the Hadamard matrix in `h`. The `N` variable indicates the size of the Hadamard matrix i.e. if `N` = 4 then a 4×4 Hadamard matrix would be filled in `h`.

   Hints:

   - To fill the Hadamard matrix with the appropriate 1's and 0's, first set the top-left element, then copy it to make a 2×2 Hadamard, then copy that to make a 4×4 Hadamard, and so on until the entire $N \times N$ matrix is filled.
   - The assignment specifies the use of a 64-bit Hadamard matrix however your function should be able to produce any 2^k-bit Hadamard matrix.

2. `void convert_to_lookup_table(unsigned long long* lookup, char **h, int N)`

   [1 mark] Using the Hadamard matrix generate a 128-element lookup array of `unsigned long long`. For i between 0 and 63, the i-th array element will hold the bit-pattern of the i-th row of the Hadamard matrix. For i between 64 and 127, the i-th array element will hold the inverted bit-pattern of the $(i-64)$-th row of the Hadamard matrix. Store the look up table in `lookup`. The `N` variable indicates the size of the hadamard matrix `h` used to generate the lookup table.

   Hints: This will require some basic bitwise operations. Some useful operations are described below.

   - To create a variable of 64 one bits: `unsigned long long ones = 0xffffffffffffffffllu;`
   - To create a variable of 64 one bits when the size of the variable type is exactly 64-bits: `unsigned long long ones = ~0llu;`
   - To create a variable of `n` ones: `unsigned long long ones = ~(~0llu << n);` Note, this will not work if `n` is greater than or equal to the size of the variable type. For instance, don't use `~(~0llu << 64)` for a 64-bit integer type.
   - To turn on bit `n` write: `x |= (unsigned long long) 1 << n;`
   - To determine whether bit `n` is on write: `if (x & ((unsigned long long) 1 << n))`
   - To invert all bits in `x` write: `y = x ^ ones;`
   - Elements 0 to 2 of the lookup array are 18446744073709551615, 6148914691236517205, 3689348814741910323, and elements 64 to 66 are 0, 12297829382473034410, 14757395258967641292. See that these are the decimal values of the Hadamard bit-patterns stored in an `unsigned long long` data type.

3. `void pgm_read(char* fin, char** img, int u, int v)`

   [0.5 mark] Read a PGM image file, `fin`, of image dimensions `u` and `v` into the array `img`. Hints:

- Don't forget to strip the header lines when reading in the image.

4. `void pgm_write(char* fout, char** img,  int format, int u, int v, int depth)`

   [**0.5 mark**] Write a PGM image from array `img` to the file `fout` with header constructed from `format`, `u`, `v` and `depth`. The `format` variable denotes the Netpbm encoding format e.g. `format` = 5 would be portable graymap encoded in binary. The `depth` variable is equal to the maximum value of the pixels used in the PGM file. Hints:

   - Don't forget to print the header line when writing out to the file. Remember that the header is terminated with a new line character '\n'.

5. `void enc_write(char* fout, char** img, int u, int v, unsigned long long* lookup);`

   [**1 mark**] Write the PGM image encoding out to a file i.e.

   (a) Read a single byte from the `img` array.

   (b) Convert the byte to a 64-bit Hadamard code. For byte value i, the Hadamard code is the i-th element in the `lookup` array.

   (c) Write code to the output file `fout`.

   (d) Continue until the end of the `img` array is reached

   Hints:

   - Files are opened and closed using `fopen()` and `fclose()`, respectively. Single bytes may be read and written using `fputc()` and `fgetc()`. Multiple bytes may be read and written using `fread()` and `fwrite()`.
   - Be sure to open your files in binary mode by passing either `"rb"` or `"wb"` to `fopen()`.
   - To write a 64-bit value in one go: `fwrite(&code, 8, 1, fp);`

6. `void enc_read(unsigned long long* encin, char* fin, int u, int v)`

   [**0.5 mark**] Read in the encoded PGM image from a file i.e.

