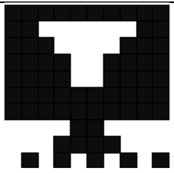
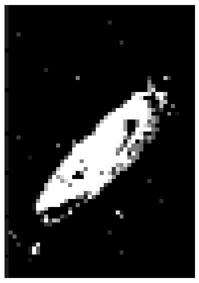
COMP2401 - Assignment #2

(Due: **Tuesday, February 7th, 2023 @ 11pm**)

In this assignment, you will work with arrays of strings and pack data into bytes based on bits.

---

(1) Data compression is an important topic in computer science because as time goes on, more and more data becomes available and there is always a need to reduce the amount of storage required to save the data. In this assignment, you will implement a basic “lossless” compression scheme called **run-length encoding**. The idea is to take an original set of byte data and reduce the amount of storage space needed to store the data without losing any data. You will write a program to compress 5 images using 3 different techniques and see which one is the best. The images will be simple binary images composed of only black and white pixels as follows:

				
Small	Solar	Mona	Puppy	Trees

The images are stored as arrays of strings in a file called **images.c**. Here on the right is the string array for the first image.

```
char  *small[] = {"1111111111",
                  "1100000011",
                  "1110000111",
                  "1111001111",
                  "1111001111",
                  "1111111111",
                  "1111111111",
                  "0000110000",
                  "0001111000",
                  "0101010101"};
```

At the bottom of the **images.c** file, the following variables have been defined. You should use these in your program to process the images:

```
char           **images[] = {small, solar, mona, puppy, trees};
char            *names[] = {"Small", "Solar", "Mona", "Puppy", "Trees"};
unsigned short    rows[] = {10, 75, 200, 181, 300};
unsigned short    cols[] = {10, 100, 134, 200, 280};
unsigned char   numImages = 5;
```

A **runLength.c** file has also been included for you as a template to begin. It has **extern** definitions for the above variables, so that you can use them and it also has the **main()** function that you will use to test your program. You should not modify the **main()** function, except temporarily to test … such as commenting/uncommenting code so that you can test one image or one compression function at a time.

Keep in mind that to compile your code … you need to include both C files:

```
student@compxxxx:~$ gcc -o runLength runLength.c images.c
```

You will need to write the following functions. They correspond to encoding and decoding an image in 4 ways. To encode an image, each function should fill up some kind of shared global array that will represent the encoded image. Then to decode, you simply read from the encoded image. Each of the 4 encoding functions must take the following 3 parameters, otherwise the **main()** function testing won't work:

```
char              *imageArray[]     // image to be encoded
unsigned short     numLines         // # of lines/rows in the image
unsigned short     lineLength       // # of pixels in a line (i.e. columns)
```

The encoding functions MUST each return an **unsigned int** representing the number of bytes that were required to compress/encode the image.

Each of the 4 decoding functions must take the following 2 parameters, otherwise the **main()** function testing won't work:

```
unsigned short numLines       // # of lines/rows in the original image
unsigned short lineLength     // # of pixels in a line (i.e. columns)
```

Follow the instructions for each function below:

basicByteEncode()

This function should store each pixel of the image (traversing from left to right, row by row) into a single byte (i.e., **unsigned char**) of the encoded image … each byte having a value of either 1 or 0 that corresponds to the 1's and 0's in the image data. It does NOT do any compression.

For example, consider the first two rows of an image as shown on the right. The encoded bytes will be:

```
1111000111
0110111000
```

1,1,1,1,0,0,0,1,1,1,0,1,1,0,1,1,1,0,0,0

basicByteDecode()

This function should take data from the encoded image (assuming that it was done using basic byte encoding) and then display the image as text using '*****' for black pixels and ' ' for white pixels. See example output at the bottom of this assignment.

bitEncode()

This function should store eight consecutive pixels (traversing from left to right, row by row) into a single byte (i.e., **unsigned char**) of the encoded image. You will need to read the pixels and store each one as a bit of an encoded byte. The first 8 pixels of the first row will represent the first byte of the encoded image. The next 8 pixels represent the next byte … and so on. For example, if the first 8 pixels of the image are “01001101” then a single encoded byte of 77 should be stored in the encoded image. The images typically have rows that are not multiples of 8. So, when reaching the end of a row, you will need to wrap-around to the next row … not skipping any bits when building the encoded byte.

For example, consider the first two rows of an image as shown on the right. After the first byte is encoded there are 2 pixels remaining in the first row, so we need to start the next encoded byte with these. The highest 2 bits will be from the first row, while the lowest 6 bits will be taken from the pixels starting in the second row. The first two encoded bytes will be: **241**, **219**. As for the last 4 pixels of row 2, we don’t know what to do with them yet. If there is another row, then we wrap around again as we just did so that ‘1000’ are the highest bits of the next encoded byte. However, if there are no more rows, then we convert the ‘1000’ to a proper 8-bit byte by ensuring that ‘1000’ are the highest bits of that last byte, with zeros for the lowest 4 bits. So in that case, the final encoded byte of the image would be **128**.

```
1111000111
0110111000
```

bitDecode()

This function should take data from the encoded image (assuming that it was done using bit encoding) and then display the image as text using '*****' for black pixels and ' ' for white pixels.

rowRunLengthEncode()(traversing from left to right, row by row)

This function should traverse the image from left to right, row by row. As encountering pixels, you should store the total count of consecutive pixels with the same value (i.e., all ones or all zeros) as the encoded byte. If, when reading pixels, the value changes from one to zero (or vice-versa), then you need to output your encoded byte and start a new encoded byte. Keep in mind that you may count more than **255** consecutive pixels on the same row, so this may be more than a byte. You will need to output multiple bytes, with a zero in between to indicate that the pixel type did not change (i.e., that they are all still 1’s or 0’s). Assume, for example, that the image had a line length of **600** and that all pixels in a row were **1**. In that case, you will have a count of **600** and you should output the following **5** bytes … **255, 0, 255, 0** and **90** for that row. The numbers add up to **600** and the **0**’s indicate not to switch pixel values in between (i.e., indicates that all **600** are the same pixel type). When you reach the end of a row while counting, write out the encoded byte and start a new encoded byte for the next row. Make sure to check if the counter was above **255** again, in case you need to write out more than one byte again (with **0** in between them). The algorithm assumes that each row starts with a **1** pixel, so if it starts with a **0** pixel, you will need to output **0** for the count of the **1** pixels that were not there.

For example, consider the first two rows of an image as shown on the right. The encoded bytes will be: 4, 3, 3, 0, 1, 2, 1, 3, 3. Notice that

```
1111000111
0110111000
```

the green **0** here, represents the fact that the 2nd row did not start with a **1** pixel. If it did … then you would not have output that **0**. The row run-length encoding will assume that the image starts with a **1** pixel. If the image starts with a **0** pixel, then you should start the encoded image with a **0** as the first byte to indicate that there were no **1** pixels.

rowRunLengthDecode()

This function should take data from the encoded image (assuming that it was done using row run-length encoding) and then display the image as text using '*****' for black pixels and ' ' for white pixels.

variableRunLengthEncode()

This function is similar to row run-length encoding in that it stores the total count of consecutive pixels with the same value as the encoded bytes. The difference is that when the row ends, we keep counting similar consecutive pixels in the next row instead of stopping to output.

For example, consider the first two rows of an image as shown on the right. The encoded bytes will be: 4, 3, 10, 5, 5, etc… The variable run-length encoding will also assume that the image starts with a **1** pixel. If the image starts with a **0** pixel, then you should start the encoded image with a **0** as the first byte to indicate that there were no **1** pixels.

```
1111000111
1111111000
0011111000
```

variableRunLengthDecode()