   (a) Read a 64-bit Hadamard code from the input file `fin`.

   (b) Store in `encin`.

   (c) Continue until all of the encoded values are read from `fin`.

7. ```
   void dec_img(char** img, unsigned long long* encin,
           unsigned long long* lookup, int u, int v, int N)
   ```

   [**1.5 mark**] Decode the encoded PGM pixel values in `encin` i.e.

   (a) Read a 64-bit Hadamard encoding from `encin`.

   (b) Search for the nearest matching code in `lookup` constructed from an `N` by `N` hadamard matrix. Compare the encoding with each `lookup` element and count the number of matching bits. If the i-th code in the array is the best matching code, then the decoded byte value is i.

   (c) Store i in the `img` array.

   (d) Continue until all of the encoded values in `encin` have been decoded.
   ```

Hints:

- Bit counting i.e. counting the number of 1's in a set of bytes. Bit counting for a single byte x:

```
for (i=0; i<8; ++i, x >>= 1)
diff += x&1;
```

- To mark non-matching bits as 1's write: `diff = x ^ y;`

8. `void enc_pack(char* fin, char* fout)`

   [**1 mark**] Pack the PGM pixel values in `fin` from 8-bits down to 7-bits and store in `fout`. Fill the last packed byte with 0's if required. Hints:

   - Remember to copy the PGM header information to the packed file UNPACKED!

9. `void enc_unpack(char* fin, char* fout)`

   [**1 mark**] Unpack the PGM pixel values in `fin` from 7-bits to 8-bits and store in `fout`. Hints:

   - Remember to copy the PGM header information to the unpacked file.

10. `void dec_msg(char* fout, unsigned long long* encin, unsigned long long* lookup, int u, int v, int N)`

    [**1 mark**] Occasionally, the space radiation that is sometimes "corrupting" the image encoding is in fact aliens trying to communicate us! The aliens have an alphabet of 32 characters in the following order

```
char alphabet[] = {'@','#','M','B','H','A','&','G','h','9','3','X','2','5','S',
    'i','s','r',';',':',',',',','.',' ','\'','"','%','=','+','!','_','\r','\n'}
```

    They have "corrupted" the encoded image values by flipping one bit out of every 32 bits compared to the corresponding `lookup` value. For example, say the first 32 bit values of an uncorrupted `encin` was equal to

    `11111111111111111111111111111111111111111111111111111111111111`

    However the aliens "corrupted" the first 32 bit values of `encin` by flipping bit 4 (starting from bit 0 on the left)

    `11110111111111111111111111111111111111111111111111111111111111`

    This would encode the letter `'H'` corresponding to the `alphabet char` array. Continue this process by checking all the encoded image values stored in `encin` and print the decoded alien message into the file `fout`. Hints:

    - As the alien alphabet is 32 characters long, there could potentially be 2 encoded letters for each `encin` value.
    - The first 32 characters of the decoded alien message is the given alien alphabet.

4 Specification

The program `hadamard.exe` is to read an input binary-file and write to an output binary-file. Depending on a command-line switch, the program will either *encode* a file with Hadamard codes, *decode* a file of (possibly corrupted) Hadamard codes to obtain an error-corrected file, *pack* the image file or *unpack* the image file and *extract* an alien message that could potentially be hidden in a Hadamard code.

You have been provided with two images `test.pgm` and `mars.pgm`. The test.pgm file contains 128 pixel values from 0 to 127 and the `mars.pgm` contains an image of mars as scene from the Mariner 9 space probe.

You have also been provided with four encoded files: `test.enc`, `mars.enc`, `marsmsg.enc` and `marsnoise.enc`. The `test.enc` and `mars.enc` files contain Hadamard encoding for the `test.pgm` and `mars.pgm` images respectively. The `marsmsg.enc` file contains a "corrupted" Hadamard encoding which has a hidden message from alien beings. The `marsnoise.enc` file contains a corrupted Hadamard encoding where 10% of the bits have been artificially flipped to simulate actual noise encountered in the Mars the Earth trip. Finally the `test.pak` file contains the `test.pgm` image but packed into 7 bits.