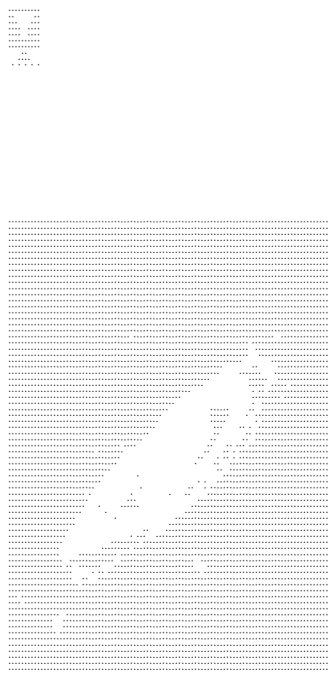
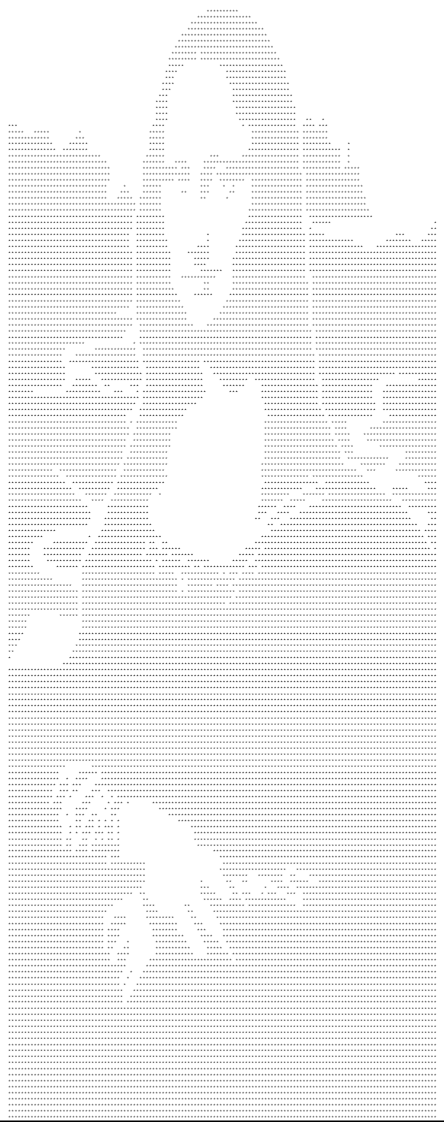
This function should take data from the encoded image (assuming that it was done using variable run-length encoding) and then display the image as text using '*****' for black pixels and ' ' for white pixels.

The following numbers should be obtained from each of the images and compression schemes. Many of you will obtain slightly different results for the row and variable run-length encoding schemes. It is easy to be off by a bit if you do not output the required bytes (especially the green zeros mentioned earlier when starting off a row or in between consecutive chunks more than 255 in length). Instead of struggling for hours trying to get the exact number, move on to the next function and come back to it later if you have time.

```
Image: Small
     Basic Byte Encoding Bytes Required = 100
     Bit Encoding Bytes Required = 13
     Row Run-Length Encoding Bytes Required = 34
     Variable Run-Length Encoding Bytes Required = 23

Image: Solar
     Basic Byte Encoding Bytes Required = 7500
     Bit Encoding Bytes Required = 938
     Row Run-Length Encoding Bytes Required = 307
     Variable Run-Length Encoding Bytes Required = 251
```

```
Image: Mona
     Basic Byte Encoding Bytes Required = 26800
     Bit Encoding Bytes Required = 3350
     Row Run-Length Encoding Bytes Required = 1377
     Variable Run-Length Encoding Bytes Required = 1213

Image: Puppy
     Basic Byte Encoding Bytes Required = 36200
     Bit Encoding Bytes Required = 4525
     Row Run-Length Encoding Bytes Required = 7724
     Variable Run-Length Encoding Bytes Required = 7527

Image: Trees
     Basic Byte Encoding Bytes Required = 84000
     Basic Bit Encoding Bytes Required = 10500
     Fixed Run-Length Encoding Bytes Required = 6638
     Variable Run-Length Encoding Bytes Required = 6303
```

Here is what the images should look like when printed. Note that the image should be identical for ALL of the compression schemes If you want to confirm the larger ones, you may have to copy/paste the output from the terminal into something like MS word and shrink it. These images here are shown with **courier** font at a very small point size of **2**.

IMPORTANT SUBMISSION INSTRUCTIONS:

Submit all of your **.c source** files and all other files needed for testing/running your programs. DO NOT TAR your files. Make sure that your name and student number is in each source file at the top as a comment.

The code **MUST** compile and run on the course VM.

- If your internet connection at home is down or does not work, we will not accept this as a reason for handing in an assignment late ... so make sure to submit the assignment WELL BEFORE it is due !

- You WILL lose marks on this assignment if any of your files are missing. So, make sure that you hand in the correct files and version of your assignment. You will also lose marks if your code is not **written neatly with proper indentation and containing a reasonable number of comments.** See course notes for examples of what is proper indentation, writing style and reasonable commenting).