You should use these given files to test your assignment code. You can use file comparison programs like `diff` to compare two files or `fc` to compare two files in a binary fashion with the /b operator. Notepad++ can also be used with the HEX-Editor plugin to view the binary values contained in the file in either hex or binary format. All given files will be similar to the files that we use to check and grade your code. Additional specifications include

- The program is to be called `hadamard.exe`.
- The program is to be run from the command-line and takes three mandatory input arguments, the function to perform, the input and output filenames. It also takes additional arguments, (`format u v depth`) to tell program the PGM format, the image size and the max pixel intensity.
- If the program is run without input arguments, it is to quit with the following message,

```
Usage: hadamard [option] <file 1> <file 2> {format u v depth}

Option:

  -e encrypt file 1 and place in file 2 using u v
  -d decrypt file 1 and place in file 2 using format u v depth
  -p pack encoded data from file 1 into file 2
  -u unpack encoded data from file 1 into file 2
  -m extract alien message from file 1 and place in file 2 using u v

Examples:

  hadamard -e input.pgm out.enc 832 700
  hadamard -d out.enc out.pgm 5 832 700 127
  hadamard -p input.pgm input.pak
  hadamard -u input.pak input.pgm
  hadamard -m out.enc msg.txt 832 700
```

- The program is to assume that the first filename refers to the input file and the second to the output file. The names will not contain whitespace.
- If the program cannot open a specified file, it is to print an error message,

```
Error: Could not open file <filename> in mode <opening-mode>.
```

  where `<filename>` will be the name of the file and `<opening-mode>` will be the mode `rb` or `wb`. The program will then quit.

- If the program is doing encoding and it reads a byte that has a value greater than 127, it is to print an error message and quit.

```
Error: Cannot handle characters with value greater than 127.
```

- You should not have any other files so that the following command will compile your program `gcc main.c hadamard.c -o hadamard.exe`

4.1 Memory allocation

The assignment will require you to use dynamic memory allocation to utilise some of the functions. This can be achieved by using the function `malloc` located in the `stdlib.h` header. For example the following code can be used to dynamically allocate a `char** img` variable with dimensions `u` and `v`

```
int i;
char** mat = malloc(u*sizeof(char*));
for (i = 0; i < u; i++)
{
    mat[i] = malloc(v*sizeof(char*));
}
```

IMPORTANT: Function parameters which require dynamic memory allocation should only be allocated OUTSIDE the function! Thus all dynamic memory allocations for function parameters should take place in your `main.c` file. It is also good programming practise to free any memory you have allocated through the `free` function located in `stdlib.h`.

4.2 Marking

4.2.1 Total Marks

The assignment will be out of 9 and scaled to 30% of your total grade. Each part of the algorithm will be marked as detailed in the Specification Section. For example, we will check your hadamard matrix and if correct you will be awarded 1 mark and if incorrect you will be awarded 0 marks. Your code will be checked on a Windows 7 machine using MinGW.

WARNING: If your code does not compile or results in a segmentation fault you will get 0 marks for the entire assignment. We will not debug your code to make it compile or run.

4.2.2 Submission

Your assignment will be due on the *20th of September* at *5pm*. Submission will be online through blackboard.

Submission Format The assignment should be submitted as a single zip file. It should contain a single folder with your student number as its name, e.g. n45726412. Within this folder should be 3 files, `hadamard.c`, `hadamard.h` and `main.c`. Running `gcc main.c hadamard.c -o hadamard.exe` must compile these files to a working `hadamard.exe` program.

WARNING: We take plagiarism very seriously if your assignment has material which has been copied from other class members or previous assignments you will be penalised.